THESE

Présentée par

Frédéric GRUAU

devant l'ECOLE NORMALE SUPERIEURE DE LYON
l'UNIVERSITE CLAUDE BERNARD-LYON I

pour l'obtention
du DIPLOME DE DOCTORAT
Spécialité: Informatique

SYNTHÉSE DE RÉSEAUX DE NEURONES PAR CODAGE CELLULAIRE ET ALGORITHMES GÉNÉTIQUES.

Date de soutenance:	le 4 Janvier 1994	
Composition du Jury:	Président:	M. Maurice Nivat
	Rapporteurs:	M. Jean-Arcady Meyer M. Jacques Demongeot
	Examinateurs:	M. Michel Cosnard M. Jacques Mazoyer M. Pierre Peretto M. Darell Whitley

Thèse préparée au Centre d'étude nucléaire de Grenoble
au sein du Département de Recherche Fondamentale matière condensée
et à l'Ecole Normale Supérieure de Lyon,
au sein du Laboratoire de l'Informatique du Parallélisme (LIP-IMAG),
Unité de Recherche Associée au CNRS

Remerciements

Je remercie Michel Cosnard pour avoir dirigé ma thèse. J'ai eu une grande chance de l'avoir comme patron de thèse. Je me sens une dette envers lui, et je ne saurais donc laisser de côté le capital qu'il m'a inculqué. Il m'a accordé sa confiance depuis avant le début jusqu'à après la fin. Je le considère comme un Bodisatthva[1] de l'informatique du parallélisme. On peut résumer l'efficacité de Michel Cosnard par une équation: 10 minutes de Cosnard = une semaine de travail, en temps et en argent. Michel Cosnard m'a donné le goût de la recherche, et de la communication par la fréquentation des laboratoires. Michel Cosnard a été le premier en France à s'intéresser aux réseaux de neurones génétiques. Aujourd'hui, plusieurs équipes travaillent sur ce sujet. Demain, je suis persuadé que cela deviendra le tronc d'arbre du connexionisme. Il m'a lancé sur ce domaine, et m'a toujours donné la bonne direction. Bien que à l'origine de beaucoup d'idées contenues dans cette thèse, il n'a jamais mis son nom sur mes publications. Je remercie Pierre Peretto pour avoir accepté d'être mon responsable CEA. Il m'a apporté une grande aide durant ma thèse. Il m'a aidé à rédiger mon premier article et m'a fait découvrir un monde nouveau: la physique statistique. Pierre Peretto m'a prodigué mille encouragements, m'a laissé ma liberté, et m'a éduqué sur la profession de chercheur. Grâce à lui j'ai bénéficié d'un cadre de travail idéal. Je lui en suis vraiment reconnaissant. Je remercie Mirta Gordon pour m'avoir fait participer à l'enseignement en cours de DEA. Mirta m'a aussi donné de bonnes idées en ce qui concerne l'apprentissage que j'ai mis au point. Jean-Louis Pautrat a été généreux avec moi, pour les missions au début de ma thèse. Alain Bourret m'a permis de garder ma bourse de thèse CFR jusqu'en Septembre. Jacque Mazoyer m'a donné de nouvelles idées. J'ai eu des échanges très fructueux avec Darell Whitley et Heinz Müehlenbein. John Koza m'a généreusement fait part des idées contenues dans son deuxième livre "GPII" deux ans avant sa publication. Je remercie les nombreuses personnes de l'ENS-LYON qui ont participé à mon environnement intellectuel. Heléne Moisy, Pascal Koiran, Alain Darte, Marianne Delorme, Luc Bougé Marc Daumas. Valérie Roger m'a fait découvrir les splendeurs de la mode. Je remercie Jean-Yves Ratajszczac et Gilles Wiber avec qui j'ai eu une expérience inoubliable de travail d'équipe. On a réalisé ensemble un de mes rêves (le compilateur neuronal). Je remercie Maurice Nivat qui m'a fait l'honneur d'être président du jury, Jean-Arcady Meyer et Jacques Demongeot qui ont acceptés d'être rapporteurs. Je remercie respectueusement André Meissner, responsable du dojo zen de Lyon. Il m'a donné une base solide et indestructible: le zen. Je remercie mes parents de leur attention bienveillante, bien que je les voie rarement, ils sont toujours présents en moi. Ils ont relu ma thèse. Bernard Tourancheau m'a prêté son login sur la machine parallèle iPSC860 128 noeuds, à Oakrige. Je remercie les copains André Mougin, Olivier Ulrich et Christian Gamrat pour la sympathie qu'il m'ont témoingnée, et en particulier, pour le fait de m'avoir généreusement attribué un bureau ensoléillé et spacieux.

[1] Bodisatthva désigne dans le Bouddhisme, quelqu'un qui bien que n'ayant pas reçu l'éveil cherche à sauver les autres avant lui-même. Pour cela, au lieu de se retirer dans un monastére, il n'hésite pas à plonger dans les phénomènes et à se salir lui même. C'est la plus haute manière de pratiquer une religion à ma connaissance.

Résumé

Autrefois considérés comme une machine vouée à un apprentissage réalisé à l'aide de petites modifications successives de paramètres, les réseaux de neurones artificiels sont en train de changer de statut. De telles méthodes d'apprentissage ne permettent pas en effet de générer de gros réseaux nécessaire à la résolution de problèmes conséquents. Cette thèse soutient les trois points de vue suivant:

- 1) Le mot clé pour sortir de l'impasse est "modularité".
- 2) L'outil clé pour générer des réseaux de neurones modulaires est le codage cellulaire.
- 3) La méthode d'optimisation clé pour explorer l'espace de recherche des codes cellulaires est l'Algorithme Génétique.

La premier point est une idée à présent répandue. Par réseaux de neurones modulaires, on entend un réseau de neurones se décomposant en sous-réseaux, d'une façon hiérarchique, avec des sous-réseaux qui peuvent se répéter. Cette thèse se décompose en deux parties. La première partie démontre le deuxième point. Le codage cellulaire est presenté comme un véritable langage machine de réseaux de neurones, avec des fondements théoriques (c'est en fait une grammaire de graphe, vérifiant un certain nombre de propriétés) et un compilateur de langage evolué. La deuxième partie démontre le troisième point. L'application des algorithmes génétiques à la synthèse de réseaux de neurones en utilisant le codage celulaire est une nouvelle technologie. Cette technologie permet de résoudre des problèmes encore jamais résolus. Elle peut automatiquement et dynamiquement décomposer un problème en une hiérarchie de sous-problèmes, et générer un réseau de neurones dont la structure hiérarchique de sous-réseaux reflète celle du problème. Elle permet d'expérimenter des domaines de recherche encore vierges comme par exemple, l'interaction entre apprentissage génétique et apprentissage de poids ou la mise au point d'apprentissages connectionistes adaptés à l'algorithme génétique.

Table des matières

Chapitre 1

Introduction aux réseaux de neurones génétiques

1.1 Les algorithmes génétiques

Les algorithmes génétiques ont pour fonction de rechercher le minimum d'une fonction f définie sur des chaînes de bits de longueur fixe. Ils utilisent pour cela une population entière de chaînes de bits qui évolue durant plusieurs générations, jusqu'à ce qu'une solution suffisamment bonne soit trouvée ou que le temps imparti soit écoulé. Les chaînes de bits sont appelées chromosomes L'évolution de la population est déterminée par trois étapes:

- Durant l'étape de sélection les individus sont reproduits avec une probabilité plus grande s'ils sont meilleurs, c'est-à-dire, s'ils ont une image par f, petite relativement aux autres individus.

- Pendant l'étape de croisement les individus sont rassemblés deux par deux. Dans chaque couple les partenaires échangent un bout de leurs chaînes de bits.

- A l'étape de mutation, chaque individu modifie un de ses bits, ceci avec une faible probabilité.

Un théorème appelé "théorème des shémas" garantit l'amélioration globale de la population d'une génération sur l'autre, à condition qu'une certaine hypothèse appelée "hypothèse des building blocks" soit vérifiée. Les algorithmes génétiques sont robustes, ils peuvent s'appliquer à n'importe quel problème de recherche à condition de pouvoir coder les solutions sur des chaînes de bits. Ils sont facilement parallélisables, et peuvent même présenter une accélération superlinéaire, l'algorithme parallèle étant différent de l'algorithme séquentiel. On a présenté ici la version de base de l'algorithme génétique qui date de 1975, époque à laquelle le père des algorithmes génétiques, John Holland a édité son livre: "Adaptation in natural and artificial environment". Depuis 1975, beaucoup de recherches ont été menées sur le sujet, aux USA et en Allemagne (stratégie évolutive). Ces recherches établissent des propriétés mathématiques, expérimentent plusieurs types d'opérateurs de croisement, mutation et sélection, explorent d'autres opérateurs issus d'analogies avec la nature ainsi que des manières de coder autres que la chaîne de bits. Le paragraphe suivant présente un exemple où l'on utilise des arbres étiquetés plutôt que des chaînes de bits. La traduction anglaise de Algorithme Génétique est "Genetic Algorithm". Par la suite on utilisera souvent "GA" pour Algorithme Génétique, car c'est une expression très usuelle.

1.2 La programmation génétique

John Koza [17] a utilisé les algorithmes génétiques pour faire évoluer des programmes LISP. LISP est un langage de programmation fonctionnel. Un programme LISP est une S-expression qui se présente comme un arbre étiqueté dont les branches sont ordonnées. Les étiquettes sont des symboles de fonctions LISP, Les feuilles sont étiquetées avec des constantes ou des entrées du programme. La sortie du programme est une valeur calculée à partir de la S-expression. Durant la recombinaison de deux arbres mère et père, un sous-arbre est coupé de l'arbre mère et vient remplacer un sous-arbre de l'arbre père. Koza n'utilise pas de mutation. Il considére que la mutation est déjà implantée par la recombinaison, lorsque les sous-arbres échangés sont réduits à des feuilles. Ainsi, Koza utilise l'algorithme génétique décrit ci-dessus sans l'étape de mutation, appliqué à des arbres étiquetés au lieu des chaînes de bits. L'espace de recherche de l'algorithme génétique est l'ensemble des arbres étiquetés. Koza a écrit un premier livre [17] où il montre que la programmation génétique permet de générer automatiquement des programmes pour résoudre un grand nombre de problèmes dans des domaines très variés. Il a écrit un deuxième livre [18] où il utilise des S-expressions LISP permettant la définition automatique de fonctions et leur réutilisation. Il a montré que l'usage de fonctions permet de résoudre des problèmes plus compliqués. Nous avons utilisé la même technique de codage sous forme d'arbres étiquetés. Nous verrons dans la conclusion que la méthode de génération de réseaux de neurones proposée dans cette thèse peut être qualifiée de programmation génétique, et pas seulement parce que le GA manipule des objets ayant la même structure de données.

1.3 Les réseaux de neurones

On peut comprendre les algorithmes génétiques comme un modèle d'apprentissage dans lequel l'information apprise est répartie sur tous les individus. Les réseaux de neurones représentent un autre modèle d'apprentissage de nature très différente, par rapport aux algorithme génétiques. Un réseau de neurones est un graphe orienté de cellules, chaque cellule calcule une activité à partir des activités provenant de ses voisines connectées en entrée, et propage à son tour son activité vers les cellules connectées en sortie. Le calcul fait par une cellule consiste à appliquer une fonction sigmoïde à la somme pondérée des entrées. Les poids représentent l'information contenue dans le graphe de cellules. On definit la fonction réalisée par le réseau, cette définition variant suivant le type de réseau utilisé. On veut faire apprendre au réseau une certaine fonction. Pour cela, le plus souvent, une procédure d'ajustement progressif des poids est utilisée: Au cours d'une itération, les poids sont modifiés par une petite quantité calculée de façon à minimiser un gradient. Plus récemment des recherches ont été menées pour essayer aussi d'optimiser la topologie du graphe de cellules. Ces recherches utilisent encore un ajustement progressif. Au cours d'une itération, un neurone ou une connexion est ajouté ou supprimé. Chang [19] a écrit tout un livre sur ce genre de méthodes, communément appelées "algorithmes constructivistes". Bien que les algorithmes constructivistes optimisent l'architecture, ils le font toujours par ajustements progressifs à partir d'une solution initiale. Les algorithmes constructivistes ne peuvent donc pas synthétiser des réseaux de neurones ayant une structure modulaire reflétant la structure du problème à résoudre. C'est à dire, un réseau de neurones se décomposant en plusieurs unités fonctionelles.

1.4 Réseaux de neurones génétiques

Il y a un nombre relativement grand de publications qui considérent plusieurs combinaisons entre algorithmes génétiques et réseaux de neurones. Une présentation de ce travail est donnée dans

Schaffer, Whitley and Eshelman [29] dans une introduction aux actes de la conférence *Combinations of Genetic algorithms and Neural Networks.* Les algorithmes génétiques ont été utilisés pour faire de l'apprentissage de poids, supervisé ou non-supervisé. Les algorithmes génétiques ont aussi été utilisés pour sélectionner les exemples à apprendre et pour interpréter le comportement d'un réseau de neurones. Finalement les GA ont aussi été appliqués au problème qui consiste à trouver des architectures de réseaux de neurones. C'est le problème qui nous intéresse. Une spécification d'architectures indique combien d'unités cachées le réseau doit avoir, et comment ces unités doivent être connectées.

Les GA ne procédent pas par ajustements progressifs. L'opérateur génétique de recombinaison permet une recherche de nature différente. Pour cette raison, le GA peut générer des architectures de réseaux de neurones modulaires. Pour appliquer les GAs à la recherche de bonnes architectures de réseaux de neurones, il y a seulement un problème à résoudre: comment coder une architecture sur des structures manipulables par l'algorithme génétique, appelées chromosomes. Trois classes de méthodes existent:

Codage direct Une structure de données de graphe est codée. Dans [33], la matrice d'adjacence[1] du réseau est codée directement. Cela donne des chromosomes très longs de taille n^2 pour un réseau comprenant n neurones. A cause de ce problème d'échelle, seulement de petits réseaux (4 unités cachées) ont été trouvés. De plus, opérer la recombinaison directement sur les connections "physiques" peut produire des réseaux de neurones non fonctionnels. Cela est appelé le problème structurel/fonctionnel dans [33]. La solution à ce problème consiste à coder sur le chromosome une collection de caractéristiques abstraites plus apte à fournir des "blocs de base". Il doit y avoir de l'abstraction dans la représentation.

Codage de paramètres . Une liste de paramètres est codée. Dans [9] les paramètres décrivent le nombre de couches, la taille des couches, et de quelle façon les couches sont connectées entre elles. Cette méthode est plus abstraite que la précedente. Elle peut produire de gros réseaux avec de petits chromosomes, mais seulement pour une classe réduite d'architectures: les architectures en couches. Elle ne peut pas générer d'architectures modulaires, la modularité étant définie formellement au paragraphe 3.4.

Codage grammatical Une grammaire de réécriture est codée. La grammaire est interprétée de manière récursive, et permet de générer une famille de réseaux de neurones. Dans [15] Kitano commence à implanter un codage ayant des propriétés d'échelle, de modularité et d'abstraction. Dans sa méthode, le chromosome code une grammaire de matrices. Une règle de cette grammaire réécrit une matrice 1×1 en une matrice de 2×2. L'application des règles est faite comme suit: On commence par une matrice 1×1 notée M_0. Cette matrice a un seul élément qui est l'axiome de la grammaire. Durant la première étape, en utilisant la règle qui réécrit l'axiome, cette matrice est transformée en une matrice 2×2 de caractères que l'on appelle M_1. A la deuxième étape, chaque caractère de M_1 est réécrit en une matrice 2×2. Le résultat est une matrice 4×4. Le processus continue jusqu'à la matrice $2^k \times 2^k$ appelée M_k, où k est le plus petit entier vérifiant $2^k > n$. Le symbole n est mis pour le nombre de neurones contenus dans le réseau de neurones cible. On extrait de la matrice M_k une sous-matrice $n \times n$ et on remplace chacun de ses éléments par 1 ou 0, suivant une convention déterminée à l'avance. On obtient finalement une matrice booléenne $n \times n$, cette matrice est la matrice d'adjacence du réseau de neurones codé. L'idée de coder une grammaire est

[1] La matrice d'adjacence d'un graphe comportant n noeuds est une matrice booléenne $n \times n$. L'élément (i, j) vaut 1 s'il y a une connexion du noeud i vers le noeud j, et 0 sinon.

nouvelle et intéressante. On peut coder des gros réseaux sur des petits chromosomes sans imposer au réseau d'avoir une forme particulière.

La dernière méthode est nettement avantageuse par rapport aux deux premières. En optimisant des grammaires qui génèrent des architectures de réseaux plutôt que directement des réseaux, on espére obtenir une meilleure propriété d'échelle, et d'une certaine façon la possibilité de réutiliser des architectures de réseaux. En d'autres termes: le but de cette méthode est de trouver des règles pour générer des réseaux de neurones qui seront utiles pour définir des architectures pour une classe générale de problèmes. En particulier, cela permettra aux chercheurs qui font du développement de définir des grammaires sur des problèmes plus petits et d'utiliser ensuite ces grammaires comme des briques de base pour attaquer des problèmes plus importants.

Cependant la méthode de Kitano présente certaines difficultés, dues au fait que Kitano utilise une grammaire de matrices. Une matrice $m \times m$ doit être développée pour un réseau de n neurones, m étant une puissance de 2 exacte,la plus petite puissance de 2 plus grande que n. Si on veut obtenir un graphe acyclique, pour un réseau sans cycle, on doit considérer la moitié de la matrice: le triangle supérieur droit, cela diminue encore plus l'efficacité.

Mjolness [23] définit des équations récursives de matrice à partir desquelles il calcule une famille de matrices entières, puis une famille de réseaux incluant les poids. Mjolness utilise aussi une grammaire de matrices. Son approche ressemble à celle de Kitano. Contrairement à Kitano, Mjolness n'est pas contraint d'utiliser des matrices dont la taille est une puissance de 2 exacte. Il n'est cependant pas facile de savoir quelles sont les différentes tailles possibles. De plus la taille des réseaux doit être déterminée à l'avance.

1.5 Plan de cette thèse

Les méthodes de codage proposées par Kitano et Mjolness sont toutes deux basées sur une grammaire de matrices. Dans cette thèse, on propose une approche basée sur les grammaires de graphes. La méthode s'appelle codage cellulaire. Un codage cellulaire développe directement une famille de réseaux de neurones, et évite de passer à travers la représentation matricielle. Au lieu d'une matrice, l'objet sur lequel les règles grammaticales sont appliquées est la cellule d'un réseau de cellules. Chaque cellule contient une copie du code. Chaque cellule lit le code à une position différente. Suivant ce qu'elle lit, une cellule peut se diviser, changer l'état de ses registres internes, et finalement devenir un neurone. Il est à la fois plus naturel et plus efficace de faire agir le développement au niveau des cellules plutôt qu'au niveau des éléments d'une matrice d'adjacence. Le codage résultant peut être considéré comme un langage permettant de décrire les réseaux de neurones d'une façon claire et compacte, et la représentation peut être recombinée efficacement par les algorithmes génétiques.

Dans la première partie de cette thèse, on démontrera l'intérêt du codage cellulaire pour luimême, en le considérant comme une méthode de codage. On présentera la méthode de codage cellulaire, et on démontrera une liste de propriétés. Avec l'utilisation du GA, malgré l'importance critique du codage, il n'y a jamais eu de tentatives faites pour établir une liste de propriétés que ce codage devrait avoir dans le cas particulier de l'optimisation des réseaux de neurones. On montrera qu'il est possible de définir la liste des propriétés formelles, qu'un codage de réseaux de neurones doit vérifier. La représentation d'un chromosome est compacte et abstraite. Tout chromosome peut développer un phénotype valide (le phénotype désigne l'entité obtenue après le décodage du chromosome, ici le phénotype est un réseau de neurones). Le processus de développement génére des architectures modulaires et interprétables, avec une propriété d'échelle puissante. On montrera qu'il est basé sur une grammaire parallèle de graphe.

Ces propriétés théoriques montrent que le codage cellulaire est adapté au GA. On va ensuite décrire un compilateur Pascal qui produit le code cellulaire d'un réseau de neurones. Ce dernier exécute les calculs spécifiés par le programme Pascal. Cela prouvera que le codage cellulaire peut vraiment être considéré comme un langage machine pour réseaux de neurones. Le compilateur en lui-même présente des intérêts. C'est un outil pour générer des gros réseaux de neurones. Il peut être utilisé pour la parallélisation automatique de programmes Pascal à condition que ceux-ci soient écrits avec une stratégie "diviser pour régner". Enfin le compilateur peut permettre de compiler des systèmes hybrides.

Dans la seconde partie de cette thèse, on montrera l'efficacité du codage cellulaire relativement à l'algorithme génétique. Une simulation sur des problèmes non-encore résolus auparavant sera décrite. Le GA est utilisé pour trouver à la fois les poids booléens (± 1) et l'architecture de réseaux booléens paramètrés qui calculent des fonctions booléennes paramètrées. Le GA tout seul peut trouver un réseau de neurones solution, ainsi il n'est pas nécessaire de faire de la rétro-propagation du gradient pour optimiser les poids, cela permet d'aller beaucoup plus vite. Cependant, on propose quand même d'utiliser un apprentissage, pour améliorer les performances. Le but de l'apprentissage et la façon dont il est combiné avec le GA sont particuliers. Le GA n'est pas utilisé pour trouver des architectures adaptées à un apprentissage particulier. Au lieu de cela, c'est l'apprentissage qui est utilisé pour accélerer le GA. On compare les accélérations obtenues suivant les différentes façons possibles de combiner apprentissage et GA. Le GA permet un parcours global de l'espace de recherche, et trouve un ensemble de sous-réseaux que le réseau solution du problème doit inclure. L'apprentissage ajuste les réseaux de neurones bruts produits par le GA, et peut ainsi accélerer la recherche. On montre qu'une certaine combinaison GA et apprentissage qui utilise l'effet Baldwin, peut être plus efficace que le mode proposé par Lamark; dans ce mode, l'information apprise est recodée sur le chromosome, et le chromosome modifié est réinjecté dans la population. La nouvelle combinaison proposée s'appelle apprentissage Developpemental. On propose un apprentissage appelé "apprentissage Switch" qui est adapté à l'usage du GA. C'est un apprentissage rapide qui se contente de changer quelques poids ± 1 et qui fonctionne pour des réseaux ayant beaucoup de couches, à condition que ceux-ci soient proches de la solution. L'algorithme génétique parallèle que nous avons programmé est décrit. Un exemple de recherche modulaire est proposé. Cet exemple utilise toutes les améliorations apportées: (apprentissage developpemental, switch learning, GA parallèle). La programmation génétique utilise le GA pour faire évoluer des S-expressions LISP (LISP est un langage de programmation particulier dans lequel un programme est une expression parenthésée appelée S-expression). Dans la conclusion on compare la synthèse génétique de réseaux de neurones utilisant le codage cellulaire et la programmation génétique.

Partie I: Le codage cellulaire

Dans cette première partie, on va présenter le codage cellulaire et étudier ses propriétés.

Chapitre 2

Présentation du Codage Cellulaire

2.1 Codage cellulaire de base

Le codage cellulaire est une méthode pour coder des familles de réseaux de neurones pareillement structurés. Dans ce chapitre on présente le codage cellulaire de base. Dans les chapitres suivants on étendra l'alphabet du codage cellulaire, au fur et à mesure que le besoin s'en fera sentir. Le codage cellulaire de base sera utilisé par l'algorithme génétique pour synthétiser des réseaux de neurones, dans la seconde partie de cette thèse. Dans les chapitres de la première partie, on va considérer des extensions du codage cellulaire indépendamment de l'utilisation de l'algorithme génétique.

Le code cellulaire est représenté par un *arbre grammaire* dont les branches sont ordonnées et dont les noeuds sont étiquetés par des noms d'opérateur cellulaire. Le lecteur ne doit pas faire la confusion entre arbre grammaire et grammaire d'arbres. Un arbre grammaire indique une grammaire codée sous la forme d'un arbre, alors qu'une grammaire d'arbres signifie une grammaire qui réécrit des arbres. Une cellule est le noeud d'un graphe orienté dont les connexions sont ordonnées: ce graphe est appelé réseau de cellules. Chaque cellule transporte sa propre copie du code cellulaire (i.e de l'arbre grammaire) et a une tête de lecture interne qui lit les étiquettes de l'arbre grammaire. En général, chaque cellule lit l'arbre grammaire à un endroit différent. Les *opérateurs cellulaires* représentent des instructions pour le développement des cellules. Ils agissent sur une cellule ou sur les connexions qui arrivent à la cellule. Pendant un pas du processus de développement, une cellule exécute l'instruction correspondant à l'opérateur cellulaire lu par sa tête de lecture, et descend sa tête de lecture d'un niveau vers le bas, dans l'arbre grammaire. Il y a une analogie entre une cellule et une machine de Turing. Une cellule lit un arbre au lieu d'un ruban, et une cellule est capable de se dupliquer elle-même. Mais cellule et machine de Turing exécutent toutes les deux les instructions lues en déplaçant leur tête de lecture d'une manière dictée par l'instruction lue.

Une cellule gère aussi un ensemble de registres internes. Certains d'entre eux sont utilisés durant le développement du graphe de cellules, d'autres déterminent les poids et les seuils des neurones du réseau final. Le registre lien est utilisé pour pointer sur une connexion particulière de la cellule en entrée ou en sortie.

Considérons le problème: "Obtenir un réseau de neurones qui calcule la fonction booléenne OU exclusif (XOR)". Les neurones peuvent avoir des seuils 0 ou 1. Les connexions peuvent être pondérées par ± 1 Si la somme pondérée de ses entrées est strictement plus grande que son seuil, l'activité du neurone est mise à 1, sinon elle est mise à 0. Les entrées d'un neurone sont donc 0 ou 1.

Le développement du graphe de cellules commence avec une cellule unique appelée cellule ancêtre connectée à la cellule pointeur entrée, et à la cellule pointeur sortie du réseau. Considérons le graphe de cellules initial représenté sur la figure 2.1. A l'étape de départ, la tête de lecture de l'ancêtre est positionnée sur la racine de l'arbre, comme le montre la flèche qui connecte les deux. Ces registres sont initialisés avec des valeurs par défaut. Par exemple le seuil est initialisé à 0. Au fur et à mesure que cette cellule se divise et se redivise, elle donne naissance à toutes les autres cellules qui vont à la fin devenir des neurones finis, et former le réseau de neurones final. On dit qu'une cellule devient un neurone fini, quand elle cesse de se développer et qu'elle perd sa tête de lecture qui est devenue inutile. La cellule pointeur entrée et la cellule pointeur sortie auxquelles est reliée la cellule ancêtre (indiquées par des rectangles dans les figures) n'exécutent pas d'instructions. Mais toutefois, à la fin du développement, la cellule pointeur entrée est connectée à la liste des unités d'entrée, et la cellule pointeur sortie est connectée à la liste des unités de sortie. Ces unités d'entrée et de sortie sont créées durant le développement elles ne sont pas rajoutées artificiellement à la fin. Quand le développement est terminé, les cellules pointeurs peuvent être détruites. Par exemple, sur la figure 2.5, le réseau de neurones décodé final comporte deux unités d'entrée étiquetées a et c et une unité de sortie étiquetée d. Nous allons à présent passer en revue la liste des opérateurs cellulaires qui sont utilisés pour développer le graphe de cellules, en un réseau de neurones.

- Un opérateur de division cellulaire crée deux cellules à partir d'une. Lors d'une division séquentielle (notée `SEQ`) la première cellule fille hérite des liens d'entrée de la cellule mère, la seconde cellule fille hérite des liens de sortie, la première cellule fille est connectée vers la seconde avec un poids 1. Comme il y a deux cellules filles, un opérateur de division cellulaire doit étiqueter des noeuds d'arité deux. La première cellule fille ira lire le sous-arbre de gauche, la deuxième cellule fille ira lire le sous-arbre de droite. L'opérateur `SEQ` est illustré aux étapes 1 et 3.

- La division parallèle notée `PAR`, est une deuxième sorte d'opérateurs de division. Les deux cellules filles héritent à la fois des liens d'entrée, et des liens de sortie de la cellule mère. (étapes 2 et 6). Enfin, il faut préciser que lorsqu'une cellule se divise, les registres de la cellule mère sont recopiés chez les cellules filles.

- L'opérateur de terminaison noté `END`, fait perdre la tête de lecture à la cellule qui l'exécute. Après cela, la cellule n'exécute plus d'opérateurs, on dit qu'elle est devenue un neurone fini. Puisque la cellule n'a plus à lire de sous-arbres, `END` étiquette les feuilles de l'arbre grammaire.

- Un opérateur de modification de registre modifie la valeur d'un registre interne à la cellule. L'opérateur `INCBIAS` incrémente (et `DECBIAS` décrémente) le seuil de la cellule. L'opérateur `INCLR` incrémente (et `DECLR` décrémente) la valeur du registre lien. Ce registre sert à repérer une connexion d'entrée particuliére. Changer la valeur de ce registre lui fait repérer une connexion différente. Le registre lien a une valeur par défaut égale 1. Il pointe donc sur le lien d'entrée le plus à gauche. Des opérations sur les autres connexions d'entrée peuvent être accomplies, en changeant la valeur du registre de liens. L'opérateur noté `VAL+` positionne le poids de la connexion d'entrée pointé par le registre lien sur 1, alors que `VAL-` positionne le poids sur -1. Les opérateurs `VAL+` ou `VAL-` n'indiquent pas explicitement à quelle connexion entrante particulière le changement de poids est destiné. Quand `VAL+` ou `VAL-` est exécuté il est appliqué sur le lien pointé par le registre lien.

- Un opérateur unaire `CUT` supprime le lien pointé par le registre lien. Cet opérateur modifie la topologie, en supprimant une connexion, alors que la modification du registre lien ne modifie pas la topologie.

Les opérateurs `INCLR`, `DECLR` et `CUT` ne sont pas illustrés. Ils ne sont pas nécessaires au développement du réseau de neurones, pour le problème du XOR. La séquence avec laquelle les cellules exécutent les opérateurs cellulaires est déterminée comme suit: Dès qu'une cellule a exécuté un opérateur, elle entre dans une file FIFO (First In First Out). La prochaine cellule qui se développera sera la tête de la file FIFO. Si la cellule se divise, la cellule qui va lire le sous-arbre de gauche entre dans la file FIFO en premier. Cet ordre d'exécution essaye de modéliser ce qui se passerait si les cellules étaient actives en parallèle. Il assure qu'une cellule ne peut pas être active deux fois, alors qu'une autre n'a pas été active du tout. Dans certains cas la configuration finale du réseau dépend de l'ordre dans lequel les cellules exécutent leurs opérateurs. Par exemple, durant le développement du XOR, effectuer l'étape 7 avant l'étape 6 produirait un réseau de neurones avec une unité de sortie ayant deux poids d'entrée à -1 au lieu d'un seul. L'opérateur d'attente `WAIT` fait attendre la cellule jusqu'au prochain tour. `WAIT` est nécessaire pour les cas où l'ordre du développement doit être controlé en générant des délais appropriés.

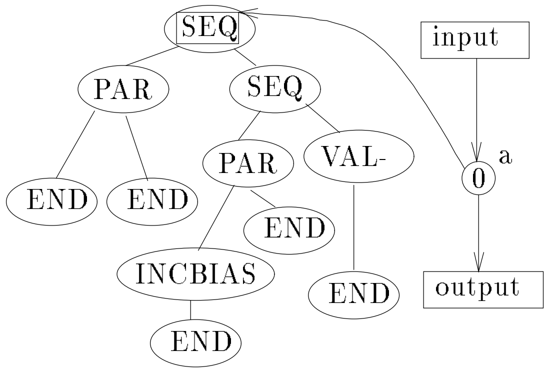

Figure 2.1: Étape 0: Développement du réseau de neurones calculant la fonction booléene OU exclusif (XOR), sur la partie droite de cette figure. Le développement commence avec une seule cellule ancêtre étiquetée *a*, qui est représentée par un cercle. Le 0 à l'intérieur du cercle indique que le seuil de la cellule est 0. la cellule ancêtre est connectée à la cellule pointeur entrée du réseau de neurones (rectangle étiqueté "input") et à la cellule pointeur sortie du réseau de neurones (rectangle étiqueté "output"). Le codage cellulaire du réseau de neurones pour le XOR est représenté sur la partie gauche de cette figure. La flèche entre la cellule ancêtre et le symbole `SEQ` de la grammaire de graphe représente la position de la tête de lecture de la cellule ancêtre. Une ligne continue indique une connexion avec un poids 1, une ligne tirets indique un poids -1. Tous les poids sont initialisés à 1.

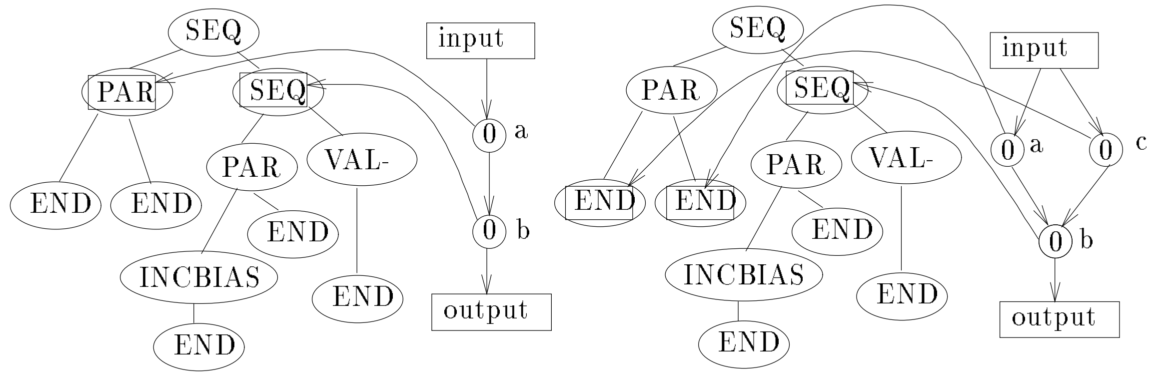

Figure 2.2: A gauche, l'exécution à l'étape 1 de la division séquentielle `SEQ` pointée par la cellule a de la figure précédente, a pour effet de diviser la cellule a en deux cellules a et b. La cellule a est connectée vers la cellule b avec un poids $+1$. La tête de lecture de la cellule a pointe maintenant vers le sous-arbre gauche du codage cellulaire à gauche (Le rectangle avec `PAR`). La tête de lecture de la cellule b pointe vers le sous-arbre droit (le rectangle au deuxième niveau vers le bas à droite avec `SEQ`). A droite l'exécution de la division parallèle `PAR` à l'étape 2 entraîne la création d'une cellule c. Les deux cellules a et c héritent des entrées qui auparavant arrivaient sur a. Les deux cellules a et c envoient leur sortie à la cellule b (l'endroit où a envoyait sa sortie auparavant) La tête de lecture de a pointe sur le sous-arbre de gauche (`END`), et la tête de lecture de la nouvelle cellule c pointe à présent sur le sous-arbre de droite qui a aussi un `END`.

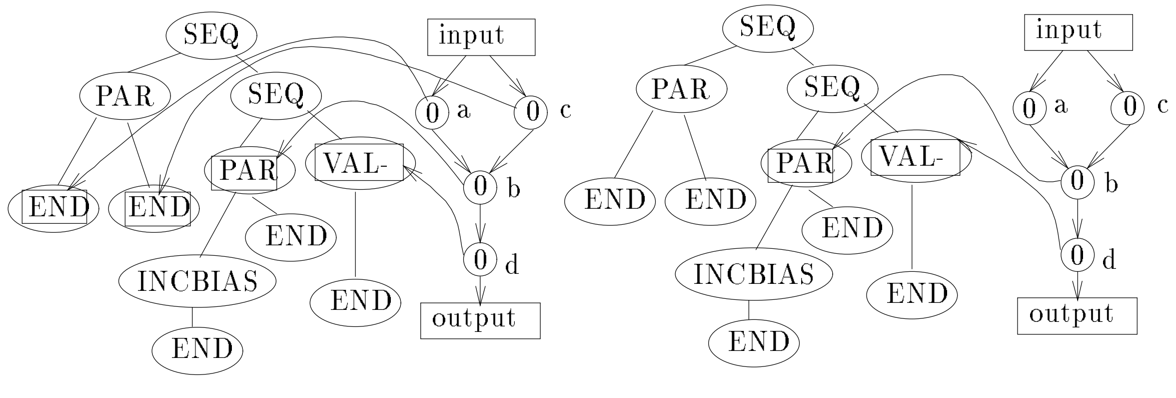

Figure 2.3: A gauche: l'exécution à l'étape 3 de la division séquentielle `SEQ` pointée par la cellule b de la figure précédente, entraîne la division de la cellule b en deux cellules b et d. La cellule b est connectée vers la cellule d avec un poids $+1$. La tête de lecture de la cellule b pointe sur le sous-arbre gauche (le rectangle étiqueté `PAR`), et la tête de lecture de la nouvelle cellule d pointe maintenant sur le sous-arbre de droite (le rectangle au troisième niveau vers le bas, à droite avec `VAL-`). A droite: exécution des deux opérateurs `END`. Ils font perdre les têtes de lecture des cellules a et b qui deviennent des neurones finis. Comme il y a deux opérateurs `END`, cela prend deux étapes (les étapes 4 et 5), une étape pour chaque opérateur `END`.

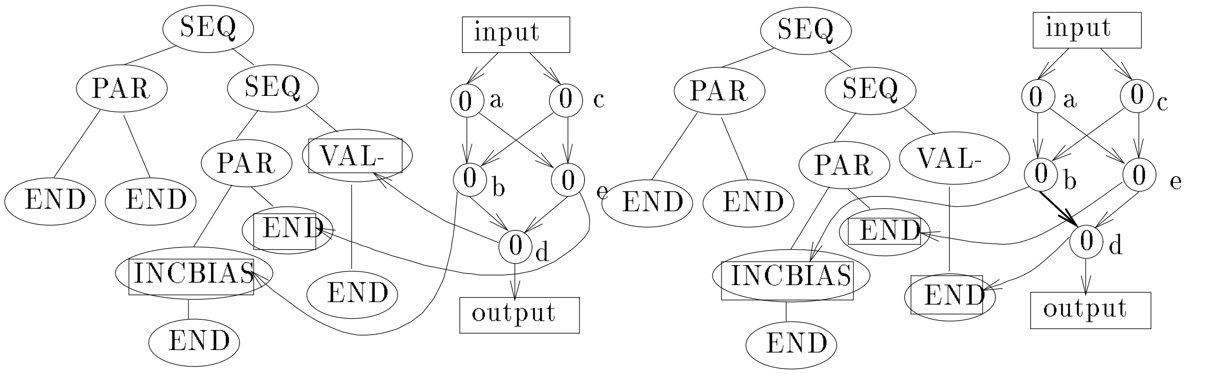

Figure 2.4: A gauche: l'exécution de la division parallèle `PAR` par la cellule b à l'étape 6 crée une cellule e. Les deux cellules b et e héritent des connexions venant des cellules a et c qui auparavant arrivaient sur b. Les deux cellules b et e envoient leur sortie sur la cellule d (endroit ou la cellule b envoyait sa sortie auparavant.) La tête de lecture de la cellule b maintenant pointe sur le sous-arbre gauche (`INCBIAS`), et la tête de lecture de la nouvelle cellule e pointe sur le sous-arbre de droite qui est étiqueté avec un `END`. A droite: la cellule c exécute un opérateur `VAL-` le registre lien vaut 1 (sa valeur par défaut). Il pointe sur la connexion la plus à droite. L'action de `VAL-` consiste à positionner le poids de la première connexion sur -1. La ligne épaisse est utilisée pour indiquer un poids -1. Après l'exécution de `VAL-` la cellule c va exécuter l'opérateur suivant, qui est un `END` situé sur le quatrième niveau de l'arbre.

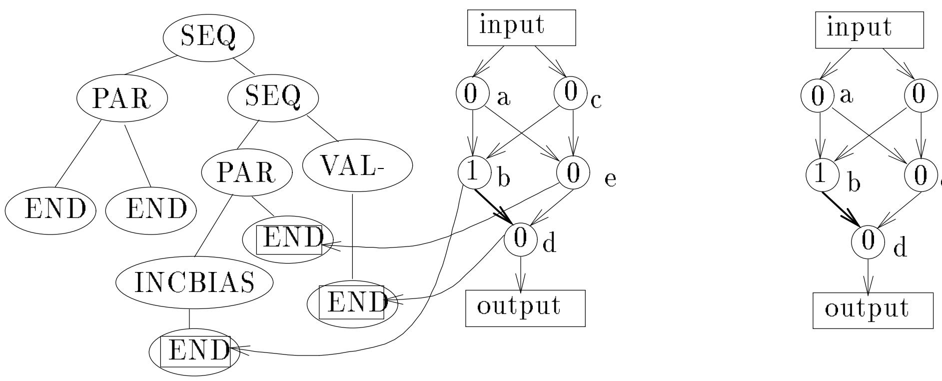

Figure 2.5: A l'étape 8, sur la gauche le neurone b exécute l'opérateur de modification de registres `INCBIAS`. l'action de `INCBIAS` consiste à augmenter le seuil de la cellule b de 1. Après l'exécution, le seuil de la cellule b est 1, et la tête de lecture de la cellule b pointe sur un `END` au cinquième niveau de l'arbre. A droite, les trois dernières étapes 9,10,11 consistent à exécuter les trois opérateurs `END`. Ces trois opérateurs `END` font perdre aux cellules b, e et d leur tête de lecture, et b, e et d deviennent des neurones finis. Puisque il n'y a plus que des neurones finis, le développement est terminé. Le réseau de neurones final possède deux unités d'entrée a et c, et une unité de sortie d. Le neurone a est le premier neurone d'entrée, parce que le lien de la cellule pointeur entrée vers a est le premier lien sortant de la cellule pointeur entrée.

2.2 Codage de la récursivité

Jusqu'à maintenant dans notre description, la grammaire n'utilise pas de récursions. (Il faut bien remarquer que le fait que la grammaire soit recurrente n'implique pas que le réseau codé soit récurrent.) On ne peut développer qu'un seul réseau à partir d'un arbre grammaire. Mais le but est de développer une famille de réseaux partageant la même structure, pour calculer une famille de problèmes structurés de la même façon. Cela permettrait d'obtenir une propriété d'échelle puissante. Dans ce but, nous avons introduit un opérateur de sauts récurrents `REC` qui permet de boucler à travers les règles d'une grammaire récurrente un nombre de fois L prédéterminé. On munit la cellule d'un registre appelé `vie` qui sert à mettre à jour L. Le registre `vie` doit être initialisé avec la valeur L initiale. La cellule qui exécute `REC` suit l'algorithme suivant:

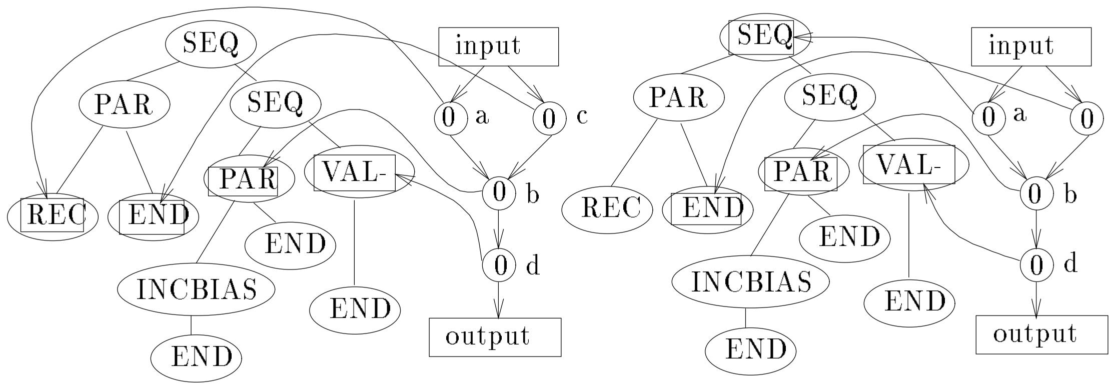

Figure 2.6: Un opérateur de développement récurrent pour un réseau de neurones calculant la parité de trois entrées. La vie de la cellule ancêtre est 2. Les trois premières étapes sont les mêmes que dans le cas du XOR. A gauche on représente le graphe de celllules à l'issue de ces trois étapes. A droite, pendant la quatrième étape, la cellule a exécute un opérateur de saut récurrent `REC` qui lui fait déplacer sa tête de lecture tout en haut sur la racine de l'arbre (`SEQ`), La cellule a recommence un développement, comme si elle était repartie à l'étape 0. La vie de la cellule a a été décrémentée et vaut à présent 1. La cellule a va donner naissance à un autre réseau de neurones XOR dont les sorties seront envoyées aux cellules filles générées après la division de la cellule b.

```
vie:=vie-1
si (vie > 0) deplacer la tete de lecture sur la racine de l'arbre.
sinon Deplacer la tete de lecture sur le sous-arbre.
```

L'utilisation de cet opérateur cellulaire est illustrée figure 2.6 et 2.7. Le code cellulaire (i.e. l'arbre étiqueté par les opérateurs cellulaires) de ces figures est presque le même que celui du sous-réseau XOR vu à la section précédente. La seule différence est qu'un opérateur `END` a été permuté avec un opérateur `REC`. Le code cellulaire résultant est capable de développer un réseau de neurones arbitrairement grand, pour calculer la fonction booléenne parité, avec un nombre arbitrairement grand d'entrées. Pour cela, il spécifie comment composer les uns derrière les autres, des sous-réseaux de neurones XOR. La figure 2.7 montre le réseau de neurones correspondant à une parité à trois entrées. Cette façon d'introduire la récurrence permet un contrôle précis du processus de croissance.

Step 22

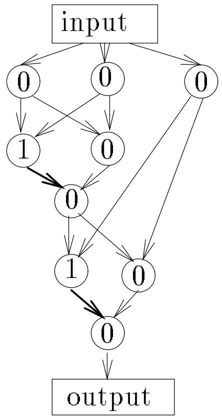

Figure 2.7: Le réseau de neurones décodé final, qui calcule la parité de trois entrées. La vie de la cellule ancêtre est 2. Le nombre total d'étapes dans le développement est 22 car chaque opérateur de l'arbre grammaire est exécuté deux fois. On peut voir clairement les deux sous-réseaux XOR qui sont cascadés. En prenant une vie initiale de L, on aurait développé un réseau de neurones pour la parité de $L+1$ entrées, comportant L sous-réseaux XOR cascadés.

Le développement n'est pas arrêté quand la taille du réseau atteint une limite prédéterminée, mais plutôt quand le code a été lu exactement L fois. L'entier positif L paramètre la structure des réseaux de neurones.

2.3 Comparaison avec la méthode proposée par Kitano

La méthode de Kitano et le codage cellulaire codent tous les deux une grammaire. Kitano code une grammaire de matrices. Le codage cellulaire code une grammaire de graphes. Au lieu de réécrire sur des matrices, la grammaire du codage cellulaire réécrit des cellules neuronales. En fait, réécrire des cellules revient à définir un véritable langage machine de réseaux de neurones. Ce langage peut décrire les architectures, ainsi que des motifs de poids, d'une façon claire et naturelle. Pendant le processus de développement, au lieu de la réécriture globale d'une matrice d'adjacence, chaque noeud d'un graphe de cellules est traité séparément, l'un après l'autre. Le résultat final de la réécriture est exactement le réseau de neurones codé. Il n'y a aucun besoin d'un processus supplémentaire. Au contraire, le processus de développement de Kitano génére une matrice brute de caractères. Cette matrice nécessite d'autres traitements faisant intervenir des conventions artificielles avant de fournir un réseau de neurones. On voudrait dans ce paragraphe souligner les avantages apportés par le codage cellulaire sur la méthode proposée par Kitano.

- Le même morceau de code représentant un sous-ensemble de règles peut être utilisé de façon répétée pendant la phase de développement. On peut ainsi reproduire le même motif de connexions en différents endroits. Cependant, s'il peut en théorie y avoir des motifs répétés dans la matrice d'adjacence, les résultats expérimentaux de Kitano ne montrent pas de réseaux comportant des sous-réseaux qui se répètent. Les réseaux de neurones produits par le GA en utilisant le codage cellulaire, ont par contre une structure clairement modulaire. De plus, une propriété théorique de modularité sera prouvée pour le codage cellulaire.

- Kitano caractérise la propriété d'échelle comme la possibilité pour le GA de générer de gros réseaux de neurones pour des problèmes paramétrés de grandes tailles. Cependant cette possibilité est testée sur un exemple atypique: une version dégénérée du problème de l'encodeur/décodeur, dans laquelle le nombre de neurones sur la couche cachée est supérieur au nombre de neurones d'entrée ou de sortie. Dans le problème original, le nombre de neurones cachés est égal au logarithme du nombre de neurones d'entrée, et c'est cela qui rend le problème intéressant. Avec le codage cellulaire une propriété d'échelle théorique est prouvée. On montre que pour une classe de familles de fonctions très large, il existe un code cellulaire de taille constante, codant une famille de réseaux de neurones qui calculent la famille de fonctions. Les résultats de simulation montrent que l'on peut de fait traiter des problèmes de taille arbitrairement grande en un temps constant.

- Le codage de Kitano nécessite des conventions artificielles qui très probablement augmentent la longueur du code. Une matrice $m \times m$ doit être développée pour coder un réseau de neurones ayant n neurones, où m est la plus petite puissance de 2 dépassant n. Dans le pire des cas, $n = m/2 + 1$. Après avoir extrait une matrice $n \times n$, on utilise une fraction qui est $(0.5 - 1/m)^2$ des bits, ce qui est approximativement 25% des bits dans le cas ou m est grand. On n'affirme pas que le nombre de neurones du réseau codé doit être une puissance exacte de 2. Toutrefois, la taille de la matrice générée doit être une puissance de 2. Cependant les calculs ci-dessus montrent qu'une grande matrice est développée, alors que seulement une petite partie de cette matrice est utilisée. Si on veut obtenir un graphe sans cycles pour un réseau de neurones feed-forward, on doit de plus extraire le triangle en haut à droite. Cela diminue encore de moitié le nombre de bits effectivement utilisés.

- Avec la méthode de Kitano, la grammaire est codée comme un ensemble de règles. Les grammaires produites par le GA ne sont pas propres. Elles peuvent avoir beaucoup de règles redondantes, qui réécrivent le même symbole de façons différentes et en même temps, d'autres règles qui ne peuvent être utilisées. Seulement un petit pourcentage des règles codées sont utilisées, et l'efficacité n'est pas bonne. Dans le codage cellulaire, la grammaire est codée comme un arbre étiqueté dont les branches sont ordonnées. Cela assure que le GA produit toujours des grammaires déterministes, où toutes les règles de réécriture sont utilisées. De plus la figure C.2 montre que les arbres peuvent fournir des codes plus courts que les règles.

- Des attributs spéciaux permettent de coder des grammaires récursives, qui produisent des réseaux de neurones infinis. Au lieu de stopper le développement quand la taille de la matrice atteint une borne donnée, un mécanisme simple permet d'appliquer les règles de la grammaire un nombre de fois L donné. Ainsi une grammaire peut développer une famille de réseaux de neurones paramétrés par L

Chapitre 3

Propriétés du codage cellulaire

3.1 Introduction

Le point crucial de l'application des algorithmes génétiques (GA) à un problème d'optimisation est la méthode utilisée pour coder les solutions de ce problème avec des structures que le GA soit capable de manipuler. Malgré l'importance fondamentale du codage, il n'y a pas encore eu d'essais, à notre connaissance pour établir une liste de propriétés que ce codage devrait avoir dans le cas particulier de l'optimisation de réseaux de neurones. Nous allons montrer qu'il est possible de définir des propriétés formelles et vérifiables permettant de caractériser un bon codage de réseaux de neurones. Les approches précédentes ont insisté chacune sur un aspect particulier. Beaucoup de codages ont été proposés, et des simulations nombreuses ont été décrites dans la littérature scientifique, avec souvent une interprétation intuitive des résultats. Il reste cependant à trouver un ensemble exhaustif de propriétés théoriques. Dans ce chapitre, huit propriétés théoriques et vérifiables mesurant l'intérêt d'une méthode de codage, seront définies. Ces propriétés sont: Complétude, compacité, fermeture, modularité, propriété d'échelle, pouvoir d'expression, abstraction, grammaire. Le codage cellulaire (CE) vérifie toutes ces propriétés.

Il n'existe pas seulement un seul CE, mais une multitude. Deux choses sont nécessaires pour définir un CE: un graphe de cellules initial, et un ensemble d'opérateurs cellulaires. La figure 2.1 représente un exemple de graphe initial. Avant que le développement ne commence, ce graphe contient une seule cellule ancêtre et deux cellules pointeurs. Dans ce chapitre, on utilisera deux sortes de graphes initiaux. Le premier exemple de graphe initial est celui qui a été vu au chapitre précédent, à la figure 2.1. Il comporte une cellule ancêtre reliée à une cellule pointeur entrée et une cellule pointeur sortie. Ce graphe est noté *ACYC*. Dans le deuxième type de graphe initial, la cellule ancêtre possède en plus un lien récurrent qui part de la sortie et arrive sur l'entrée. Ce lien récurrent est utilisé pour développer des réseaux de neurones récurrents. Ce graphe est noté *CYC*.

Un CE particulier comprenant r opérateurs est déterminé par un $(r+1)$uplet de symboles. Le premier symbole indique le nom du graphe initial de cellules. Les autres r symboles indiquent les noms des opérateurs utilisés. Le premier symbole sera noté en italique pour le distinguer des autres symboles. Au cours de ce chapitre, on montrera qu'augmenter le nombre d'opérateurs permet de réaliser un nombre croissant de propriétés. Nous allons réintroduire en les précisant, certains opérateurs définis au chapitre précédent. L'appendice D.5 donne une liste de tous les opérateurs utilisés dans cette thèse.

3.2 Codage d'architectures

Dans ce paragraphe on étudie les opérateurs qui développent l'architecture. Une architecture est un graphe orienté. A chaque sommet se trouve un neurone. L'architecture inclut aussi deux ensembles ordonnés de neurones. Ces ensembles spécifient les unités d'entrée et de sortie du réseau. Tout d'abord un ensemble minimal de trois opérateurs cellulaires est décrit, à partir desquels n'importe quelle architecture peut être codée.

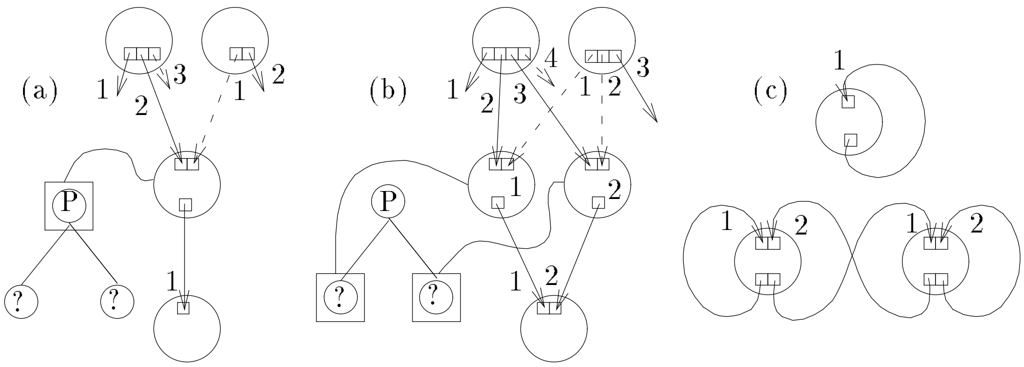

Figure 3.1: La division parallèle (P est mis pour PAR). (a) Avant l'exécution, (b) après l'exécution, (c) cas du lien récurrent. Le poid +1 est représenté par —- et le poid -1 par - - - .

Le premier opérateur est la division parallèle notée `PAR`. C'est un opérateur d'arité deux. Dans une division parallèle, les deux cellules filles héritent toutes les deux des liens d'entrée et de sortie de la cellule mère. Si de plus, la cellule mère a un lien récurrent, les deux cellules filles sont connectées entre elles dans les deux sens, et possèdent aussi un lien récurrent sur elles mêmes. La figure 3.1 (b) montre comment les liens doivent être réordonnés, une fois que la nouvelle cellule a été ajoutée. Le deuxième opérateur noté `CUT` coupe une connexion en entrée. Il a un argument entier qui indique le numéro de la connexion à couper. Le troisième opérateur noté `END` fait perdre à la cellule sa tête de lecture, la cellule devient donc un neurone fini.

La figure 3.2 montre comment coder un graphe donné. Comme toutes les feuilles et seulement les feuilles sont étiquetées avec l'opérateur `END`, l'opérateur `END` ne sera plus représenté dans les figures suivantes. Le développement d'un graphe ayant n noeuds est réalisé en deux étapes. Pendant la première étape, un graphe complétement connecté de m noeuds est développé, où $m = 2^{\lceil \log_2(n) \rceil}$ et $\lceil x \rceil$ est la fonction qui calcule l'entier plus grand que x, le plus proche de x. Cette étape est réalisée à l'aide d'un arbre binaire de profondeur $\lceil \log_2(n) \rceil$ étiqueté avec des `PAR`. Durant la deuxième étape, chaque cellule coupe les liens qui sont en excés. Tous les liens vers les $n - m$ cellules en excés sont coupés. Le graphe final posséde bien n noeuds. La propriété suivante permet de formaliser le résultat:

Définition de Propriété 1 (complétude) *Un codage est dit complet par rapport à un ensemble d'architectures donné, (resp. de réseaux de neurones), si n'importe quel élément de l'ensemble peut être codé.*

La construction précédente s'écrit:

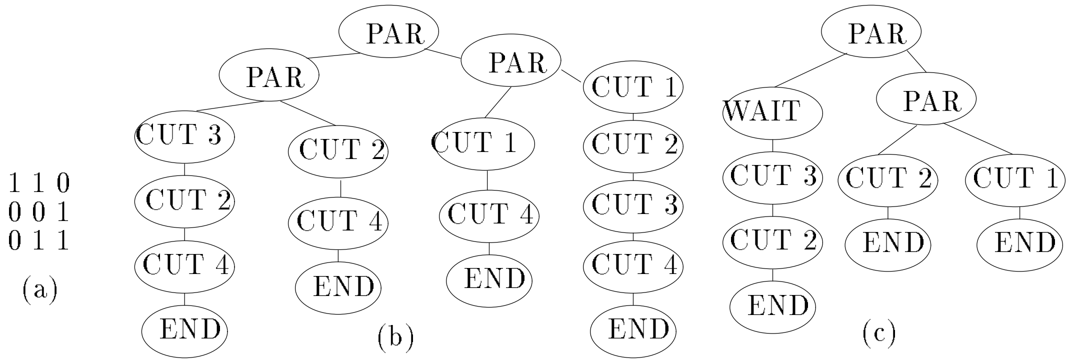

Figure 3.2: Codage d'un réseau de neurones récurrent arbitraire. (a) la matrice d'adjacence du réseau que l'on veur coder. (b) Code qui utilise deux opérateurs `PAR` `CUT`. (c) Code qui utilise trois opérateurs `PAR` `CUT` `WAIT`

Proposition 1 { *CYC,* `END`, `PAR`, `CUT` } *est complet par rapport à l'ensemble de toutes les architectures.*

A cause de l'opérateur `CUT`, la topologie du réseau final dépend de l'ordre de réécriture des cellules. Dans la construction précédente, les cellules commencent à couper leurs liens d'entrée seulement lorsque la croissance du réseau est terminée. C'est pour cette raison qu'un réseau dont la taille est une puissance de deux doit être développé. Ce développement peut être évité en utilisant un opérateur cellulaire unaire `WAIT`, qui fait attendre un pas à la cellule. L'opérateur `WAIT` est nécessaire pour pouvoir coder un ordre d'exécution particulier en générant des délais appropriés. La figure 3.2 montre comment `WAIT` peut être utilisé pour coder un réseau de neurones de façon plus compacte. Il n'y a plus besoin de créér plus de cellules que n'en contient le réseau final. Chaque cellule commence à couper ses liens en même temps que les autres. Les cellules qui ont fini de se diviser en avance, sont simplement retardées en utilisant l'opérateur `WAIT`. Il est clair que cette construction permet de diminuer la taille du code. Définissons la propriété de compacité comme suit:

Définition de Propriété 2 (compacité) *Une méthode de codage d'architectures $\mathcal{E}$ est dite topologiquement plus compacte qu'une autre méthode de codage $\mathcal{F}$ si toute architecture codée par $\mathcal{F}$ avec un code de taille s, peut être codée par $\mathcal{E}$ avec un code de taille $O(s)$. On utilise ici le mot compact avec un sens qui n'a aucun rapport avec le sens mathématique ordinaire. Compact signifie simplement de longueur courte. De plus, si la méthode permet aussi de coder les poids, on a une autre définition de la compacité. Une méthode de codage de réseaux de neurones $\mathcal{E}$ est dite fonctionnellement plus compacte qu'une autre méthode $\mathcal{F}$, si pour tout réseau N codé par $\mathcal{F}$ avec un code de taille s, il existe un réseau codé par $\mathcal{E}$, avec un code de taille $O(s)$ qui calcule la même fonction que N.*

La taille d'un code est le nombre de bits nécessaires pour le stocker. Comme il y a moins de 31 opérateurs utilisés dans ce chapitre, cinq bits suffisent pour coder chaque opérateur plus un caractère NULL, car $2^5 > 31$. Ainsi la taille du code d'un arbre contenant α opérateurs est $5\alpha + 1$. Nous allons en prouver la compacité par rapport aux représentations de graphes classiques. Tout d'abord, considérons le codage direct $\mathcal{C}_1$ de la matrice d'adjacence. Celui-ci produit un code de

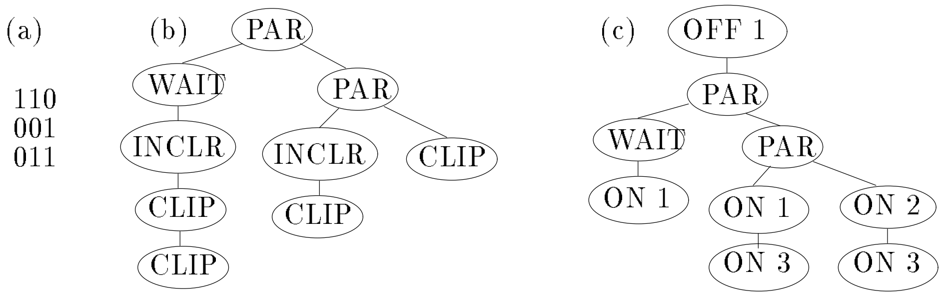

Figure 3.3: Codage compact d'un réseau de neurones récurrent arbitraire. (a) La matrice d'adjacence du réseau que l'on veut coder. (b) Compacité par rapport à $\mathcal{C}_1$. (c) Compacité par rapport à $\mathcal{C}_3$.

taille n^2 pour un réseau ayant n neurones. Munissons la cellule d'un registre lien qui contient le numéro d'un lien. Définissons l'opérateur cellulaire unaire `INCLR` qui incrémente le registre lien et l'opérateur `CLIP` qui coupe le lien pointé par le registre lien. Comme cela est montré à la figure 3.3, chaque cellule du graphe totalement connecté peut sélectionner son sous ensemble de connexions comportant moins de n `CLIP` alternés avec moins de n `INCLR`. Comme le graphe totalement connecté comporte n cellules, la taille du code cellulaire utilisant `INCLR` et `CLIP` est $O(n^2)$. D'autre part, la taille du code direct est n^2. On en déduit:

Proposition 2 { *CYC,* `END`, `CLIP`, `INCLR`, `PAR`, `WAIT` } *est topologiquement plus compact que* $\mathcal{C}$*1.*

Comme l'opérateur `CUT` peut être implanté en utilisant `CLIP`, `CLIP` ne sera plus utilisé par la suite. Dans les figure suivantes, `CLIP` sera noté comme un `CUT` sans argument. Maintenant considérons la représentation $\mathcal{C}_2$, où la liste des connexions est codée. Les numéros des neurones peuvent être codés en utilisant $\log(n)$ bits. Pour chaque connexion, les numéros des neurones connectés sont codés. Ainsi $\mathcal{C}_2$ produit un code dont la taille est $O(l\log(n))$ pour un réseau qui comporte n neurones et l connexions. Supposons qu'un lien ait deux états: `on/off`. Comme les cellules, les liens ont aussi des registres. Cet état est stocké dans un registre de lien appelé `state`. Quand un lien se duplique, les registres du lien sont aussi dupliqués. A la fin de la réécriture, les liens dont l'état est `off` sont enlevés. Définissons l'opérateur `ON` qui positionne le registre `state` du lien spécifié par un argument entier à `on`, et sa contrepartie `OFF`. La figure 3.3 (c) montre comment coder un réseau arbitraire en utilisant les opérateurs `ON` et `OFF`. Par convention la valeur initiale du registre `state` est `on`. L'opérateur `OFF 1` qui étiquette la racine de l'arbre grammaire positionne l'état du lien récurrent à `off`. Les opérateurs `PAR` développent un réseau complétement connecté où tous les liens sont à `off`. Enfin, chaque cellule c construit son ensemble de l_c connexions en appliquant l_c fois l'opérateur `ON` avec le numéro de ses voisins. Comme chacun de ces numéros sont codés avec $\log(n)$ bits, la taille totale du code est $O(l\log(n))$. La proposition suivante est correcte:

Proposition 3 { *CYC,* `END`, `ON`, `OFF`, `PAR`, `WAIT` } *est topologiquement plus compacte que* $\mathcal{C}_2$*.*

Enfin considérons la méthode de codage $\mathcal{C}_3$ qui code un réseau de neurones en couches avec une interconnexion compléte entre les neurones de deux couches consécutives. $\mathcal{C}_3$ spécifie le nombre de couches et le nombre de neurones dans chaque couche. Définissons deux autres opérateurs de

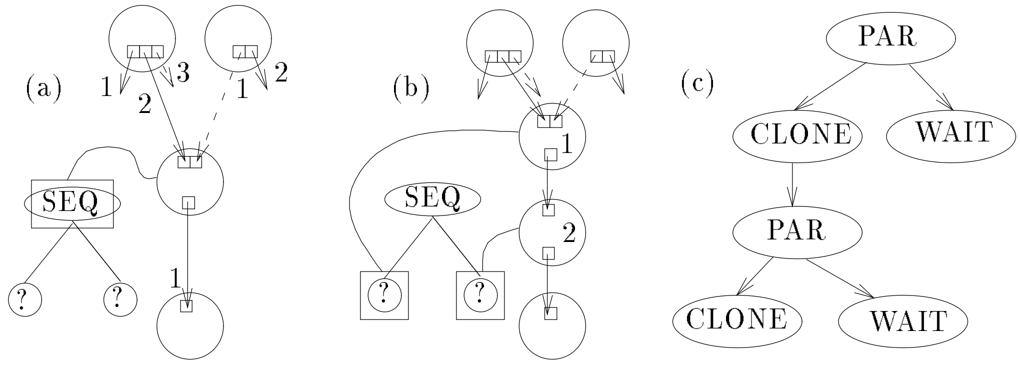

Figure 3.4: Comment coder efficacement un réseau de neurones en couches. (a) Avant la division séquentielle, (b) Après la division séquentielle, (c) Le code d'une couche de 7 cellules en utilisant l'opérateur `CLONE`. Le poid 1 est représenté par — et le poid -1 par - - - .

division. La division séquentielle notée `SEQ` fait hériter la première cellule fille de tous les liens d'entrée de la cellule mère. La deuxième cellule fille hérite de tous les liens en sortie de la cellule mére. De plus la première cellule fille est connectée vers la deuxième avec un poids 1 et un état `on`. Un exemple de division séquentielle est montré à la figure 3.4 (a) (b). La division `CLONE` est comme la division `PAR`, sauf que les deux cellule filles placent leur tête de lecture sur le même noeud de l'arbre grammaire. Comme il y a un seul sous-arbre au noeud étiqueté avec `CLONE`, cet opérateur est unaire. Son nom est justifié parce que les deux cellules filles vont exécuter exactement le même code, et vont donc donner naissance au même sous réseau de neurones. Un réseau de neurones comprenant c couches est développé comme suit: Appliquer $c-1$ fois l'opérateur `SEQ` pour produire une chaîne de c cellules. A l'issue de cette étape, chaque cellule représente potentiellement une couche. La place de la cellule dans la chaîne correspond au numéro de la couche. Maintenant, chaque cellule représentant la couche i va exécuter n_i fois l'opérateur `PAR`, où n_i est le nombre de neurones contenus dans la couche i. Il y a une façon plus astucieuse d'opérer. Comme cela est montré à la figure 3.4 (c), une couche de n_i neurones peut être développée avec une combinaison de moins de $3\log_2(n_i)$ opérateurs choisis parmi `PAR` et `CLONE`. Le nombre d'opérateurs cellulaires est: $c + 3\sum_{i=1}^{c}\log_2(n_i)$. Cela est inférieur à trois fois la taille du code produit par $\mathcal{C}_3$ Donc:

Proposition 4 { *CYC*, `END`, `PAR`, `SEQ`, `CLONE`, `WAIT` } *est topologiquement plus compact que* $\mathcal{C}3$.

Définissons maintenant une autre propriété:

Définition de Propriété 3 (fermeture) *Un codage d'architectures est fermé par rapport à un ensemble d'architectures, si tout code possible développe une architecture dans cet ensemble.*

Quels que soient les opérateurs utilisés, un Codage cellulaire donné est toujours fermé par rapport à l'ensemble de toutes les architectures. Cela signifie que pour chaque chromosome produit par le GA, le GA pourra développer et évaluer un réseau, et ainsi recevoir un feed-back de la fonction de fitness, et donc une information exploitable.

Supposons connu que l'architecture que l'on recherche appartienne à un ensemble particulier. Il est possible de faire en sorte que le GA utilise cette information a priori. On doit pour cela utiliser

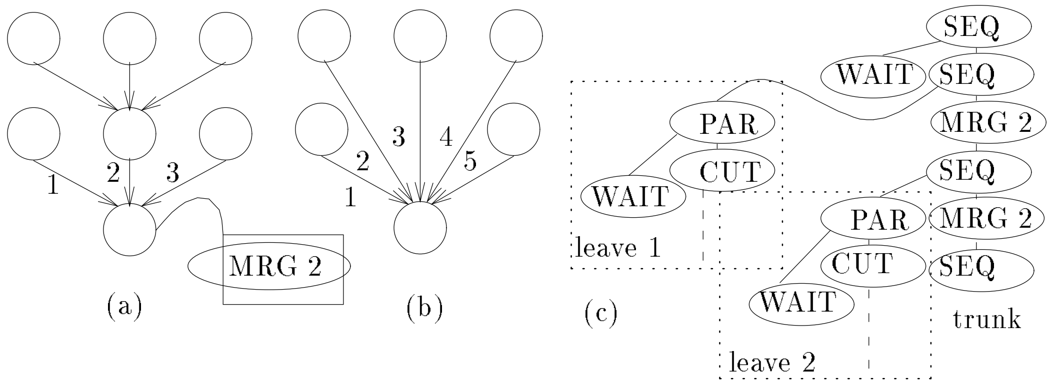

Figure 3.5: L'opérateur `MERGE`. (a) Avant l'application de l'opérateur. (b) Après l'application. (c) Le code d'un graphe acyclique arbitraire.

une méthode de codage fermée par rapport à cet ensemble, et le GA placera automatiquement toutes les architectures codées par les individus de sa population dans cet ensemble. Par exemple on peut rechercher une architecture acyclique pour faire un apprentissage du type "rétro propagation" du gradient. Si on commence à réécrire à partir d'un graphe de cellules acylique, le graphe de cellules restera acyclique, donc: { *ACYC*, `END`, `PAR`, `SEQ`, `CUT`, `INCLR`, `WAIT` } est fermé par rapport à l'ensemble des architectures sans cycle, mais il n'est pas complet. Pour obtenir la complétude, on introduit l'opérateur `MRG`. Cet opérateur a un argument entier i, et réécrit une cellule de la manière suivante: Soit l le lien d'entrée de numéro i et c la cellule voisine qui est connectée à travers l. L'opérateur `MRG` remplace l par la liste des liens d'entrée de c, comme cela est montré figure 3.5 (a) et (b). Si après cette opération c n'a plus de liens en sortie, c est supprimée. La figure 3.5 (c) montre comment construire le code d'un réseau acyclique quelconque. Dans un réseau acyclique, il est possible de numéroter les noeuds pour que chaque noeud ait des connexions orientées seulement vers d'autres noeuds ayant un numéro plus grand. Le code proposé comprend un tronc et n branches, où n est le nombre de noeuds. A une étape intermédiaire du développement du réseau de cellules, toutes les cellules sont connectées vers la cellule qui lit le tronc. Périodiquement, la cellule qui lit le tronc se divise, et crée une cellule c qui va lire la branche numéro i, et représente le noeud de numéro i. A ce moment précis, toutes les cellules qui représentent des noeuds avec des numéros plus petits que i sont connectées à c. La cellule c va lire sur la branche numéro i la liste des liens superflus à couper. Les opérateurs d'élagage correspondant sont situés à l'intérieur des lignes pointillées dans la figure 3.5 (c).

Ainsi:

Proposition 5 *Le codage cellulaire* { *ACYC*, `END`, `PAR`, `SEQ`, `CUT`, `INCLR`, `MRG`, `WAIT` } *est fermé et complet par rapport à l'ensemble des architectures acycliques. Si on rajoute* `ON`, `OFF` *et* `CLONE`, *on obtient une propriété de compacité par rapport à $\mathcal{C}1$, $\mathcal{C}2$ et $\mathcal{C}3$.*

Pour chaque neurone, un chemin le reliant à une unité d'entrée doit exister, sans quoi le neurone aurait une sortie constante et serait inutile. Il faut aussi un chemin vers une unité de sortie (sinon le neurone serait inutilisé) . Une architecture qui vérifie ces deux conditions sur chacun de ses neurones, est appelée une architecture valide. On peut caractériser la validité comme suit: une architecture est valide si, et seulement si, elle reste connexe quand on lui retire ou bien la cellule pointeur entrée, ou bien la cellule pointeur sortie.

Prouvons d'abord la propriété de fermeture, par rapport aux architectures valides acycliques. L'action des opérateurs cellulaires qui coupent des liens, doit être légèrement modifiée. Une cellule de lecture peut appliquer les opérateurs `CUT`, `OFF`, `MRG` sur un lien dont l'état est `on`, seulement si le nombre de liens d'entrée dont l'état est `on`, est strictement plus grand que 1. Prenons *ACYC* comme graphe de cellules initial. Essayons de supprimer ou bien la cellule pointeur entrée, ou bien la cellule pointeur sortie. Le graphe élagué est un graphe connexe ayant deux noeuds. Une étude de tous les cas montre que l'application de n'importe lequel des opérateurs définis, conserve la connexité. Ainsi, si on effectue le développement à partir du graphe élagué, le graphe final est toujours connexe, et c'est le graphe final que l'on obtiendrait à partir du graphe initial non élagué, en supprimant la cellule pointeur entrée ou bien la cellule pointeur sortie seulement à la fin. En utilisant la caractérisation de la validité que l'on a donnée, on peut conclure.

Proposition 6 { *ACYC*, `END`, `PAR`, `SEQ`, `CUT`, `INCLR`, `MRG`, `WAIT` } *est fermé par rapport à l'ensemble des architectures acycliques valides.*

Maintenant considérons le cas des architectures valides récurrentes. La construction d'une architecture récurrente donnée, avec les opérateurs d'elagage modifiés, utilise le fait qu'une architecture récurrente valide contient une architecture acyclique valide. Prenons le graphe initial *CYC*. Définissons un ordre (i.e une relation "inférieur") sur les cellules en utilisant les deux règles suivantes:

- Pendant une division de cellules, la cellule qui lit le sous-arbre gauche est inférieure à celle qui lit le sous-arbre droit.
- Si une cellule c_1 est inférieure à une cellule c_2, toutes les cellules descendantes de c_1 seront inférieures à toutes les descendantes de c_2.

On munit chaque cellule de deux sites d'entrée. Le premier contient tous les liens connectés à des cellules inférieures. Le deuxième contient tous les liens connectés à des cellules supérieures où égales. Quand une cellule se divise, ces deux sites d'entrée doivent être gérés de façon à ce que le précédent cahier des charges soit respecté. Pour être sûr de développer une architecture valide il faut appliquer les restrictions concernant les opérateurs `CUT`, `OFF`, `MRG` relativement au site d'entrée qui est connecté aux cellules inférieures. On peut appliquer `CUT`, `OFF`, `MRG` sur un lien de ce site dont l'état est `on`, seulement si le nombre de liens de ce site dont l'état est `on`, est strictement plus grand que 1.

Avec `ON`, `OFF` et `CLONE`, on peut encore déduire une propriété de compacité relativement à $\mathcal{C}_1$, $\mathcal{C}_2$ et $\mathcal{C}_3$. La construction avec $\mathcal{C}_2$ utilisant `ON` et `OFF` est plus difficile parce que la nouvelle sémantique ne permet pas de positionner tous les liens à `off`. A chaque étape, pour chaque cellule, au moins un lien d'entrée reliant une cellule inférieure doit être `on`. Heureusement, il est possible d'assurer cela sans augmenter l'ordre de complexité de la taille du code.

Enfin, on peut vouloir imposer le nombre d'unités d'entrée ou de sortie. Dans ce cas il suffit de commencer le développement à partir d'un graphe dans le genre de celui qui est montré à la figure 3.6.

3.3 Codage de réseaux de neurones

Dans ce paragraphe nous allons décrire comment coder un réseau de neurones comprenant une architecture mais aussi des poids et des sigmoïdes différentes. Une façon simple de coder les poids

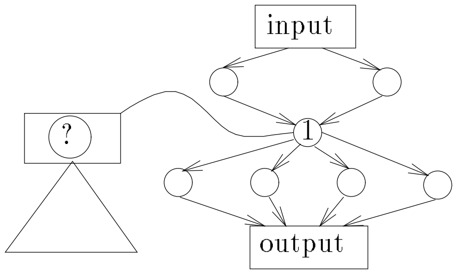

Figure 3.6: Graphe initial utilisé pour imposer deux unités d'entrée et 4 unités de sortie. La cellule étiquetée 1 est la cellule ancêtre.

est d'utiliser un opérateur unaire `VAL` qui positionne le poids du lien, pointé par le registre de lien, à une valeur spécifiée par un argument. Cet argument peut être ± 1, un entier ou un réel, suivant la précision désirée pour les poids. Considérons les codage $\mathcal{C}_1$ et $\mathcal{C}_2$ définis au paragraphe précédent. Le codage $\mathcal{D}_1$ est le codage $\mathcal{C}_1$ complété par la liste de valeurs de poids. De même, le codage $\mathcal{D}_2$ est le codage $\mathcal{C}_2$ complété par la liste des valeurs de poids. Nous avons le résultat suivant:

Proposition 7 { *CYC,* `END`, `PAR`, `SEQ`, `CUT`, `INCLR`, `ON`, `OFF`, `MRG`, `WAIT`, `VAL`, } *est complet par rapport à l'ensemble de tous les réseaux de neurones où les unités ont toutes la même sigmoïde. Il est fonctionnellement plus compact que $\mathcal{D}1$ et $\mathcal{D}2$.*

Pour montrer ces propriétés, on raisonne en deux étapes. D'abord l'architecture est developpée, avec tous les poids égaux à 1, et tous les seuils égaux à 0. Puis les poids différents de 1 et les seuils différents de 0 sont modifiés. Du point de vue du GA, un codage utilisant l'opérateur `VAL` présente des désavantages. Le GA forme sa population initiale avec des codes aléatoires. Comme Koza le décrit dans [17], lorsqu'un opérateur du genre de `VAL` a besoin d'un argument, une valeur aléatoire est générée dans un intervalle spécifié. S'il n'y a pas de mutation, ce qui est le cas dans la méthode proposée par Koza, la valeur possible de l'argument, dans les générations suivantes, est limitée à l'ensemble des valeurs existantes dans la population initiale. Dans le cas de l'opérateur `VAL`, cela signifie que les valeurs possibles des poids sont restreintes à cet ensemble fini. Si un poids particulier est nécessaire, et que ce poids n'est pas échantilloné dans la population initiale, le GA ne pourra pas le générer. De plus l'opérateur `VAL` ne donne pas la possibilité de trouver des valeurs en utilisant un affinage progressif, lié à la formation de blocs de bases (building blocks dans le jargon des GA) de taille croissante. Une solution au problème consiste à utiliser deux autres opérateurs: `MULT` qui multiplie le poids par son argument et `ADD` qui additionne au poids son argument. Il est possible d'aller plus loin dans cette direction en utilisant un ensemble de quatre opérateurs: `INC`, `DEC`, `MULT2` et `DIV2`, qui respectivement incrémentent, décrémentent, multiplient par deux et divisent par deux le poids du lien pointé par le registre lien. Ces opérateurs sont intéressants parce qu'il n'ont pas d'argument. En effet, si on utilise seulement cet ensemble sans `VAL`, alors la détermination des poids sera faite par une combinaison de ces quatre opérateurs. La précision avec laquelle les poids sont codés n'est pas fixée à priori. Si le codage d'un nombre prend b bits avec une notation en virgule fixe, un codage de ce nombre peut être construit avec $O(b)$ opérateurs choisis parmi { `INC`, `DEC`, `MULT2`, `DIV2` }. On conserve donc toujours la même propriété de compacité par rapport à $\mathcal{D}_1$ et $\mathcal{D}_2$.

Une technique similaire peut être utilisée pour supprimer l'usage d'arguments concernant d'autres opérateurs. Par exemple, si l'argument est un numéro de lien, l'opérateur peut utiliser la valeur du registre de liens. Pour conserver la propriété de compacité, il faut un opérateur `MULT2LR` qui multiplie par 2 la valeur du registre de lien, et positionne sa valeur à 0 si elle dépasse le nombre de liens entrant. Pour être plus clair, dans les prochaines figures, nous continuerons cependant à utiliser des arguments entiers.

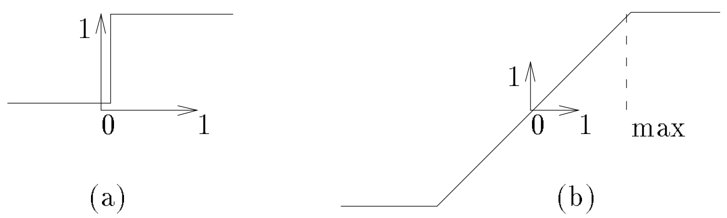

Figure 3.7: Deux sigmoïdes. (a) La fonction signe, (b) La fonction linéaire bornée. La constante *max* est un paramètre du modèle.

Un réseau de neurones n'est pas entièrement déterminé par une architecture et des poids. Chaque neurone peut avoir des caractéristiques particulières. Par exemple un seuil différent de 0, une sigmoïde spéciale, ou bien une procédure d'apprentissage, ou bien une dynamique spéciale (on verra au chapitre suivant que chaque neurone peut suivre une dynamique différente). Pour coder ces caractéristiques, on ajoute à la cellule autant de registres internes qu'il y a de caractéristiques. Par exemple, supposons qu'il y ait le choix entre les deux sigmoïdes indiquées sur la figure 3.7. On utilise un registre interne `sigmo` qui stocke le type de la sigmoïde. Un opérateur `SIGMO` positionne la valeur de ce registre avec la valeur de son argument. Quand la cellule devient un neurone fini, son registre `sigmo` détermine la sigmoïde.

3.4 Simulation d'une machine de Turing

Dans ce paragraphe on construit le code d'un réseau de neurones qui simule une machine de Turing. La sigmoïde utilisée sera la fonction signe; Cela nous permettra ensuite d'obtenir d'autres propriétés intéressantes. Comme première étape nous voulons montrer la propriété suivante:

Définition de Propriété 4 (Modularité) *Soit un réseau de neurones N_1 qui inclut à différents endroits plusieurs copies d'un même sous réseau de neurones N_2. Un code de N_1 est dit modulaire s'il inclut le code de N_2 une seule fois.*

Cette définition de propriété essaye de formaliser l'idée intuitive de la modularité. La modularité doit aider à réduire la complexité apparente d'un réseau de neurones initial, en le décomposant en un ensemble de sous réseaux de complexité moindre. Ce processus est répétitif, les sous réseaux peuvent eux mêmes être redécomposés. La modularité est liée au problème de la réutilisabilité: Le but est de transformer la conception de réseaux de neurones en une activité de jeu de construction où chaque réseau serait construit en combinant des réseaux déjà existants. Comme chacun des réseaux peut être compris séparément par un lecteur humain, les architectures produites seraient aisément compréhensibles.

Pour pouvoir inclure un sous réseau N_2, il nous faut définir un opérateur qui puisse déplacer la tête de lecture d'une cellule sur la racine de l'arbre grammaire (code cellulaire) définissant N_2. On peut utiliser une méthode conseillée par Ray dans [28] appelée "addressage par motifs complémentaires", et inspirée de la biologie moléculaire. Ray écrit:

> Dans la plupart des langages machines, quand le pointeur d'instructions saute sur un autre bout de code, l'adresse numérique exacte du code est spécifiée dans le programme. Remarquons qu'au contraire, dans les systèmes biologiques, lorsqu'une molécule A intéragit avec une molécule B, elle ne spécifie pas exactement les coordonnées où B se trouve. Au lieu de cela, la molécule A présente un motif sur sa surface, qui est complémentaire avec un motif se trouvant sur la cellule B. La diffusion améne les deux molécules à se rencontrer, et leur conformation complémentaire leur permet d'interagir.

L'adressage par motifs complémentaires est accompli grace à l'opérateur unaire `JMP`. Chaque `JMP` est suivi par une séquence d'opérateurs choisis parmi `NOP_0` et `NOP_1`. Considérons le bout de code suivant: `JMP(NOP_0(NOP_0(NOP_1(..))))..` La cellule qui va exécuter l'opérateur `JMP` cherchera dans l'arbre l'occurence la plus proche du motif complémentaire: `NOP_1(NOP_1(NOP_0(..)`. Ici, "plus proche" est défini relativement à la topologie naturelle sur un graphe où la distance entre deux noeuds est égale à la longueur du plus court chemin qui les joint. Si le motif complémentaire est trouvé, la cellule positionne sa tête de lecture à la fin du motif complémentaire, sinon le `JMP` est ignoré. Ce mécanisme de branchement peut boucler indéfiniment. Pour éviter une croissance infinie du réseau, la cellule est munie d'un registre `vie`. La `vie` de la cellule ancêtre est initialisée avec L. A chaque saut récurrent de la tête de lecture, la vie est décrémentée. Lorsque la vie vaut 0, les sauts de tête de lecture sont interdits. Avec cet algorithme un code développe toute une famille de réseaux de neurones paramétrés par L. La taille des réseaux augmente avec L.

Pour achever l'édification de la propriété de modularité, il faut montrer que l'on peut coder séparément un sous réseau N_2 du réseau principal N_1. Dans ce but on utilise un opérateur binaire: `DEF`. Quand une cellule exécute l'opérateur `DEF`, elle place sa tête de lecture sur le sous-arbre de droite. Pour pouvoir être utilisé, le sous-arbre de gauche doit contenir une étiquette faite avec des `NOP_0`s et des `NOP_1`s. Le sous-arbre de droite correspond au réseau principal et le sous-arbre de gauche correspond au sous réseau inclus en plusieurs endroits.

Proposition 8 *Tout codage cellulaire utilisant les opérateurs* `JMP`, `NOP_0`, `NOP_1` *et* `DEF` *est modulaire.*

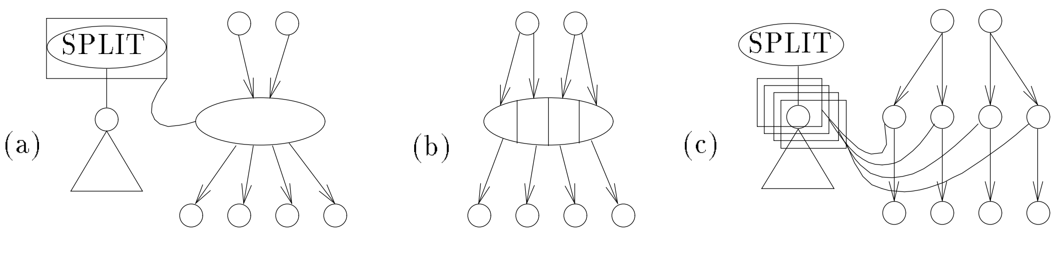

Figure 3.8: La division `SPLIT` (a) Avant la division. (b) Duplication des liens sur le site d'entrée pour que le nombre total de liens soit égal à quatre. (c) Création de quatres neurones.

Pour pouvoir connecter un sous-réseau au reste du réseau de façon fonctionnelle, on a défini un opérateur de division `SPLIT`. Une cellule qui exécute `SPLIT`, se divise comme suit:

Tout d'abord, supposons que le nombre de liens à l'entrée soit égal au nombre de liens à la sortie, comme sur la figure 3.8 (b). La cellule se divise en n cellules filles, où n est le nombre de liens à l'entrée ou à la sortie. L'héritage des liens est montré figure 3.8 (c). La cellule fille numéro i hérite du lien d'entrée numéro i et du lien de sortie numéro i. L'opérateur `SPLIT` est unaire, il étiquette un noeud qui a un seul sous-arbre. Chaque cellule fille place sa tête de lecture sur ce même sous-arbre.

A présent considérons le cas où il y a moins de liens en entrée qu'en sortie. Certains des liens en entrée sont dupliqués d'une façon homogéne, jusqu'à ce que le nombre de liens en entrée soit égal au nombre de liens en sortie. Dupliqué de façon homogène, signifie que chaque lien l est reproduit un nombre n_l de fois, et on a toujours $|n_{l_1} - n_{l_2}| \leq 1$ pour deux liens l_1 et l_2 différents. On est ainsi ramené au cas précédent. Si la cellule a plus de liens en entrée que de liens en sortie, ce sont les liens en sortie qui sont dupliqués.

Le codage cellulaire peut être utilisé pour concevoir un réseau de neurones destiné à calculer une fonction connue. Par exemple la figure 3.9 montre un réseau de neurones qui simule une machine de Turing; la méthode de conception consiste à construire des réseaux de complexité croissante en spécifiant comment connecter des réseaux de complexité moindre. Le mode utilisé pour représenter le codage est basé sur cette décomposition. Chacun des sous réseaux est codé, et nommé séparément. Pour représenter un sous réseau h, plutôt que d'utiliser le sous-arbre grammaire qui code h, on représente un graphe de cellules obtenu de la façon suivante: prenons comme graphe initial de cellules, une cellule de lecture ancêtre connectée à un ensemble de cellules étiquetées. Ces cellules étiquetées représentent les voisines des cellules qui incluront le sous réseau h, durant le développement complet du réseau. On développe ce graphe initial en utilisant deux conventions:

- Une cellule qui lit un opérateur `JMP` se comporte différemment suivant sa vie. Si sa vie est strictement positive, elle inclut le sous réseau indiqué par `JMP`. Sinon elle continue à lire le même code cellulaire, et inclut le sous réseau dont le code est le sous arbre en dessous de `JMP`. Pour rendre compte des deux cas possibles, à chaque fois qu'une cellule lit un opérateur `JMP`, on gèle le développement de la cellule, et on introduit un sous réseau g dont le code est le sous-arbre qui commence en dessous de `JMP`. Dans le cercle représentant la cellule gelée, on écrit le nom du sous réseau qui est inclus par `JMP` ainsi que le nom de g, à moins que le sous réseau g ne se réduise à un neurone.

- Lorsqu'une cellule c exécute un opérateur `SPLIT`, le nombre de cellules créées est égal au nombre de liens de c. Comme ce nombre de liens dépends de l'état d'avancement du développement, on ne peut savoir avant la compilation, en combien de cellules filles, la cellules c va se diviser. Pour cette raison, à chaque fois qu'une cellule exécute un opérateur `SPLIT`, on gèle le développement, et on introduit un sous réseau g, dont le code est le sous-arbre qui commence en dessous de `SPLIT`. Dans le cercle représentant la cellule gelée, on écrit `SPLIT` suivi du nom de g, à moins que g ne se réduise à un neurone.

Si le sous-arbre commençant à `JMP` ou à `SPLIT` est le noeud `END`, il n'est pas nécessaire d'introduire de nouveaux sous réseaux virtuels, car ceux ci se réduiraient à un seul neurone. Dans le réseau final de cellules, les cellules étiquetées indiquent comment le sous réseau h doit être connecté au reste du réseau. Il ne faut pas oublier que le réseau de neurones final peut dépendre de l'ordre dans lequel les cellules exécutent leurs instructions. De façon à compléter notre mode de représentation, l'ordre d'exécution des cellules gelées doit être indiqué si besoin est. Dans le cas de la machine de Turing, on doit écrire que les cellules gelées sur les opérateurs `SPLIT` doivent exécuter cet opérateur

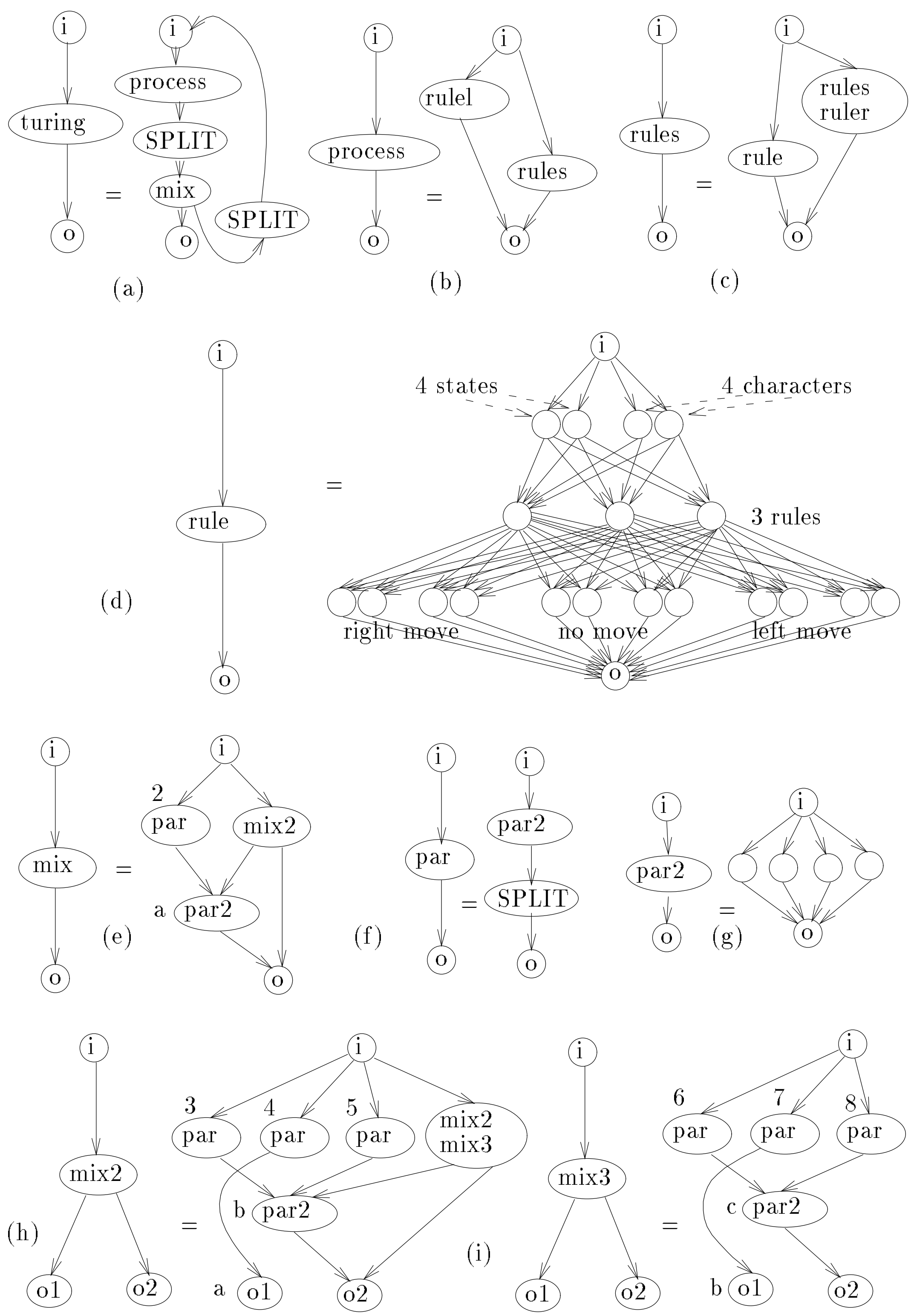

Figure 3.9: Codage d'une machine de Turing.

seulement lorsque tous leurs voisins sont des neurones finis. Donc, au niveau du code cellulaire, il doit se trouver un certain nombre d'opérateurs `WAIT` pour réaliser ces délais. Le nombre de ces opérateurs est une fonction linéaire de la valeur initiale de la vie que l'on appelle L. Ces opérateurs `WAIT` peuvent donc être implantés à l'aide d'un sous réseau qui ne fait rien, à part s'inclure lui même, et qui est utilisé seulement pour retarder le développement.

La figure 3.9 (a) montre un réseau qui est récurrent, et qui a deux couches: une couche de traitement et une couche pour recombiner les informations. Les deux couches sont connectées entre elles de façon bijective, grace à l'utilisation d'un opérateur `SPLIT` inséré entre les deux couches. A chaque fois que les activités font un tour complet à travers le réseau, une transition de la machine de Turing est simulée. Les figures 3.9 (b) et (c) montrent que la couche de traitement applique en parallèle L sous réseaux de neurones appelés `rule` à un ruban de L registres. A cause d'une légère différence, le sous réseau `rule` le plus à gauche est appelé `rulel`, et le plus à droite est appelé `ruler`. Le sous réseau `rule` code les règles de transition de la machine de Turing. Chaque sous réseau `rule` correspond à un registre de la machine de Turing. Pour chaque registre, un caractère et un état sont codés. Si la tête de lecture est positionnée sur ce registre, l'état codé est l'état de la machine de Turing, sinon l'état est 0. Sur les entrées du i^{eme} sous réseau `rule`, sont appliqués le code binaire de l'état et le caractère du registre correspondant. Sur l'exemple, deux bits sont utilisés pour coder l'état, et aussi deux bits pour coder le caractère. Ainsi, s'il y a q états et un alphabet de c caractères, le nombre d'entrées du sous réseau `rule` est $i = \lceil \log_2(q) \rceil + \lceil \log_2(c) \rceil$. Le nombre de sorties vaut $3 * i$, trois rangées de i cellules. Une rangée représente chacun des trois mouvements possibles de la tête de lecture de la machine de Turing. Les sous réseaux `ruler` et `rulel` ont seulement $2 * i$ sorties. Comme ils traitent un registre en bout de ruban, seulement deux mouvements sont permis. Si la tête de lecture de la machine de Turing n'est pas sur le registre, l'état est 0. Son code est la concaténation de $\lceil \log_2(q) \rceil$ valeurs 0, et la sortie de `rule` est 0 partout, excepté pour les $\lceil \log_2(c) \rceil$ sorties de la rangée du milieu dont les activités sont les mêmes que les $\lceil \log_2(c) \rceil$ entrées. Cela rend compte du fait que le caractère sur le ruban ne change pas. Si la tête de lecture est sur le registre, elle bouge dans une direction, à gauche ou à droite, ou bien elle peut choisir de rester. La rangée d'unités de sortie correspondante indiquera l'état suivant, ainsi que le caractère à écrire. Les deux autres rangées vont sortir des zéros. Chaque unité cachée du sous-réseau `rule` implèmente une règle de transition de la machine de Turing.

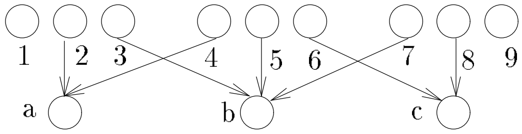

Figure 3.10: L'action de la couche qui recombine les informations.

La couche de recombinaison doit calculer le prochain état et le caractère à écrire sur le ruban, à partir de toutes les sorties de la couche de traitement. Ainsi $3i * L$ activités doivent être recombinées en $i * L$ activités. La figure 3.10 (a) montre comment cela doit être fait. La figure 3.9 (e,f,g,h,i) montre comment cela est fait. Chacune des $i * L$ activités peut être calculée en appliquant un OU logique à trois des $3i * L$ activités.

Maintenant définissons précisément ce qu'on entend par simuler un programme sur une machine qui peut avoir une mémoire arbitrairement grande, avec une famille de réseaux de neurones paramétrés, chacun étant de taille finie. Tout d'abord définissons les fonctions que peut calculer un réseau de neurones comme suit:

Définition 1 *Un réseau de neurones N calcule une fonction $f : \{0,1\}^n \to \{0,1\}^p$ s'il a plus de n unités d'entrée et plus de p unités de sortie, et si le processus suivant peut avoir lieu avec succés. Un vecteur v de n bits est appliqué sur les premières unités d'entrée de N, toutes les autres unités d'entrée étant initialisées à 0. Le réseau de neurone N effectue ses calculs avec une dynamique parallèle synchrone. Après stabilisation, les activités des premières p unités de sorties de N sont $f(v)$.*

Maintenant, considérons une machine $M(L)$ ayant une mémoire de L bits, et un programme P qui fait des calculs, sur des chaînes de bits. Si on lance le programme sur certains vecteurs de bits particuliers, le programme P aura assez de mémoire pour faire ses calculs, et fournira un vecteur binaire résultat. L'ensemble de ces vecteurs de bits forme le domaine de définition de la fonction calculée par P sur $M(L)$; et la valeur de la fonction est donnée par P.

Définition 2 *Un réseau de neurones $N(L)$ paramétré par un entier L simule un programme P sur une machine $M(L)$ ayant une mémoire de L bits, si $N(L)$ peut calculer la fonction définie par P sur $M(L)$.*

Définition de Propriété 5 (Pouvoir d'expression) *Une méthode de codage de famille de réseaux de neurones paramétrés a un pouvoir d'expression supérieur à un langage de programmation, si pour chaque programme P écrit dans ce langage, il existe le code d'un réseau paramétré qui simule le programme P. De plus, la taille du code est $O(s(P))$ où $s(p)$ est le nombre de bits nécessaire pour stocker le code source de P.*

Dans le cas d'une machine de Turing, un programme est une liste de règles de transition. La construction précédente fournit le code d'un réseau de neurones paramétré qui simule n'importe quel programme de machine de Turing, moyennant l'ajustement suivant: Les unités d'entrée de la couche de traitement ne doivent pas être connectées à la cellule pointeur entrée. Leur activité initiale doit être zéro, excepté pour le registre du ruban sur lequel pointe la tête de lecture. Initialement, l'état codé dans ce registre doit être l'état initial de la machine de Turing q_0. Maintenant il nous faut comparer la taille du code cellulaire avec la taille mémoire nécessaire pour coder une machine de Turing particulière. Le nombre de bits nécessaire pour coder r règles de transition est $s = (\log_2(q) + \log_2(c))r$. La taille du code de `par2` est $O(\log_2(q) + \log_2(c))$.. La taille du code de respectivement `rule`, `ruler`, `rulel` est $O(s)$.

Proposition 9 { *CYC,* `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *a un pouvoir d'expression plus grand que le langage machine de Turing.*

Le fait que le codage cellulaire ait un pouvoir d'expression plus grand que le langage des machine de Turing, permet de prouver une sorte de propriété de compacité maximale.

Proposition 10 { *CYC,* `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *est fonctionnellement plus compact que n'importe quel autre codage de réseaux de neurones.*

Pour prouver cela, considérons une méthode de codage de réseaux de neurones arbitraire $\mathcal{E}$. Soit e le code d'un réseau de neurones particulier codé avec $\mathcal{E}$, et $s(e)$ sa taille. Il existe un programme pour machine de Turing, qui sur les entrées e et i construit le réseau de neurones codé par e, effectue

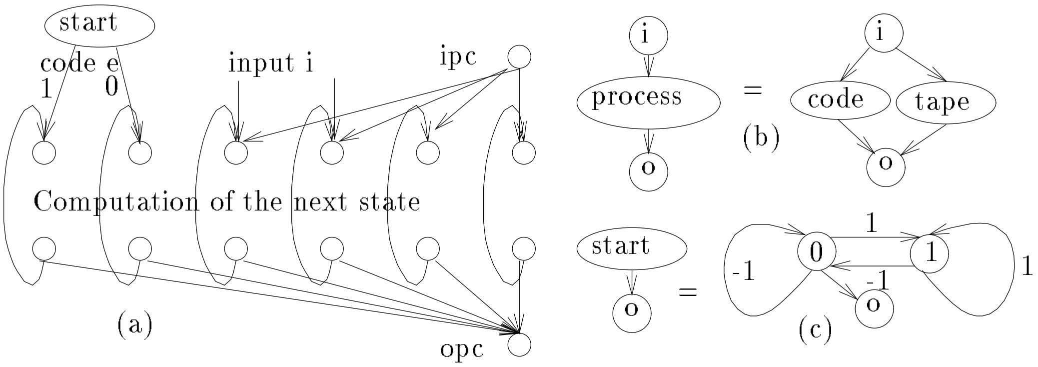

Figure 3.11: Un réseau de neurones qui simule une machine de Turing qui elle même simule un réseau de neurones. (a) Le réseau de neurones qui simule. (b) Modification de la couche de traitement (c) Le sous réseau `start`.

les calculs du réseau à partir des entrées i, et renvoie la sortie du réseau. Ce programme peut être exécuté sur une machine T. Si le ruban initial de T contient la concaténation de e et i, le résultat calculé par T sera constitué par les sorties du réseau de neurones. Soit N le réseau de neurones qui simule T. Une chaîne de $s(e)$ neurones doit être ajoutée à N de telle sorte que les $s(e)$ registres de la machine de Turing soient automatiquement initialisés avec e. La figure 3.11 (a) montre à quoi ressemble le réseau de neurones final. Il représente un réseau de neurones qui simule une machine de Turing développant et simulant le réseau de neurones codé par $e = \overline{10}$ (notation d'un code binaire à deux bits, le premier bit vaut 0, le deuxième bit vaut 1). Le réseau de neurones simulé a deux unités d'entrée. Dans le but de simplifier la figure, on a representé seulement les neurones qui stockent les caractères du ruban. Le codage du réseau de neurones final doit spécifier la chaîne des $s(e)$ neurones, donc il est un peu plus complexe que le codage de la figure 3.9. La figure 3.11 (b) montre la première partie du code de la couche de traitement. Le sous-réseau `code` va développer $s(e)$ sous réseaux du type de `rule`. Le i^{eme} sous réseau n'est pas connecté à la cellule pointeur entrée. Il est connecté au sous réseau `start` par l'intermédiaire d'un neurone, dont le poids est 0 ou 1 suivant la valeur du i^{eme} bit de e. Le rôle du sous réseau `start` est d'envoyer un 1, puis ensuite des valeurs 0. Le sous-réseau `tape` développe un ruban de L registres. La couche de recombinaison doit être modifiée de façon à recombiner $s(e) + L$ registres au lieu de seulement L. La taille du code total est de façon évidente une fonction linéaire de $s(e)$, elle est donc de complexité $O(s(e))$. Une analyse technique (non indiquée ici pour résumer) montre que la pente de la fonction linéaire ne dépend pas de T, et donc ne dépend pas non plus de $\mathcal{E}$.

Une autre propriété intéressante est la propriété d'échelle. Considérons un problème de taille variable, et une méthode de codage de famille de réseaux de neurones paramétrés. Supposons que le même code puisse être utilisé pour résoudre le problème de taille 1 et les problèmes de taille plus grande, en utilisant un paramètre plus grand. Si le GA peut trouver un code pour un réseau de neurones qui résoud le problème de taille 1, il ne sera pas beaucoup plus difficile pour le GA de trouver un réseau de neurones pour le problème de taille 1.000.000 parce que le code correspondant est le même. La propriété permettant de résoudre des problèmes de grande taille s'appelle la propriété d'échelle. La remarque précédente nous permet de formaliser la propriété d'échelle d'une méthode de codage.

Définition de Propriété 6 (Échelle) *Soit $f(L)$ une famille de fonctions définies chacune sur un domaine fini, l'index L est un entier. Une méthode de codage a la propriété d'échelle relativenent à $f(L)$ s'il existe un code tel que pour tout L, le réseau de neurones codé avec un paramètre L, calcule la fonction $f(L)$.*

Considérons les définitions suivantes:

Définition 3 *Une famille de fonctions f_n définies chacune sur un domaine fini est dite récursive s'il exite une machine de Turing T qui calcule $f_n(x)$ lorsque n et x sont initialement écrits sur son ruban. Si f_n est récursive, $TM_{space}(f_n)$ est le nombre minimal de registres nécessaire à T pour calculer toutes les valeurs de f_n.*

Nous allons montrer la proposition suivante:

Proposition 11 { *CYC,* `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *a la propriété d'échelle relativement à l'ensemble des familles de fonctions récursives telles que $TM_{space}(f_n)$ est plus petit que k^n pour un certain k.*

Pour prouver cette proposition, nous devons simplement fournir un codage d'une machine de Turing arbitraire, telle que le réseau développé avec un paramètre L simule la machine de Turing avec un ruban de k^L registres, au lieu de seulement L comme cela a déjà été fait. Le codage est similaire à celui qui a déjà été proposé. Comme le codage est technique, on ne le présente pas ici. La famille de langages que peut reconnaître une machine de Turing en utilisant moins de k^L registres pour un certain k s'appelle PSPACE. Cet espace PSPACE est tellement grand qu'il inclut tous les langages usuels [12], en d'autres termes, les langages qui ne sont pas PSPACE sont des exceptions. On en déduit que le codage cellulaire a la propriété d'échelle relativement à toutes les familles de fonctions usuelles.

Un automate cellulaire peut aussi être simulé avec une construction similaire aux machines de Turing. Cette fois-ci, la couche de recombinaison précéde la couche de traitement, et son but est de reproduire à l'emplacement i la concaténation des états des cellules voisines de la cellule i.

Proposition 12 { *CYC,* `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *a un pouvoir d'expression plus grand que les automates cellulaires.*

Les machines de Turing et les automates cellulaires sont des dispositifs de calculs universels. Les automates cellulaires sont particulièrements puissants. En général, un algorithme donné peut être efficacement programmé sur un automate cellulaire. Il est possible de représenter l'ensemble des règles de transitions d'un automate cellulaire, par le code cellulaire d'un réseau de neurones. On peut cependant considérer un dispositif de calculs encore plus puissant que les automates cellulaires. La définition de propriété suivante décrit ce pas supplémentaire:

Définition de Propriété 7 (abstraction) *Un codage est dit abstrait, s'il a un pouvoir d'expression plus grand qu'un langage de programmation évolué. Un tel langage utilise des variables, des instructions comportant diverses structures de contrôle pour modifier les valeurs de ces variables, et des fonctions et procédures pour programmer de façon modulaire. Des exemples de tels langages sont Algol, Pascal, C, Fortran.*

L'utilisation d'un codage abstrait permet de remplacer la recherche d'un réseau de neurones, par celle d'un programme calculant la fonction cible. S'il existe un petit programme qui calcule la fonction cible, l'ensemble des programmes à rechercher est plus petit que l'ensemble des réseaux de neurones de taille bornée contenant un réseau de neurones solution. Ainsi un codage abstrait est succeptible d'être plus efficace, relativement à l'utilisation de l'algorithme génétique, si la fonction à apprendre présente un certain degré de régularité qui la rend calculable par un programme de petite taille. Comme la propriété d'abstraction est difficile à démontrer, nous le ferons au cours du prochain chapitre. En fait, la preuve est un compilateur qui transforme le programme Pascal en un code cellulaire dont la longueur est proportionnelle à la longueur du programme Pascal.

La dernière définition de propriété est la suivante:

Définition de Propriété 8 (Grammaire) *Une méthode de codage de réseaux est grammaticale, si l'objet qui est codé par elle peut être associé à une grammaire (un système de réécriture).*

Proposition 13 *Le codage cellulaire est grammatical, il code une grammaire de graphe.*

La démonstration de cette proposition se trouve dans l'appendice C. Avec la propriété grammaticale, la recherche génétique de réseaux de neurones via le codage cellulaire apparaît comme un processus d'inférence grammaticale, où le langage décrit par la grammaire est implicitement défini au lieu d'explicitement, par une série d'échantillons appartenant ou non au langage (des échantillons positifs et négatifs. Cela ouvre trois nouvelles pespectives. La méthode (utilisée dans la seconde partie de cette thèse) qui consiste à coder une grammaire en un ensemble d'arbres et à faire évoluer des ensembles d'arbres avec l'algorithme génétique, peut être utilisée pour inférer d'autres types de grammaire. A l'inverse, on peut s'inspirer des méthodes existantes dans le domaine de l'inférence grammaticale pour améliorer le codage cellulaire. Enfin l'étude du codage cellulaire en tant que grammaire, permet de classifier le codage cellulaire parmi les classes de complexité existantes. Le codage cellulaire est une grammaire de graphe parallèle et contextuelle.

3.5 Conclusion

Lorsque l'on applique l'algorithme génétique à la synthèse de réseaux de neurones, il est maintenant reconnu que le point crucial concerne la manière dont les réseaux sont codés sur les structures manipulées par l'algorithme génétique. Huit propriétés théoriques qui permettent d'évaluer l'efficacité d'un codage de réseaux de neurones ont été proposées dans ce chapitre. Un codage particulier de réseaux de neurones qui posséde toutes les propriétés a été étudié. Ce codage, appelé codage cellulaire, comporte un alphabet de 20 allèles. Chaque allèle correspond à un opérateur cellulaire qui modifie légérement la topologie et les étiquettes d'un graphe de cellules. Ces opérateurs sont: { `END`, `PAR`, `CLONE`, `SEQ`, `SPLIT`, `MRG`, `CUT`, `ON`, `OFF`, `INC`, `DEC`, `MULT2`, `DIV2`, `INCLR` , `MULT2LR`, `WAIT`, `NOP_0`, `NOP_1`, `JMP`, `DEF` }. Ces opérateurs n'ont pas d'argument. Le codage cellulaire génére des neurones avec un seul type de sigmoïde qui est la fonction signe. Il est facile de rajouter des opérateurs si d'autres sigmoïdes sont requises.

Les propriétés du codage cellulaire peuvent être résumées comme suit:

Complétude N'importe quel réseau de neurones peut être codé. Donc l'algorithme génétique peut en principe toujours atteindre un réseau de neurones solution.

Compacité ou Parcimonie . Le codage cellulaire est topologiquement plus compact que les représentations classiques de réseaux de neurones. Il est fonctionnellement plus compact que n'importe quelle autre méthode de codage. Cela assure que les codes manipulés par l'algorithme génétique seront de taille minimale. Plus les codes sont petits, plus l'espace de recherche est petit. L'algorithme génétique a moins de calculs à faire pour explorer un petit espace de recherche.

Fermeture Le codage cellulaire développe toujours des architectures de réseaux valides (connexes) où chaque neurone est relié à la fois aux unités d'entrée et aux unités de sortie. Il produit des architectures acycliques ou cycliques suivant le graphe de cellules initial. Ces deux propriétés de fermeture assurent que l'algorithme génétique produit des architectures intéressantes. L'algorithme génétique ne rejette pas de codes. C'est une garantie d'efficacité.

Modularité Si un réseau de neurones peut être décomposé en plusieurs copies d'un sous réseau, le code du réseau complet comporte le code du sous réseau plus une partie de code qui spécifie comment connecter entre elles les copies du sous réseau. Cette propriété favorise la formation de blocs de base neuronaux (building blocks). Les régularités d'une architecture codée de façon modulaire rendent l'architecture facilement interprétable.

Propriété d'échelle Un code de taille fixe peut générer un famille de réseaux de neurones. Pour n'importe quelle famille de problèmes raisonables, il existe un code tel que le i^{eme} réseau résoud le problème i. Un problème paramétré de taille par exemple 1.000.000 peut être aussi facile à résoudre que le même problème de taille 2 ou 3, parce que le code des réseaux de neurones correspondant est le même, c'est seulement le paramètre de taille qui change.

Pouvoir d'expression Le codage cellulaire peut être interprété comme un langage machine pour réseaux de neurones. Son pouvoir d'expression est supérieur à celui des machines de Turing ou bien à celui des automates cellulaires. C'est une autre preuve de l'extrême compacité du codage.

Abstraction Il est possible de compiler un programme Pascal en un code cellulaire d'une famille de réseaux, qui simule le programme Pascal. L'espace de recherche des réseaux de neurones qui est très grand, peut être remplacé par l'espace de recherche des programmes plus petit. Cela aide l'algorithme génétique à trouver des réseaux de neurones, dans le cas où la fonction cible peut être calculée par un programme de petite taille.

grammaire Le codage cellulaire est une grammaire de graphe, cette propriété est très importante car c'est la racine de toutes les autres.

Chapitre 4

Un compilateur neuronal

4.1 Introduction

Une méthode de codage appelée codage cellulaire a été décrite dans les chapitres précédents. Le codage cellulaire peut être compris comme un langage machine de réseaux de neurones, capable de décrire un programme de machine de Turing, ou d'automate cellulaire. On peut utiliser le codage cellulaire comme un outil pour concevoir des réseaux de neurones simulant une machine de Turing ou un automate cellulaire. Il est déjà connu qu'on peut simuler une machine de Turing si on utilise un nombre infini de neurones. Voir [7] pour des variations sur ce thème, et la relation avec des automates cellulaires. Dans [30], les machines de Turing sont simulées par des réseaux de neurones ayant un nombre fini de neurones. Pour stocker un nombre infini de données, on utilise des activités de neurones qui sont des nombre réels ayant un nombre infini de décimales. Les activités des neurones doivent donc avoir une précision infinie. Ces résultats ont un intérêt uniquement théorique. Il est en effet inhabituel et inconfortable de programmer une machine de Turing ou un automate cellulaire. Dans ce chapitre, on propose une méthode pour compiler un programme écrit dans un langage procédural, vers un réseau de neurones. Plus qu'une preuve que le programme peut être simulé par un réseau de neurones, la compilation fournit une méthode effective afin de construire un réseau de neurones simulant un programme donné. On verra qu'elle présente plusieurs autres intérêts, comme la conception de gros réseaux de neurones, la parallélisation automatique, la compilation de systèmes hybrides naïfs. Comme les réseaux compilés comprennent un nombre fini de neurones, et leurs activités ont une précision limitée, ils ont une mémoire bornée. On ne peut donc pas compiler des instructions pour l'allocation[1] dynamique de mémoire. Les procédures récursives peuvent cependant être compilées, mais, durant l'éxécution du programme, la profondeur d'imbrication sera bornée par une constante donnée à l'avance. La performance du compilateur est mesurée en fonction de la taille du code généré. Le résultat confirme l'efficacité du compilateur, et permet de démontrer la propriété d'abstraction numéro 7 définie au chapitre précédent.

4.2 Les étapes de la compilation

Le procédé de compilation que l'on va présenter a été programmé. Le logiciel obtenu s'appelle JaNNeT (Just an Automatic Neural Network Translator) Le but de ce logiciel est uniquement de prouver la validité de la compilation, et de mesurer les performances du produit, c'est à dire par exemple, le nombre de neurones des réseaux compilés. JaNNeT comprend trois étapes montrées

[1] allouer de la mémoire signifie réserver une partie de la mémoire de l'ordinateur pour y ranger et y lire des données.

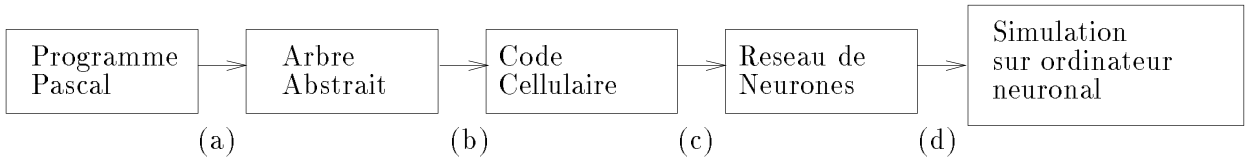

Figure 4.1: Etape du procédé de compilation proposé: (a) La génération de l'arbre abstrait à partir du programme Pascal, (b) La réécriture de l'arbre abstrait en un code cellulaire à l'aide d'une grammaire d'arbre, (c) Le décodage du code cellulaire en un réseau de neurones, (d) L'ordonnancement et l'allocation du réseau de neurones sur une machine physique. Cette étape n'est pas faite par JaNNeT

figure 4.1. L'entrée du compilateur est un programme écrit en langage Pascal "amélioré". Le produit est un réseau de neurones qui calcule ce qui est décrit par le programme Pascal. Le Pascal utilisé est dit "amélioré" parce qu'il propose des instructions supplémentaires par rapport au Pascal standard.

La première étape est l'extraction de l'arbre syntaxique à partir du programme Pascal. C'est une technique standard utilisée comme première étape pour compiler n'importe quel langage de programmation. Notons que cet arbre syntaxique n'a pas une forme habituelle. Dans l'appendice D.6, une grammaire décrit les arbres syntaxiques utilisés. En fait, l'arbre syntaxique utilisé a été obtenu à partir d'un arbre syntaxique standard, en faisant une suite de modifications. La section D.12 décrit les modifications apportées par rapport à un arbre syntaxique standard. La troisième étape (c) utilise le codage cellulaire. C'est le décodage du code cellulaire obtenu après la deuxième étape. Ce décodage fait intervenir le développement d'un graphe de cellules, vu au chapitre précédent.

La deuxième étape (b) constitue le coeur du compilateur. Cette étape est une réécriture d'arbre. Réécrire signifie écrire différemment, c'est à dire remplacer une suite de lettres par une autre suite de lettres, dans le cas simple de la réécriture de mots. Ainsi en partant d'un mot initial, on le réécrit en un autre mot, puis, encore en un autre mot, ainsi de suite jusqu'au mot final. La réécriture simplifiée d'arbres consiste à remplacer un noeud d'un arbre par un sous-arbre. Le sous-arbre recollé doit spécifier où recoller les sous-arbres fils du noeud réécrit. La deuxième étape consiste donc à remplacer chaque noeud de l'arbre syntaxique par un sous arbre étiqueté avec des opérateurs cellulaires. Lorsque les opérateurs constituant ce sous-arbre vont être exécutés par une cellule, il vont faire une transformation locale du graphe, en remplaçant la cellule initiale par plusieurs autres cellules, connectées aux voisines de la cellule initiale. Chaque noeud de l'arbre syntaxique peut donc être associé à une transformation locale d'un graphe de cellules. On associe à chaque noeud de l'arbre abstrait un macro-opérateur qui effectue cette transformation. Un macro-opérateur effectue une transformation plus importante que celle réalisée par un opérateur du code cellulaire. Dans tous les cas, excepté pour l'opérateur `SPLIT`, un opérateur cellulaire crée au plus une cellule, ou modifie au plus un registre, et il est paramètré par au plus un seul entier. Il faut plusieurs opérateurs cellulaires pour coder un macro-opérateur. Cela justifie l'appellation "macro-opérateur". L'exposé de la méthode de compilation pourra être fait sans faire référence au codage cellulaire. On se placera au niveau des macro-opérateurs, ceux-ci étant implémentés par des sous-arbres d'opérateurs cellulaires, décrits dans l'appendice D.9. Ainsi, ce chapitre peut être lu indépendamment des chapitres précédents.

Le compilateur JaNNeT prouve la proprieté d'abstraction vue au chapitre précédent. La taille du code cellulaire généré par JaNNeT peut en effet être bornée par k fois le nombre de noeuds de l'arbre abstrait, où k est le nombre maximum d'opérateurs nécessaires pour coder un macro-opérateur. Le codage cellulaire a donc un pouvoir d'expression supérieur au langage Pascal.

Dans sa version actuelle JaNNeT ne va pas "jusqu'au bout", c'est à dire qu'il ne produit pas une suite d'instructions exécutables par un ordinateur neuronal cible précis. Il produit un réseau de neurones, c'est une représentation adaptée à un ordinateur neuronal comprenant plusieurs processeurs spécialisés dans la simulation de neurones. Ce réseau de neurones est actuellement simulé sur un ordinateur classique. La quatrième étape figure 4.1 (d) consiste à allouer ce réseau sur une architecture physique d'ordinateur neuronal. Cette étape doit prendre en compte des considérations de taille mémoire, de communications entre microprocesseurs et de précision de calculs.

4.3 Les neurones utilisés

4.3.1 Sigmoïde

Un neurone est caractérisé par le calcul qu'il fait à partir de ses entrées. Un neurone classique fait la somme des entrées pondérées par les poids des connexions, puis applique à cette somme une fonction appelée sigmoïde. On utilise plusieurs sigmoïdes. Chacune à un numéro. Une cellule possède un registre `sigmo` dans lequel est rangé le numéro de la sigmoïde du futur neurone. Les quatre sigmoïdes utilisées sont indiquées figure 4.2.

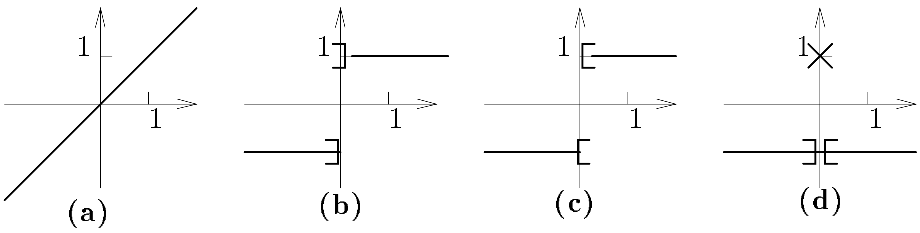

Figure 4.2: Différentes sigmoïdes utilisées. (a) identité,(b) marche-escalier-L (L comme Left) (c) marche-escalier-R (R comme Right),(d) égalité à zéro

Par ailleurs, on utilise pour la compilation de programmes écrits en Pascal des neurones non classiques qui font le produit de leurs entrées (Pi-unit), ou le quotient d'une entrée par une autre entrée (Div-unit). On pourrait simuler la multiplication et la division avec des neurones classiques, cependant, cela coûterait cher en neurones!

4.3.2 Dynamique

Pour assurer qu'un neurone calculant une opération arithmétique à deux opérandes attende que les deux opérandes soit arrivées avant de calculer son activité, on utilise une dynamique particulière. Les neurones décident quand déclencher le calcul de leur activité suivant 3 règles:

- Un neurone déclenche le calcul de son activité, dés qu'il a reçu les activités de chacun de ses voisins.
- Les neurones d'entrée sont initialisés avec le vecteur d'entrée

- Un neurone envoie son activité à ses voisins en sortie, lorsqu'il est initialisé, ou s'il vient de calculer son activité.

On n'utilisera pas d'unités spéciales pour coder les seuils. A un instant donné, tous les neurones qui peuvent calculer leur activité le font. Ainsi cette dynamique particulière (notée $\mathcal{D}_0$) est moitié parallèle, moitié séquentielle. Il n'y a pas besoin d'entités globales pour contrôler le flot d'activités. Deux autre dynamiques seront utilisées par le compilateur. La dynamique $\mathcal{D}_2$ est applicable sur un neurone ayant exactement deux entrées. Comme pour $\mathcal{D}_0$ le neurone attend d'avoir reçu les deux opérandes avant de procéder au calcul. Si la première entrée est strictement positive, le neurone se comporte comme un neurone $\mathcal{D}_0$, sinon le neurone est dit bloquant, il ne propage aucune activité. La dynamique $\mathcal{D}_1$ permet de bloquer un flux d'activité. Les neurones EAND que nous allons voir plus tard utilisent cette dynamique. La dynamique $\mathcal{D}_2$ est la suivante: le neurone remet à jour son activité dés qu'il reçoit une activité de l'un de ses voisins. Cette dynamique est utilisée par les neurones AOR, pour récupérer une activité sur un seul lien à partir de plusieurs liens, lorsqu'on ne peut savoir à l'avance par quel lien l'activité va arriver.

4.4 Microcodage

On est intéressé par une implantation pratique des opérateurs cellulaires et du codage cellulaire. Dans le but de produire un code cellulaire plus compact avec le compilateur, on a du introduire beaucoup d'autres opérateurs de division, ainsi que des opérateurs qui modifient les registres des cellules, ou qui font des transformations locales de la topologie sans changer le nombre de cellules, ou encore qui influent sur l'ordre d'exécution des opérateurs par les cellules. Chaque nouvel opérateur peut avoir un argument entier qui spécifie le plus souvent un numéro de lien, ou la valeur avec laquelle on modifie le registre, ou le numéro d'un arbre. L'usage d'arguments entiers favorise la représentation compacte.

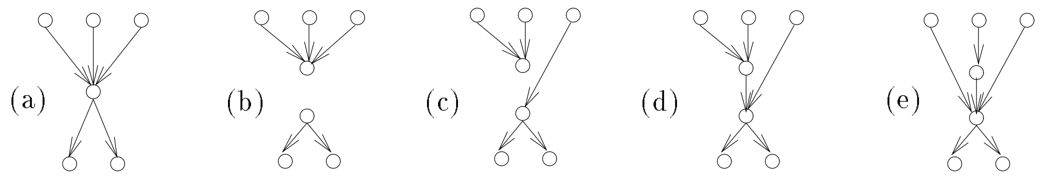

Figure 4.3: Microcodage de l'opérateur ADL 2. (ADd on Link). L'argument vaut la valeur 2. Cet opérateur intercale une cellule sur le lien numéro 2. Le microcode est: DIVm<sm> 2. (a) Avant la division ADL, (b) Division en deux fils, (c) le segment "m>" est analysé, (d) le segment "s" est analysé, (e) le segment "m<" est analysé

Pour plus de clarté, on a microcodé les opérateurs cellulaires. Un microcode est composé de deux parties. La première partie spécifie la catégorie de l'opérateur. Elle utilise trois lettres majuscules. DIV indique une division de cellules, TOP: une modification locale de la topologie, CEL, SIT, LNK: une modification d'un registre de respectivement: une cellule, un site, un lien. Une cellule possède une suite ordonnée de sous-sites d'entrée et une suite ordonnée de sous-sites de sortie. EXE, BRA indiquent des opérateurs utilisés pour gérer l'ordre d'exécution. BRA teste une condition, et suivant le résultat, la tête de lecture est placée sur le sous-arbre gauche ou sur

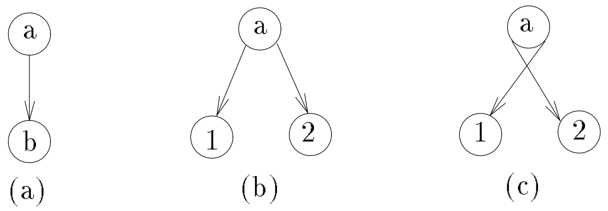

Figure 4.4: La différence entre un microcode de segment utilisant "d" ou utilisant "r". (a) On représente la situation avant la division de cellule. la cellule qui se divise est etiquetée "b". Elle a un voisin en entrée étiqueté "a". (b) le lien en entrée est dupliqué en utilisant un segment du type "d". (c) le lien en entrée est dupliqué en utilisant un segment du type "r". La seule différence est l'ordre des liens fait par la cellule "a" entre la cellule fille "1" et la cellule fille "2".

le sous-arbre droit. HOM indique une division en plus de deux fils. AFF indique des opérateurs utilisés seulement par le logiciel d'affichage de réseaux de neurones. Les différents opérateurs sont répertoriés à l'appendice D.5.

La seconde partie du microcode est une opérande. Pour CEL, SIT, LNK, l'opérande est une lettre minuscule codant le nom du registre. S'il y a un signe arithmétique, l'opérateur applique l'opération arithmétique sur le contenu actuel du registre et sur l'argument, puis il place le résultat dans le registre. Sinon il se contente de charger le registre avec la valeur de l'argument. Pour EXE, l'opérande est le nom particulier de l'ordre d'exécution codé par une lettre minuscule. Pour BRA, l'opérande indique le type de conditions à tester, codé par une lettre minuscule. Pour DIV, l'opérande est une liste de segments, chaque segment est composé d'une lettre et éventuellement d'une liste de signes arithmétiques. Au début, tout opérateur de division crée deux cellules. La cellule "1" hérite des liens en entrée, et la cellule "2" hérite des liens en sortie. La suite de segments est alors analysée dans l'ordre inverse pour déplacer ou dupliquer des suites de liens d'une cellule vers l'autre. La figure 4.3 donne un exemple d'une telle analyse. Si la lettre est minuscule, la suite de liens est déplacée ou dupliquée du site d'entrée de la cellule fille "1" vers le site d'entrée de la cellule fille "2". Si c'est une majuscule, la suite de liens est déplacée ou dupliquée du site de sortie de la cellule numéro 2, vers le site de sortie de la cellule numéro 1.

- La lettre "m" ou "M" indique: déplacer une suite de liens.
- La lettre "d" ou "D" indique: dupliquer une suite de liens. La figure 4.4 explique la légère différence entre "d" et "r". La lettre "r" ou "R" a presque le même effet que "d" ou "D". La seule différence est que le lien ajouté n'est pas ordonné de la même façon chez les cellules voisines.
- La lettre "s" indique qu'il faut connecter le site de sortie de la cellule fille numéro "1" au site d'entrée de la cellule fille numéro "2". Elle ne nécessite pas de signes arithmétiques.

Les signes arithmétiques spécifient la suite de liens qui va être déplacée ou dupliquée, en utilisant l'argument de l'opérateur.

- '<', '>', '=' indiquent les liens plus petit, égaux au supérieur à l'argument. '*' indique tous les liens.

- '#' et '$' indiquent le premier lien et le dernier lien.
- ˆ. indique qu'il faut considérer les liens de la cellule voisine, reliée à travers le lien dont le numéro est l'argument.
- '−' est utilisé lorsqu'il est nécessaire de compter les liens dans l'ordre décroissant. Si '−' est placé avant les trois lettre majuscules qui indiquent le type de l'opérateur, son interprétation est différente. Il indique alors qu'il faut échanger les rôles du site d'entrée et du site de sortie.
- '_' est utilisé pour remplacer l'argument par la valeur du registre de lien.
- '|' est utilisé pour remplacer l'argument par le nombre total de liens, divisé par deux.

Un microcodage similaire est utilisé pour les opérateurs du type TOP qui modifient localement la topologie en coupant ou en réordonnant les liens. Dans ce cas, la cellule fille unique hérite des liens en sortie, et l'analyse des segments indique des mouvements de liens du site d'entrée de la cellule mère vers le site d'entrée de la cellule fille.

Dans l'appendice D.5 il y a aussi un grand nombre d'opérateurs utilisés seulement pour étiqueter les cellules. Pendant le développement du réseau de cellules, le logiciel que nous avons programmé peut afficher les noms des opérateurs lus par les cellules. Les neurones gardent leur tête de lecture sur le code cellulaire, sur le dernier opérateur lu. Ainsi on utilise des opérateurs qui ne servent à rien pour étiqueter les feuilles du code cellulaire. Cela permet de voir à l'écran la fonction que chaque neurone réalise.

4.5 Nouveaux operateurs

Dans ce paragraphe on décrit quelques opérateurs cellulaires utilisés pour la compilation de programmes Pascal. Ces opérateurs ne peuvent être décrits de façon précise par leur microcode; c'est pourquoi on les explique séparément.

L'opérateur `BLOC` permet de bloquer le développement d'une cellule. Une cellule qui lit `BLOC` attend que toutes ses cellules voisines en entrée soient devenues des neurones. Cet opérateur est souvent utilisé dans le compilateur pour éviter qu'on ne commence à développer trop tôt certains morceaux de code cellulaire.

Le compilateur génére plusieurs codes cellulaires, un pour le programme principal et un par procédure où fonction. Ces codes aussi appelés "arbres-grammaire" ont un numéro. L'opérateur `JMP` utilise un argument entier et sert à relier les différents arbres entre eux. L'argument indique un numéro d'arbre-grammaire. La cellule de lecture qui exécute l'opérateur `JMP x` doit positionner sa tête de lecture sur la racine de l'arbre grammaire dont le numéro est x.

On a besoin de répartir les liens d'entrée en paquets de liens pour pouvoir les traiter séparément. Chacun des sites d'entrée et de sortie est divisé en plusieurs sous-sites. La façon dont ces sous-sites sont gérés est assez technique. La description qui suit n'est pas nécessaire pour la compréhension de la suite de l'exposé. Les sous-sites sont codés grâce à un registre de lien booléen appelé `begin-sub-site`. Quand ce registre contient la valeur 1, il indique qu'un nouveau sous-site commence à partir de ce lien. Par convention, le premier lien d'un site a son registre `begin-sub-site` positionné sur 0. Pour coder la place d'un lien entre deux cellules, une connexion est représentée par deux éléments, chacun inséré dans deux listes chaînées, Chacun de ces éléments contient un registre `begin-sub-site` différent. Car une connexion peut commencer un sous-site dans le site d'entrée, sans pour autant commencer un sous-site dans le site de sortie. Chacun des sites d'entrée et de sortie possède un registre booléen `divisible`.

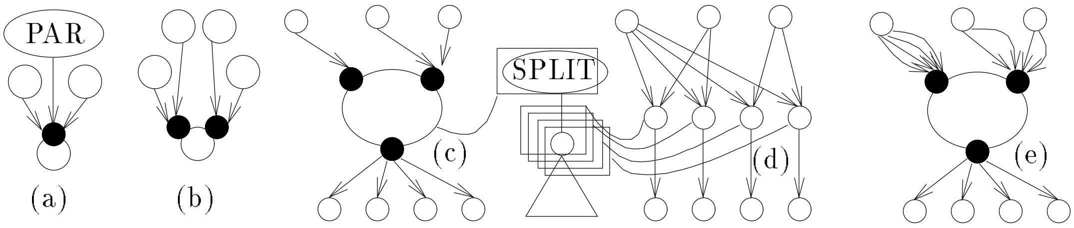

Figure 4.5: Propagation des marques de début de sous-site. Un sous-site est représenté par un petit disque noir. (a) Avant une division parallèle de la cellule étiquetée PAR (b) Après la division parallèle, on voit qu'un sous-site a été créé chez le voisin, dont l'état du registre `divisible` du site d'entrée était positionné sur 1. (c) Reprèsente la situation avant l'exécution de l'opérateur SPLIT, (d) Après l'exécution de l'opérateur SPLIT, (e) Représente une étape intermédiaire.

Lors de la division d'une cellule, un lien peut-être dupliqué, la règle suivante détermine la propagation du contenu du registre `begin-sub-site` de chacun des deux éléments de la connexion. Si le registre `divisible` du site auquel un élément de la connexion est rattaché, contient la valeur 1, le registre `begin-sub-site` est dupliqué. Sinon, le registre `begin-sub-site` est recopié dans l'élément de plus petit numéro, tandis-que l'autre élément a son registre `begin-sub-site` mis à 0. Cette règle signifie que si `divisible` est à 0, il n'y a pas création de sous-sites, mais il y a conservation des anciens sous-sites, si `divisible` est à 1, un sous-site se divise en deux sous-sites. Figure 4.5 (a) (b) donne un exemple de division de sous-sites.

D'autre part, il faut aussi mettre à jour les registres `begin-sub-site` pour les éléments de connexion correspondant aux cellules nouvellement créées par une division en deux (microcode commençant par DIV) ou une modification de la liste des connexions (microcode commençant par TOP). Tout d'abord, s'il y a division, les registres `divisible` de la cellule mère sont dupliqués. On a vu au paragraphe précédent que pour DIV et TOP, la liste des connexions est créée par la concaténation de plusieurs segments de connexions. Dans le cas où le registre `divisible` est à 1, des marques de début de sous-site sont insérées de façon à ce que chaque segment forme un sous-site. Dans le cas contraire, aucune marque de début de sous-site n'est insérée.

La répartition en sous-sites n'a d'intérêt que par rapport à l'opérateur cellulaire `SPLIT` défini dans le chapitre précédent. Cet opérateur a un comportement bien particulier dans le cas où il y a plusieurs sous-sites. La figure 4.5 (c) (d) (e) montre l'exécution du `SPLIT` dans le cas où la cellule a deux sous-sites d'entrée. C'est la généralisation du cas d'un seul sous-site. Certains liens sont dupliqués de manière à ce que le nombre de liens n sur chaque sous-site soit identique, avant que la cellule ne se divise en n cellules. Chaque cellule fille aura autant de liens en entrée qu'il y a de sous-sites d'entrée, de même pour les liens en sortie. Le registre `divisible` est dupliqué chez les cellules filles créées. Dans le cas où `divisible` est à 1, la cellule a autant de sous-sites qu'il y a de liens, dans le cas contraire il n'y a aucun sous-site.

4.6 Principe du Compilateur

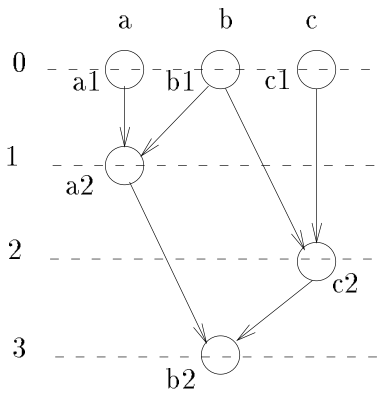

Figure 4.6: Résultat de la compilation du programme Pascal: *var a,b,c: integer; begin a=a+b; c=c+b; b=a+c; end.* Chaque ligne pointillée correspond à la compilation d'une instruction. Les neurones réalisent une simple addition de leurs entrées (Les poids sont à 1, la sigmoïde est l'identité). La ligne 0 représente l'environnement initial. Il est composé de trois valeurs: "a", "b", "c" contenues dans les neurones a_1, b_1 et c_1 . A la ligne 1, la valeur de "a" a changé, elle est contenue dans un nouveau neurone a_2 qui contient la somme de "a" et "b". Pour faire cette somme, le neurone a_2 est connecté en entrée aux neurones a_1 et b_1. La valeur de "c" change à la ligne suivante, puis la valeur de "b". On voit que les deux premières instructions: *a=a+b; c=c+b* peuvent s'exécuter en parallèle par les deux neurones a_2 et c_2. Sur cet exemple, chaque variable est représentée par deux neurones au cours du temps.

4.6.1 Structure du réseau compilé

Comment faire pour mimer un programme Pascal avec un réseau de neurones? La figure 4.6 montre un exemple simple de compilation. Chaque variable V du Programme Pascal contient une valeur. La variable est initialisée au début du programme, puis elle varie suivant les affectations qui lui sont faites. A chaque variable V, on fait correspondre autant de neurones qu'il y a de changements possibles de la valeur de V. Tous ces neurones représentent la variable V au cours du temps. A un instant donné du déroulement du programme, V a une certaine valeur v, celle ci est contenue dans un des neurones qui représente V. Ce neurone est connecté en entrée aux neurones contenant la valeur d'autres variables ayant servi à calculer v, et il est connecté en sortie à des neurones qui utilisent la valeur v pour faire d'autres calculs.

L'environnement, en compilation, désigne l'ensemble des variables accessibles, en un point donné du programme. Avant l'exécution de chaque instruction, l'environnement est représenté par une rangée r de neurones, autant de neurones qu'il y a de variables. Chacun des neurones contient la valeur de la variable qu'il représente à ce moment là particulier.

Une instruction va modifier l'environnement, en modifiant les valeurs de certaines variables. Une instruction se compile en un réseau intermédiaire calculatoire de neurones, qui va calculer les valeurs modifiées, à partir des anciennes valeurs contenues dans la rangée r, et va stocker ces nouvelles valeurs dans une nouvelle rangée de neurones r_1. Ainsi La totalité du programme

Pascal se compile en une suite de rangées de neurones, intercalées entre des couches de neurones intermédiaires calculatoires.

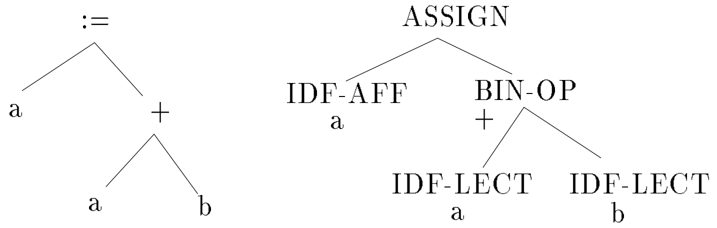

Figure 4.7: L'arbre syntaxique de l'instruction "*a:=a+b*". A gauche, l'arbre syntaxique est étiqueté par les mots du programme Pascal, à droite, les étiquettes comprennent deux parties: la première partie contient la nature du noeud, et la deuxième partie contient un attribut précisant la fonction du noeud. Par exemple IDF-AFF indique que le noeud contient le nom d'une variable dans laquelle on va écrire; IDF-LECT: le nom d'une variable dans laquelle on va lire; BIN-OP signale que l'on va appliquer un opérateur binaire comme l'addition, ASSIGN indique que l'on fait une affectation, le sous arbre gauche doit contenir le nom de la variable à modifier, le sous-arbre droit contient le calcul de la nouvelle valeur. Pour les noeuds IDF-AFF et IDF-LECT l'attribut du noeud contient le nom de la variable à écrire ou à lire. Pour le noeud BIN-OP l'attribut contient le nom de l'opérateur binaire, le noeud ASSIGN n'a pas d'attribut.

4.6.2 Compilation d'une instruction PASCAL

L'idée de la méthode de compilation consiste à traduire chaque mot du langage Pascal par une modification effectuée sur un graphe de neurones. En se plaçant à une grosse granularité, la compilation du programme se décompose en la compilation de chaque instruction, qui se traduit par l'ajout d'une couche calculatoire intermédiaire de neurones pour chaque instruction et celui d'une rangée de neurones contenant l'environnement modifié par l'instruction. Ainsi la compilation peut être vue comme un processus de construction. On construit la rangée de neurones contenant l'environnement initial, à partir des déclarations de variables du programme Pascal. Puis on dépose des couches de neurones calculatoires successives (c'est à dire, aptes à faire des calculs pour modifier les valeurs contenues dans l'environnement), en compilant le programme Pascal, instruction par instruction.

Cette idée de construction progressive du réseau de neurones compilé peut être appliquée avec une granularité plus fine qu'une instruction PASCAL. Une instruction PASCAL peut être décomposée en plusieurs mots du langage PASCAL, ces mots sont organisés suivant une structure d'arbre appelée arbre syntaxique (voir figure 4.7. On peut associer à chaque mot du langage PASCAL étiquetant l'arbre syntaxique une modification locale du réseau de neurones, de façon à ce que l'ensemble des petites modifications mises bout à bout aboutissent à l'ajout au réseau de neurones de la couche calculatoire correspondant à l'instruction. Ceci est mis en place en utilisant un formalisme à peu près identique au Codage Cellulaire. La couche de neurones va être développée en plusieurs étapes intermédiaires. Certains neurones en cours de développement vont posséder chacun une copie de l'arbre syntaxique, et une tête de lecture interne qui lit sur l'arbre syntaxique. On les appellera des cellules plutôt que des neurones, car leur rôle n'est pas de faire les calculs correspondant à un neurone, mais de se diviser de façon à créer la couche calculatoire de neurones. Chaque cellule lit sur l'arbre syntaxique à une position différente. Les étiquettes de l'arbre syntaxique vont représenter des instructions pour le développement des cellules qui agissent localement sur la cellule ou sur les connexions de la cellule. De façon plus formelle, Nous verrons par la suite que ces instructions peuvent se décomposer en une suite d'opérateurs cellulaires définis dans le Codage Cellulaire. Elles seront donc appelées des macros-opérateurs. Pendant un pas de développement, une cellule exécute le macro-opérateur référencé par l'étiquette qu'elle lit, et déplace sa tête de lecture vers le bas, dans l'arbre syntaxique.

Une cellule gère aussi un ensemble de registres internes. Certains sont utilisés pendant le développement, d'autres déterminent les poids et seuils du réseau de neurones final.

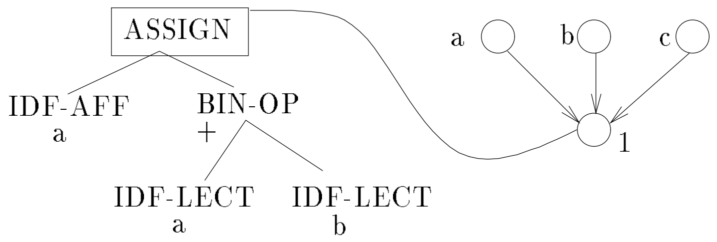

Figure 4.8: Développement de la couche neuronale correspondant à l'instruction PASCAL "*a:=a+b*", sur la moitié droite de cette figure. Le développement commence avec une cellule ancêtre étiquetée "1" connectée en entrée à trois neurones étiquetés "a", "b" et "c". Ces neurones contiennent respectivement les valeurs des variables "a", "b" et "c".. A gauche se trouve l'arbre syntaxique de l'instruction PASCAL. La flèche entre la cellule ancêtre et le symbole ASSIGN de l'arbre syntaxique, représente la tête de lecture de la cellule ancêtre. Les connexions sortantes et rentrantes sont ordonnées. C'est la position de la flèche d'arrivée sur le cercle symbolisant la cellule qui code le numéro du lien. Par exemple, le lien vers "b" est le lien numéro 2.

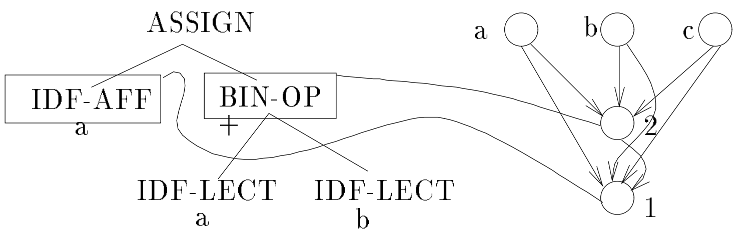

Figure 4.9: Exécution à l'étape 1 du macro-opérateur ASSIGN, lu par la cellule ancêtre "1" de la figure précédente. La cellule ancêtre "1" se divise en deux cellules filles "1" et "2". Les deux filles héritent des liens en entrée de la mère. La fille "2" est connectée vers la fille "1". A présent la tête de lecture de la cellule "1" est dirigée sur le noeud IDF-AFF, et celle de "2" est dirigée sur le noeud BIN-OP.

Considérons le problème de la construction d'un réseau pour compiler l'instruction Pascal

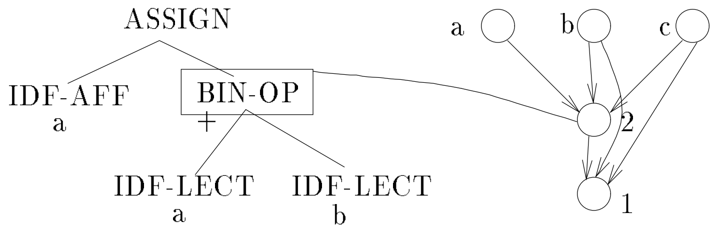

Figure 4.10: Exécution à l'étape 2 du macro-opérateur IDF-AFF, lu par la cellule "1". La cellule "1" supprime son premier lien, et déplace son quatrième lien pour le mettre en première place. Elle perd sa tête de lecture, et devient neurone.

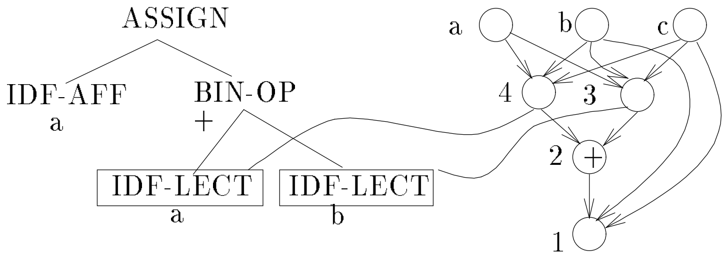

Figure 4.11: Exécution à l'étape 3 du macro-opérateur BIN-OP lu par la cellule "2". La cellule "2" se divise en trois cellules "2", "3" et "4". Les cellules "3" et "4" héritent des liens d'entrée de leurs mère, et sont connectées en sortie vers la cellule "2", tandis que "2" garde seulement son lien en sortie, perd sa tête de lecture et devient neurone. Le signe "+" à l'intérieur du cercle indique que le neurone "2" fait une simple addition de ses entrées.

"a:=a+b". Le développement du réseau correspondant à cette instruction entraîne la création de 4 cellules, et cependant un seul neurone reste en place à la fin du développement.

Le développement commence avec une seule cellule appelée cellule ancêtre connectée aux neurones contenant l'environnement courant. Considérons le réseau de départ indiqué sur la partie droite figure 4.8. Au départ la tête de lecture de l'ancêtre est positionnée sur la racine de l'arbre syntaxique, comme le montre la flèche connectant les deux. Ses registres sont initialisés avec des valeurs par défaut. Au fur et à mesure que la cellule ancêtre exécute les macro-opérateurs, elle donne naissance à toutes les autres cellules. A la fin, les cellules perdent leur tête de lecture et deviennent des neurones finis. Quand il ne reste plus de cellules de lecture, on obtient le réseau de neurones. L'exemple décrit dans les figures 4.8 à 4.12 donne une première idée du processus de compilation d'une instruction Pascal. Il est important de ne pas oublier que dans les figures, les connexions sortantes et rentrantes sont ordonnées. C'est la position de la connexion sur le cercle symbolisant le neurone, qui code le numéro de la connexion.

4.6.3 Compilation du programme PASCAL

A la sous-section précédente, on a montré que l'arbre syntaxique d'une instruction Pascal

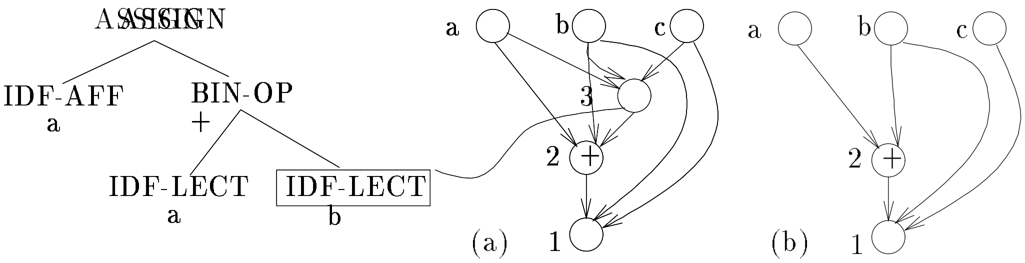

Figure 4.12: (a) Exécution à l'étape 4 du macro-opérateur IDF-LECT, lu par la cellule "4". La cellule "4" disparaît après avoir connecté son premier voisin d'entrée avec son voisin de sortie. (b) Exécution à l'étape 5 du macro-opérateur IDF-LECT, lu par la cellule "3". La cellule "3" disparaît après avoir connecté sa deuxième voisine d'entrée avec son voisin de sortie. Il ne reste plus de cellules de lecture, la compilation de l'instruction PASCAL "a:=a+b" est terminée. La cellule ancêtre "1" est bien connectée à trois neurones contenant l'environnement modifié. Elle peut commencer à lire et à développer l'instruction suivante.

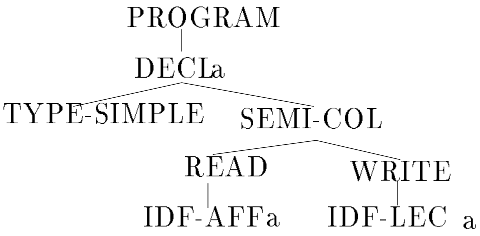

Figure 4.13: L'arbre syntaxique du programme PASCAL: "`program p; var a : integer; begin read(a); write (a); end.`" Cet arbre syntaxique n'a pas une forme habituelle. Il contient les déclarations de variables, et les étiquettes ne sont pas rangées dans l'ordre classique. Cet arbre a une forme particulière dans le but de pouvoir réaliser facilement la compilation du réseau de neurones. L'appendice contient une grammaire BNF décrivant la forme de ces arbres syntaxiques. Pratiquement, on obtient ces arbres syntaxiques à partir des arbre syntaxiques standard en faisant des rajouts et quelques modifications, décrites dans l'appendice

peut être interprété comme un arbre étiqueté avec des macros opérateurs cellulaires. Lorsque ces macros-opérateurs sont exécutés par des cellules, ils permettent de développer une couche de neurones qui calcule ce qui est spécifié par l'instruction Pascal. Cette démarche peut être généralisée à tout le programme Pascal. L'ensemble du programme Pascal, (ou d'un programme écrit dans n'importe quel langage d'ailleurs) peut être représenté par un arbre syntaxique. La figure 4.13 représente l'arbre syntaxique du programme Pascal "`program p; var a : integer; begin read(a); write (a); end.`". Les premier noeuds de l'arbre syntaxique correspondent à la partie déclaration de variables du programme Pascal. Ils vont créer la première couche de neurones contenant les valeurs par défaut de ces variables. Les noeuds suivants de l'arbre syntaxique correspondent aux instructions. Ils vont créer plusieurs couches de neurones calculatoires, les unes à la suite des autres.

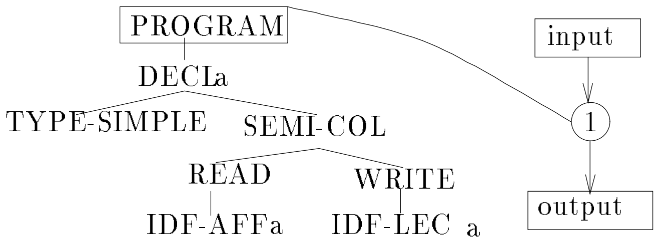

Figure 4.14: Développement du réseau de neurones simulant le programme Pascal "`program p; var a : integer; begin write (a); end.`" sur la moitié droite de cette figure. Le développement commence avec une cellule ancêtre étiquetée "1" et représentée par un cercle. La cellule ancêtre est connectée à la cellule pointeur entrée (rectangle étiqueté input) et à la cellule pointeur sortie (rectangle étiqueté output). A gauche se trouve l'arbre syntaxique de l'instruction PASCAL. La flèche entre la cellule ancêtre et le symbol PROGRAM de l'arbre syntaxique représente la tête de lecture de la cellule ancêtre.

Considérons le réseau de départ sur la moitié droite de la figure 4.14. La cellule ancêtre est reliée a au neurone pointeur d'entrée et au neurone pointeur de sortie. Celles-ci, indiquées par des rectangles sur la figure, n'exécutent pas de macro-opérateurs. A la fin du développement, la cellule pointeur d'entrée pointe sur les neurones d'entrée du réseau, et la cellule pointeur de sortie pointe sur les neurones de sortie du réseau. Le développement du réseau est décrit dans l'appendice D.1. Comme sur le premier exemple, on constate que le graphe de cellules passe par des étapes intermédiares où il atteint une taille de 6 cellules, et qu'à la fin il se réduit à 2 neurones.

4.7 Les macro-opérateurs

On a vu aux paragraphes précédents des petit exemples introductifs à la méthode de compilation. Celle-ci est basée sur la mise en correspondance des étiquettes de l'arbre syntaxique avec les macro-opérateurs cellulaires. On appelle chaque macro-opérateur du nom de l'étiquette qui lui correspond. Un macro-opérateur m remplace la cellule c qui l'exécute par un graphe d de cellules et de neurones. Ce graphe est rattaché au graphe global dans lequel se trouvait la cellule c par l'intermédiaire des voisines de c. Le terme "voisins" désigne deux cellules connectées entre elles, directement ou par l'intermédiaire de neurones spéciaux appelés neurones pointeurs. Le graphe d doit indiquer où va lire chaque cellule de lecture. Comme ces cellules de lectures vont lire chacune un des noeuds fils du noeud étiqueté par m, et que chacun des noeuds fils est lu en général une et une seule fois, le nombre des cellules de lecture du membre droit est égal au nombre de sous-arbres.

Les arbre-syntaxiques utilisés sont décrits par une grammaire d'arbre, appendice D.6. Cette grammaire précise la liste des macro-opérateurs, et la façon dont ils sont agencés entre eux. On fait référence par la suite aux non-terminaux de cette grammaire. On distingue deux types de macro-opérateurs: les macro-opérateurs de type expression, ils correspondent aux étiquettes générées à partir du non-terminal <expr> et génèrent des neurones servant à calculer des valeurs, et les macro-opérateurs du type "modification environnement" qui servent à gérer l'environnement.

Dans ce paragraphe on va présenter les macro-opérateurs non encore illustrés dans les exemples introductifs. On mettra en évidence un motif invariant qui est le contexte de reécriture. Le contexte de reécriture d'une cellule par un macro-opérateur est un invariant: Si l'environnement contient n variables, la cellules a $n+2$ liens en entrée et 2 liens en sortie. Le premier lien en entrée et en sortie pointe vers la cellule pointeur entrée, et la cellule pointeur sortie. Le deuxième lien en entrée pointe vers une cellule appelée cellule "start", le deuxième lien en sortie pointe vers une cellule étiquetée "next". La cellule "next" est un neurone si le macro-opérateur est du type "expression", ou une cellule de lecture bloquée si le macro-opérateur modifie l'environnement. La cellule "next" est chargée de faire la liaison entre le graphe généré par le macro-opérateur et le reste du réseau de neurones en amont.

Le premier macro-opérateur exécuté lors du développement du graphe de cellules est le macro-opérateur PROGRAM (figure D.1). Ce macro-opérateur a pour but de créer le motif invariant. Par récurrence cet invariant est ensuite conservé car pour chaque macro-opérateur m, les cellules de lectures contenues dans son membre droit (destinées à exécuter d'autres macro-opérateurs) ont leurs voisines qui vérifient le motif invariant.

Comment fait-on de façon pratique pour associer un macro-opérateur à une étiquette de l'arbre syntaxique? On respecte le motif invariant, on essaye de minimiser le nombre de neurones créés, et on s'assure que l'effet du macro-opérateur est bien celui souhaité. Dans le cas d'un macro-opérateur type "modification d'environnement", on s'assure que la modification est correcte, dans le cas d'un macro-opérateur type "expression", on s'assure que le sous-réseau généré calcule la bonne sous expression.

Dans l'implantation pratique, les macro-opérateurs sont décomposés en de plus petits opérateurs du code cellulaire. Pour chaque macro-opérateur, ces décompositions sont indiquées dans l'appendice. Les opérateurs du code cellulaire réalisent des petites modification de graphes. Par exemple, ils ne créent jamais plus de deux cellules, et sont paramètrables par un seul entier. Les macro-opérateurs, eux, créent un grand nombre de cellules. La décomposition d'un macro-opérateur en opérateurs est un moyen simple, efficace et rapide de programmer les macro-opérateurs. Cela permet de résumer tous les macros-opérateurs en deux pages d'expressions parenthésées, dans l'appendice D.8 et D.9. Dans la compilation, il arrive fréquemment que certaines cellules après exécution d'un macro-opérateur, bloquent leur développement jusqu'a ce que toutes leurs voisines soient des neurones, puis aillent exécuter un code cellulaire pour effectuer une certaine transformation.

4.7.1 Noyau du Pascal Standard

On considére dans ce paragraphe, les programmes écrits avec le noyau du langage Pascal standard, indispensable pour écrire des programmes non-triviaux. Pour le moment, on considére les variables scalaires, on considérera le cas des variables de type composé plus tard (tableaux à plusieurs dimensions, structures avec pointeurs). Considérons d'abord les macro opérateurs correspondants à la partie déclaration du programme Pascal. Ces macro opérateurs (notés `DECL`) créent la première couche du réseau de neurones, celle qui contient les valeurs initiales des variables de l'environnement. Les macro-opérateurs `DECL` sont chaînés les uns à la suite des autres. Chacun de ces macro-opérateurs a pour effet de rajouter un lien en entrée sur la cellule ancêtre, ce lien étant relié à une cellule, elle-même reliée à la cellule "start", par un poids 0 (Figure D.2 et D.3). Cette connexion assure que les neurones correspondant aux variables seront activés, et pourront faire démarrer le réseau. On suppose que toute variable est initialisée à 0.

L'exécution séquentielle repose sur une chaîne de cellules lectrices, associée à une liste d'instructions. La cellule de la chaîne à la position i lit le sous-arbre de l'instruction numéro i de la liste. Sur la figure D.4, les voisins d'entrée de la cellule "1" sont les neurones contenant l'environnement courant.

La cellule à la position $i+1$ possède un seul voisin d'entrée à savoir la cellule i. La cellule "2" en position 2 doit être retardée jusqu'a ce que la cellule "1" ait terminé le développement du sous-réseau qui modifie les variables suivant l'instruction 1. Pour ce faire, elle bloque son développement jusqu'a ce que toutes ses cellules voisines en entrée soient devenues des neurones.

L'affectation est l'exemple le plus simple d'instruction. Pour bien comprendre la figure 4.9, il faut se souvenir que chaque variable est repérée par son numéro de lien. Dans ces conditions, ranger une valeur dans l'environnement revient à connecter un neurone qui contient cette valeur, à la cellule ancêtre qui traduit les instructions, ceci en vérifiant bien que le numéro de la connexion est le bon. Cette gestion de lien est effectuée par l'opérateur IFD-AFF, figure 4.10

De la même façon pour pouvoir lire une variable, il suffit de connaître son numéro de lien et de connecter le neurone qui contient la valeur de la variable vers le neurone appelé "next" qui a besoin de cette valeur.

Les unités d'entrée du réseau correspondent aux valeurs écrites durant l'exécution du programme Pascal. Si dans le programme Pascal l'instruction *read a* apparaît, il y aura un neurone d'entrée créé. En positionnant la valeur initiale de ce neurone à v, la valeur de a sera initialisée à v. Les unités de sortie correspondent aux valeurs écrites durant l'exécution du programme Pascal. Si l'instruction *write a* apparaît, il y aura un neurone de sortie créé. A chaque fois que l'instruction *write a* est exécutée, la valeur de ce neurone sera la valeur de a. Un neurone devient unité d'entrée, s'il est connecté à la cellule pointeur entrée. Il devient unité de sortie, s'il est connecté à la cellule pointeur sortie. Le macro-opérateur READ établit une connexion vers la cellule pointeur entrée (figure D.5), de même que l'opérateur WRITE établit une connexion vers la cellule pointeur sortie (figure D.8).

Les opérateurs binaires (figure 4.11) se compilent en développant un arbre de neurones implémentant chacun une opération arithmétique particulière, grâce à une sigmoïde adaptée. Les opérateurs unaires se compilent de la même façon que les opérateurs binaires. Seulement 2 cellules sont créées au lieu de trois. L'une va développer le sous-réseau correspondant à la sous-expression à laquelle l'opérateur unaire est appliqué. L'autre va positionner sa tête de lecture sur le code cellulaire répondant au nom de l'opérateur unaire.

Jusqu'à présent, les macro-opérateurs réécrivaient une cellule en 0,1,2 ou 3 cellules au maximum. Le macro-opérateur IF reécrit une cellule en un graphe dont la taille est proportionnelle à la taille de l'environnement. Cela représente un pas dans la complexité de la reécriture. Cette nouvelle classe de macro-opérateurs peut être implantée grâce à l'utilisation de plusieurs sous-sites, et de l'opérateur `SPLIT`. Nous expliquons un peu plus en détail le macro opérateur `IF`. La ligne Pascal `if a then b else c` peut se traduire par la formule booléenne `( a AND b) OR ( (NOT a) AND c)` ou par la formule neuronale `( a EAND b) AOR ( (NOT a) EAND c)`. La valeur vraie est codée par 1, et la valeur faux par -1. Un NOT logique peut ainsi se réaliser avec un poids -1. Sur la figure D.12, on voit une couche de 6 neurones EAND, divisée en deux sous-couches de 3. Ces sous-couches permettent d'aiguiller le flot de valeurs contenues dans l'environnement vers le réseau calculant le corps du THEN, ou le réseau calculant le corps du ELSE. Rappelons que les neurones EAND (Extended AND) ont une dynamique spéciale. C'est un AND entier qui a pour entrées un booléen et un nombre entier naturel. Si le booléen est faux le neurone EAND ne sort aucune valeur: ni 0, ni 1. Si le booléen est vrai, la sortie vaut l'entier d'entrée. Ensuite un AOR (Arithmetic OR) récupère soit l'environnement de sortie du corps du THEN soit celui du corps du ELSE, pour le transmettre à l'instruction qui suit le "if then else". Les AOR ont aussi une dynamique spéciale. Ils remettent à jour leurs activités dès que l'une de leurs entrées a changé. Ces dynamiques sont décrites au paragraphe 4.3.2.

Lorsque dans un calcul, on a besoin d'une constante, celle-ci est stockée par le seuil d'un neurone dont la sigmoïde est l'identité. Ce neurone est relié à la cellule "start", de façon à contrôler son activité, c'est à dire, à quel moment et combien de fois il envoie la constante qu'il contient aux neurones qui en ont besoin

Le `while <condition> do <body>` correspond à un réseau de neurones récurrent dans lequel une boucle infinie de calculs peut avoir lieu, tandis que l'architecture reste bornée (figure D.15). Aucune allocation mémoire ne se produit dans le corps de la boucle. Le flot de valeurs contenues dans l'environnement commence par passer dans une couche de neurones AOR qui sert de tampon. Il passe ensuite dans un réseau qui calcule la condition d'arrêt. Suivant la valeur de la condition, le flot d'activités sort de la boucle ou bien est envoyé dans le corps de la boucle. Dans le dernier cas, il est réinjecté dans la couche de neurones tampon, après avoir passé une couche de neurones synchronisatrice (figure D.16). La synchronisation évite qu'un neurone donné du corps de la boucle ne remette à jour son activité deux fois, alors qu'un autre neurone ne l'a pas encore fait. Dans le cas de deux boucles imbriquées, les neurones correspondant à la boucle interne reviennent à leur état initial lorsqu'ils ont terminé la boucle, et sont prêts à en exécuter une autre si la boucle externe l'exige. Le neurone "start" trouve son utilité principale dans le corps d'une boucle. Il est considéré comme une variable locale, il a donc une copie de lui dans la couche AOR tampon. Cette copie est connectée à tous les neurones contenant des constantes utilisées dans le corps de la boucle. Cette copie réactive ces neurones à chaque tour de boucle. Cela est indispensable au bon fonctionnement du réseau. La compilation de l'instructin REPEAT est similaire à celle du WHILE. Le calcul de la condition se fait à la fin, au lieu du début.

4.7.2 Appels de procédure

Nous allons expliquer comment se compilent les procédures et les fonctions à l'aide d'un programme Pascal "fil rouge" qui nous servira à la fois pour préciser le passage des paramètres, l'appel, l'exécution et le retour de la procédure ou de la fonction. Voici ce programme.

```
program proc_func

Var glob:integer               (*variable globale*)

Procedure proc2(paf2:integer)  (*parametre formel*)
var loc2:integer;              (*variable locale de la procedure proc2*)
   begin
   ...
   end;

Procedure proc1(paf1:integer)  (*parametre formel *)
var loc1:integer;              (*variable locale de la procedure proc1*)
   begin
   proc2(pef2);                (*parametre effectif a instancier *)
   end;                        (*passes par valeurs a proc2*)

BEGIN                          (*corps programme principal*)
   proc1(pef1)                 (*parametre effectif a instancier*)
END.                           (*passe par valeurs a proc1*)
```

On utilisera un motif invariant légèrement modifié, comportant 3 variables de l'environnement. La première est une variable globale, la deuxième est un paramètre d'appel, la troisième est une variable déclarée localement dans la procédure appelée. Dans toutes les figures, le premier lien de sortie de la cellule "1" pointe vers la cellule pointeur sortie, et le deuxième lien vers la cellule pointeur entrée. Si on représente parfois l'inverse, c'est seulement pour obtenir un graphe planaire plus lisible.

Considérons les noeuds de l'arbre syntaxique du programme, correspondant à l'instant d'appel à proc2 dans proc1. Imaginons que nous sommes en train d'exécuter les macro-opérateurs associés à ces noeuds sur le graphe de cellules en cours de développement. L'environnement, du côté neuronal, comporte alors les variables globales, c'est à dire la cellule pointeur entrée, la cellule "start", la variable Pascal "glob"; plus celles qui sont locales a proc1, "pef1" et "loc1". Pour traduire l'appel de la procédure proc2, on commence par retirer de l'environnement les variables locales à proc1. Ceci est fait par le macro-opérateur CALL-P (figure D.17). Puis on développe les paramètres passés par valeur de proc1 à proc2 (macro-opérateur COMMA, figure D.19), enfin, la cellule ancêtre va développer le corps de proc2. Avant de développer les instructions de `proc2`, la cellule ancêtre commence par exécuter le macro-opérateur PROCEDURE qui insère une cellule chargée de dépiler les variables locales à proc2, lorsque proc2 rendra la main (code cellulaire POP, figure D.22). Après cela, une cellule insèrée par CALL-P exécutera un code cellulaire RESTORE qui permet de restaurer les variables locales à proc1.

Il n'est conceptuellement pas nécessaire de couper et de remettre les variables locales à proc1. Cependant, cela permet de pouvoir faire correspondre toujours le même numéro de lien à une variable donnée du programme.

Les macro-opérateurs CALL-P, CALL-F, et POP sont paramètrés par deux attributs. L'attribut GLOBAL est le nombre de variables globales de l'environnement. L'attribut LOCAL est le nombre de variables locales.

Le macro-opérateur CALL-F est plus simple que CALL-P. Il n'y a pas besoin de créer de cellule pour restaurer l'environnement correct, car une fonction ne renvoie pas un environnement mais juste une valeur. Le macro-opérateur FUNCTION a un effet nul. En effet, pour la même raison que précédemment, il n'est pas nécessaire de créer de cellule pour dépiler l'environnement. Le macro-opérateur RETURN a aussi un effet nul.

4.7.3 Les tableaux

Une des grandes richesses des langages de programmation est la structure de données tableau. Pour manipuler les tableaux, on se sert des neurones pointeurs. Ce sont des neurones utilisés uniquement pour structurer des données, en arbre. Ces neurones se trouvent aux noeuds de l'arbre. Les données se trouvent aux feuilles. L'utilisation de neurones pointeurs assure que la cellule ancêtre posséde un seul lien d'entrée par variable de l'environnement. Cela permet de traiter les variables séparément, ce qui est nécessaire lorsqu'on veut lire une variable ou la modifier. On peut représenter des tableaux Pascal à plusieurs dimensions, ou des structures (mot clé Pascal record). Le motif invariant doit prendre en compte ces arbres neuronaux. La figure D.25 montre un exemple de motif invariant étendu comportant un tableau 1D de deux éléments. La figure D.26 utilise un arbre de profondeur 2, pour représenter un tableau à 2 dimensions. Un arbre de profondeur d peut représenter un tableau à d dimensions. L'utilisation de plusieurs couches de neurones pointeurs pour stocker un tableau à plusieurs dimensions est nécessaire si on veut pouvoir manipuler les colonnes suivant chaque dimension.

Ce sont les macro-opérateurs correspondant à la partie du programme Pascal où la variable tableau est déclarée, qui créent cette structure d'arbre de neurones pointeurs. Pour cela, on code le type d'une variable par une chaîne de macro-opérateurs TYPE-ARRAY, (voir figure D.27). Chacun de ces macro-opérateur est paramètré par le nombre de colonnes existant dans la dimension qu'il représente. Cette chaîne se termine par un macro-opérateur TYPE-SIMPLE dont on a déjà vu l'effet figure D.3.

Beaucoup de macro-opérateurs doivent gérer les neurones pointeurs, à l'aide d'une exploration récursive de l'arbre des neurones pointeurs. Par exemple le IF, le WHILE, le X-AOR le X-SYNC. Ce sont des macro-opérateurs qui reconstruisent tous les neurones contenant l'environnement courant. La figure D.28 montre l'effet du code cellulaire X-AOR lorsque l'environnement contient un tableau. D'une façon générale, les macro-opérateurs déjà introduits: le IF, le WHILE, le X-SYNC doivent lire et ensuite reconstruire les arbres de pointeurs neurones structurant l'environnement. Comme on ne connaît pas à priori la profondeur de ces arbres, la seule façon de procéder est de remonter récursivement le long de ces arbres. Ce n'est pas un problème important, car le code cellulaire rend possible l'emploi de procédés récursifs. Pour cela on utilise l'opérateur `BPN x`: il teste si la cellule voisine en entrée de numéro x est un pointeur neurone, et exécute des codes cellulaires différents selon le résultat du test.

La lecture d'un tableau (par exemple: `a:=t[i1,i2]`) se traduit par un arbre syntaxique inhabituel. L'arbre syntaxique utilisé pour: `t[i1,i2]`, est indiqué dans l'appendice, figure D.29. La lecture d'un tableau est interprétée comme le résultat d'une fonction `1lect-index`, écrite en code cellulaire prenant $d+1$ entrées pour un tableau à d dimensions: les d indices et le nom du tableau. La fonction `1lect-index` renvoie la valeur lue dans le tableau.

L'écriture d'un tableau (par exemple `t[i1,i2]:=v`) se traduit par un arbre syntaxique du même genre que pour la lecture d'un tableau. Il est indiqué dans l'appendice, figure D.30. L'écriture d'un tableau à d dimensions est interprétée comme la valeur d'une fonction `1affect-index` écrite en code cellulaire à $d+2$ entrées et une sortie qui est un tableau. Les entrées comprennent les d indices, la valeur à affecter, et le nom du tableau. La fonction `1affect-index` renvoie le tableau modifié après l'écriture du tableau.

On peut voir sur les figures que les réseaux développés pour lire ou écrire dans un tableau sont de taille importante. Lorsque les indices utilisés pour lire ou écrire dans le tableau ne peuvent être connus statiquement (c'est à dire avant l'exécution), la complexité du réseau développé, pour lire ou pour écrire, est proportionnelle au nombre d'éléments du tableau. Dans le cas contraire, à cause des simplifications de neurones (appendice D.14), la complexité du sous-réseau ajouté est bornée indépendamment de la taille du tableau.

4.7.4 Langage Pascal Amélioré

Dans ce paragraphe, nous décrivons la compilation de nouvelles instructions rajoutées au langage Pascal standard, pour mettre en valeur le compilateur neuronal. On a d'abord rajouté l'instruction CALLGEN, qui permet de faire appel à une fonction définie en code cellulaire. Préalablement, au début du programme Pascal doit figurer une instruction `#include <nom-fig>` qui indique un nom de fichier où doivent se trouver les codes cellulaires des fonctions appelées par CALLGEN. La syntaxe de CALLGEN est la suivante: `<idf0> := CALLGEN("nom-code ", <idf1>,..,<idfk>)`, le résultat de l'appel sera une valeur affectée à la variable <idf0>, le nom "nom-code " indique le nom du code cellulaire que l'on veut développer, <idf1> .. <idfk> sont des paramètres d'appels. Le macro-opérateur CALLGEN a le même effet que CALL-F, sauf que le corps de la fonction appelée n'est pas défini par un arbre syntaxique mais par un code cellulaire préexistant à la compilation. Les neurones contenant les valeurs des paramètres <idf1> .. <idfk> sont connectés en entrée du réseau inclus. Les sorties du réseau inclus sont connectée aux neurones représentant la variable <idf0>. Une autre différence entre `CALLGEN` et `CALL-F` est qu'avant d'exécuter le corps du CALLGEN, tout l'environnement est supprimé. En effet CALLGEN a été conçu pour pouvoir appeler un certain nombre de fonctions prédéfinies et réutilisables dans plusieurs contextes, ou inclure des réseaux de neurones "blocs de bases" prés-appris. Les réseaux développés par un appel CALLGEN n'ont jamais besoin des variables globales. Ils est donc plus simple de programmer la supression des liens reliant les variables globales, au niveau de l'exécution du macro-opérateur CALLGEN plutôt qu'à chaque fois, dans le code cellulaire appelé par CALLGEN. Des exemples de fonctions prédéfinies dans l'appendice D.10 sont:

- La fonction "gauche" qui extrait la partie gauche d'un tableau
- La fonction "droite" qui extrait la partie droite d'un tableau
- La fonction "concat" qui permet de concaténer deux tableaux
- la fonction "int-to-array" qui rajoute un pointeur neurone qui introduit une dimension supplémentaire. Si le paramètre passé est un tableau à d dimensions, "int-to-array" qui renvoie un tableau à $d+1$ dimensions.
- La fonction "array-to-int" qui supprime un pointeur neurone et enléve une dimension. Si le paramètre passé est un tableau à d dimensions, "int-to-array" renvoie un tableau à $d-1$ dimensions.
- La fonction "randomize" qui permet d'initialiser un tableau avec des valeurs aléatoires dans $\{-1, 0, 1\}$.

Un des buts principaux du compilateur est de parallèliser automatiquement un programme écrit avec une stratégie de résolution "diviser pour régner". Dans cette stratégie, un problème de taille n est divisé en deux sous-problèmes de taille $n/2$ puis en quatre sous-problèmes de taille $n/4$ etc... On peut utiliser les fonctions de tableaux prédéfinies pour compiler de tels problèmes. Les fonctions "gauche " et "droite " permettent de diviser les données contenues dans un tableau en deux parties égales. Pour cela, on répartit les colonnes suivant la première dimension en deux sous-tableaux contenant moitié moins de colonnes. Cela peut fonctionner même si le nombre de colonnes est impair, auquel cas l'un des sous-tableaux a une colonne de plus que l'autre. Cependant, il faut pouvoir tester de façon statique (c'est à dire avant l'exécution des calculs du réseau, durant la compilation) l'arrêt de la récursion. Dans ces divisions en deux et encore en deux, on s'arrête de diviser les problèmes en sous-problèmes lorsque la taille des sous-problèmes devient un. On applique alors une procédure particulière. Il faut donc pouvoir tester statiquement, au moment du développement du réseau de neurones, si le nombre de colonnes d'un tableau suivant la première dimension vaut 1. Suivant la réponse, on va décider de développer une partie différente de l'arbre syntaxique. La syntaxe du IF statique est la même que celle du IF normal, sauf qu'on remplace IF, THEN et ELSE par #IF, #THEN et #ELSE. Toutefois, la condition du #IF est toujours la même expression. Cette expression s'écrit `t=1` (où t est un tableau). Les macro-opérateurs #IF et #THEN testent si le tableau t a 1 seule ou plusieurs colonnes suivant sa première dimension. Dans un cas la cellule ancêtre va lire le sous-arbre gauche du #THEN, dans l'autre cas elle va lire le sous-arbre droit.

4.8 Résultat de compilation

Dans ce paragraphe, on propose quelques exemples de compilation illustrant le fonctionnement du compilateur et son intérêt. Les unités d' entrée du réseau correspondent aux valeurs lues durant

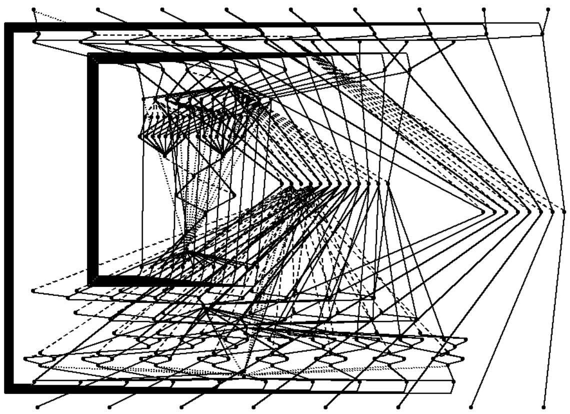

Figure 4.15: Un réseau de neurones pour trier 8 nombres, en utilisant la méthode du tri par bulles.

l'exécution du programme Pascal. Si dans le programme Pascal l'instruction *read a* apparaît, il y aura un neurone d'entrée créé. En donnant v comme valeur initiale à ce neurone, la valeur de a sera initialisée à v. Les unités de sortie correspondent aux valeurs écrites durant l'exécution du programme Pascal. Si l'instruction *write a* apparaît il y aura un neurone de sortie créé. A chaque fois que l'instruction *write a* est exécutée, la valeur de ce neurone sera la valeur de a. Il y a un neurone d'entrée particulier appelé *start*, utilisé pour démarrer le réseau. Pour faire fonctionner le réseau, on doit initialiser les unités d'entrées avec les valeurs voulues, puis on relaxe le réseau (C'est à dire, on effectue les calculs du réseau) avec une dynamique à mi-chemin entre parallèle et séquentielle. Celle-ci est définie à la sous-section 4.3.2. Avec la dynamique parallèle, tous les neurones mettent à jour leur activité à chaque itération, avec la dynamique séquentielle, un seul neurone met à jour son activité à chaque itération. Avec la dynamique utilisée, plusieurs neurones mettent à jour leur activité à chaque itération.

4.8.1 Compilation de programme Pascal Standard

La figure 4.8.1 montre un réseau compilé avec le programme suivant:

```
program p;
type tab = array [0..7] of integer;
var t : tab; i, j, max, aux : integer;
begin
  read (t);
  while i < 7 do
  begin
    j := i + 1;max := i;
    while j <= 7 do
    begin
      if t[max] < t[j] then max := j  fi;
      j := j + 1;
    end;
    aux := t[i];t[i] := t[max];
    t[max] := aux; i := i + 1;
  end;
```

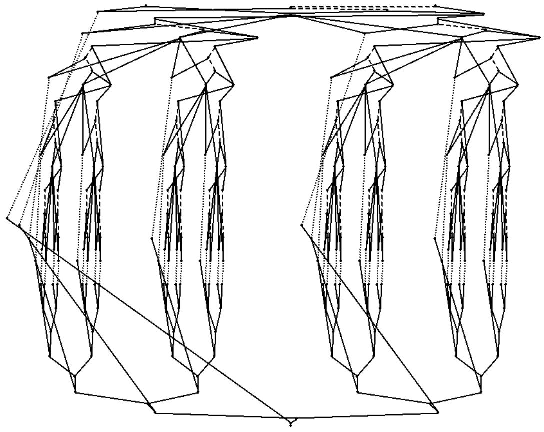

Figure 4.16: Un réseau de neurones pour calculer la suite de fibonaci, développé jusqu' à une profondeur 6.

```
  write (t);
end.
```

Il y a 9 unités d'entrée, la cellule start et les 8 valeurs à trier. Il y a 8 unités de sortie correspondant à l'instruction `write(t)`. On peut voir les deux boucles encapsulées qui sont représentées par deux jeux de liens récurrents. La figure 4.16 montre le réseau compilé avec le programme suivant:

```
program Fibonacci;
var resul:integer;a : integer;
function fib(n:integer):integer;
begin
  if (n<=1) then resul := 1
  else resul := fib(n-1)+fib(n-2) fi;
  return (resul);
end;
begin
  read (a); write (fib (a));
end.
```

Cet exemple illustre que la profondeur de récursion doit être bornée avant que ne commence la compilation, si le programme à compiler utilise les appels récursifs. Ici la profondeur de recursion est bornée par 6, donc, ce réseau permet le calcul de `fib(i), i<6`. En fait, le chiffre 6 correspond à la valeur initiale de la vie de la cellule ancêtre.

4.8.2 Utilisation de l'instruction CALLGEN

Nous avons ajouté au langage Pascal deux instructions *CALLGEN* et *#include* qui permettent d'inclure dans le réseau final des petits réseaux dont les poids ont été préalablement ajustés à l'aide d'un algorithme d'apprentissage. Le code de ces réseaux se trouve dans des fichiers dont les noms sont spécifiés par une instruction *#include* Ci dessous, un exemple de programme Pascal utilisant l'instruction CALLGEN. Le réseau compilé est montré figure 4.17.

```
#include "bete.a"
```

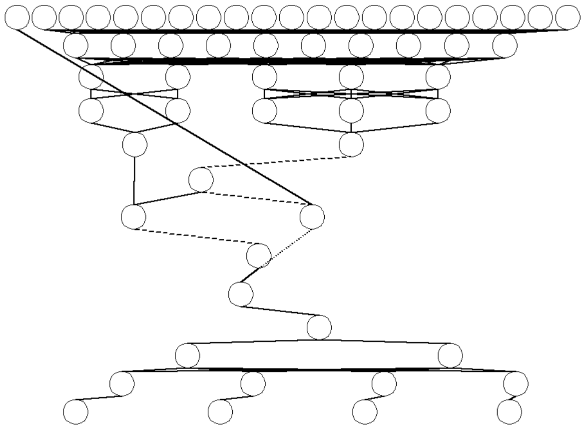

Figure 4.17: Un réseau de neurones hybride simulant le comportement d'une bête.

```
program bete;
type tab=ARRAY[0..19] of integer; tab2=ARRAY[0..9] of integer;
var pixel:tab; feature:tab2; position,est_mechant,mouvement:integer;

function opposer(position:integer):integer;
begin  return(1-position); end;

begin
  read(pixel); feature:=CALLGEN("retine",pixel);
  position:=CALLGEN("position_objet",feature);
  est_mechant:=CALLGEN("predateur",feature);
  if(est_mechant=1) then mouvement:=opposer(position)
  else mouvement:=0 fi;
  write(CALLGEN("moteur",mouvement));
end.
```

Dans le fichier *bête.a* se trouvent les codes de plusieurs réseaux en couches. Le réseau *rétine* prend en entrée des pixels et extrait des caractéristiques intéressantes. Supposons qu'un objet se trouve dans le champs visuel. Le réseau *position-objet* détermine à partir de ces caractéristiques, la position de l'objet, le réseau *prédateur* détermine si l'objet est un prédateur, le réseau *moteur* commande les 4 pattes de l'animal à partir d'une direction de mouvement. L'ensemble du réseau compilé permet de simuler le comportement de fuite de l'animal en face d'un prédateur. Le réseau final *bete* a en entrée des pixels, et en sortie les neurones moteurs des muscles des pattes. Ce réseau peut être stocké dans une bibliothèque pour être à son tour inclus dans un autre réseau plus grand. Cet exemple illustre la souplesse de l'interface entre le langage Pascal et le connexionisme. Le réseau compilé comprend plusieurs couches de neurones apprises, plus une partie qui est purement symbolique. C'est la partie correspondant à l'instruction IF. Ainsi le compilateur, non seulement peut relier entre eux différents sous-réseaux, mais peut aussi faire des calculs à partir de leurs sorties et plus généralement, gérer les entrées/sorties.

4.8.3 Utilisation de l'instruction #IF

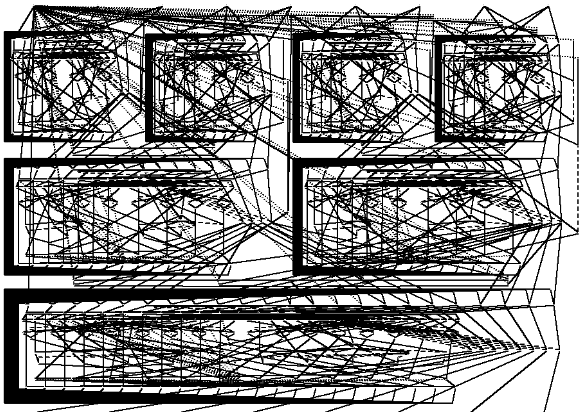

Figure 4.18: Un réseau de neurones pour trier 8 entiers en parallèle

Nous avons ajouté au langage Pascal une instruction *#IF #THEN #ELSE* qui permet de tester une condition au moment de la compilation. La condition testée a toujours la même forme. On teste si la première dimension d'un tableau contient plus d'une colonne. Cette instruction permet de parallèliser automatiquement un programme écrit avec une stratégie de résolution "diviser pour régner". Dans cette stratégie, un problème de taille n est divisé en deux sous-problèmes de taille $n/2$ puis en quatre sous-problèmes de taille $n/4$ ... etcr. On utilise toujours l'instruction #IF conjointement avec des fonctions appelées par CALLGEN qui enrichissent la structure de données tableau. Par exemple, "gauche " et "droite " permettent de diviser les données contenues dans un tableau en deux parties égales. "int-to-array" et "array-to-int" permettent de rajouter ou de supprimer une dimension. Ci dessous, un exemple de programme Pascal utilisant l'instruction CALLGEN. Le réseau compilé est montré figure 4.18.

```
#include "array.a"
program Trifusion;
const infini=10000; type tab = ARRAY [0..7] of integer; var w : tab;

function fusion(t:tab;n:integer;u:tab;m:integer): tab;
var v:tab; i,j,a,b:integer;
begin
  v:=CALLGEN("concat",t,u); a:=t[0];b:=u[0];
  while(i+j<n+m) do
  begin
    if(a<b) then
    begin  v[i+j]:=a;i:=i+1; if (i<n) then a:=t[i] else a:=infini  fi; end
    else
     begin v[i+j]:=b;j:=j+1; if (j<m) then b:=u[j] else  b:=infini fi; end
    fi;
  end;
  return(v);
end;

function tri(t:tab;n:integer) : tab;
begin  #IF t=1
       #THEN return(t)
```

```
        #ELSE return(fusion(tri(CALLGEN("gauche",t),n DIV 2),  n DIV 2,
                  tri(CALLGEN("droite",t),n-(n DIV 2)), n-(n DIV 2)))
        #FI;
end;

begin read(w); write(tri(w,8)); end.
```

La fonction fusion est écrite en Pascal standard, la fonction tri utilise le Pascal amélioré pour programmer la stratégie "diviser pour régner". Le fichier *array.a* contient la bibliothèque des codes cellulaires, définissant les fonctions manipulant les tableaux. Le réseau compilé prend en entrée 8 entiers et sort la liste triée. La structure du réseau est assez claire. La première couche de réseaux récursifs sort 4 listes triées de deux nombres chacune, la deuxième sort deux listes triées de 4 nombres. Bien qu'étant récursive, la fonction tri se compile en un graphe fini, parce qu'elle est appliquée sur un tableau fini d'entiers. Comme on a utilisé une stratégie "diviser pour régner", le réseau trie n nombres en un temps $O(n)$, à condition que les neurones puissent faire leurs calculs en parallèle. Nous avons aussi compilé beaucoup d'autres programmes utilisant la stratégie "diviser pour régner". La convolution, le produit matriciel, le maximum d'une liste d'entiers ont été programmés avec une stratégie "diviser pour régner" et compilés en un réseau capable d'exécuter la tâche en un temps $O(\ln(n))$.

4.9 Application du compilateur

La compilation neuronale est un procédé basé sur un paradigme nouveau: la génération automatique de réseaux de neurones à partir d'une description algorithmique. Ce paradigme est à l'opposé exact du courant tout-puissant et unilatéral qui à l'heure actuelle présente les réseaux de neurones sous l'aspect "machine qui apprends toute seule". Nous allons à présent montrer que la compilation neuronale peut reproduire et même améliorer trois sortes de compilations existantes. La compilation neuronale est donc un procédé capable de cumuler en les améliorant les intérêts de ces trois sortes de compilations.

4.9.1 Conception de réseaux de neurones

L'intérêt d'un langage de description de réseaux de neurones tel que SESAME [20] est de faciliter la conception des réseaux de neurones de grande taille. Ces langages proposent de définir des réseaux de neurones "blocs de bases" puis ensuite d'assembler ces blocs pour construire des réseaux de neurones plus complexes, ces derniers pouvant à leurs tour être assemblés etc... Cette conception hiérarchique est très pratique. Nous avons voulu obtenir le même effet avec la compilation neuronale. Pour ce faire, on a complété le langage Pascal avec l'instruction CALLGEN, qui permet d'inclure des sous-réseaux de neurones prédéfinis. Au début du programme Pascal doit figurer une instruction `#include <nom-fig>` qui indique un nom de fichier où doivent se trouver les codes cellulaires des sous-réseaux à inclure. La syntaxe de CALLGEN est la suivante: `<idf0> := CALLGEN("nom-réseau ", <idf1>,..,<idfk>)`. Le "nom-réseau" indique le nom du réseau à inclure. Les neurones de sortie du réseau qui est inclus, seront connectés aux neurones représentant la variable <idf0>. D'autre part, <idf1> .. <idfk> sont des paramètres d'appel contenus dans des neurones qui seront connectés aux neurones d'entrée du réseau inclus. Lorsqu'il compile un réseau, le compilateur neuronal produit son code cellulaire, ce code cellulaire peut être stocké dans une bibliothéque, sous forme d'un fichier. Ainsi le réseau compilé pourra être inclus dans un autre réseau de plus grande taille, qui à son tour pourra être codé et inclus dans un réseau d'ordre supérieur. Cette technique permet une compilation modulaire et une conception hiérarchique des réseaux de neurones. Mais la compilation neuronale va beaucoup plus loin que des langages de description tels que SESAME. La compilation neuronale fournit la possibilité de combiner des réseaux briques de bases sans faire référence à des connexions physiques entre réseaux, mais en se plaçant à un niveau logique, soft. Un être humain comprend beaucoup mieux une description soft où il suffit de décrire les étapes logiques d'un calcul, plutôt qu'un ensemble de connexions enchevêtrées. Ainsi la compilation neuronale permet d'envisager la conception de très grands réseaux. Dans un futur proche, des machines avec des millions de neurones seront disponibles, et les avantages d'une description logique et soft deviendront évidents. De plus, la compilation neuronale ne se contente pas de fournir le réseau de neurones, elle fournit en même temps une représentation graphique de ce réseau. A notre connaissance, c'est le seul logiciel permettant d'afficher un graphe quelconque en rendant compte des régularités de ce graphe.

4.9.2 Un outil pour systèmes hybrides naïfs

La compilation neuronale est un procédé permettant de compiler un programme pascal en un réseau de neurones. Pour compiler une base de règles utilisée en Intelligence Artificielle en un réseau de neurones, il suffit d'écrire la base de règles et le moteur d'infèrences en langage Pascal. La compilation neuronale inclut donc les compilateurs de bases de règles, utilisés dans les systèmes dits hybrides. Ces systèmes tentent de combiner intelligence artificielle et réseaux de neurones. Cependant la compilation neuronale peut intervenir dans la construction de systèmes hybrides d'une autre façon. Ce compilateur permet de représenter dans le même modèle de réseaux de neurones artificiels: 1) de la connaissance sous forme d'algorithme de résolution de problèmes qui est compilée; 2) de la connaissance provenant d'un apprentissage dans les réseaux de neurones. Le compilateur permet ainsi de construire des modèles hybrides naïfs, en construisant des réseaux en deux couches. La première couche est apprise et elle est sub-symbolique, elle calcule des symboles en les ancrant sur des données d'entrée floues et distribuées. La seconde couche est compilée, elle fait un calcul à partir de ces symboles. Le systèmes hybride obtenu est naïf car les symboles ne sont pas reliés entre eux.

4.9.3 Parallélisation automatique

Les exemples numériques montrent comment la compilation neuronale réussit à parallèliser automatiquement des problèmes tels que le tri ou la multiplication matrice-vecteur, ou matrice-matrice. La compilation neuronale peut parallèliser les algorithmes du type "diviser pour régner". Cette classe générale d'algorithmes est définie comme suit: On divise un problème en deux problèmes de taille moitié. Chacun de ces sous-problèmes est alors redivisé en deux sous-sous problèmes de taille moitié etc, jusqu'a ce qu'on arrive à un problème indivisible qui lui, peut se résoudre en temps constant. Comme les sous-problèmes sont tous indépendants, il peuvent être résolus par deux sous-réseaux de neurones différents, en même temps. D'ou le parallélisme qui en résulte.

Dans sa version actuelle le compilateur neuronal ne peut cependant être considéré comme un compilateur parallèle, car il ne produit pas une suite d'instructions exécutables par un ordinateur parallèle cible particulier. Il produit un réseau de neurones qui est une représentation adaptée à un ordinateur parallèle, à cause de la nature intrinsèquement parallèle d'un réseau de neurones. L'étape finale consiste à allouer ce réseau sur une architecture physique d'ordinateur parallèle. Cette étape doit prendre en compte des considérations de taille mémoire, de communications entre processeurs et de précision arithmétique des calculs.

On ne peut donc trop s'avancer en ce qui concerne la puissance du compilateur neuronal comme compilateur parallèle. Cependant, les réseaux de neurones utilisés ont une structure 3D. Une des dimensions de cette structure représente le temps, il paraît possible de projeter le réseau sur un plan 2D, suivant la dimension du temps, pour associer le réseau sur une grille 2D de processeurs. On est en train de travailler sur cette dernière étape avec comme ordinateur cible une carte DECPerle1 de Digital Equipment Corporation. Le compilateur neuronal n'est pas totalement un compilateur-paralléliseur, car il ne fonctionne que si les algorithmes sont écrits avec une stratégie "diviser pour régner". Cependant beaucoup d'algorithmes peuvent se programmer ainsi, et on peut dire que l'avantage de la compilation neuronale par rapport a l'approche traditionnelle décrite dans [27] est que l'utilisateur n'a pas besoin de rajouter des indications dans son programme. Ces indications précisent le travail de chaque processeur et l'endroit où vont être stockées les données.

Partie II: Utilisation du codage cellulaire avec l'algorithme génétique

On va étudier dans cette deuxième partie la synthèse de réseaux de neurones en utilisant l'algorithme génétique (GA) avec le codage cellulaire (CE). On suppose que le lecteur est familier avec le CE. Le premier chapitre de cette deuxième partie décrit des simulations faites avec l'algorithme génétique brut, sans utiliser d'apprentissage. L'utilisation d'apprentissages sera introduite dans les chapitres suivants.

Chapitre 5

Synthèse de réseaux de neurones génétiques sans apprentissage

5.1 La programmation génétique

Dans l'introduction, on a présenté le GA comme une procédure d'optimisation qui manipule des codes appelés chromosomes. On a cité l'exemple d'école, dans lequel les chromosomes utilisés sont des chaînes de bits de longueur fixe. Il est possible d'utiliser n'importe quelle structure pour les chromosomes, pourvu que deux opérateurs soient définis sur ces structures. Un opérateur binaire de recombinaison, et un opérateur unaire de mutation. La recombinaison crée un nouveau chromosome fils en mélangeant de l'information provenant de deux chromosomes parents. La mutation modifie légérement un chromosome. La recombinaison permet un échange d'information entre les chromosomes. La mutation réalise une recherche locale. La tendance actuelle, dans la communauté des gens qui utilisent des stratégies évolutionnistes comme le GA, est de mettre le maximum d'information sur le chromosome et dans les opérateurs, pour assurer que les chromosomes générés par le GA sont toujours valides. Autrement dit, les chromosomes doivent avoir une syntaxe correcte, et doivent toujours être évaluables; et les opérateurs de recombinaison et de mutation doivent générer à partir de structures syntaxiquement correctes, des structures syntaxiquement correctes

En resumé, la structure du chromosome doit contenir toute l'information de nature syntaxique, pour que dans la recherche menée par le GA, les problèmes de syntaxe soient résolus. Koza a montré que la structure de donnée "arbre" pouvait être efficacement recombinée par le GA. La programmation génétique a été définie par Koza comme l'évolution de programmes écrits en LISP (langage de programmation utilisant des expressions parenthésées appelées S-expressions) avec l'algorithme génétique [17]. Dans le paradigme de la programmation génétique, les chromosomes sont des S-expressions LISP. Celles-ci peuvent être décrites graphiquement comme des arbres étiquetés dont les branches sont ordonnées. Comme les codes cellulaires ont exactement la même structure que les S-expressions LISP, (en effet, un code cellulaire est un arbre grammaire, qui est un arbre étiqueté dont les branches sont ordonnées) la même approche que celle que Koza utilise pour les S-expressions LISP est utilisée pour générer et recombiner des codes cellulaires. L'ensemble des allèles est l'ensemble des opérateurs cellulaires utilisés. L'espace de recherche est l'espace de tous les arbres étiquetés possibles. Pendant la recombinaison de deux arbres, un sous-arbre est coupé de l'un des arbres parents et va remplacer un sous-arbre de l'autre arbre parent. Le résultat est un arbre fils. Les sous-arbres qui sont échangés sont sélectionnés aléatoirement. La figure 5.1 donne un exemple de recombinaison.

Nous avons utilisé la programmation génétique, avec toutefois quelques différences par rapport

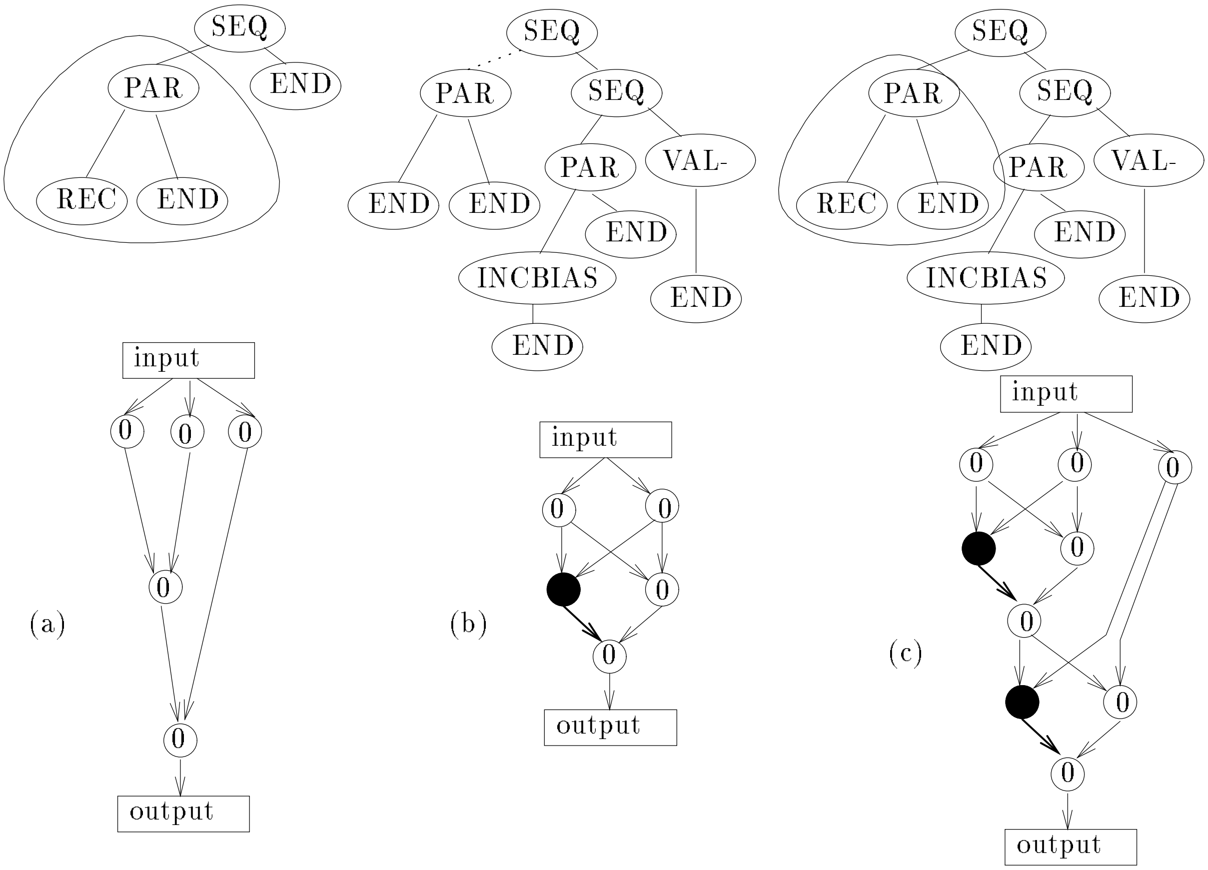

Figure 5.1: Une illustration de recombinaison entre deux arbres. (a) le père (b) la mère (c) le fils. La vie de la cellule ancêtre a été initialisée à 2. L'arbre père code un réseau qui calcule le OU logique à trois entrées. L'arbre mère code un réseau qui calcule le OU exclusif (XOR) à deux entrées. La ligne pointillée indique le sous-arbre qui va être élagué et remplacé. Le réseau de neurones numéro 2 est développé à partir de l'arbre résultant du croisement entre l'arbre père et l'arbre mère. Il calcule la parité de trois entrées. L'arbre fils résultant, code la solution générale pour le problème de la parité.

à l'approche de Koza. On n'arrête pas le GA après un nombre fixé de générations, mais après qu'un certain temps T se soit écoulé. Koza n'utilise pas de mutations. On utilise un taux de mutation $t_m = 0.005$; chaque allèle a la probabilité t_m d'être muté en un autre allèle de même arité. L'expérience montre que la mutation permet d'augmenter considérablement les performances du GA. Koza considère la profondeur des arbres pour générer la population initiale. Dans notre travail, nous ne prenons en compte que le nombre de noeuds, et nous générons une population initiale d'individus ayant tous exactement le même nombre de noeuds l_0. L'arbre résultant de la recombinaison de deux autres arbres, doit avoir un nombre de noeuds inférieur à une certaine constante l_{max}. Nous avons conçu un GA séquentiel qui utilise une sélection locale et qui recherche des couples de parents en respectant la répartition géographique des individus. Les individus sont répartis sur une grille 2-D, et se recombinent avec le meilleur individu trouvé dans le voisinage. Ceci permet d'implanter l'isolation par la distance [4], de telle façon que différentes régions sur la grille puissent présenter des tendances locales d'évolution. Les individus sont détruits en utilisant une approximation de la méthode GENITOR [32]. On sélectionne aléatoirement 50 individus dans la population et on détruit le moins bon de cet échantillon, au lieu de détruire le moins bon de toute la population. La taille de la population p et le nombre de générations g ne sont pas fixés à l'avance. C'est le GA qui calcule p et g à partir d'une fonction qui délivre le nombre de générations correspondant à une taille de population donnée ($g = p/10$ dans notre simulation); et ceci à partir du temps total T alloué. Au lieu de remplacer une population par une autre population, le GA renouvelle la population, individu par individu. La "fitness" est une valeur réelle qui mesure la qualité d'un chromosome. Le GA passe la majeure partie de son temps à calculer les "fitness". Le GA évalue le temps moyen mis pour calculer la "fitness" d'un chromosome. Il calcule ensuite le nombre de générations qui pourraient encore avoir lieu si la taille de la population était inchangée. Si ce nombre est plus grand que $p/10$, le GA crée un nouvel individu, sinon il détruit un individu. En utilisant cette stratégie, lorsque le temps de calcul alloué est écoulé, la taille finale p de la population et le nombre de générations g effectuées vérifient $g = p/10$. A chaque fois qu'un nouvel individu est créé et évalué avec un temps t, le temps moyen d'évaluation est mis à jour en utilisant $t_{moy} = (p * t_{moy} + t)/(p+1)$. Comme le temps moyen d'évaluation augmente au fur et à mesure des générations, le nombre d'individus de la population décroit. Ainsi, la taille de la population peut être dynamiquement adaptée au changement de temps requis pour évaluer la "fitness". Dans les premières générations, les chromosomes sont rapidement évalués et un grand nombre de chromosomes peuvent être testés. Dans les générations de la fin, l'évaluation peut devenir comme nous allons le voir, très coûteuse, et seuls quelque chromosomes restent en compétition.

5.2 La "fitness"

Un point très important concernant les réseaux de neurones génétiques est la fonction de "fitness" utilisée pour évaluer les réseaux de neurones. Ce paragraphe décrit la "fitness" que l'on utilise. On se restreint pour le moment aux réseaux de neurones booléens, ayant des poids ± 1 et des seuils entiers. Soit f_{tar} la fonction que le réseau de neurones booléen doit calculer, et f_{nn} la fonction qu'il calcule. Le GA produit des chromosomes en forme d'arbre qui développent toujours un phénotype valide, c'est à dire, un graphe connexe de neurones. Ensuite les unités d'entrée et de sortie doivent être déterminées. La figure 2.1 montre le graphe initial de cellules. Les cellules pointeur entrée et pointeur sortie définissent l'ensemble des cellules d'entrée et de sortie du réseau. A la fin du développement, la liste ordonnée des liens de sortie de la cellule pointeur entrée définit un ordre sur les unités d'entrée. De façon similaire, la liste ordonnée des liens d'entrée de la cellule pointeur sortie définit un ordre sur les unités de sortie. Donc le nombre d'unités d'entrée et de sortie du réseau est aussi codé par le chromosome. L'avantage de coder ces deux nombres est que le code obtenu est intrinséquement cohérent; il peut être développé sans connaître la taille de l'espace d'entrée ou de sortie de la fonction cible. Grâce à cette cohérence intrinsèque, le code peut être utilisé comme un bloc de base et être concaténé dans un autre code, de façon à inclure le réseau de neurones correspondant. Si le nombre d'unités d'entrée ou de sortie trouvé par le GA n'est pas le bon, La fonction f_{nn} calculée par le GA est définie comme suit: Si il y a trop d'unités visibles, détruire les unités surnuméraires dans l'ordre décroissant. S'il n'y en a pas assez, la partie basse du vecteur de bits d'entrée est appliquée sur les unités d'entrée du réseau, les composantes du vecteur de sortie qui ne sont pas définies prendront la valeur 0.

Puis, on mesure à quelle distance f_{nn} se trouve de f_{tar}. Au lieu d'utiliser l'erreur quadratique moyenne (approche classique), on utilise une distance basée sur la théorie de l'information, pour la raison suivante: nous pensons que l'information varie de façon plus continue, relativement à la topologie définie sur les arbres étiquetés, en convenant que les voisins d'un arbre sont obtenus par l'ajout, la modification ou la suppression d'un noeud. Un exemple plus concret nous permettra de mieux comprendre le genre de continuité apportée par la théorie de l'information: Considérons un arbre t qui développe un réseau de neurones calculant l'opposé exact de f_{tar}. Supposons que l'espace de sortie est {0,1}. Pour obtenir un réseau correct, il suffit d'ajouter un seul neurone à la sortie qui effectue un NON logique. En ajoutant un ou deux noeuds à l'arbre t, il est possible de construire un arbre t' qui ajoute ce neurone. Relativement à la topologie sur les arbres, t et t' sont très proches. Une "fitness" basée sur l'erreur quadratique moyenne donnerait la note la plus basse à t et la note la plus haute à t'. Bien qu'il y ait beaucoup d'informations utiles contenues dans t, t n'aurait pas la possibilité de se reproduire, et cette information utile serait perdue. Au contraire une "fitness" basée sur l'information donnerait la même note à t et à t'.

Voici maintenant une définition résumée de l'information et de l'information mutuelle. Soit un ensemble fini E muni d'une densité de probabilité d et deux variables aléatoires X et Y à valeurs dans E, ayant une densité pd et qd. L'information de X est:

$$I(X) = \int_E \varphi(p(x))d(x); \varphi(u) = u * \log_2(u)$$

L'information mutuelle entre X et Y est:

$$H(X,Y) = \int_{E\times E} \varphi(\frac{p \otimes q(x,y)}{p(x)q(y)})p(x)d(x)q(y)d(y)$$

Les inégalités suivantes sont toujours vraies:

$$0 \leq H(X,Y) \leq I(Y); H(X,Y) = I(Y) \iff \exists g : E \to E; p \otimes q(x,y) \neq 0 \Rightarrow y = g(x)$$

Nous allons montrer maintenant comment utiliser la théorie de l'information pour définir la "fitness". Tout d'abord, supposons que l'espace de sortie est {0,1}. Soit un vecteur aléatoire V dans l'espace d'entrée, ayant une densité de probabilité uniforme. La mesure de la "fitness" est définie par:

$$\gamma(f_{nn}, f_{tar}) = \frac{H(f_{nn}(V), f_{tar}(V))}{I(f_{tar}(V))}$$

La quantité $H(f_{nn}(V), f_{tar}(V))$ peut être considérée comme l'information apportée par f_{nn} à f_{tar}. Nous la divisons par $I(f_{tar}(V))$ pour obtenir une "fitness" normalisée entre 0 et 1. Une analyse mathématique montre que quand la "fitness" atteint la valeur 1, le réseau de neurones calcule la bonne fonction (sinon il calcule le complément à 1).

Si maintenant l'espace de sortie est $\{0,1\}^p$, on applique l'approche précédente à chacune des p composantes, et on fait la moyenne des p "fitness" résultantes.

Finalement, considérons une famille de fonctions paramétrées que l'on suppose partager le même type d'algorithmes. Soit f_{tar}^L un élément de cette famille, où L indexe la taille du problème à résoudre et est stocké dans le registre `vie` de la cellule ancêtre. Pour chaque L, $L = 1, \ldots, L_{max}$, un réseau de neurones N_L est développé. Ce réseau N_L calcule une approximation f_{nn}^L de la cible f_{tar}^L. La "fitness" totale est:

$$\Gamma(L_{max}) = \sum_{L=1}^{L_{max}} \gamma(f_{nn}^L, f_{tar}^L) \tag{5.1}$$

L'expérience montre que les individus générés pendant les premières générations font beaucoup d'erreurs. Le réseau de neurones N_1 a une "fitness" basse, et les réseaux N_L pour $L > 1$ ont une "fitness" nulle. Il n'est donc pas nécessaire de développer les réseaux N_L pour $L > 1$ durant les premières générations. La règle suivante détermine exactement quand il devient intéressant de développer le réseau N_2 puis N_3 etc.. jusqu'à $N_{L_{max}}$

> **Règle de développement:** Si N_1, N_2, ..., N_L ont déjà été développés, continuer à développer N_{L+1} si la "fitness" de N_L est plus grande qu'un certain seuil r. Si $L > L_{max}$, stopper le développement.

Le seuil r est un paramètre à ajuster en fonction du problème particulier que l'on cherche à résoudre. Avec cette règle, dans les premières générations, seulement un seul réseau est développé. Tandisqu'à la dernière génération, jusqu'à L_{max} réseaux sont développés. Ainsi on économise beaucoup de temps machine. En général la taille du réseau N_L est aussi plus grande que L fois la taille du réseau N_1. Donc l'évaluation de la "fitness" des réseaux lorsque L réseaux sont développés prend au minimum $L*(L+1)/2$ fois plus de temps. Si on va jusqu'à $L = L_{max} = 50$ comme ce sera le cas, le rapport est de 2500.

La règle de développement présente un autre intérêt: Chaque terme de la "fitness" est compris entre 0 et 1. Donc la "fitness" elle même est comprise entre 0 et L_{max}. Avec la règle de développement, seulement les premiers termes des L_{max} termes sont additionnés. Si le seuil r est proche de 1, la règle de développement dit que l'algorithme continue à développer le réseau suivant seulement si le réseau courant calcule la fonction exacte. Finalement, si la "fitness" est comprise entre L_0 et $L_0 + 1$, cela signifie que les L_0 premiers réseaux de neurones ont été développés et calculent la bonne fonction, et le $L_0 + 1^{ieme}$ n'est pas encore totalement correct. Ainsi la "fitness" peut mesurer le niveau de résolution d'un problème.

5.3 Simulation avec la parité

Le tableau 5.1 résume les caractéristiques du problème de l'évolution d'une famille de réseaux de neurones, calculant la parité d'un nombre d'entrées arbitrairement grand. Le GA fait évoluer des chromosomes qui ont une structure d'arbre. On reprend dans ce tableau, la terminologie utilisée par Koza. Les allèles sont les symboles etiquetant les arbres, les allèles terminaux étiquettent les feuilles des arbres, les allèles de fonctions étiquettent les noeuds internes de l'arbre. Les points de fitness sont les points où l'on évalue la fonction de fitness. La "fitness" brute, donne une mesure de la qualité du réseau. La "fitness" normalisée est calculée à partir de la fitness brute, elle permet de réajuster la fitness brute pour obtenir une fitness comprise dans un intervalle connu. Les paramètres sont les valeurs principales qui déterminent la nature du GA particulier utilisé pour résoudre le problème. Le prédicat de succès indique quand la recherche menée par le GA doit être arrêtée.

Le GA essaye de trouver un chromosome qui développe une famille de réseaux de neurones N_L telle que N_L calcule la parité de $L + 1$ entrées. La parité de L entrées binaires est la somme des entrées modulo 2. Le seuil utilisé dans la règle de développement a été choisi $r = 0.7$. On a développé les $L_{max} = 50$ premiers réseaux de neurones. Ainsi, les solutions ont été testées pour le problème de la parité jusqu'à 51 entrées. Cependant, le GA génére des solutions pour la parité d'un nombre arbitrairement grand de bits, parce que c'est la récurrence qui est "apprise" par le GA pour calculer la parité de $n + 1$ bits à partir de celle de n bits. Les opérateurs `PAR SEQ WAIT VAL- REC` ont été décrits au chapitre 2. Comme on n'utilise pas d'opérateurs modifiant le registre de lien, celui-ci reste sur sa valeur initiale qui est 1, il indique toujours le lien le plus à gauche. Dans ces conditions, l'opérateur `VAL-` a pour effet de mettre le poids du lien le plus à gauche à -1. Les opérateurs `SETBIAS 0` et `SETBIAS 1` positionnent le seuil du neurone à 0 et à 1. On a utilisé pour cette première expérience un jeu d'opérateurs très réduit pour simplifier le problème.

Objectif:	générer une famille de réseaux de neurones qui calculent la parité de taille arbitraire
Allèles terminaux:	`SETBIAS 0 SETBIAS 1`
Alèles de fonctions:	`PAR SEQ WAIT VAL- REC`
point de Fitness :	vecteurs aléatoires de $L+1$ bit à rentrer dans le réseau N_L. Le nombre de vecteurs est déterminé par le GA
Fitness brute d'un réseau	L'information mutuelle entre la fonction calculée par le réseau et la fonction cible
Fitness normalisée d'un réseau	Fitness brute divisée par l'information de la fonction cible; elle est comprise entre 0 et 1
Fitness normalisée	La somme des "fitness" normalisées des réseaux développés; elle est comprise entre 0 et 50.
Paramètres:	$T = 1800s, t_m = 0.005, l_0 = 20, l_{max} = 30$
Prédicat de succès:	La "fitness" normalisée vaut 50, c'est à dire les 50 premiers réseaux sont développés et leur "fitness" est maximum,

Tableau 5.1: Tableau décrivant la recherche génétique d'une famille de réseaux de neurones pour résoudre le problème de la parité jusqu'à 51 entrées.

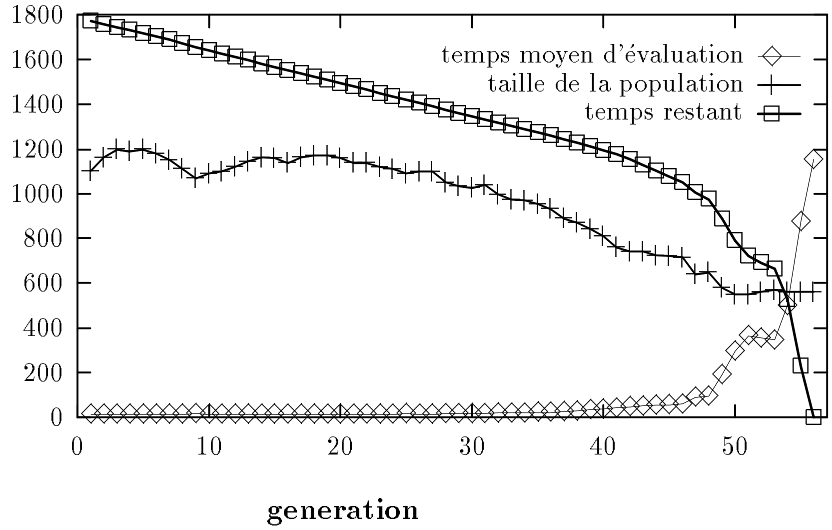

Figure 5.2: Courbe du temps moyen d'évaluation de la "fitness" (en milli secondes) de la taille de la population, et du temps machine qui reste (en secondes), en fonction du numéro de la génération.

Comme nous l'avons déjà signalé, la taille de la population et le nombre de générations ne sont pas spécifiés. Au lieu de cela, une variable T spécifie le temps alloué au GA. Le GA déduit p et g à partir d'une fonction qui donne le nombre de générations en fonction de la taille de la population. ($g = p/10$ dans cette simulation) et du temps alloué T. Le GA calcule p et g en évaluant le temps moyen mis pour calculer la "fitness" d'un chromosome. Ce temps moyen change au cours des générations. Le GA doit mettre à jour en permanence l'estimation de p et g. Comme le temps moyen de calcul de la "fitness" augmente, le nombre d'individus décroit (de 1100 à 550 dans la figure 5.2). Ainsi la taille de la population peut dynamiquement s'adapter à un changement du coût en temps du calcul de la "fitness". Dans les premières générations, les chromosomes sont évalués rapidement, et un grand nombre de solutions peuvent être explorées, de façon à sélectionner beaucoup de pics différents dans le paysage de la "fitness". Dans les dernières générations, l'évaluation devient très coûteuse (jusqu'à 2500 fois plus longue comme on l'a déjà remarqué), et seulement quelques individus restent en compétition sur chaque pic sélectionné.

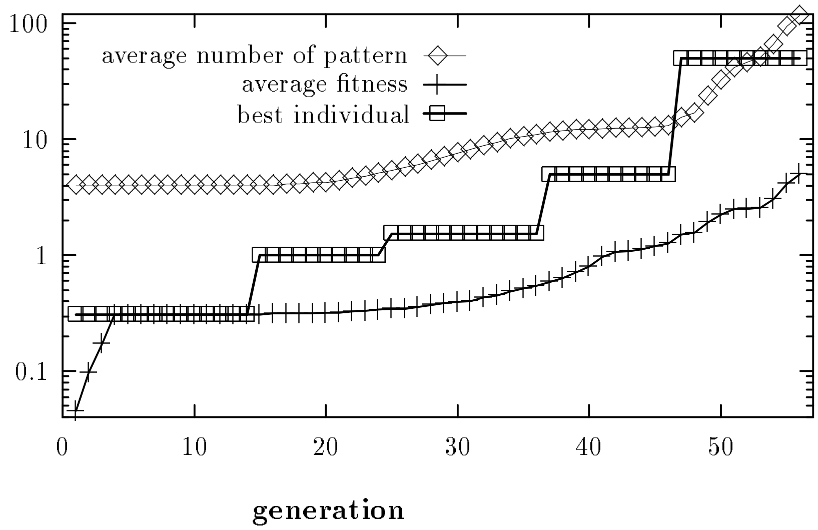

Figure 5.3: Courbe du nombre moyen de vecteurs utilisés pour évaluer la "fitness", de la "fitness" moyenne, et de la meilleure "fitness" de la population en fonction du numéro de la génération.

Le nombre de vecteurs de bits utilisés pour évaluer la "fitness" avec une précision suffisante n'a pas à être spécifié non plus par l'utilisateur. Le GA détermine ce nombre en comparant la précision de la "fitness" avec la variance de la population. La figure 5.3 montre que lorsque les individus s'améliorent, le nombre de vecteurs utilisés pour évaluer leur "fitness" croît.

Comme le GA produit des réseaux de neurones de grande taille, on a du introduire un autre paramètre pour gagner du temps. Si le rapport entre le nombre de cellules du graphe de cellules et le nombre d'unités visibles dépasse 15, le développement est stoppé et la "fitness" du réseau est mise à zéro. Sur 20 expérience de 1800 secondes, sur une station de travail de la catégorie sun 4, le GA trouve toujours une solution au problème de la parité de taille 50 (à 51 entrées). Sur la figure 5.3, l'axe vertical correspond à la "fitness" des chromosomes. Cette "fitness" peut s'interpréter pour connaître le degré d'avancement de la résolution du problème. A la génération 15, la fitness atteint 1, le GA résoud la parité de taille 1 qui est le XOR. A la génération 37, la fitness atteint 5, la parité jusqu'à la taille 5 est résolue, une analyse de la solution montre qu'à ce stade, le meilleur chromosome génére une famille de réseaux de neurones N_L telle que N_L calcule correctement la parité de $L+1$ bits. Mais le nombre de neurones de N_L est proportionel à 2^L, alors que le nombre d'unités visibles est proportionnel à L. Le rapport entre ces deux nombres tend vers l'infini. Il dépasse le seuil de tolérance pour $L = 6$ de telle sorte que les réseaux de neurones N_L pour $L > 5$ ont tous une "fitness" nulle. C'est la raison pour laquelle la "fitness" est inférieure à 50. A la génération 47, le GA produit une famille de réseaux N_L qui ont seulement $O(L)$ neurones et qui résolvent le problème de la parité. Le GA a appris la récurrence. Il pourrait développer un réseau de neurones pour calculer la parité d'un nombre arbitrairement grand d'entrées.

La figure 5.4 montre le chromosome construit par le GA. Le réseau de neurones numéro L correspondant, calcule la parité de $L + 1$ entrées avec $5L - 1$ unités d'entrée et $10L$ connexions.

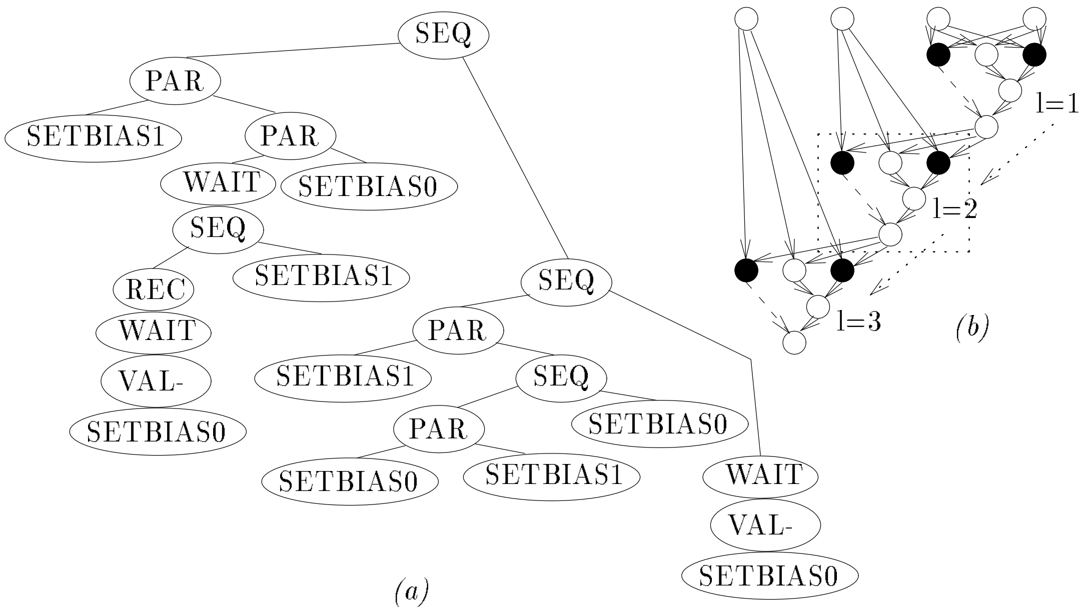

Figure 5.4: Solution du problème de la parité: (a) le chromosome trouvé par le GA. (b) Le réseau de neurones pour $L = 3$ Le rectangle en pointillés indique un bloc de base neuronal qui est recopié plusieurs fois. — poids +1 • seuil +1 - - poids -1 ∘ seuil 0

C'est clairement la combinaison de L blocs de base neuronaux. Chacun de ces blocs de base neuronaux calcule séparement un OU exclusif logique (XOR). L unités cachées peuvent être enlevées sans dommages, malgré le faible nombre de connexions. En effet, la complexité en nombre de connexions ordinairement quadratique, est ici seulement linéaire. La solution reste tolérante aux pannes, contrairement à l'intuition que le réseau ne fonctionne plus dès la suppression du moindre neurone. Ces réseaux sont en effet comme de la dentelle, il sont associés à un programme et il est vrai que la moindre modification d'un programme perturbe, le plus souvent gravement, son fonctionnement.

Nous avons conçu un logiciel qui permet de visualiser le développement du réseau de cellules ainsi que le réseau de neurones final. La figure 5.9 montre une image générée par ce logiciel. Elle représente le réseau de neurones pour $L = 20$ qui résoud la parité de 21 entrées. Les lignes tirets représentent des poids -1 et les lignes continues des poids 1. Un disque noir indique un seuil de 1, un cercle indique un seuil de 0. La solution brute comporte 41 unités d'entrée. Elle produit le réseau de neurones correct modulo l'élimination des 20 derniéres unités d'entrée surnuméraires, suivant le procédé décrit à la section 5.2.

5.4 Simulation avec la symétrie

Une analyse des solutions trouvées par le GA pour le problème de la parité montre que ce problème est particulièrement facile à résoudre comparativement à d'autres problèmes booléens. Tout d'abord il est possible de coder un réseau de neurones calculant la parité en utilisant un arbre

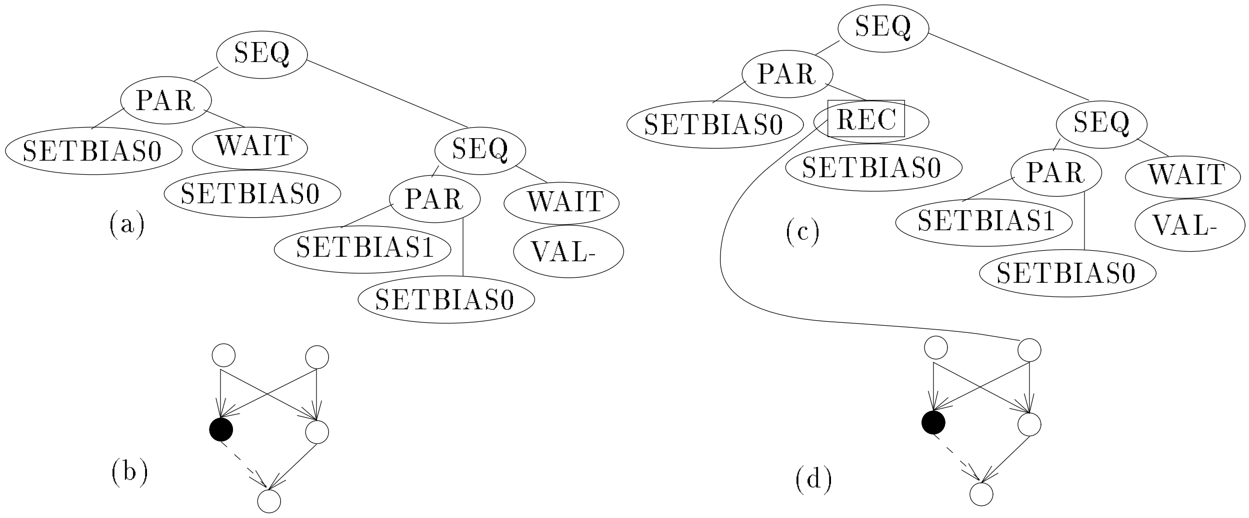

Figure 5.5: (a) Code cellulaire d'un réseau calculant le XOR (b) Le réseau de neurones correspondant (c) Un code cellulaire d'un réseau calculant la parité (d) Le réseau de neurones correspondant avec le développement gelé sur le programme de récursion. — poids +1 • seuil +1 - - poids -1 ∘ seuil 0

de seulement 11 noeuds. La deuxième raison est moins évidente. A la figure 5.5 (a) on montre un code qui développe le réseau de neurones (b). Ce réseau calcule le XOR. Le XOR est aussi la parité de deux entrées, c'est à dire, le problème de la parité de taille 1 que le GA essaye de résoudre dans les expériences précédentes. Ainsi le code cellulaire (a) développe le réseau de neurones N_1 de la famille N_L correctement. Il a donc une "fitness" au moins égale à 1. La figure 5.5 (d) développe le réseau de cellules codé par (c) avec les conventions suivantes: quand une cellule lit l'opérateur cellulaire de saut récurrent `REC`, elle géle son développement. Geler signifie qu'elle garde sa tête de lecture sur le noeud étiqueté par `REC`, et ne se réécrit plus. Le réseau de cellules (d) est presque le même que (b), sauf qu'une des cellules d'entrée est gelée. Si on la dégéle, elle donnera naissance à un sous réseau qui calculera la parité d'ordre inférieur. Le réseau de cellules (d) explicite le processus de récurrence. Il exprime la règle de récurrence suivante: pour calculer la parité de $L+1$ bits, on applique un XOR sur le premier bit et sur la parité des L bits suivants. Les chromosomes (a) et (c) ne différent que pour une seule position qui est `WAIT` dans (a) et `REC` dans (c). Comme le code cellulaire pour le problème de taille 1, et le code cellulaire pour le problème de taille arbitraire sont presque les mêmes, cela facilite la recherche du GA. Le GA commence par trouver un réseau de neurones qui ressemble à (b) pour résoudre le problème de taille 1, puis à l'aide d'une mutation correctement placée, le GA introduit un opérateur `REC` qui permet d'aller de (a) à (c) et d'achever la récurrence. On peut exprimer cela d'une autre manière: un code cellulaire développe un réseau de neurones en utilisant une récurrence. Cela met en jeu la résolution de deux problèmes, le problème de taille 1, et le problème de la récurrence qui consiste à trouver un moyen de combiner un réseau solution du problème de taille L avec quelque neurones supplémentaires pour construire un réseau solution du problème de taille $L+1$. Dans le cas de la parité, le problème de taille 1 et le problème de la récurrence sont presque identiques, car tout les deux utilisent un réseau de neurones XOR.

Pour résoudre d'autres problèmes booléens qui n'ont pas ce trait particulier, on a du augmenter le codage cellulaire de base. On a introduit un opérateur binaire de branchement noté `BLIFE`. Une cellule qui lit BLIFE exécute l'algorithme suivant:

```
1-Si (vie >1) tete-lecture := sous-arbre gauche du noeud courant
2-Sinon  tete-lecture := sous-arbre droit du noeud courant
```

Cet opérateur rend compte du fait que les calculs correspondant au problème de taille 1 peuvent n'avoir rien en commun avec ceux correspondant à la récurrence.

Objectif:	Trouver le CE d'une famille de réseaux de neurones qui calculent la symétrie de taille arbitraire
Allèles de terminaux:	END
Allèles de fonctions:	PAR SEQ WAIT VAL+ VAL- INCLR DECLR INCBIAS DECBIAS CUT REC BLIFE
Points de Fitness :	Le nombre de vecteurs est déterminé par le GA. Pour chaque réseau de neurones, Si le réseau a moins de 6 unités d'entrée, tous les vecteurs d'entrée sont traités Sinon un nombre égal de vecteurs aléatoires symétriques et non symétriques sont traités
Fitness brute d'un réseau	L'information mutuelle entre la fonction calculée par le réseau et la fonction cible
Fitness normalisée d'un réseau	Fitness brute divisée par l'information de la fonction cible elle est comprise entre 0 et 1
Fitness normalisée	La somme des "fitness" normalisées des réseaux développés, elle est comprise entre 0 et 20.
Paramètres:	$T = 10800s, t_m = 0.005, l_0 = 30, l_{max} = 50$
Prédicat de succès:	La "fitness" normalisée vaut 20, c'est à dire les 20 premiers réseaux sont développés et leur "fitness" est maximum,

Tableau 5.2: Tableau décrivant la recherche génétique d'une famille de réseaux de neurones, pour résoudre le problème de la symétrie jusqu'à 40 entrées.

Le tableau 5.2 résume les caractéristiques principales du problème de l'évolution d'un code cellulaire (CE) qui génére une famille de réseaux de neurones, pour calculer la symétrie d'un vecteur de bits de taille arbitrairement grande. Le GA essaye de trouver le code d'une famille de réseaux (N_L) où N_L calcule la symétrie de $2 * L$ entrées. Si l'entrée est symétrique, le réseau de neurones doit calculer 0, sinon il doit calculer 1. Les solutions sont testées pour la symétrie allant jusqu'à 40 entrées. Les opérateurs END PAR SEQ WAIT VAL+ VAL- INCLR DECLR INCBIAS DECBIAS CUT ont été décrits au chapitre 2.

Après des echecs répétés, on a remarqué que la plupart du temps, les réseaux de neurones bruts produits par le GA n'avaient pas le nombre correct d'entrées ou de sorties. Ils étaient évalués modulo la convention décrite dans la section 5.2. Nous avons ajouté un terme à la "fitness" qui récompense les réseaux de neurones ayant le nombre correct d'unités visibles. En faisant ainsi, on espére que deux sortes d'individus vont coexister: les individus qui développent des réseaux calculant la bonne fonction mais dont le nombre d'unités visibles n'est pas correct, les individus qui développent des architectures ayant un nombre d'unités visibles correct mais ne calculant pas la bonne fonction. La recombinaison entre ces deux sortes d'individus produit des individus ayant les deux caractéristiques correctes. En fait, cette modification de la "fitness" a permis enfin d'obtenir le succés, là où auparavant le GA n'arrivait pas à générer de solution pour le problème de la symétrie. Plus précisément, la nouvelle "fitness" est la somme pondérée de l'ancienne plus un terme t. Si le nombre d'entrées (respectivement de sorties) est correct, on ajoute 0.5 à t. Au cours de la simulation on a utilisé:
nouvelle "fitness" = ancienne "fitness"$*0.85 + t * 0.15$.
La nouvelle "fitness" est toujours située entre 0 et 1.

Le problème qui consiste à générer une famille de réseaux pour la symétrie, est beaucoup plus difficile que dans le cas de la parité, parce que la récurrence nécessite un calcul sans rapport avec le calcul correspondant au problème de taille 1. Le problème de taille 1 est un XOR à deux entrées.

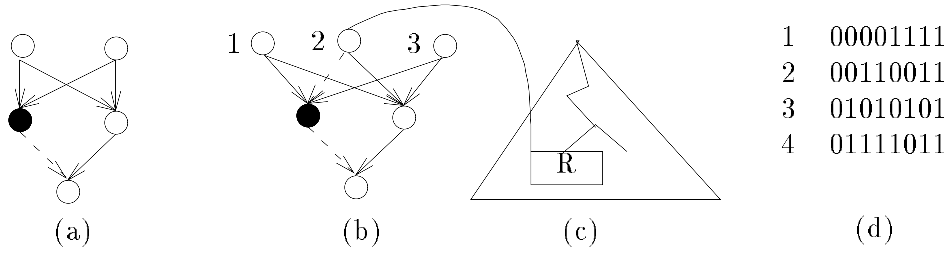

Figure 5.6: La fonction booléenne symétrie (a) Un réseau de neurones pour résoudre le problème de taille 1. (b) Le graphe de cellules pour le problème de la récurrence, (c) Allure générale de l'arbre grammaire,

Pour faire une récurrence calulant la symétrie de $n+2$ bits à partir de la symétrie de n bits, on peut par exemple utiliser le réseau montré à la figure 5.6 (b), où le développement de la cellule de lecture que lit `REC` est gelé. Le réseau montré sur la figure 5.6 (b) exprime la récurrence suivante: Un vecteur de n bits n'est pas symétrique dans deux cas: ou bien les $n-2$ neurones du milieu ne sont pas symétriques, ou bien la première et la dernière entrée sont différentes. Dans la figure 5.6, le graphe de cellules (a) est nettement différent du graphe de cellule (b), parce que le problème de taille 1 est différent du problème de la récurrence. Le GA doit donc résoudre deux problèmes différents au lieu d'un seul. N'oublions pas que pendant les premières générations, le réseau de neurones N_1 est le seul à être développé. Donc la "fitness" prend en compte uniquement le problème de taille 1. Lorsque les codes générés par le GA deviennent assez bons, le problème de taille 1 est résolu, et les réseaux N_L pour $L > 1$ sont développés. La "fitness" prend alors en compte le problème de la récurrence. La forme de la fonction de "fitness" change complétement au cours du temps. On peut dire en termes techniques, que le GA doit résoudre un problème où l'environnement est changeant. Ce genre de problème est particuliérement difficile, car la population peut converger prématurément vers une solution adaptée au premier environnement. Pour cette raison, on a échangé la station de travail sun 4 pour une machine parallèle plus puissante. Le nom de la machine est: iPSC860. Nous avons utilisé 4 noeuds de cette machine. Un noeud de cette machine a la puissance d'approximativement 1,5 sun 4. Le même GA est exécuté sur chaque noeud. Régulièrement, les GA s'échangent leurs meilleurs individus. Le modèle de GA parallèle est décrit au chapitre 8. Il permet de lutter contre la convergence prématurée.

Comme on l'a déjà dit le GA déduit la taille de la population p, et le nombre de générations g, à partir d'une fonction qui fournit le nombre de générations optima, en fonction de la taille de la population. Nous avons choisi $g = p/50$ dans cette série d'expériences. Sur 9 essais de 3 heures chacuns, le GA a trouvé une solution dans 6 cas pour la symétrie de taille 20 (40 entrées). Nous décrivons maintenant un essai réussi. La taille de la population décroit de 6000 à 1400. Le nombre moyen de points où les réseaux sont évalués croît de 4 à 1800. Sur la figure 5.7, l'axe vertical correspond à l'information apprise sur le problème de la symétrie. A la génération 5, le GA résoud la symétrie de taille 1 (le XOR), à la génération 17, il résoud la symétrie de taille 2, en faisant ce qui pourrait s'appeler une approximation de la récurrence. A la génération 20, le GA produit une famille (N_L) de réseaux qui ont seulement $O(L)$ unités, et qui résoud le problème de la parité. Le GA a appris la récurrence.

Le chromosome trouvé par le GA est: `VAL-(SEQ (SEQ (PAR(END )(WAIT (BLIFE(PAR(REC(END`

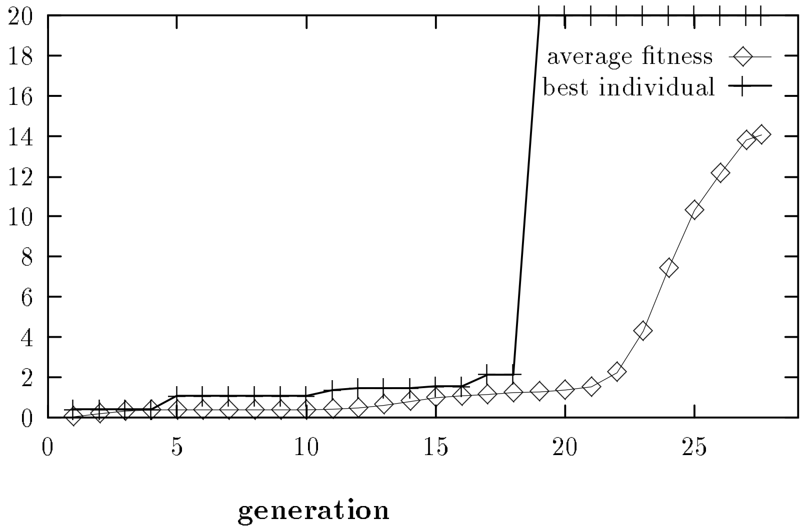

Figure 5.7: Courbe de la "fitness" moyenne (trait fin), et de la meilleure "fitness"(trait épais), en fonction du numéro de génération

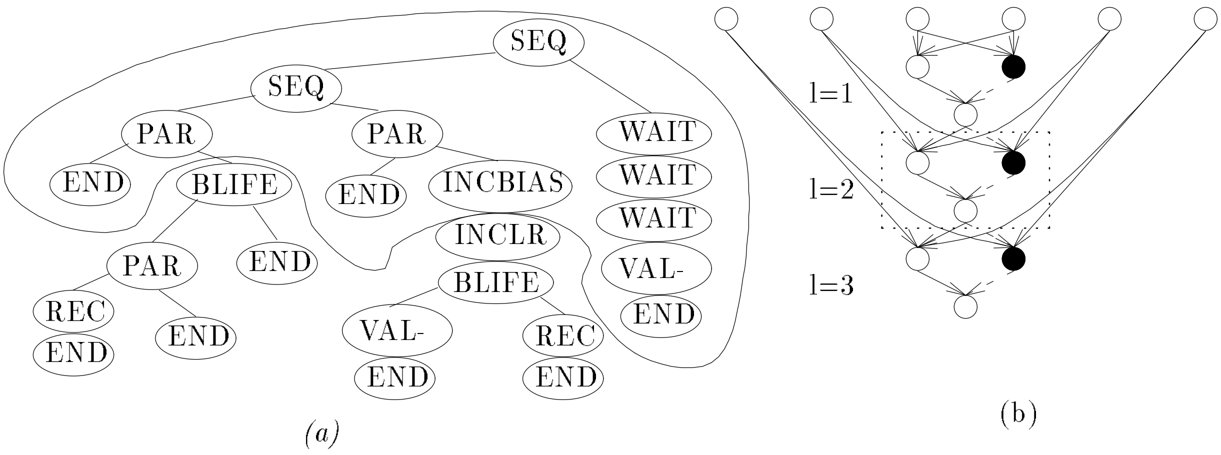

Figure 5.8: Résolution de la symétrie: (a) Le chromosome trouvé par le GA (b) Le réseau de neurones pour $L = 3$ Le rectangle en pointillé indique le bloc de base neuronal qui est recopié plusieurs fois.

`))( VAL+(VAL+(WAIT(VAL+(END )) )) ))( DECLR (BLIFE( INCLR( VAL+( WAIT (VAL− (END )) )) )( DECLR (INCLR (END ))))))))(PAR( INCLR ( DECLR ( END ))) (INCBIAS( INCLR (BLIFE( VAL−(END ))(REC ( BLIFE( END )(VAL+ (WAIT( BLIFE (END )(END ))))))))))(DECLR( VAL + (INCLR (−(END ))))))` Dans ce chromosome, beaucoup de gênes sont inutiles. En gardant seulement les gênes qui modifient le développement du graphe de cellules, il est possible de le simplifier. La figure 5.8 montre le chromosome simplifié et le réseau de neurones qui résulte de son décodage pour $L = 3$. On utilise les mêmes conventions pour dessiner le réseau, que pour la figure 5.4. Le chromosome simplifié possède 24 gènes, il est plus court que la solution obtenue à la main! Pour concevoir une solution, la procédure la plus simple est de mettre un opérateur `BLIFE` à la racine, de coder un réseau XOR sur la branche de droite, et un réseau de cellules comme celui représenté figure 5.6 (b) pour calculer la récurrence, sur la branche de gauche. Cela donne le chromosome de 29 bits qui suit: `BLIFE (SEQ (PAR (PAR( END ) (REC(END )) ) (END ))(SEQ (PAR ( INCLR (INCBIAS (VAL- (END )) )) (END ))(WAIT (VAL-(END ) )) ))(SEQ (PAR (END )(END ))( SEQ (PAR ( INCBIAS ) (END ))(WAIT (VAL- (END )) ))` Le GA a pu résumer l'information mieux que nous, en utilisant l'astuce suivante: il a utilisé deux fois l'opérateur `BLIFE` pour générer une modification aux deux endroits précis où différent le réseau XOR et le graphe de cellules calculant la récurrence. Il a produit un morceau de code de 11 gênes commun à la fois au réseau XOR et au graphe de cellule calculant la récurrence. Il a utilisé un opérateur `BLIFE` deux fois pour générer une modification exactement aux deux endroits où le réseau XOR et le graphe de cellules calculant la récurrence, différent. Le réseau de neurones correspondant de numéro L calcule la symétrie de $2 * L$ entrées avec $3 * l - 1$ unités cachées, et $8 * L - 2$ connexions. On pense que cette solution est optimale. Elle est clairement structurée comme une combinaison de L copies du même bloc de base neuronal. La figure 5.10 représente le réseau de neurones pour $L = 20$, qui résoud le problème de la symétrie de 40 entrées.

5.5 Conclusion

Ce chapitre montre que l'algorithme génétique brut, combiné avec le codage cellulaire, a permis de décomposer automatiquement et dynamiquement un problème en sous problèmes, en utilisant une sorte de stratégie "diviser pour régner"; le premier sous problème correspond à l'initialisation d'une récurrence, et le deuxième correspond au calcul de la récurrence. Le GA a pu générer un réseau de neurones ou de cellules pour chacun des sous problèmes, puis a pu recombiner plusieurs copies du réseau de neurones pour construire une architecture résolvant le problème dans son ensemble. Il est de manière évidente, impossible de générer un réseau de neurones pour la parité de 50 entrées, en utilisant un algorithme d'apprentissage basé sur une technique de gradient telle que la rétro-propagation, parce que la complexité en temps d'un tel apprentissage croît trop vite avec la taille du problème; comme le remarque Mühlenbein dans son article [24]. Ainsi cet exemple montre clairement la supériorité de la technologie GA+CE, dans le cas particulier des fonctions booléennes présentant un certain degré de régularité.

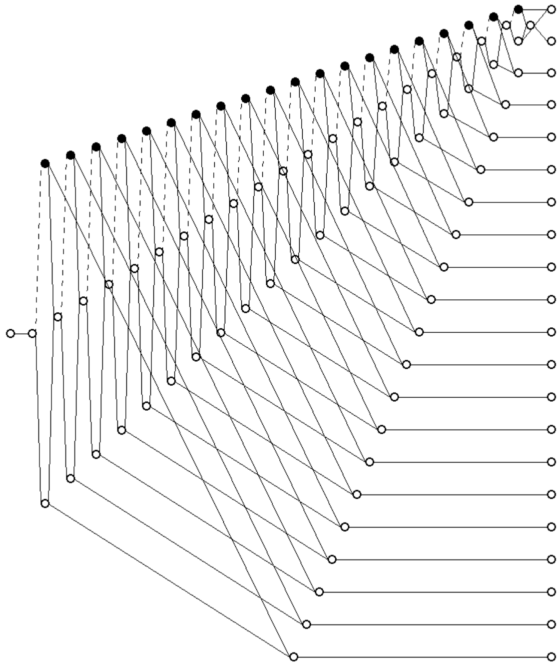

Figure 5.9: Un réseau de neurones pour la parité de 21 entrées.

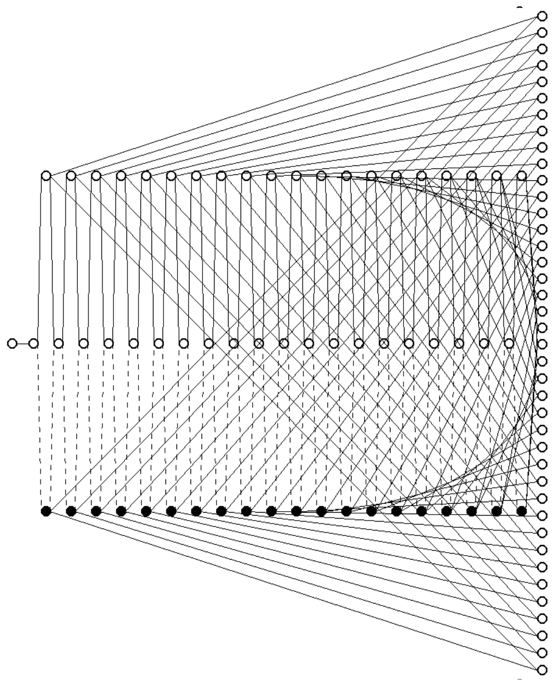

Figure 5.10: Un réseau de neurones pour la symétrie de 40 entrées.

Chapitre 6

Interaction entre apprentissage et développement

6.1 Introduction

Nous avons vu au chapitre précédent comment générer des réseaux de neurones avec le GA brut, sans apprentissage. Le principal inconvénient de l'utilisation d'un algorithme d'apprentissage basé sur une méthode de gradient, est le coût élevé de chaque évaluation. Pour effectuer une rétro-propagation (ou une autre forme de descente de gradient plus rapide) dans l'évaluation de chaque réseau, le nombre d'évaluations requis pour trouver des architectures de réseaux optimisées, devient rapidement prohibitif en temps de calcul. La quantité de calculs est en fait si énorme que le GA ne peut être utilisé, sauf pour optimiser de petites topologies. Par exemple, si l'évaluation d'une architecture de réseaux de neurones de taille modeste, pour résoudre un problème complexe, prend une heure de calcul (ce qui est loin d'être suffisant pour un problème difficile); il faudra une année pour faire seulement 9000 évaluations d'architectures durant la recherche génétique. Si l'architecture est réellement complexe (par exemple, si elle comporte 500 connexions), 9000 évaluations ne seront pas suffisantes pour la recherche génétique.

Le travail présenté dans ce chapitre concerne l'introduction d'apprentissage en plus du processus de l'évolution des arbres grammaires. Nous proposons des solutions différentes de l'approche usuelle qui vient d'être décrite. Le GA n'est pas utilisé pour trouver des architectures adaptées à un apprentissage. C'est l'apprentissage qui est utilisé pour accélérer le GA. On compare les accélérations obtenues avec les différentes manières de combiner un apprentissage et le GA. En particulier, les quatres modes suivants d'apprentissage sont étudiés: 1) Utiliser le GA tout seul, sans apprentissage, 2) Utiliser l'apprentissage de façon simple. L'apprentissage change la fonction de fitness qui mesure la performance brute, en faisant des modifications de poids sur des réseaux de neurones simples (méthode appelée mode d'apprentissage Baldwin, du nom de son inventeur). 3) Utiliser une forme plus complexe d'apprentissage qui change les poids des cellules de réseaux simples, puis réutilise ces changements pour le développement de réseaux de taille plus importante (méthode appelée mode d'apprentissage développemental) et 4) Utiliser une forme d'évolution Lamarckienne où le comportement appris est rétro-codé sur les chromosomes de l'individu, puis transmis à ses descendants durant la recombinaison génétique (méthode que nous appelons mode d'apprentissage Lamarckien).

On peut réaliser des tests comparatifs de l'apprentissage Lamarckien, l'apprentissage Baldwin et l'apprentissage développemental, parce qu'on utilise le codage cellulaire (CE) pour coder les réseaux de neurones. Trois qualités du CE sont utilisées: en premier lieu, le CE code à la fois l'architecture et les poids, en second lieu le CE permet de rétro-coder un changement de poids sur les arbres grammaires, en troisième lieu, le CE est récursif. Le premier point permet d'étudier l'apprentissage Baldwin. Le deuxième point, l'apprentissage Lamarckien, et le troisième point rend possible l'étude de l'apprentissage développemental.

Le paragraphe 2 détaille le rôle joué par l'apprentissage dans l'évolution, et les différentes formes possibles d'apprentissage et de combinaison entre apprentissage et évolution, qui seront utilisées dans les expériences.

6.2 Comment combiner l'apprentissage et le développement cellulaire.

Ajouter une forme d'apprentissage au processus de développement permettrait d'accélérer la recherche de réseaux de neurones fonctionnels En général, pour être efficace, le processus d'apprentissage doit être moins coûteux que celui qui consite à reproduire et tester un nouvel individu; si le coût de l'apprentissage approche ou dépasse celui de la reproduction et du test d'un nouvel individu, l'apprentissage doit augmenter les chances de trouver un comportement amélioré.

Il y a plusieurs façons d'"ajouter" de l'apprentissage au développement, et plusieurs questions doivent être considérées. En premier lieu, on travaille avec des réseaux de neurones booléens. L'apprentissage doit aussi produire des poids booléens, pour que l'information apprise puisse être rétro-codée sur le code cellulaire (chromosome) manipulé par le GA. Le rétro-codage des poids sur le code cellulaire est utile si on doit employer une forme d'évolution Lamarckienne. Par évolution Lamarckienne, on veut dire que les chromosomes produits par la recombinaison d'autres chromosomes, sont améliorés avec l'apprentissage, et ces améliorations sont rétro-codées sur le chromosome. Les chromosomes améliorés sont replacés dans la population, et ont la possibilité de se reproduire.

Une critique évidente de l'évolution Lamarckienne est qu'elle n'est pas Darwinienne: la plupart des scientifiques pensent que l'évolution ne se déroule pas de cette manière. Une autre critique estime que pour conserver les capacités du GA à faire des calculs basés sur les shémas (théorème des shémas), l'évolution Lamarckienne ne doit pas être utilisée. Considérons le cas simple où les chromosomes sont des chaînes de bits. Dans ce cas, un shéma est un hyperplan de l'espace vectoriel des chaînes de bits. Un hyperplan est dit "être représenté" dans la population du GA, si plusieurs individus de la population appartiennent à cet hyperplan. On peut s'abstraire des individus, et ne considérer que les hyperplans représentés dans la population du GA. Le GA apparaît comme un outil à manipuler des hyperplans. La solution du problème est définie comme étant contenue dans l'intersection d'un ensemble $\mathcal{E}$ hyperplans. Le théoréme des shémas prouve que le GA fait ses calculs en échantillonant des individus dans les hyperplans de $\mathcal{E}$. On dit qu'il fait de "l'échantillonage d'hyperplans". Changer le codage d'un individu, modifie l'information statistique contenue dans toute la population, et l'information statistique concernant la répartition des individus dans les hyperplans. Théoriquement, l'application d'une optimisation locale diminue la capacité du GA à rechercher en utilisant l'échantillonage d'hyperplans. Le reproche que l'on peut faire est que l'optimisation locale change l'information héritée des parents par les descendants, en privilégiant l'aspect descente de gradient sur l'aspect échantillonage d'hyperplans. Ainsi, l'exploration de l'espace de recherche est moins globale, car c'est l'échantillonage d'hyperplans qui confére à l'algorithme génétique sa capacité à mener une recherche globale. Malgré cette objection théorique, les algorithmes génétiques hybridés avec une optimisation locale, sont des méthodes d'optimisation intéressantes, pour plusieurs raisons. Tout d'abord, l'algorithme génétique hybride ne fait pas une descente de gradient ordinaire, il fait une descente de gradient simultanée en plusieurs points de l'espace. A moins que la fonction cible ne soit sévèrement multimodale, il paraît probable que certains descendants seront dans le bassin d'attraction de la solution globale. Deuxièmement une stratégie hybride altère l'échantillonage d'hyperplans, mais ne le supprime pas totalement. Par exemple, utiliser l'apprentissage ou l'optimisation pour améliorer la population initiale de chaînes de bits, influence la répartition initiale, mais n'interfére pas avec les échantillonages d'hyperplans à venir. Troisièmement, en général, la descente de gradient peut trouver un petit nombre d'améliorations locales, sans pour autant changer complètement le code. Dans ce cas, l'altération de l'échantillonage d'hyperplans sera minime.

6.2.1 L'effet Baldwin

Il y a d'autres façons de coupler évolution et apprentissage, qui nécessitent chez les individus l'emploi d'un mode d'apprentissage de nature non Lamarckienne. Une méthode qui connait actuellement un intérêt renouvelé, est *l'effet Baldwin* [1] et [10]. D'un point de vue implantation, l'exploitation de l'effet Baldwin recquiert que les individus emploient une forme d'apprentissage, mais le comportement appris n'est pas recodé sur le code génétique, comme dans le cas de l'apprentissage Lamarckien. Au lieu de cela, le comportement appris affecte la forme de la fonction de fitness. En d'autres termes, la fitness est fonction simultanément du comportement hérité et du comportement appris. Le comportement appris correspond à des caractéristiques directement codées sur les gênes du chromosome. De ce point de vue, l'apprentissage change la fonction de fitness. En particulier, supposons que le code génétique, (ou plus précisément le comportement de l'individu développé à partir d'un code génétique particulier) puisse être amélioré de manière significative, à l'aide d'un petit nombre de changements. Sous cette hypothése, n'importe quelle forme d'apprentissage corrigeant le comportement hérité, change aussi le paysage de fitness associé à la fonction à optimiser. Plus précisément, elle récompense les solutions qui sont proches du comportement optimal, lequel peut être atteint avec l'apprentissage. L'idée que le comportement appris peut influencer l'évolution, en créant une pression sélective vers des codes génétiques prédisposant au comportement appris, a déjà été proposée par J.M. Baldwin, [14] il y a presque cent ans.

Hinton et Nowlan [10] présentent l'exemple simple suivant, pour illustrer l'effet Baldwin. Supposons que le paysage de fitness soit plat, avec un seul pic qui représente la solution du problème. Supposons aussi que les individus puissent reconnaitre s'ils sont près de la solution et puissent modifier leur comportement de façon à exploiter la solution. Alors, l'apprentissage construit un bassin d'attraction autour du pic, de telle façon que le paysage de fitness soit transformé en un problème d'optimisation moins difficile. Comme le remarque Hinton et Nowlan, il est important que les individus soient capables de reconnaître quand un comportement est meilleur. Plus un individu est proche de la solution, plus il a de chance d'apprendre la solution.

Hinton et Nowlan reconnaissent que cet exemple est un cas extrême non réaliste. Son but est d'illustrer les principes de l'effet Baldwin, en utilisant l'algorithme génétique pour générer des réseaux de neurones. Une marche aléatoire tient lieu d'apprentissage. Le code génétique spécifie la topologie d'un réseau de neurones, en indiquant quelles connexions parmi 20 connexions potentielles, doivent être utilisées pour connecter un ensemble de neurones de façon particulière. La fonction objet (sans apprentissage) est telle qu'un réseau de neurones est déclaré correct et a une fitness élevée, pendant que tous les autres réseaux sont incorrects et ont la même fitness qui est basse. Cette fonction objet crée un pic à l'endroit de la solution correcte, dans un paysage de fitness qui est plat partout ailleurs. Le codage génétique utilise un alphabet de trois caractères: (0, 1, ?) qui spécifie l'existence d'une connexion, l'absence d'une connexion, ou bien la connexion non spécifiée (représentée par ?). Si la connexion n'est pas spécifiée, elle peut être positionnée aléatoirement ou positionnée en utilisant un apprentissage. Dans ce cas, l'apprentissage est simplement une recherche aléatoire d'affectations possibles des connexions non-spécifiées.

Deux résultats intéressants ont étés signalés par Hinton et Nowlan. Tout d'abord, la recherche génétique sans apprentissage ne permet pas de trouver de solutions; Cela n'est pas surprenant puisque la solution est comme une aiguille dans une botte de foin. Deuxièmement les solutions finales trouvées par l'algorithme génétique avec apprentissage sont partiellement spécifiées. En d'autres mots, un petit nombre de caractères non spécifiés vont rester dans le code génétique, à partir du moment où la solution est assez proche de la cible, pour que l'apprentissage trouve la solution avec une probabilité proche de un.

Le travail de Hinton et Nowlan illustre bien l'effet Baldwin, mais ne permet pas de savoir si l'effet Baldwin peut être utilisé pour accélérer l'apprentissage génétique. Belew [3] donne une bonne critique des suppositions sous jacentes au modèle de Hinton et Nowlan. En particulier, le nombre de cycles d'apprentissage attribués aux individus doit être suffisamment grand pour permettre de localiser le pic; sinon, cela reste un problème du type "aiguille dans une botte de foin".

Dans le contexte du codage cellulaire, nous aimerions exploiter une certaine forme d'apprentissage qui puisse corriger la spécification du réseau faite par l'arbre grammaire, correspondant à un codage cellulaire particulier. On veut ensuite tester l'hypothése selon laquelle l'apprentissage (qui est utilisé pour changer la valeur retournée par la fonction de fitness mais ne modifie pas le code) va aboutir à une accélération de la recherche génétique de réseaux de neurones, pour calculer la parité et la symétrie.

6.2.2 Apprentissage développemental

Finalement, il y encore d'autres moyens d'utiliser l'apprentissage dans le contexte du code cellulaire, qui est intermédiaire entre l'apprentissage Lamarckien et l'apprentissage direct des poids. On appelera cette forme d'apprentissage intermédiaire, *Apprentissage développemental.* Cette forme d'apprentissage exploite une caractéristique unique des grammaires récursives utilisées par le codage cellulaire. Comme le codage permet des mouvements récursifs dans l'arbre grammaire, l'apprentissage pour le réseau de neurones N_1 qui est appliqué pour $L = 1$, peut être exploité au cours du développement de réseaux N_L pour des valeurs de L plus grande que 1 ($L = 1$ correspond à une itération du développements sans récursion). En d'autres termes, l'apprentissage a lieu pour l'itération $L = 1$. Cela peut être accompli avec un coût relativement faible, parce que le réseau N_1 est très petit, conséquence d'un développement court. L'apprentissage doit modifier seulement quelques poids (un ou deux poids dans notre expérience). L'information apprise peut être rétro-codée sur la grammaire, de telle façon que la grammaire modifiée génère le réseau avec les poids modifiés. La grammaire modifiée peut alors être utilisée pour développer des réseaux de neurones N_L plus grands, où $L > 1$. Cela peut peut être efficace sous l'hypothése de régularité suivante:

> Si une grammaire développe correctement N_1, elle a en moyenne plus de chances de développer correctement N_L pour $L > 1$.

Cette hypothése n'est valide que si la fonction que N_L calcule est elle même régulière, comme c'est le cas pour la parité ou la symétrie. La figure 6.1 illustre comment une grammaire peut enregistrer une modification des poids pour $L = 1$ et la répliquer deux fois pour $L = 2$. Le réseau pour $L = 2$ est représenté à la figure 6.1(e). Il a deux poids positionnés à -1. Sur cet exemple, le réseau N_L aura L poids positionnés sur -1. La méthode utilisée pour le rétro-codage est expliquée au paragraphe 6.2.5.

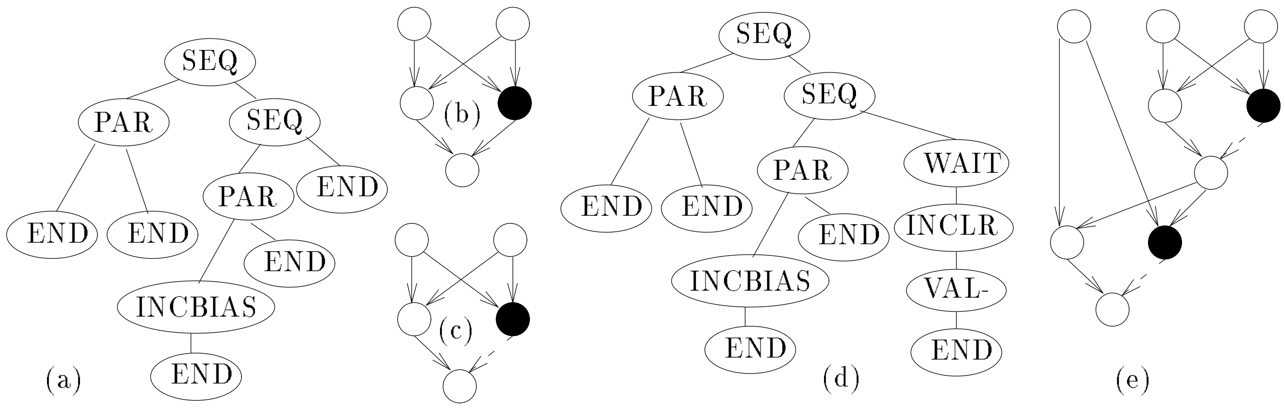

Figure 6.1: Algorithme pour l'apprentissage développemental: (a) chromosome produit par le GA, (b) développement du réseau N_1, pour une vie $L = 1$, (c) après l'apprentissage de N_1, un poids reliant l'unité de sortie a été modifié en -1 (ligne tirets). (d) rétro-codage sur le chromosome (e) développement du réseau de neurones N_2 en utilisant le chromosome modifié, le poids -1 est répété deux fois.

La règle suivant laquelle des réseaux de neurones moins développés nécessitent moins d'efforts de calculs à évaluer, est sans doute importante du point de vue biologique. On n'a pas fait de statistiques à ce sujet, mais la raison de ce coût moindre est claire: la taille de N_L croît avec L. S'il y a k opérateurs de récursion `REC`, la taille va croître comme k^L, sauf dans le cas où $k = 1$, où la taille croît linéairement. Le temps d'évaluation croît avec la taille des réseaux.

L'apprentissage développemental défini ci-dessus, n'est pas Lamarckien. En effet, l'information résultant de l'apprentissage est rétro-codée sur le chromosome, dans le but de l'utiliser pour développer d'autres réseaux. Les descendants de l'individu ne porteront pas cette information dans leur code. En fait, le code génétique est utilisé pour stocker l'information apprise pendant la "vie de l'individu", durant laquelle le développement de différents réseaux a lieu, mais cette information apprise n'est pas transmise aux générations suivantes. On appellera ce mode d'apprentissage, *l'apprentissage développemental*.

On ne revendique pas la possibilité biologique de cette méthode, puisque nous supposons qu'il n'y a pas d'évidences biologiques pour utiliser des codage cellulaires récursifs de la manière exacte dont ils sont utilisés ici. Cependant, il existe des espèces qui, effectivement, recodent de l'information apprise, sur leur ADN, en utilisant l'ADN comme moyen de stockage de l'information apprise; Ce stockage reste local, et n'a pas d'impact sur la reproduction.

On doit aussi souligner que *l'apprentissage developpemental* est efficace, parce qu'il utilise *l'effet Baldwin*. En d'autres termes, *l'apprentissage développemental* est seulement une façon plus fonctionnelle d'apprendre, une façon plus fonctionnelle d'exploiter le comportement appris durant la vie et le développement de l'individu. Le comportement appris n'est pas transmis aux descendants.

6.2.3 Étude comparative

Notre but est d'examiner l'apprentissage Lamarckien, l'impact de l'effet Baldwin sur l'apprentissage, et l'utilisation de la méthode d'apprentissage développemental. Pour l'apprentissage Lamarckien et développemental, il est nécessaire de rétro-coder le comportement appris sur le chromosome. Comme le codage cellulaire spécifie un réseau de neurones booléen, l'apprentissage que nous utilisons doit aussi produire des poids booléens. De plus, il faut que le processus d'apprentissage soit très rapide. Pour exploiter l'effet Baldwin, l'analogie proposée par Hinton et Nowlan suggère aussi un apprentissage rapide et en quelque sorte, minimal. L'apprentissage doit donner une information sur la proximité du but. Plus spécifiquement, un apprentissage idéal, pour le jeu d'expériences que nous proposons, serait un apprentissage en un seul cycle effectuant un ou plusieurs changements dans le réseau, pour l'améliorer. Ces changements peuvent alors être testés.

Étant donné le cahier des charges que nous venons de faire ressortir, nous avons choisi d'utiliser un apprentissage du type hebbien. Cela signifie aussi que nous allons utiliser l'apprentissage d'une façon assez restreinte.

6.2.4 Réalisation de l'apprentissage Hebbien

Donnons nous un réseau de neurones booléen avec une couche cachée munie de seuils corrects, et tel que la première couche ait aussi un jeu de poids et de seuils corrects, "correct" signifie que le réseau peut calculer la fonction cible, à condition d'ajuster les poids des unités de sortie. Il est clair que le problème de l'apprentissage est ramené à celui d'apprendre une fonction linéairement séparable relativement aux sorties des unités cachées. A ce stade, on pourrait utiliser l'algorithme du perceptron qui garantirait la convergence de l'apprentissage. Le problème est que cet algorithme nécessite plus d'un cycle et ne produit pas de poids booléens. Si toutefois, on limite l'apprentissage à un seul cycle, on peut observer les activations simultanées des unités cachées et des unités de sortie, et collecter des informations pour prévoir si chaque connexion entre une unité cachée et une unité de sortie particulière doit être inhibitrice ou excitatrice. Car, pour chaque exemple à apprendre, et pour chaque paire (unité cachée, unité sortie), on voudrait que le poids soit positif (i.e +1) si les deux unités de cette paire sont activées simultanément, et négatif (i.e. -1) si l'unité cachée est activée mais pas l'unité de sortie. (Il n'y a pas lieu de considérer le cas où l'unité cachée n'est pas activée, car alors le changement du poids ne modifie pas les calculs du réseau.)

N'oublions pas que les poids des réseaux sont booléens (± 1). Pour la parité et la symétrie, les réseaux de neurones ont une seule unité de sortie. On utilise l'algorithme suivant pour calculer de nouveaux poids sur les connexions qui arrivent sur l'unité de sortie. Chaque exemple à apprendre est présenté sur les unités d'entrée. Le niveau d'activation de chaque neurone de la couche cachée est calculé, jusqu'à ce que toutes les unités cachées soient traitées. L'activité de l'unité de sortie est forcée sur la sortie que l'on désire associer à l'exemple. Pour chaque lien l relié à l'unité de sortie, l'information hebbienne suivante est calculée: Chaque lien l possède une variable d_l. Si les activités des unités reliées par l sont les mêmes, d_l est incrémenté, sinon d_l est décrémenté. La variable d_l contient une mesure de la corrélation des activités des neurones reliés par l. Avant l'apprentissage, toutes les variables d_l sont initialisées à zéro.

Cet algorithme est différent de l'algorithme du perceptron car on enregistre des statistiques sur tous les exemples d'apprentissage, pas seulement sur ceux qui produisent des erreurs. Cet apprentissage ressemble à celui utilisé dans la machine de Boltzman pour réaliser une recherche globale de l'espace des poids [5].

Aprés avoir traité tous les exemples, la variable d_l est utilisée pour déterminer si on doit changer le signe du poids w_l. On considére le sous-ensemble des liens l pour lesquels d_l et w_l ont un signe opposé (on rapelle que $w_l = \pm 1$). On modifie seulement les poids des liens de ce sous ensemble. Ce sont les liens pour lesquels les poids sont différents des corrélations, et le but de l'apprentissage hebbien est précisément d'égaliser la valeur des poids et celle des corrélations. C'est la raison pour laquelle on dit que cet apprentissage est de type hebbien. Pour les liens qui ont été sélectionnés, $|d_l|$ peut être considéré comme une mesure de l'erreur commise sur le poids, parce que c'est la quantité de changements à faire pour ramener d_l sur le signe correct. Donc, on trie les liens du sous ensemble, par d_l décroissants. Le premier lien, c'est à dire celui qui a le d_l le plus grand, a son poids modifié avec la probabilité 1. Le deuxième lien a son poids modifié avec la probabilité 0.05 (que nous avons choisie). Plus généralement, le i^{eme} poids est modifié avec une probabilité $(0.05)^{i-1}$. Cet algorithme d'apprentissage assure que la plupart du temps, le poids d'un lien sera modifié. La seule condition pour cela est qu'au moins un lien l ait son d_l et w_l de signes opposés. Sous cette condition, le nombre moyen de liens modifiés est compris entre 1 et $1/(1-0.05) \approx 1.05$. Lorsque les changements de poids sont effectués, on traite une fois de plus tous les exemples en acceptant les changements effectués seulement si la fitness s'est améliorée. Seulement un cycle d'apprentissage est utilisé.

En moyenne moins de deux liens ont leurs poids modifiés, et cela devrait être suffisant pour créer un effet Baldwin, puisque l'apprentissage doit seulement contruire des bassins d'attraction autour des minimas, dans le paysage de fitness associé à la fonction d'évaluation brute (sans apprentissage). Dans nos expériences, les seuils (ou les biais ces deux noms désignent les mêmes quantités) ne sont pas appris mais doivent être calculés par le GA.

L'expérience montre que cet apprentissage est suffisant dans le cas du réseau de neurones XOR qui calcule un "OU" exclusif (XOR). En effet, si l'architecture et les seuils sont corrects, l'apprentissage est capable de corriger les mauvais poids. Un exemple d'architecture correcte (où tous les poids valent 1) comporte deux unités cachées. L'une de ces unités calcule un ET logique, l'autre calcule un OU logique. Pour obtenir un réseau qui calcule le XOR, il suffit de positionner le poids de la connexion entre l'unité ET et l'unité de sortie sur -1, de façon à ce que l'unité de sortie produise le résultat de l'unité OR à moins que l'unité ET ne soit à 1 auquel cas la sortie est 0. Le GA produit des réseaux ayant tous leurs poids initialisés à la valeur 1; donc pour aider le GA à produire un réseau de neurones XOR, il suffit de modifier les poids des connexions reliant l'unité de sortie. Ainsi, dans le cas du XOR, et en fait dans le cas de tous les problèmes de parité (parmi lesquels le XOR est le représentant le plus simple), la forme d'apprentissage hebbien limité que nous utilisons suffit, parce qu'elle exploite un trait fortuit du problème: le fait qu'il suffise de modifier les poids des connexions reliant l'unité de sortie. Nous allons cependant montrer au chapitre suivant que de tels traits fortuits ne sont pas indispensables pour qu'une stratégie d'apprentissage soit efficace.

Un travail fameux fait par Hinton Ackley et Sejnowsky au sujet de la machine de Boltzman [5], a montré que l'apprentissage hebbien essaye d'optimiser l'information mutuelle entre la fonction calculée par le réseau et la fonction cible. Comme la fitness que nous utilisons est aussi basée sur l'information mutuelle, le GA et l'apprentissage optimisent la même quantité et sont donc cohérents.

6.2.5 Réalisation de l'apprentissage développemental

Un des avantages du codage récursif est que l'évaluation peut être faite de façon incrémentale. Pour la parité ou la symétrie, un réseau de neurones peut être évalué relativement à sa capacité à résoudre le problème de taille 1 où deux entrées sont nécessaires. Si on connaît une solution pour le problème à deux entréesl'itération suivante dans le développement peut être effectuée, et le réseau à trois entrées, être évalué. De cette manière, l'évaluation est rapide dans les premières générations, pour traiter les problèmes avec un petit nombre d'entrées, mais elle devient plus coûteuse au fur et à mesure que les réseaux deviennent de plus en plus performants et ont une taille croissante.

De façon analogue, on utilise l'apprentissage uniquement sur les réseaux durant la première itération du processus de développement récursif. Ce réseau de neurones est le plus petit de toute une famille de réseaux de neurones qui peut être générée par un chromosome. C'est le plus petit, à la fois en termes de poids et en termes d'unités, donc, peu de temps est dépensé pour faire de l'apprentissage. Mais nous voudrions aussi que les autres réseaux de neurones bénificient du processus d'apprentissage. Pour cela, on doit rétro-coder sur le chromosome les changements de poids qui ont été faits durant l'apprentissage du premier réseau; et ceci, en utilisant une combinaison appropriée d'opérateurs cellulaires. On peut alors développer la famille toute entière de réseaux de neurones, de façon à exploiter l'apprentissage via le chromosome modifié. L'apprentissage a introduit de nouveaux opérateurs cellulaires qu'une cellule (du réseau de cellules intermédiaire) va exécuter pour modifier ses poids; Ainsi cette cellule va reproduire un sous-réseau identique à celui obtenu après apprentissage. Une des exigences du rétro-codage sur le chromosome est que le nouveau chromosome développe correctement le réseau de neurones correspondant à la première itération, en incluant les modifications apportées par l'apprentissage. Cela n'est pas toujours simple. En général, quand une cellule devient un neurone, elle exécute un opérateur cellulaire `END`, et perd sa tête de lecture de telle façon qu'elle ne peut plus se réécrire elle même. Mais pour faire du rétro-codage, on doit garder la tête de lecture. Cette tête pointe en effet sur le noeud exact où le sous-arbre d'opérateurs cellulaires codant les modifications de poids doit être inséré. Un exemple concret est montré à la figure 6.2.

Supposons que l'apprentissage change la valeur du second poids du neurone de sortie, en la valeur -1. Considérons la valeur du registre lien au moment où la cellule devient un neurone. Si cette valeur est 1, on doit générer le sous-arbre, `INCLR(VAL-(END))`. L'opérateur `INCLR` positionne le registre lien sur 2. L'opérateur `VAL-` positionne le poids du lien numéro 2 sur -1. Soit N le noeud pointé par la tête de lecture du neurone. Ce noeud a une arité 0, et vient juste de transformer la cellule en un neurone fini. Pour insérer le nouveau sous-arbre, on doit transformer ce noeud feuille en un noeud interface d'arité 1. On remplace l'opérateur cellulaire `END` par un opérateur unaire `WAIT`, puis on insère le sous-arbre qui effectue le changement de poids. Quand une cellule exécute l'opérateur `WAIT`, elle se contente de positionner sa tête de lecture sur le sous-arbre nouveau, et de se mettre dans la file d'attente pour attendre son prochain tour de réécriture.

On ne peut être sûr que ce genre de changements dans l'arbre grammaire puisse exactement recoder le comportement appris. Le code modifié produira le comportement appris si les deux conditions suivantes sont vérifiées:

- L'apprentissage est effectué sur le premier réseau de la famille
- Pendant le développement du code modifié, pour chaque cellule c du graphe de cellules intermédiaire qui commence à lire le sous-arbre inséré, les cellules ayant des connexions qui arrivent sur c doivent avoir perdu leur tête de lecture et être des neurones finis.

  Cela assure que les changements nouveaux apportés par c ne vont pas intéragir de manière inapropriée avec les cellules qui connectent c en sortie.

Le premier réseau de neurones de la famille a la caractéristique suivante: une feuille donnée de l'arbre grammaire est lue par une cellule au maximum. Ce n'est pas le cas pour les réseaux de neurones d'ordre plus grand. Si la première condition n'était pas vérifiée, deux neurones distincts pourraient lire la même feuille, et les deux pourraient essayer de recoller à l'arbre principal deux sous-arbre distincts, sur une même feuille. Il n'est pas possible d'insérer deux sous-arbres distincts sur une même feuille car ces deux sous-arbres pourraient coder des modifications de poids contradictoires.

Si la seconde condition est satisfaite, alors, le changement introduit par le sous-arbre ajouté reste local. Sinon il pourrait produire des effets de bord inattendus. Par exemple sur la figure 6.2,

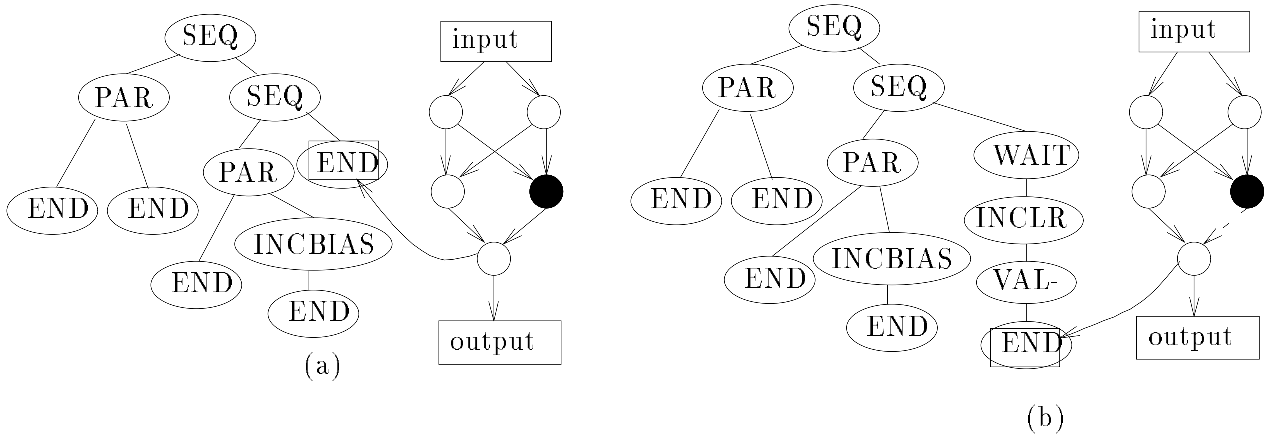

Figure 6.2: Rétro-codage sur l'arbre grammaire, qui permet de coder le code d'un XOR. (a) Le réseau et le code avant le rétro-codage. (b) L'arbre grammaire a été modifié et le changement de poids correspondant est montré. Le réseau de neurones modifié calcule le XOR.

supposons que l'opérateur `INCBIAS` de l'arbre grammaire soit remplacé par un `PAR`, Le noeud correspondant qui est peint en noir sur la figure 6.2 va maintenant se diviser au lieu de changer son seuil, et il y aura trois connexions d'entrée sur l'unité de sortie. Aussi, supposons que nous désirions changer le poids de la seconde connexion d'entrée de l'unité de sortie; si le changement de poids est réalisé avant la division parallèle des voisins, alors, la seconde et la troisième connexion auront simultanément le poids -1. Si le changement de poids est réalisé après la division parallèle, seul le poids de numéro 2 sera négatif.

Pour assurer une séquence de développement correcte, on pourrait introduire plusieurs opérateurs `WAIT` à des positions clé, de telle façon que l'arbre nouveau qui est inséré s'exécute dans la séquence correcte. Cependant développer les réseaux de neurones d'ordre suppérieur demanderait un coût supplémentaire important en temps de calcul. De plus, cela rendrait l'apprentissage Lamarckien difficile parce que les chromosomes deviendraient rapidement trés gros. Cela signifie que le codage ne serait plus aussi compact. Et la recombinaison devrait traiter des arbres de profondeur croissante. Remarquons que cette stratégie d'insérer de longs délais ne pose pas de difficultés vis à vis de l'apprentissage développemental, puisque le chromosome modifié ne "retourne jamais" dans la population du GA. Il n'y a pas d'inconvénient à ajouter le nombre de `WAIT` nécessaire pour être sûr que toutes les autres cellules aient fini de se développer, avant de commencer à développer le nouveau code cellulaire (correspondant aux poids appris).

Cependant, dans les expériences, nous avons retenu le compromis suivant: Nous remplaçons chaque opérateur 0-aire `END` par un opérateur unaire `WAIT`, puis on raccorde sur cet opérateur `WAIT` le sous-arbre codant le changement de poids. On ajoute un opérateur cellulaire `WAIT`, et on espére ainsi suffisamment augmenter la probabilité de vérifier la seconde condition. Pour rester cohérent, on a utilisé le même rétro-codage simultanément pour l'apprentissage Lamarckien et développemental. Ce rétro-codage ne garantit pas que le comportement appris ne va pas intéragir de façon inattendue avec le processus de développement. Empiriquement on constate que dans la plupart des cas analysés, il n'y a pas d'interactions nuisibles.

6.3 Simulation de différentes combinaisons

Nous décrivons à présent le résultat d'expériences qui comparent différentes façons d' ajouter de l'apprentissage. Le GA brut désigne le GA sans apprentissage. Nous avons décrit le codage au paragraphe 2.1. On rapelle au lecteur que ce codage ne code pas seulement l'architecture mais aussi les poids. Grâce à cette caractéristique, il est possible d'utiliser le GA seul, sans apprentissage. Dés que le réseau de neurones est développé, on fait un apprentissage de type hebbien. Si on utilise seulement la fitness améliorée du réseau de neurones N_1 obtenu avec une vie 1, (dans le cas $L = 1$ il n'y a pas de récursion) dont les poids ont été modifiés, on apelle cela *Apprentissage Baldwin.* Si on rétro-code l'information sur le chromosome et qu'on l'utilise pour développer des réseaux d'ordre plus grand, c'est l'*Apprentissage développemental.* Si les chromosomes améliorés sont remis en circulation dans la population, cela s'appelle *Apprentissage Lamarckien.* Les expériences suivantes comparent les quatres modes d'apprentissages sur des tâches de complexité croissante.

6.3.1 Mutation stratégique

Au cours des expérimentations on a remarqué que la tâche la plus difficile rencontrée par le GA consiste à placer correctement l'opérateur cellulaire de branchement récurrent `REC`. Dans le cas du problème de la parité, une estimation correcte de l'endroit où placer `REC` suffit à résoudre le problème de la récurrence, comme l'indique la figure 5.5. D'autre part, un placement incorrect empêche le GA de développer un seul des réseaux de neurones de la famille, excepté le tout premier. Ce caractére "tout ou rien" provoque une discontinuité dans le paysage de fitness, et le placement de `REC` est donc très difficile. Pour surmonter le problème on a utilisé une forme de mutation stratégique pour `REC`. En mutant `REC` 10 fois plus souvent que les autres opérateures cellulaires, la plupart des opérateurs `REC` sont rapidement supprimés, et l'allèle `REC` se trouve représenté 10 fois moins que les autres opérateurs d'arité 1. Pour contrebalancer cet effet de rareté, chaque opérateur d'arité 1 a une probabilité $4 * t_{mut}$ de muter en l'opérateur `REC`, où t_{mut} est le taux de mutation des autres allèles. L'effet combiné de ces deux stratégies de mutation aboutit à explorer un grand nombre de places différentes pour l'opérateur `REC`.

6.3.2 La fonction parité

Dans ce jeu d'expériences, le GA "tourne" pendant un temps maximum de 900 secondes sur un noeud de la machine parallèle iPSC860. Pour résoudre le problème de la parité, le GA doit trouver un code celullaire qui développe une famille de réseaux (N_L) telle que N_L calcule la parité de $L + 1$ entrées. La fonction booléenne parité calcule la somme de ses entrées modulo 2. La table 5.1 donne une description des paramètres de l'expérience. Pour établir une comparaison valide, on a fait 50 esssais, et on a calculé la moyenne et l'écart type de deux variables:

- le temps requis pour trouver la solution
- le nombre total de réseaux de neurones évalués.

Pour obtenir une estimation de l'écart type du résultat, il suffit de diviser par $\sqrt{50}$ l'écart type de la variable. On indique ici l'écart type de la variable pour mettre en évidence le fait qu'il est grand. Cette information permet de mieux comprendre comment fonctionne la recherche génétique.

Le problème de la parité est un peu spécial relativement à l'apprentissage développemental. Le réseau de neurones N_1 est développé avec une vie $L = 1$, il calcule la fonction XOR. Ce réseau XOR a tous ses poids égaux à 1, excepté le poids de la connexion reliant le AND à l'unité de sortie,

Apprentissage	temps	ecart type	EVALUATIONS	ecart type	SUCCES
GA brut	240	197	23865	20317	36
Baldwin	234	205	14307	10699	28
Développemental	131	195	8404	8829	47
Lamarckian	66	85	5700	5220	49

Tableau 6.1: Expériences initiales pour le problème de la parité qui se sont déroulées chacune pendant un temps maximum de 900 secondes sur un noeud de la machine paralléle iPSC860, c'est à dire, avec une seule population. Les succés mesurent combien de fois une solution a été trouvée pour le problème de la parité. Le nombre d'essais est 50. Tous les temps sont mesurés en secondes dans le chapitre.

comme le montre la figure 2.6. Comme on "apprend" seulement le réseau N_1, et que les poids sont initialisés à 1, l'apprentissage n'a besoin de changer que les poids des connexions arrivant sur l'unité de sortie.

La première ligne de la table 6.1 présente les résultats de l'expérience avec le GA brut, sans utiliser d'apprentissage. Les trois lignes suivantes utilisent de l'apprentissage. Les simulations sont divisées en deux classes, certaines trouvent une solution rapidement, d'autres prennent un temps très long. Cela explique l'écart type très élevé observé dans les expériences. Le GA est souvent bloqué dans une solution sous-optimale.

Bien que l'apprentissage Baldwin ne permette pas d'améliorer le temps de calcul, il nécessite en moyenne 1700 évaluations de réseaux de neurones avant d'obtenir un succés, alors que le GA brut en nécessite 24000. Cela signifie que le temps moyen d'évaluation est plus important pour l'apprentissage Baldwin. Le temps supplémentaire dépensé pour l'apprentissage hebbien reste aux alentours de 0.5 millisecondes, alors que le temps moyen pour développer et évaluer les réseaux de neurones, est de 7 millisecondes dans les premières générations et augmente substantiellement pour atteindre finalement 14 millisecondes. Ainsi l'algorithme d'apprentissage hebbien prend moins de 8% du temps nécessité pour développer et évaluer un réseau de neurones. Le temps moyen requis pour l'apprentissage Baldwin est plus grand, à cause de la règle utilisée pour évaluer la fitness d'un code d'une famille (N_L) de réseaux de neurones. Le GA brut ne développe pas tous les réseaux N_L pour $L > 1$ dans les premières générations. Pour toutes les expériences la règle de développement présentée au chapitre précédent explique à quel moment on commence à développer N_2 puis N_3 ...etc jusqu'à $N_{l_{max}}$. Quand l'apprentissage hebbien est utilisé, le réseau N_1 est obtenu trés rapidement. Par exemple, avec l'apprentissage Baldwin sur l'unité de sortie, le temps moyen pour trouver le premier réseau de neurones est de 19 secondes, alors que le GA brut prends 176 secondes. Ainsi, avec l'apprentissage Baldwin, le GA commence à développer N_2 trés tôt, à partir de la deuxième génération, en général. Cela a pour effet secondaire de rendre l'évaluation des chromosomes plus coûteuse. L'apprentissage Lamarckien et l'apprentissage développemental sont beaucoup plus rapides que le GA brut ou l'apprentissage Baldwin simple. Le mode Lamarckien paraît plus performant que le mode développemental sur cet exemple.

Pour pallier au fait que certaines simulations ne donnent pas de solution, nous avons utilisé une population plus grosse et un algorithme génétique parallèle. Le GA parallèle est décrit au chapitre 8. Le nombre d'expériences est encore 50. Pour chaque expérience, le GA tourne 3 heures sur 4 noeuds de la machine parallèle iPSC860. La taille de la population varie entre 3500 et 2000 individus par noeud. On a donc utilisé l'équivalent de 12 heures de calculs sur un noeud avant de

Apprentissage	temps	ecart type	EVALUATIONS	ecart type	SUCCES
basic GA	173	64	17808	5818	50
Baldwin	135	86	12511	4799	50
Developmental	59	19	7160	1805	50
Lamarckian	55	18	6800	1763	50

Tableau 6.2: Deuxième série d'expériences pour le problème de la parité. Le GA a tourné 3 heures sur 4 noeuds de la machine parallèle iPSC860

Apprentissage	temps	ecart type	EVALUATIONS	ecart type
basic GA	91	55	7332	3869
Baldwin	74	36	5811	2292
Developmental	48	20	4352	1840
Lamarckian	53	29	4381	2229

Tableau 6.3: Expérience avec le problème de la symétrie. Le GA a tourné 300 secondes sur 32 noeuds de la machine parallèle iPSC860

décider qu'il y avait échec.

Les résultats sont exprimés en termes de calculs par processeurs, à la table 6.2. Comme quatre processeurs ont été utilisés dans ces expériences, les temps de calcul et les nombres d'évaluations doivent être multipliés par un facteur 4. Le GA obtient un résultat dans toutes les expériences. L'écart type est plus petit que dans la première série d'expériences, parce que la taille de la population est suffisamment grande pour pouvoir échantilloner correctement des blocs de base intéressants. L'apprentissage Baldwin produit une accélération de 1.3 sur le GA brut. L'apprentissage Lamarckien et l'apprentissage développemental produisent des résultats encore bien meilleurs. Les performances de ces deux apprentissages sont semblables.

6.3.3 La fonction symétrie

Le problème consiste à trouver un code qui développe une famille de réseaux de neurones (N_L) telle que N_L calcule la fonction symétrie de $2 * L + 1$ entrées binaires. La symétrie de $2 * L + 1$ entrées binaires calcule 1 si, et seulement si, l'entrée est symétrique, et si le bit du milieu vaut 1. Ce problème est une version différente de la symétrie vue au chapitre précédent qui utilisait $2 * L$ bits. Cette nouvelle version est beaucoup plus facile à apprendre par le GA, que celle du chapitre précédent, car le problème de taille 1 est le même que le problème de la récurrence. Le GA tourne 300 secondes sur 32 noeuds d'un iPSC860. La taille de la population varie entre 300 et 200 individus sur chaque noeud, et donc entre 9200 et 6400 sur l'ensemble des noeuds combinés. La table 6.3 présente les résultats sans apprentissage, et l'utilisation d'un apprentissage sur les poids de l'unité de sortie. Ces résultats sont basés sur 30 expériences. Chaque essai a fourni une solution.

On a indiqué les temps de calcul de chaque processeur. Il faut les multiplier par 32 pour trouver le temps de calcul total, Comme on utilise des populations assez grandes, l'écart moyen est raisonnablement petit. L'apprentissage Baldwin est cette fois plus performant que le GA brut. L'apprentissage développemental surpasse l'apprentissage Lamarckien.

6.4 Conclusion

Faisons le point sur les différentes sortes d'apprentissage.

Nous avons introduit l'apprentissage développemental: s'il est réalisé tôt, il sera d'autant plus efficace qu'il influence le développement ultérieur du réseau; il permet d'éviter le coût élevé d'un apprentissage sur des réseaux de plus grande taille développés ultérieurement dans la vie de l'arbre grammaire.

D'autre part, l'apprentissage développemental constitue un outil bien adapté pour comparer une évolution Lamarckienne avec une Darwinnienne, en exploitant l'effet Baldwin. Comme dans l'apprentissage Lamarckien, c'est la famille entière des réseaux développée à partir d'un code cellulaire, qui va profiter de l'apprentissage, pas seulement ceux de la première génération développés avec une seule itération de l'arbre grammaire.

En second lieu, l'apprentissage développemental est compétitif avec l'apprentissage Lamarckien, dans la recherche d'une famille de réseaux de neurones booléens. C'est en particulier vrai pour le problème de la symétrie. Les résultats de la parité ne sont pas aussi probants, mais la parité est un problème assez particulier, car l'apprentissage des poids des connexions arrivant sur l'unité de sortie suffit pour trouver un jeu de poids corrects. L'utilisation de l'apprentissage (ou d'une optimisation locale) pour changer le paysage de fitness sans utiliser d'apprentissage Lamarckien, présente un intérêt général pour les chercheurs travaillant sur les GA et concernés par l'optimisation de fonctions.

Finalement, le travail présenté dans ce chapitre montre que le codage cellulaire est un banc d'essai correct, et peut être unique, pour tester et comparer apprentissage Lamarckien et effet Baldwin. L'apprentissage hebbien que nous avons utilisé est très simple. D'autres formes d'apprentissages pourraient être considérées. Le prochain chapitre présentera un apprentissage qui a été élaboré spécialement pour les réseaux de neurones produits par le GA en utilisant le code cellulaire.

Chapitre 7

L'apprentissage "Switch"

Ce chapitre propose un apprentissage spécialement adapté aux réseaux de neurones générés par l'algorithme génétique en utilisant le codage cellulaire. On va d'abord décrire le cahier des charges d'un tel algorithme. On propose ensuite un algorithme d'apprentissage qui apprend des poids ± 1. Une extension de cet algorithme permet d'apprendre des poids dans $\{1, 0, -1\}$ mais l'efficacité de l'apprentissage est amoindrie. Finalement, on présente deux sortes d'expériences qui démontrent que l'apprentissage proposé remplit les conditions du cahier des charges. La première sorte d'expériences utilise l'apprentissage sur une architecture fixée, et étudie la convergence de l'apprentissage. La deuxième sorte d'expérience utilise l'apprentissage en le combinant avec le GA.

7.1 Le cahier des charges de l'apprentissage

L'algorithme d'apprentissage proposé est conçu pour apprendre les poids des connexions des réseaux de neurones générés par le GA, en utilisant le codage cellulaire. Il doit donc avoir des caractéristiques spéciales car:

- les réseaux de neurones générés par le GA ont un aspect particulier
- l'apprentissage n'est pas appliqué sur un seul réseau, mais sur plusieurs milliers de réseaux.

Le GA produit des réseaux de neurones avec des seuils entiers et des poids booléens. L'apprentissage doit donc aussi trouver des poids booléens, pour que l'on puisse par la suite rétro-coder l'information apprise, sur le code cellulaire. Le rétro-codage est nécessaire pour implanter le mode d'apprentissage Lamarkien et le mode d'apprentissage développemental vus au chapitre précédent.

L'apprentissage n'optimise pas les seuils parce que le GA peut les calculer de manière avantageuse. En effet: le code cellulaire d'un seuil est compact. En utilisant un opérateur cellulaire tel que `INCBIAS`, il n'est pas nécessaire de spécifier un numéro de lien, ce qui est le cas si on veut coder un changement de poids. Donc le GA a moins d'efforts à faire pour calculer les seuils corrects que pour calculer les poids corrects.

La sigmoïde des neurones est la fonction signe. Un neurone calcule son champ qui est la somme pondérée des activités de ses voisins. Si le champ est inférieur ou égal à zéro, l'activité du neurone devient zéro, sinon elle devient 1. Le GA génére des milliers de réseaux de neurones, donc il n'est pas nécessaire que l'apprentissage soit systématiquement convergent sur chaque réseau. D'un autre côté l'apprentissage doit être rapide, car le temps dépensé pour une expérience avec le GA, sera le temps mis par l'apprentissage multiplié par des milliers. Le but de l'apprentissage est d'améliorer les réseaux de neurones produits par le GA. Il doit changer quelques poids pour améliorer les performances du réseau. Il ne peut trouver l'optimum global, que si le réseau, avant apprentissage, en était proche. Les réseaux de neurones générés par le GA en utilisant le codage cellulaire ont quelques particularités: les neurones sont faiblement connectés, le nombre de connexions en entrée ne dépasse pas trois, en général. Le nombre de couches est grand; une rétro-propagation standard ne serait donc probablement pas efficace. La plupart des poids valent 1, seuls quelques poids valent -1. La raison en est que les poids sont initialisés à 1, et seuls les poids négatifs doivent être codés en utilisant des opérateurs cellulaires tels que `VAL-` et `INCBIAS`. L'algorithme d'apprentissage doit tirer parti de ces particularités.

7.2 L'algorithme d'apprentissage Switch

```
Soit c, le neurone en cours de traitement
Pour chaque neurone n qui  touche en sortie c, avec le lien l
  essayer de modifier l'activite de n et calculer le nouveau champ.
  switch(Comp(nouveau champ, ancien champ))
    case 0 : if(Petit(ancien champ)) r_n := r_n + r_c;
             w_n := w_n + w_c;
    case + : r_n := r_n + w_c;
    case - : w_n := w_n + q * w_c / 2;
  Si l'activite de n est non nulle
    essayer de modifier le poids de l
    switch(Comp(nouveau champ, ancien champ ))
       case 0 : if(Petit(ancien champ)) r_l := r_l + r_c;
                w_l := w_l + w_c;
       case + : r_l := r_l + w_c;
       case - : w_l := w_l + q * w_c ;
  Sinon
    Essayer de modifier le poids de l et l'activite de n
    Calculer le nouveau champ
    switch(Comp(nouveau champ, ancien champ ))
      case 0 : if(Petit(ancien champ)) r_n := r_n + q * r_c;
                                       r_l := r_l + q * r_c;
               w_n := w_n + q * w_c; w_l := w_l + q * w_c;
      case + : r_n := r_n + q * w_c; r_l := r_l + q * w_c;
      case - : w_n := w_n + q * q * w_c; w_l := w_l + q * q * w_c;
```

Figure 7.1: L'algorithme d'apprentissage. Les indices c, n, l sont attribués respectivement au neurone en cours de traitement, au neurone voisin et au lien. Le champs est la somme pondérée des entrées d'un neurone. La fonction `Comp` renvoie 0 si le champ change de signe, et sinon elle retourne le signe de la différence entre son premier argument et son deuxième argument. La fonction `Petit` renvoie 1 si son argument est 0 ou 1, sinon elle renvoie 0.

Nous allons maintenant décrire l'algorithme d'apprentissage: A chaque cycle, tous les exemples à apprendre sont passés à travers le réseau. Aprés le passage de chaque exemple, les valeurs de deux variables sont calculées pour chaque neurone et pour chaque lien: r et w. La lettre r est mise pour "right" qui signifie "correct" en anglais. La lettre w est mise pour "wrong" qui signfie "incorrect".

Soit o une unité de sortie, si l'activité de o est la sortie désirée, r_o est mis à 1 et w_o est mis à 0, sinon c'est le contraire, r_o est mis à 0 est w_o est mis à 1. Pour une unité cachée h, les variables r_h et w_h sont calculées de façon à indiquer si l'activité courante calculée par h est correcte. Si r_h est élevé, une modification de l'activité de h est susceptible de diminuer les performances du réseau de neurones sur l'exemple en cours de traitement. Si w_h est élevé, c'est le contraire, une modification de l'activité de h est susceptible d'augmenter les performances du réseau de neurones. De même, pour chaque lien l, les variables r_l et h_l sont calculées de façon à indiquer si le poids courant du lien est correct.

Les calculs de toutes ces variables se font en traitant les neurones les uns après les autres dans le sens contraire de la propagation des activités. On commence par les unités de sorties, comme pour la rétro-propagation, et on finit par les unités d'entrée. L'algorithme décrit à la figure 7.1 montre comment traiter un neurone. Soit c le neurone en cours de traitement, il y a trois façons différentes de modifier l'activité de c

- t_1 Modifier le poids d'une connexion entrante.
- t_2 Modifier l'activité d'un voisin d'entrée n.
- t_3 Modifier les deux en même temps.

L'algorithme d'apprentissage essaye successivement t_1 t_2 et t_3. Pour chaque essai t_i, il analyse l'effet de la modification sur le neurone c. Supposons que l'effet soit désirable (i.e. il augmente le nombre d'exemples que le réseau calcule correctement), l'algorithme augmente w_n si l'activité de n a été modifiée, ou w_l si le poids de l a été changé. Sinon, supposons que l'effet soit mauvais (i.e. il augmente le nombre d'exemples où le réseau fait une erreur de calcul), l'algorithme augmente w_n si l'activité de n a été modifiée, ou w_l si le poids de l a été changé. Le champ d'un neurone est la somme pondérée de ses entrées. La sortie du neurone est 1 si le champ est strictement positif, et 0 sinon. Il y a trois effets possibles, suivant la façon dont le champ du neurone est modifié:

- Le champ de c a changé de signe (par convention, ici, zéro a le signe -), cela a pour effet de changer l'activité de c. Si $w_c > 0$, on désirerait que l'activité de c change avec une pondération w_c. On enregistre le fait que l'essai t_i est avantageux avec une pondération égale à w_c. Si t_i a modifié l'activité de n, on ajoute w_c à w_n; si t_i a modifié le poids de l, on ajoute w_c à w_l.

  Si $r_c > 0$ il faut éviter que l'activité de c change avec une pondération r_c. On enregistre le fait que t_i est désavantageux avec une pondération r_c. On ajoute r_c à r_n ou bien à r_l suivant l'essai t_i, mais cette fois ci seulement si c a une stabilité minimale, c'est à dire, si son champ avant l'essai t_i valait 1 ou 0. Si le champ valait -1 ou 2, les deux autres cas possibles, obligatoirement l'activité de n valait 1, et t_i a modifié le poids. Dans ce cas, l'effet de t_i peut toujours être contrebalancé en positionant l'activité de n à 0; il n'est donc pas irrémédiable.

- Le champ augmente en valeur absolue: si on veut que l'activité de c change, cet effet est indésirable parce que le changement sera plus difficile à obtenir. On ajoute w_c à r_n ou r_l suivant l'essai t_i.

- Le champ diminue en valeur absolue. Si on veut que l'activité de c change, cet effet est désirable car le changement sera plus facile à faire. Mais on est moins satisfait que dans le premier cas où le changement est complétement réalisé. Aussi, au lieu d'ajouter w_c, on ajoute $w_c * q * \delta/2$ à w_n ou à w_l, où q est un paramètre compris entre 0 et 1, strictement plus petit que 1, et δ est la diminution de la valeur absolue du champ. La valeur de δ dépend de t_i.

Elle est 2 si on a fait un changement de poids, et 1 sinon. On utilise le facteur $\delta/2$ pour tenir compte du fait qu'un progrès du champ de 2 vers la valeur correcte, est meilleur qu'un progrès de seulement 1.

L'essai numéro t_3 permet de tenir compte d'une information pertinente dans le seul cas où on effectue deux changements de poids: Un changement de poids sur le lien l qui relie n vers c, et un autre changement de poids en amont de n pour modifier l'activité de n. D'un autre côté, l'information calculée pendant t_1 et t_2 est pertinente pour un seul changement de poids. Comme l'information calculée pendant l'essai t_3 est moins précise, on multiple les quantités qui sont rétro-propagées par q.

Une fois que tous les exemples ont été traités, les w_l et r_l correspondant à tous les exemples sont additionnés. On calcule une fitness f_l pour chaque lien qui vaut:

$$f_l = r * r_l - w * \log(1 + w_l) + d * d_l \tag{7.1}$$

Les coefficients positifs r, d, w sont des paramètres de l'apprentissage. Dans toutes les expériences, on obtient de bons résultats en utilisant toujours le même jeu de coefficients. La variable d_l contient la profondeur du lien l qui est la longueur du chemin minimal entre ce lien et une unité de sortie. Les deux liens de tout le réseau qui ont la fitness la plus basse, sont sélectionnés. Un de ces deux liens est choisi pour être modifié. Le lien qui a la fitness la plus basse est choisi avec une plus grande probabilité que l'autre, car on veut le modifier. On note cette probabilité p. On a donc $p > 0,5$. La performance du réseau de neurones dont un poids a été modifié est calculée. Cette performance est ici, simplement le nombre de sorties correctes du réseau, divisé par le nombre d'exemples et aussi par le nombre de neurones de sorties. Si la performance augmente, la modification de poids est acceptée. Sinon, la modification peut encore être acceptée, mais cette fois avec une probabilité $\exp(-\Delta/T)$. T est une température, Δ est la diminution de la performance. D'un cycle de l'apprentissage à l'autre, T est diminué, de façon à faire du recuit simulé. Si le changement est accepté, alors le nouveau poids qui a été ajouté est gelé pendant e cycles, où $e = e_1 + e_2 * d_l$. e_1 et e_2 sont des paramètres positifs. A cause du terme $e_2 * d_l$ les liens qui sont proches des unités de sortie seront gelés pendant un temps plus long. Cela contrebalance le fait qu'ils sont davantage susceptibles d'être modifiés, à cause du terme $d * d_l$ qui apparaît dans l'expression de la fitness. Ainsi, l'algorithme commence par modifier les poids des unités près des unités de sortie, ce qui est préférable car l'information y est plus pertinente. En effet, plus on s'éloigne des unités de sortie, plus les calculs émanant de différentes unités interférent, et plus les imprécisions sur les calculs s'ajoutent. L'algorithme continue en essayant de modifier les poids d'unités cachées de plus en plus profondes, grâce au gel.

7.3 Simulation avec l'algorithme d'apprentissage Switch de base

On décrit maintenant un jeu d'expériences, durant lesquelles on a essayé d'apprendre respectivement la symétrie, la parité et le décodeur. Pour chacune de ces fonctions cibles, on considère un code cellulaire (CE) élaboré sans machines, qui produit une famille de réseaux de neurones ayant l'architecture et les seuils convenables. On teste l'algorithme switch sur les quatre premiers réseaux de la famille. Ces réseaux représentent un banc d'essai qui est pertinent, puisque l'algorithme d'apprentissage Switch est destiné à être utilisé en étant combiné avec le CE. On a représenté le quatrième réseau avec les poids convenables à la figure 7.3. Comme ces architectures sont minimales, il n'y a qu'une seule façon de choisir les poids correctement. Nous avons utilisé deux modes

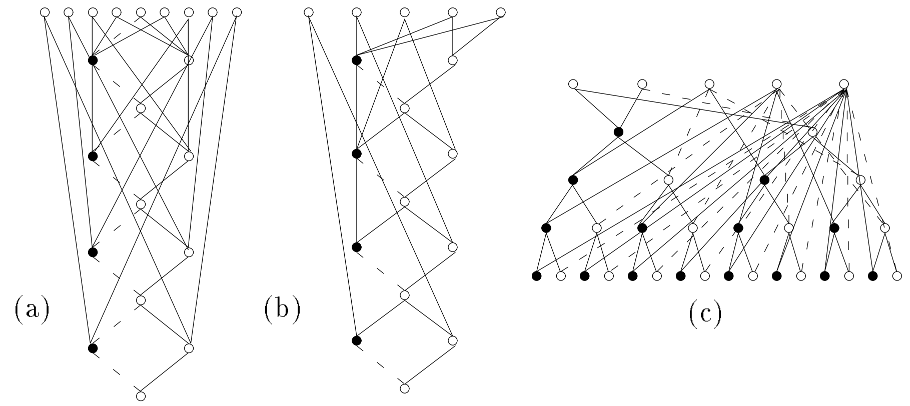

Figure 7.2: (a) la symétrie (b) la parité (c) le décodeur. Les cercles représentent des neurones. Si le seuil est 0, le cercle est vide, si le seuil est 1, le cercle est peint en noir. Une ligne continue indique un poids de 1, une ligne de tirets indique un poids de -1.

différents pour l'initialisation des poids. Dans la premier mode, tous les poids sont positionnés à la valeur 1, dans le deuxième, on positionne les poids sur une valeur aléatoire ± 1.

Nous cherchons à mettre en évidence l'influence du paramètre q. Si q vaut 0, l'algorithme utilise seulement les essais t_1 et t_2. Il calcule une information qui est pertinente si un seul changement de poids est effectué. Si q est strictement positif, l'algorithme calcule une information qui est pertinente si au moins deux changements de poids sont effectués. En d'autres termes, la fitness d'un lien indique si le réseau de neurones va augmenter sa performance avec deux modifications de poids: la valeur du poids de ce lien, et la valeur du poids d'un certain autre lien particulier. Dans toutes les expériences décrites, on utilise les valeurs de paramètres suivantes: $r = 5, w = 5, d = 1, e_1 = 2, e_2 = 1, p = 0.8$. La température initiale est 1, nous la divisons par 0.01 aprés chaque cycle. Ces paramètres donnent de bons résultats pour les trois jeux d'expériences. Nous avons effectué 32 essais pour chaque expérience, et nous avons calculé la moyenne et l'écart type (ET) du nombre de cycles d'apprentissage nécessaires pour atteindre la solution. L'écart type de la moyenne peut être obtenu en divisant ET par $\sqrt{32}$. On veut que l'apprentissage soit rapide, on ne veut pas qu'il soit convergent avec certitude. Nous fixons donc le nombre maximum de cycles d'apprentissage à 40. Dans le tableau 7.1, nous présentons dans cet ordre, la valeur de q, le mode d'initialisation des poids, le numéro du réseau qui est L, le nombre d'essais réussis, et enfin, le nombre moyen de cycles utilisés par les essais réussis, avec l'écart type ET.

Dans le premier jeu d'expériences, le GA apprend la fonction symétrie. Le réseau de neurones N_L calcule la symétrie de $2 * L + 1$ entrées, il a $8 * L$ connexions, $2 * L$ couches; $2 * L$ poids seront modifiés par l'apprentissage, dans le cas où tous les poids initiaux sont égaux à 1. La symétrie de $2 * L + 1$ bits calcule 1, si et seulement si l'entrée est symétrique, et si le bit du milieu vaut 1. La fonction étudiée suivante est la parité. Le réseaux de neurones N_l a $L + 1$ entrées, $2 * L$ couches, $6 * L$ connexions, et seulement L poids vont être modifiés si tous les poids sont initialisés à 1. La parité renvoie la somme de ses entrées modulo 2. Avec la fonction décodeur, N_L a $L + 1$ unités d'entrée, 2^L unités de sortie, L couches , $2^{L+2} - 4$ connexions, $2^L - 1$ poids doivent être modifiés. Si la première entrée est 0, le décodeur calcule 0; sinon l'unité de sortie dont le numéro est codé sur les L bits d'entrée suivants, a son activité qui vaut 1. Les activités de toutes les autres unités de sortie valent 0. Les résultats avec le quatrième réseau N_4, pour le problème du décodeur, sont

q	0.00								0.02							
poids	un				aléatoire				un				aléatoire			
L	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
cible	simétrie															
succés	32	27	30	22	31	18	6	2	32	32	31	28	32	30	14	12
cycles	2.8	4.4	7.4	13	11	20	21	8	4.3	5.8	11	12	9.4	18	30	30
ET	2.2	1.6	4.2	5.5	7.1	10	9.6	12	5.5	4.2	7.1	7.2	8.6	8.6	6.3	6.9
cible	parité															
succés	32	32	32	32	30	18	8	2	32	32	32	32	32	32	23	18
cycle	2	2.9	4.4	6.5	6.2	16	17	2	1.7	2.4	3.2	4.7	6.8	13	25	21
ET	3.5	3	3	6	5.6	11	9.2	5.5	2.9	2.3	0.7	3.5	8.4	7.7	9.9	9.0
cible	décodeur															
succés	32	32	32	32	32	32	13	0	32	32	32	32	32	28	15	3
cycle	1	3	7	15	3.9	5.5	4.4	-	1.6	3	7	15	3.6	7.2	14.2	11.5
ET	0	0	0	0	5.1	2.4	8.3	-	2.1	0	0.3	0.3	5.4	4.4	6.8	12.3

Tableau 7.1: Résultats expérimentaux avec l'algorithme Switch de base.

très médiocres lorsque les poids sont initialisés de manière aléatoire. Cela vient probablement du fait que le réseau a 60 poids. L'apprentissage doit modifier en moyenne 30 poids, ce qui est bien plus que dans les autres cas.

Nos expériences montrent que commencer avec des poids initiaux égaux à 1 aide l'apprentissage. Quand il y a beaucoup de poids -1, les activités des neurones ont une plus grande probabilité d'être nulles, l'apprentissage ne dispose plus d'informations exploitables. Nos expériences montrent aussi qu'utiliser un coefficient q non nul permet d'augmenter significativement les performances, dans le cas de l'initialisation aléatoire. Dans ce cas, l'information supplémentaire apportée en positionnant q à 0.02 est précieuse car peu d'informations sont disponibles.

Nous avons présenté un algorithme d'apprentissage dont le but est d'améliorer les réseaux de neurones produits par le GA avec le CE. Cet apprentissage exploite la forme particulière des réseaux de neurones générés par le CE. Ces réseaux de neurones ont un petit nombre de connexions. Pendant un cycle d'apprentissage, l'algorithme sélectionne un poids à modifier et le modifie. L'algorithme produit des poids valant ± 1, il est donc possible de faire du rétro-codage. (avec des poids valant d'autres valeurs entières, il serait encore possible de rétro-coder, mais il faudrait des opérateurs cellulaires plus complexes que `VAL+` et `VAL-`). La sélection du poids à changer est rendue plus difficile s'il y a des connexions redondantes. En outre, plus il y a de connexions, plus il y a de poids à modifier, et plus il faut faire de cycles d'apprentissage. En résumé, l'algorithme d'apprentissage est plus efficace avec un petit nombre de connexions, et même, avec le nombre minimal de connexions. Les réseaux de neurones produits par le GA ont un grand nombre de poids égaux à 1; l'apprentissage proposé produit des performances meilleures dans le cas où les poids sont initialisés à 1. Les réseaux de neurones produits par le GA ont de nombreuses couches (jusqu'à 8 dans nos expériences); l'apprentissage supporte autant de couches. En outre, l'apprentissage remplit certaines conditions imposées par l'usage du GA: Il doit être rapide, mais ne doit pas nécessairement être convergent à 100%. Nos expériences montrent qu'un taux de succés au dessus de 50% est réalisé avec un nombre moyen de 20 cycles d'apprentissage, sur des réseaux comportant jusqu'à 12 unités cachées.

7.4 Extension de l'algorithme d'apprentissage Switch

Dans les deux paragraphes précédents nous avons décrit un apprentissage qui convient aux réseaux de neurones générés par le GA en utilisant le CE. Cet apprentissage cherche des poids égaux à ± 1. Chercher des poids parmis $\{-1, 1\}$ est restrictif. Dans ce paragraphe, on étend cet apprentissage de façon à choisir les poids parmis $\{-1, 0, 1\}$. Positionner un poids à 0 revient à supprimer la connexion. Donc le rétro-codage est encore possible, en utilisant l' opérateur cellulaire `CUT` qui coupe une connexion.

L'algorithme d'apprentissage switch étendu est similaire à l'algorithme switch de base présenté dans les paragraphes précédents. A chaque cycle d'apprentissage, tous les exemples sont propagés à travers le réseau. On calcule deux variables positives pour chaque neurone n: r_n et w_n et 6 variables positives pour chaque lien l: $r_l[v]$ et $w_l[v]$, $v \in \{-1, 0, 1\}$. En fait on ne considére pas $r_l[v_0]$ et $w_l[v_0]$, où v_0 est la valeur courante du poids. Soit o une unité de sortie, si l'activité de o est la sortie désirée, on donne à r_o la valeur 1 et à w_o la valeur 0, sinon, on donne à r_o la valeur 0 et à w_o la valeur 1. Pour une unité cachée h, les variables r_h et w_h sont évaluées pour indiquer si l'activité courante calculée par h est correcte. si r_h est élevée, une modification de l'activité de h est susceptible de diminuer la performance du réseau de neurones sur l'exemple en cours de traitement. Si w_h est élévée, c'est le contraire, une modification de l'activité de h est susceptible d'augmenter la performance du réseau de neurones. De façon similaire, pour chaque lien l, si $r_l[v]$ (resp. $w_l[v]$) est élevé, positionner la valeur du poids de l sur v, est susceptible d'augmenter (resp. de diminuer) la performance du réseau de neurones. Les calculs de toutes ces variables se font en traitant les neurones les uns après les autres, dans le sens contraire de la propagation des activités. On commence par les unités de sorties, comme pour la rétro-propagation, et on finit par les unités d'entrée. L'algorithme décrit à la figure D.15 dans l'appendice, montre comment traiter un neurone. Soit c le neurone en cours de traitement, il y a trois façon différentes possibles pour modifier l'activité de c:

- t_1 Modifier le poids d'un lien entrant l.
- t_2 Modifier l'activité d'un neurone n voisin en entrèe.
- t_3 Modifier les deux en même temps.

L'algorithme d'apprentissage essaye successivement t_1, puis t_2 ou t_3 suivant la valeur de l'activité de n. Pour chaque essai t_i, il analyse l'effet de la modification sur le neurone c. Supposons que l'effet soit désirable. L'algorithme augmente w_n si l'activité de n a été modifiée, ou $w_l[v]$ si le poids de l a été changé en v. Sinon supposons que l'effet soit mauvais. L'algorithme augmente w_n si l'activité de n a été modifiée, ou $w_l[v]$ si le poids de l a été changé en v. Il y a trois effets possibles suivant les changements du champ de c:

- e_1: Le champ de c a changé de signe (par convention, ici, zéro a le signe -). Ainsi l'activité de c change. Si $w_c > 0$, il faut que l'activité de c change avec une pondération w_c. On enregistre le fait que l'essai t_i est avantageux avec une pondération w_c. Si t_i a modifié l'activité de n, on ajoute w_c à w_n, si t_i a positionné le poids de l sur v, on ajoute w_c à $w_l[v]$.

  Si $r_c > 0$ il ne faut pas que l'activité de c change avec une pondération r_c. On enregistre le fait que t_i est désavantageux avec une pondération r_c. On ajoute r_c à r_n ou bien à $r_l[v]$ suivant t_i

- e_2 Le champ augmente en valeur absolue. Si on veut que l'activité de c change, cet effet est indésirable car le changement sera plus difficile à faire. On ajoute w_c à r_n ou $r_l[v]$ suivant t_i.

- e_3 Le champ diminue en valeur absolue. Si on veut que l'activité de c change, cet effet est désirable car le changement sera plus facile. On ajoute donc w_c à w_n ou $w_l[v]$ suivant t_i. Si on ne veut pas que l'activité de c change, cet effet est indésirable, parce que l'activité de c est moins stable et pourrait être changée plus facilement. On ajoute r_c à r_n ou $r_l[v]$ suivant t_i.

Jusqu'à maintenant, l'algorithme d'apprentissage Switch ressemble à une rétro-propagation standard. Cependant, il y a une différence. Les quantités r_c et w_c qui sont rétro-propagées, sont multipliées par un coefficient qui est utilisé pour pondérer l'importance relative du changement de poids ou d'activité correspondant. L'algorithme d'apprentissage switch de base utilise aussi des coefficients, mais de façon moins précise. En effet, l'algorithme switch étendu considére davantage de cas. Les coefficients sont produits en utilisant 6 paramètres p_i, $i = 1, \ldots, 6; 0 < p_i < 1$. Chacun de ces paramètres correspond à une heuristique particulière. Le paramètre p_1 est utilisé pour changer de préférence l'activité d'un neurone de faible stabilité. Cette sorte de stratégie a été proposée par Gordon dans [8]. Elle a des bases théoriques et expérimentales. En fait une preuve issue de la physique statistique montre que si l'on veut minimiser le nombre d'erreurs du réseau, il faut changer de préférence l'activité d'un neurone ayant une faible stabilité. Les paramètres p_2, p_3 et p_4 pondérent les importances relatives des effets e_1, e_2, et les deux possibilités de e_3. On préfére apprendre de nouveaux exemples plutôt que de consolider les exemples déjà appris. Consolidé signifie "plus difficile à désapprendre". Cette heuristique est réalisée en choisissant $p_3 >> p_4$. L'activité d'un neurone d'entrée est fixe. On peut donc mieux évaluer l'effet d'un changement de poids sur un lien qui sort d'une unité d'entrée. Le paramètre p_5 permet de prendre cela en compte. L'essai numéro t_3 permet de tenir compte d'une information qui est pertinente à la condition qu'au moins deux changements de poids soient effectués: un changement de poids sur le lien l qui relie n vers c, et un autre changement de poids en amont de n pour modifier l'activité de n. D'un autre côté, l'information calculée pendant t_1 et t_2 est pertinente pour un seul changement de poids. Comme l'information calculée pendant l'essai t_3 est moins précise, on multiplie les quantités rétro-propagées par p_6. Le paramètre p_6 est l'équivalent du paramètre q utilisé dans l'apprentissage switch de base. L'effet de e_3 est plus important si le changement du champ à pour valeur 2 que si sa valeur est de 1. On multiplie donc les quantités rétro-propagées par $\delta/2$, où δ est la valeur absolue de la différence entre l'ancien champ et le nouveau champ. L'algorithme dans l'appendice D.15 peut être implanté avec 8 multiplications, 7 additions et 4 branchements conditionnels pour le traitement de chaque lien. La méthode pour cette implantation consiste à calculer à l'avance tous les coefficients pour tous les cas de figures possibles, c'est à dire: tous les champs possibles, et toutes les valeurs des activités de n et des poids de l possibles.

Une fois que tous les exemples ont été traités, les $w_l[v]$ et $r_l[v]$ correspondant à tous les exemples sont additionnés. On calcule une fitness $f_l[v]$ pour chaque lien $l, v \in \{-1, 0, 1\} \setminus \{v_0\}$, où v_0 est la valeur courante du poids. On utilise la formule suivante:

$$f_l[v] = w * w_l[v] - r * r_l[v] - d * d_l \tag{7.2}$$

Les coefficients positifs r, d, w sont des paramètres de l'apprentissage. La fitness $f_l[v]$ mesure l'intérêt qu'il y a à remplacer par v la valeur du poids de l. La variable d_l contient la profondeur du lien l, qui représente la longueur du chemin minimal entre ce lien et une unité de sortie. La fitness f_l d'un lien est le maximum des $f_l[v]$. Sont sélectionnés les deux liens dans tout le réseau, qui ont la fitness la plus basse. Un de ces deux liens est choisi pour être modifié. Le lien qui a la fitness la plus basse est choisi avec une probabilité $p > 0.5$. Ce lien étant le moins adapté, il aura alors une plus grande probabilité d'être modifié. Le poids du lien selectionné est positionné sur la valeur v telle que $f_l[v]$ soit la fitness maximum. Aprés le changement de poids, la performance du réseau

target	parité						symétrie					
weights	un			aléatoire			un			aléatoire		
L	1	2	3	1	2	3	1	2	3	1	2	3
succés	32	32	32	32	21	7	32	32	31	32	15	8
cycles	3	6	17	8	22	24	3	7	16	16	41	34
ET	1	1	7	6	24	22	0	4	12	14	22	26

Tableau 7.2: Résultats expérimentaux avec l'algorithme Switch étendu.

avec le poids modifié est calculée. Cette performance est le nombre de sorties correctes du réseau, divisé par le nombre d'exemples, et aussi par le nombre de neurones de sorties. Si la performance augmente, la modification de poids est acceptée. Sinon, la modification peut encore être acceptée, mais avec une probabilité $\exp(-\Delta/T)$. T est une température, Δ est la diminution de la performance. D'un cycle à l'autre, T est diminué, de façon à faire du recuit simulé. Si le changement est accepté, alors le nouveau poids qui a été ajouté est gelé pendant e cycles, ou $e = e_1 + e_2 * e_3^{d_l}$ e_1, e_2 et e_3 étant des paramètres positifs. A cause du terme $e_2 * e_3^{d_l}$ les liens qui sont près des unités de sortie seront gelés pendant un temps plus long. Cela contrebalance le fait qu'ils sont davantages susceptibles d'être modifiés, à cause du terme $d * d_l$ qui apparaît dans l'expression de la fitness.

7.5 Simulation avec l'algorithme d'apprentissage Switch étendu

On décrit maintenant un ensemble d'expériences durant lesquelles le GA apprend respectivement la parité et la symétrie. Pour chacune de ces fonctions cibles, on considére un code cellulaire (CE) trouvé manuellement, qui produit une famille de réseaux de neurones telle que certaines connexions doivent être supprimées par l'apprentissage. On teste l'algorithme switch sur les trois premiers réseaux de la famille. Ces réseaux représentent un banc d'essai pertinent, puisque l'algorithme d'apprentissage Switch est destiné à être utilisé en étant combiné avec le CE. On a représenté le troisième réseau avec les poids corrects trouvés par l'apprentissage à la figure 7.3. Nous avons utilisé deux modes différents pour l'initialisation des poids. Dans le premier mode tous les poids sont positionnés sur 1, dans l'autre mode, on positionne les poids sur une valeur aléatoire ± 1. Pour la symétrie les paramètres sont: $p_1 = 0.3$, $p_2 = 0.5$, $p_3 = 0.04$, $p_4 = 0.006$, $p_5 = 0.17$, $p_6 == 0.02$, $r = 5$, $w = 7$, $p = 0.8$, $d = 0.5$, $e_1 = 2$, $e_2 = 1$, $e_3 = 0.7$. Nous avons gardé les mêmes paramètres pour la parité, exceptés $w = 6$, $d = 2$, $p_1 = 0.7$. La température initiale est 1, elle est divisée à chaque cycle par 1.01. Nous avons fait 32 essais pour chaque expérience, et nous avons calculé la moyenne et l'écart type du nombre de cycles d'apprentissage nécessaires pour atteindre la solution. Comme ce problème est plus difficile que le problème précédent, on fixe le nombre maximum de cycles d'apprentissage à 100 cycles. Dans le tableau 7.2, on présente dans cet ordre, la fonction cible, le mode d'initialisation des poids, le numéro du réseau qui est L, le nombre d'essais réussis, le nombre moyen de cycles d'apprentissages utilisées par les essais réussis, et l'Ecart Type (ET), du nombre de cycles.

Dans le premier jeu d'expériences, le GA apprend la parité. Le réseau de neurones N_L posséde $L + 1$ entrées, $2 * L$ couches et $8 * L$ connexions. $3 * L$ poids doivent être modifiés si tous les poids sont initialisés à 1. La parité calcule la somme de ses entrées modulo 2.

La fonction étudiée suivante est la fonction symétrie. Le réseau de neurones N_L calcule la symétrie de $2 * L + 1$ entrées, il posséde $11 * L$ connexions et $2 * L$ couches. $4 * L$ poids doivent

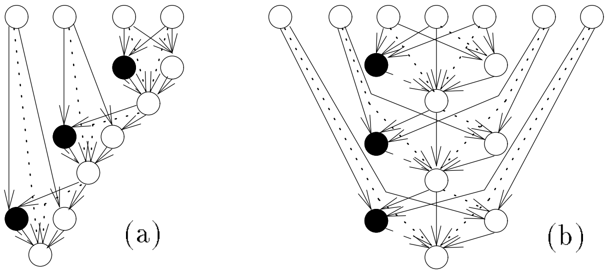

Figure 7.3: Le troisième réseau de neurones après apprentissage, pour la parité (a) et la symétrie (b). Les cercles représentent des neurones. Si le seuil est 0, le cercle est vide, si le seuil est 1, le cercle est peint en noir. Une ligne continue indique un poids de 1, une ligne de tirets indique un poids de -1, une ligne pointillée indique un poids 0.

être modifiés par l'apprentissage, dans le cas où tous les poids initiaux valent 1. La symétrie de $2 * L + 1$ bits calcule 1 si et seulement si l'entrée est symétrique, et le bit du milieu est 1.

Les résultats montrent qu'un taux de succés au dessus de 20% est réalisé avec un nombre moyen de 40 cycles, sur des réseaux de neurones comportant jusqu'à 8 unités cachées. L'apprentissage switch étendu n'est pas aussi performant que l'apprentissage switch de base. Cependant il faut tenir compte du fait que la complexité du problème étudié est plus grande. Supposons que l'on commence avec tous les poids initialisés à 1. Il y a 3 fois plus de poids à modifier dans le cas de la parité, et deux fois plus que dans le cas de la symétrie, En outre, il y a deux choix à faire au lieu d'un seul, pour la nouvelle valeur du poids. Comme dans le cas de l'apprentissage de base, les expériences montrent qu'il vaut mieux initialiser tous les poids à 1, plutôt qu'aléatoirement.

7.6 Simulation avec l'algorithme génétique

Dans cette série d'expériences, on combine l'algorithme génétique avec l'apprentissage Switch de base. On a utilisé la version simplifiée de l'apprentissage Switch qui cherche des poids ± 1, parce que les résultats du paragraphe précédent indiquent que la version étendue (qui cherche des poids dans $\{-1, 0, 1\}$) est moins efficace sur les problèmes particuliers que l'on veut résoudre. On a utilisé le GA parallèle décrit au chapitre 8, et on l'a fait tourner sur 4 noeuds d'un iPSC860. On compare les résultats obtenus avec l'apprentissage hebbien sur les poids de l'unité de sortie, et ceux obtenus avec l'apprentissage switch de base appliqué à tous les poids. On considère les problèmes de la parité et de la symétrie, de taille arbitrairement grande. Les trois combinaisons possibles entre apprentissage et évolution, étudiées au chapitre précédent, sont simulées. Ce sont les modes d'apprentissage: Baldwin, développemental et Lamarkien. Le premier tableau présente les résultats relatifs à la parité. Le deuxième tableau présente les résultats relatifs à la symétrie. Ici, la population est de 1024, le GA tourne 300 secondes avant de diagnostiquer un échec.

Ces expériences montrent clairement que l'apprentissage switch permet d'obtenir une meilleure accélération de la recherche génétique que l'apprentissage hebbien, spécialement pour le problème de la symétrie. Cela n'est pas surprenant, car apprendre seulement sur les poids de l'unité de sortie exploite une propriété spéciale de la parité; Pour la parité de taille 1, qui est un XOR à deux entrées, le seul poids -1 qui ait besoin d'être appris est un poids qui appartient à l'unité de

MODE	TEMPS	EVALUATIONS	TAUX DE SUCCÉS	NOMBRE D'ESSAIS
basic GA	110	11500	0.14	100
Apprentissage hebbien				
Baldwin	260	17700	0.07	100
Développemental	21	1800	0.64	76
Lamarkien	13	1300	0.79	63
Apprentissage Switch				
Baldwin	180	6500	0.08	100
Développemental	11	520	0.84	59
Lamarkien	10	610	0.86	58

Tableau 7.3: Les expériences avec la parité

MODE	TEMPS	EVALUATIONS	TAUX DE SUCCÉS	NOMBRE D'ESSAIS
basic GA	790	44000	0.27	71
Apprentissage hebbien				
Baldwin	960	48000	0.15	84
Développemental	360	20000	.57	14
Lamarkien	420	24000	0.5	40
Apprentissage Switch				
Baldwin	2280	45000	.12	170
Développemental	62	1400	1	23
Lamarkien	98	2850	0.74	27

Tableau 7.4: Les expériences avec la symétrie

sortie. Dans le codage cellulaire , les poids sont initialisés à la valeur 1. Donc les codes où tous les poids sont positifs sont plus compacts que ceux comportant des poids négatifs. En revanche, la symétrie est un problème pour lequel l'apprentissage doit modifier les poids des unités cachées, or l'apprentissage hebbien proposé ne le fait pas. C'est la raison pour laquelle l'apprentissage switch se révéle être plus performant sur le problème de la symétrie. Sur ce problème, on peut remarquer que l'apprentissage développemental est plus performant que l'apprentissage Lamarkien.

7.7 Conclusion

L'apprentissage switch a été conçu pour améliorer les réseaux de neurones produits par l'algorithme génétique avec le codage cellulaire. Pendant un cycle d'apprentissage, l'algorithme switch modifie un seul poids. L'algorithme de base cherche les poids dans $\{-1, 1\}$, la version étendue dans $\{-1, 0, 1\}$. L'apprentissage switch convient aux réseaux de neurones générés par le GA. Ces réseaux de neurones sont constitués de neurones qui ont peu de connexions entrantes, et donc, le nombre total de connexions d'un réseau est petit; il est de l'ordre du nombre de neurones. Ces réseaux de neurones ont beaucoup de poids égaux à 1, et beaucoup de couches. En outre, l'apprentissage switch est rapide, et le temps machine dépensé en apprentissage, est comparable au temps dépensé par le GA tout seul pour la recherche génétique et pour le développement des réseaux de neurones.

On a effectué deux sortes d'expériences pour mesurer l'efficacité de l'apprentissage switch: des mesures de convergence et de vitesse de convergence sur une architecture de neurones fixée, ayant déjà les seuils corrects, et des mesures de l'accéleration apportée au GA. Le premier type d'expériences avec l'apprentissage switch de base montre qu'un taux de succés au dessus de 50% est réalisé avec un nombre de cycles d'apprentissage avoisinant 20, sur des réseaux comportant jusqu'à 12 unités cachées.

L'apprentissage switch de base combiné avec le GA, améliore d'un facteur 13 le temps de résolution du problème de la symétrie, l'apprentissage switch de base permet d'accomplir un facteur d'accélération de 13. L'apprentissage hebbien n'améliore que de 1.5. L'apprentissage switch permet donc une accélération significative de la recherche génétique.

Chapitre 8

L'algorithme génétique parallèle mixte

8.1 État de l'art des algorithmes génétiques parallèles

L'algorithme génétique (GA) est une technique de recherche basée sur les principes de la génétique des populations. Le GA a déjà été décrit au paragraphe 1.1, dans l'introduction. Un GA conserve en mémoire une population de solutions pour un problème donné. Pour construire de nouvelles solutions, le GA sélectionne deux solutions de sa population (appariement), puis les recombine pour former un descendant, et finalement il mute le descendant. Ce processus itératif de sélection, recombinaison et mutation, fait évoluer une population de solutions initialement médiocres (aléatoirement construite) vers une population de solutions meilleures. Le processus s'arrête lorsqu'une solution suffisamment bonne est trouvée. On reporte le lecteur à [11] pour un livre de John Holland qui fait référence sur les GA.

Implanter un GA sur un système multi-processeur consiste à diviser la population en sous-populations, en affectant un processeur à chacune d'elle. Les processeurs échangent leurs meilleures solutions. Ces communications ont lieu, en respectant une **répartition spatiale** de la population. Différents modèles de GA parallèles ont déjà été étudiés. Müehlenbein distingue trois modèles dans [25]. Dans le modèle des *îles*, la population est divisée en un ensemble d' îles disposées aléatoirement. Les migrations d'individus entre les îles sont aussi aléatoires. Dans le modèle *"stepping stone "*, la migration a lieu seulement entre îles voisines. Le modèle d'*isolement par la distance*, aussi connu sous le nom de *massivement parallèle*, considère une distribution continue des individus. Dans ce modèle, les individus sont souvent placés sur une grille à deux dimensions et s'apparient avec leur voisinage.

8.2 Le GA parallèle mixte

Dans ce chapitre on introduit un nouveau modèle appelé le GA parallèle mixte qui combine le modèle "stepping stone" avec le modèle de l'isolement par la distance. Ainsi les avantages des deux modèles peuvent être retenus.

- L'appariement local permet d'obtenir un haut degré de ressemblance entre les individus qui se recombinent, comme le montre R. Collins et D. Jefferson dans ([4]). Si les parents sont très différents, ils risquent de produire un descendant qui n'est pas fonctionnel. Ainsi, un haut degré de similarité entre parents favorise la création d'individus intéressants.

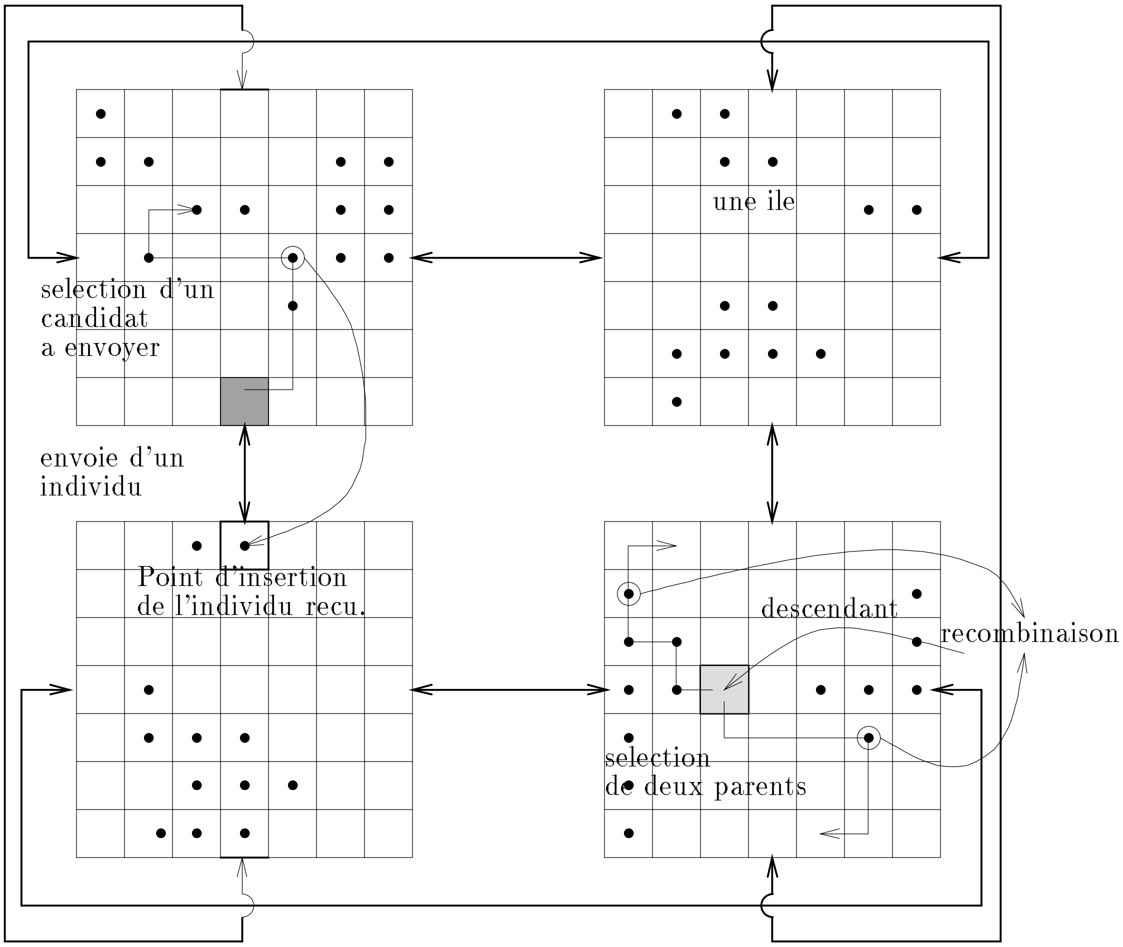

Figure 8.1: Structure spatiale de la population, pour le GA parallèle mixte: C'est un tore 2-D de grilles 2-D. Chaque petit carré représente un site. Un point dans un carré représente un individu occupant un site. Un cercle autour d'un point indique que l'individu représenté par le point est le meilleur rencontré au cours d'une marche aléatoire. Un carré gris représente un site qui a été aléatoirement choisi pour démarrer une marche aléatoire. Les carrés en trait gras représentent un site où l'on doit insérer un individu.

- L'utilisation d'îles séparées aide à maintenir la diversité génétique (H. Muehlenbein, M. Schomish, et J.Born. [26]). Les individus sur une île donnée sont relativement isolés des individus qui se trouvent sur une autre île. Donc chaque sous-population de chaque île peut explorer une région différente de l'espace de recherche. Cela permet d'éviter la convergence prématurée de la population vers une solution sous optimale, et donc, de mener une recherche plus globale.

Dans le GA parallèle mixte, les individus sont répartis sur des îles. Ces îles forment un tore 2D. Chaque île est associée à un processeur d'une machine MIMD (Mutiple Instruction Multiple Data). Un processeur peut envoyer des individus à ses quatre processeurs voisins. A l'intérieur d'une île, les individus sont arrangés sur une grille 2-D. La structure spatiale complète est un tore 2-D de grilles 2-D, représenté à la figure 8.1.

Les sites de la grille 2D ne sont pas tous occupés. La densité de la population est maintenue autour de 0.5. Ce qui signifie qu'un site donné est occupé avec une probabilité 0.5. L'appariement est fait comme suit: un site s est choisi aléatoirement sur la grille. A partir de ce site, deux marches aléatoires consécutives sont effectuées. Les deux meilleurs individus trouvés au cours de chacune de ces marches aléatoires, sont retenus. Avant la recombinaison, on vérifie que les deux parents sont différents. Le descendant est ensuite placé sur le site s. Si ce site est occupé par un individu i, i est placé sur un site voisin de s, choisi aléatoirement, s'. Si maintenant s' est occupé par i', i' est placé sur un site voisin de s', choisi aléatoirement, s''; et ainsi de suite jusqu'à ce qu'un site vide soit trouvé. Cette façon de procéder permet de simuler le modèle parallèle de l'isolement par la distance sur une machine qui n'est pas parallèle (i.e. sur un seul processeur).

Tanese [31] introduit deux paramètres de migration: l'intervalle de migration (le nombre de générations entre deux migrations consécutives) et le taux de migration (Le nombre d'individus qui migre à chaque migration). Cette approche est généralement suivie. Nous avons adopté une approche nouvelle ne nécessitant qu'un seul paramètre. Le GA renouvelle la population individu par individu plutôt que de renouveler toute la population en même temps. Chaque fois qu'un individu est créé, il peut être construit soit par la recombinaison de deux individus, soit par le moyen d'un échange. Le GA choisit de faire un échange avec une probabilité que l'on appelle le taux d'échange.

Quand un processeur échange un individu, un site s est sélectionné sur le bord de la grille 2D, une marche aléatoire commence à partir de ce site, le meilleur individu trouvé est envoyé au processeur voisin qui se trouve en face du bord de la grille 2D. Il est envoyé avec les coordonnées de s. Lorsqu'il est reçu, ces coordonnées sont lues, et l'individu est placé exactement sur le site qui se trouve en face de s. Ainsi les échanges sont effectués en respectant la topologie 2D de l'ensemble.

8.3 Une accélération super linéaire

Nous avons appliqué le GA au problème de la recherche d'un réseau de neurones booléen calculant la parité de 51 entrées. Nous avons codé le réseau de neurones en utilisant le codage cellulaire. Ce codage permet de développer une famille de neurones (N_L), le L^{eme} réseau calcule une fonction booléenne régulière de taille L comme par exemple, la parité de $L+1$ entrées. Müehlenbein [26] insiste sur les avantages que présente l'utilisation d'une procédure d'optimisation locale pour accélérer le GA. Dans le cas d'une génération de réseaux de neurones, on peut utiliser un algorithme d'apprentissage pour faire de l'optimisation locale. L'apprentissage switch de base a été utilisé. Cet apprentissage est appliqué au premier réseau de la famille de réseaux de neurones codé par chaque chromosome (solution) produit par le GA. L'information apprise est rétro-codée sur le chromosome et peut être utilisée pour développer les réseaux de neurones de taille croissante de toute la famille. Ce mode de combinaison entre recherche génétique et apprentissage appelé apprentissage développemental, a été étudié au chapitre 6.

La machine parallèle MIMD utilisée est un iPSC860 comportant jusqu'à 64 processeurs. Un noeud de cette machine fait 40 MIPS, ce qui équivaut à peu près à 1.5 Sparc 2 (SUN4). Les communications entre les noeuds sont asynchrones, et sont recouvertes par les calculs. Donc, le coût des communications est nul! Ces prédictions théoriques sont confirmées par les mesures faites pour chaque expérience. Si le GA n'arrive pas à trouver une solution après 20 secondes, on redémarre le GA à partir de zéro. Le temps pour trouver la solution comprend le temps dépensé dans ces échecs. La taille optimale de la population est fonction du nombre de processeurs utilisés. Pour une étude correcte de l'accélération, il est donc indispensable de tester plusieurs tailles de population pour un nombre de processeurs donné. La figure 8.2 montre que la taille optimale de la population varie en fonction du nombre de processeurs. Certains auteurs déclarent avoir trouvé des accélérations super-linéaires, mais leur résultat ne sont pas corrects, car ils choisissent une taille de population favorisant un grand nombre de processeurs. Un calcul d'accélérations, où l'auteur n'a pas cherché à optimiser la taille de la population présente peu d'intérêt.

La figure 8.3 montre que l'on obtient une accélération superlinéaire en dessous de 16 processeurs. Cela est possible parce que le GA parallèle n'a pas le même algorithme que le GA séquentiel. La pente de l'accélération diminue après 16 processeurs, non à cause du coût des communications qui reste nul, mais parce qu'une population de 128 individus est assez grosse pour résoudre le problème posé. Le GA parallèle mixte fonctionne efficacement, si les sous-populations comportent au minimum 8 individus. Avec un problème plus difficile à résoudre par le GA (où la taille des building block est plus grande), une taille de population plus grande serait nécessaire pour échantillonner correctement l'espace de recherche et les building block. On devrait observer une accélération super linéaire pour un nombre de processeurs encore plus grand.

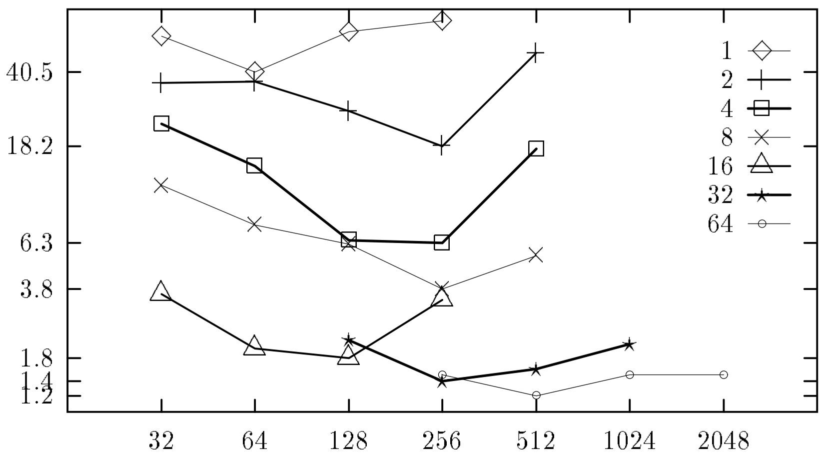

Figure 8.2: L'axe horizontal indique la taille totale de la population. (sur tous les processeurs). Cette taille est toujours une puissance de 2. L'axe vertical indique le temps moyen en secondes, pris par le GA parallèle mixte pour trouver un réseau de neurones calculant la parité de 51 entrées. La moyenne est faite sur dix exemples qui dépendent de la population initiale, cette dernière est en effet établie aléatoirement. Les tirets sur l'axe vertical indiquent les meilleurs temps pour chaque nombre de processeurs.

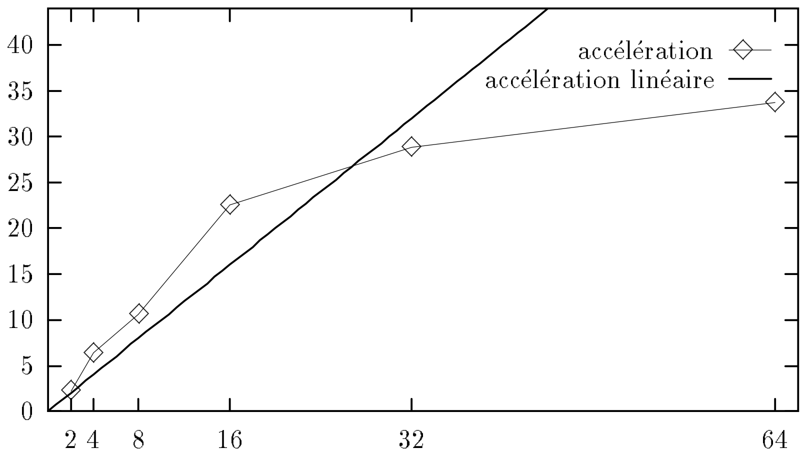

Figure 8.3: L'axe horizontal indique le nombre de processeurs utilisés, l'axe vertical montre l'accélération par rapport à l'utilisation d'un seul processeur.

Chapitre 9

La modularité

Ce chapitre décrit comment incorporer la modularité au codage cellulaire. D'abord nous montrons que la modularité est présente dans la nature, et que sa présence est la preuve que la nature utilise un langage génétique similaire pour coder le phénotype sur le génotype. Un tel langage permet de définir séparément des structures pour les réutiliser ensuite. On explique comment incorporer la modularité au codage cellulaire de base. Un algorithme génétique qui comporte toutes les améliorations apportées aux chapitres précédents est utilisé. le GA recherche un code modulaire qui fournit à la fois l'architecture et les poids ± 1 d'un réseau de neurones particulier, pour calculer la fonction décodeur de 10 entrées et 1024 sorties. Ce résultat suggère que le GA peut exploiter la modularité, pour trouver des architectures d'un niveau de complexité supérieur.

9.1 Validité biologique

Pourquoi les cinq doigts de la main se ressemblent ils? Pourquoi les vertèbres sont-elles presque identiques? Pourquoi le corps humain est-il symétrique? Pourquoi les poumons ont-ils une structure fractale? Parce que la nature utilise un langage de programmation procédural pour coder le développement du corps humain sur les chromosomes. La procédure "doigt" est définie une seule fois, elle est "appelée" dix fois. La procédure "vertèbre" est appelée autant de fois qu'il y a de vertèbres. La procédure "poumon" est récursive. La procédure de haut niveau "moitié de corps humain" est appelée deux fois. Si les doigts, les vertèbres, les poumons ont une structure bien identifiée, il n'en va pas de même pour le cerveau. La complexité du cerveau rend difficile l'analyse de sa structure. Toutefois, les expériences menées par Hubel [13] montrent qu'une partie du cerveau (concernée par la vision de bas-niveau) peut se décomposer en colonnes. Chaque colonne repére sur une zone particulière de l'image rétinienne, une zone contrastée noir/blanc, la frontiére entre le noir et le blanc est approximée par une ligne et fait un angle particulier. Ces colonnes se ressemblent donc, de la même façon que les doigts se ressemblent.

Pourquoi la nature utilise-t-elle un langage de programmation procédural? Un tel langage permet au programmeur de factoriser le code informatique, et d'obtenir des programmes concis. De la même façon il permet à la nature de factoriser le code génétique et d'obtenir un chromosome petit. De fait, la taille du chromosome humain est très petite, relativement à la quantité d'information nécessaire qu'il faudrait pour coder le cerveau, en représentant chaque connexion explicitement. La nature a probablement un autre intérêt à utiliser un langage procédural. Lorsque deux chromosomes se recombinent, il peuvent échanger des procédures, plutôt qu'un code chimique qui ne contient pas un sens intrinsèque. Les procédures peuvent être alors considérées comme des blocs de base ("building blocks" dans le jargon des algorithmes génétiques). L'hypothèse des blocs de bases, pour le problème qui nous intéresse, est la suivante: Un individu bien adapté doit comporter une collection de procédures correctes. Si une procédure correcte, en moyenne, aide l'individu à s'adapter à son milieu, le théorème des shémas garantit alors que les procédures correctes se reproduisent à grande vitesse par le jeu de la sélection et de la reproduction. Elles vont se retrouver dans le code de nombreux individus après plusieurs générations. Par le jeu du croisement, un individu pourra alors posséder toutes les procédures correctes et être bien adapté. Nous avons vu au chapitre 4 que le codage cellulaire est comme un langage de programmation procédural. L'argumentation qui précède plaide en faveur de la validité biologique d'une telle sorte de codage.

9.2 Comment incorporer la modularité au codage cellulaire

Pour que le codage cellulaire puisse être considéré comme un langage procédural, il faut compléter la version de base décrite au chapitre 2. Au paragraphe 3.4 on a montré que l'utilisation de l'opérateur JMP permet d'obtenir un codage cellulaire vérifiant la propriété de modularité (définie dans ce même paragraphe). On a testé cet opérateur JMP sans obtenir de succès. On a donc opté pour un autre type d'opérateur JMP en vue d'implanter la modularité. L'opérateur que nous allons introduire, est similaire à celui défini au chapitre 4.

Le code cellulaire doit comporter une liste d'arbres étiquetés au lieu d'un seul arbre, comme auparavant. Comme chaque arbre correspond à un sous-ensemble de règles de grammaire, il est appelé arbre grammaire, comme auparavant. Chaque arbre code un sous réseau qui sera inclus dans le réseau final. L'algorithme génétique devra automatiquement définir ces sous réseaux, et en inclure une ou plusieurs copies dans le réseau final. Koza, dans [18], décrit comment le GA peut automatiquement définir des fonctions, et insérer un ou plusieurs appels de fonction dans le programme principal qui est aussi trouvé par le GA. Koza appelle cela ADF (Automatic Definition of Function). On appellera un sous réseau automatiquement défini un "ADSN" (Automatic Definition of Sub Network). Chaque arbre grammmaire a un nom qui est aussi le nom du sous réseau codé. Ce nom est donné de façon systématique. Le nom de l'arbre grammaire *axiome* (celui qui est lu en premier) sera `main`, le nom des sous réseaux sera `ADSN0, ADSN1, ADSN2,` etc.... Au début de la réécriture, la cellule ancêtre lit la racine de l'arbre grammaire axiome `main`. Pour chaque arbre grammaire `ADSNi`, il existe un opérateur cellulaire du même nom, qui permet d'inclure le sous réseau `ADSNi`. Cet opérateur est analogue à l'opérateur `JMP x` défini au paragraphe 4.5. La seule différence est que l'opérateur `ADSNi` n'a pas d'arguments. Il y a autant d'opérateurs possibles qu'il y a d'arbres grammaire. Quand une cellule lit l'opérateur `ADSNi`, elle positionne sa tête de lecture sur la racine de l'arbre grammaire qui code le sous réseau `ADSNi`. Si un sous réseau s'inclut lui-même, le développement continue indéfiniment. Cela revient à utiliser l'opérateur `REC` défini à la section 2.2. Dans ce cas particulier, la cellule exécute l'algorithme suivant qui a déjà été exposé pour `REC`:

```
vie:=vie-1
si (vie > 0) deplacer la tete de lecture sur la racine de l'arbre ADSNi.
sinon Deplacer la tete de lecture sur le sous-arbre du noeud courant.
```

La figure 9.1 montre un code modulaire comprenant trois arbres grammaire, et son développement en un réseau qui calcule le décodeur. Cette figure est complexe. On prie le lecteur de se reporter au chapitre 2 pour un exemple plus simple de développement et une explication détaillée du codage cellulaire. On présente ici un code plus complexe pour trois raisons: 1) Montrer la puissance du code cellulaire; 2) Présenter un exemple de fonction booléenne (la fonction décodeur),

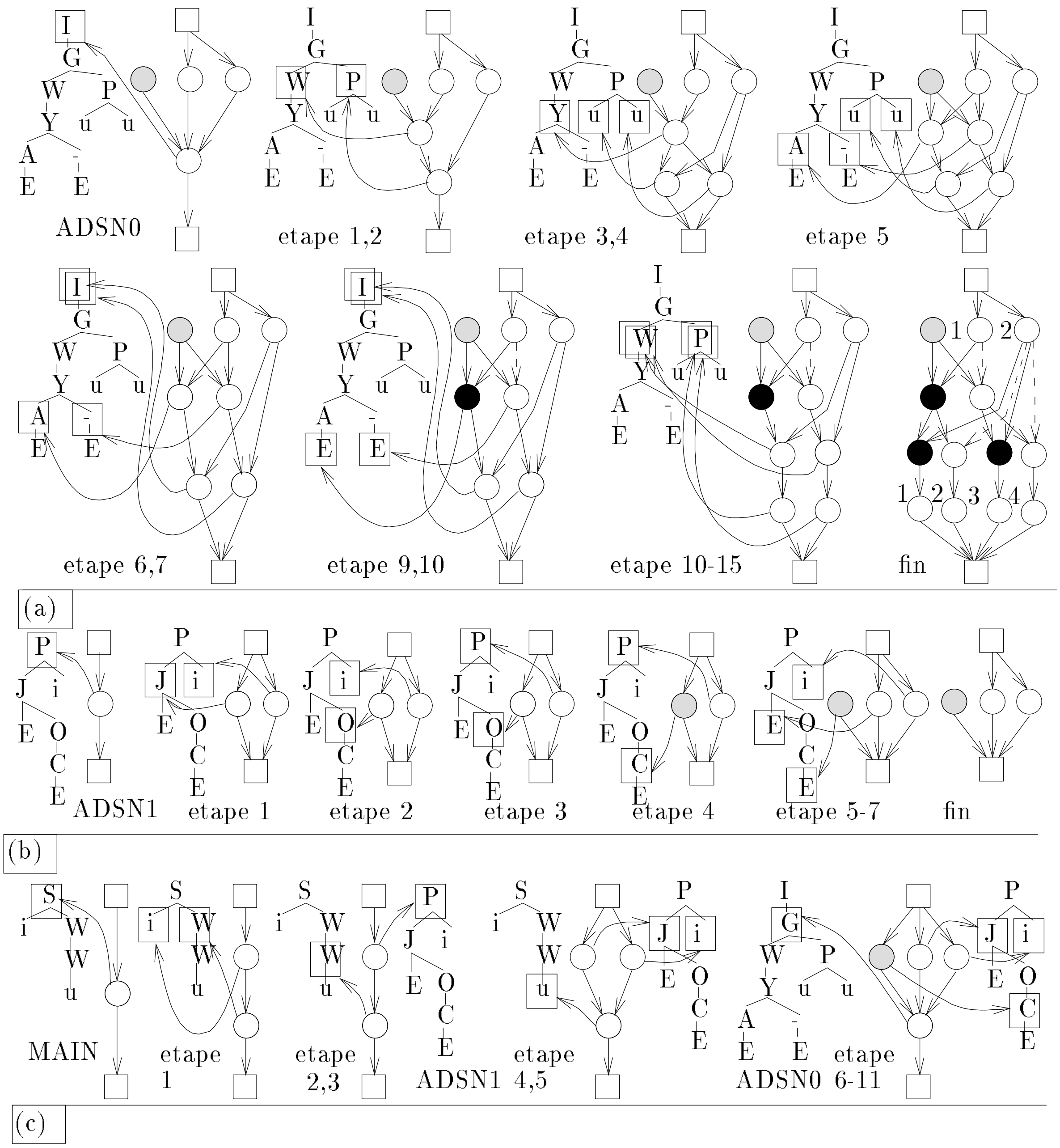

Figure 9.1: Le réseau de neurones pour le décodeur à deux entrées. (a) Développement du sous réseau ADSN0 en commençant le développement à partir d'un graphe de cellules contenant déjà le réseau ADSN1. (b) développement du réseau ADSN1 (c) les 11 premières étapes du développement du décodeur à deux entrées. A l'étape 11, le graphe de cellules est similaire au graphe de cellules initial montré en (a), les dernières étapes peuvent donc être déduites de (a). Les conventions de représentation sont les conventions habituelles, à l'exception du disque gris qui indique que le seuil vaut -1. On utilise une notation abrégée pour les opérateurs cellulaires: `PAR, SEQ, WAIT, END, INCBIAS, DECBIAS, VAL-,VAL+, INCLR, DECLR, CUT, XSPL, SEP, ADF1, ADF0` qui sont remplacés par `P, S, W, E, A, O, -, +, I, D, C, G, Y, u, i,`. Les cellules pointeur entrée et pointeur sortie sont représentées par un simple carré.

pour laquelle une décomposition modulaire est nécessaire afin de coder la famille de réseaux de neurones correspondante; 3) Permettre de mieux comprendre comment le GA peut trouver une solution au problème du décodeur.

La figure 9.1 (c) montre que le réseau de neurones calculant le décodeur comprend deux sous-réseaux reliés séquentiellement. Le premier sous réseau `ADSN1` développe les unités d'entrée, ainsi qu'une unité qui produit la constante 1. Ce sous réseau s'inclut lui-même une seule fois, sa taille est donc $O(L)$. Le deuxième sous réseau `ADSN0` utilise une à une les unités développées par `ADSN1`. Il s'inclut lui même deux fois, sa taille est donc $O(2^L)$. Le réseau de neurones qui calcule la fonction décodeur de 2 entrées est développé dans la figure 9.1. Le développement peut être fait pour un entier L arbitrairement grand. En outre, le code est modulaire suivant la définition donnée au paragraphe 3.4. En général, un code cellulaire est toujours modulaire puisqu'il utilise un arbre grammaire par sous réseau de neurones.

On utilise deux nouveaux opérateurs de division cellulaire. Soit r la valeur du registre de lien. Dans la division graduelle notée $\hat{\mathrm{X}}$SPL, la première cellule fille hérite des r premiers liens d'entrée; la seconde cellule fille hérite des autres liens d'entrée et de tous les liens de sortie. La première cellule fille est connectée à la seconde avec un poids égal à 1. (étapes 2, 14, et 15, figure 9.1 (a)). Dans la division de séparation (notée `SEP`), les deux cellules filles héritent de tous les liens d'entrée, la première cellule fille hérite de la première moitié des liens de sortie, la deuxième cellule fille hérite de la deuxième moitié des liens de sortie (étape 5, figure 9.1 (a)). Les opérateurs `XSPL` et `SEP`, et leur microcode sont décrits à l'appendice D.5. On utilise aussi un autre opérateur de branchement `BLIFE2`. Lorsqu'une cellule lit `BLIFE2`, elle place sa tête de lecture sur le sous arbre droit si on effectue la première itération de l'arbre grammaire, sinon elle place sa tête de lecture sur le sous arbre gauche.

9.3 Simulation utilisant la modularité

Objectif:	Trouver le CE d'une famille de réseaux de neurones qui calculent le décodeur de taille arbitraire
Allèles de terminaux:	`END, ADSN0, ADSN1, CONST1`
Allèles de fonctions:	`PAR, SEQ, WAIT, VAL+, VAL-, INCLR, DECLR, INCBIAS, DECBIAS,`
	`CUT BLIFE2, XSPL, SEP`
points de Fitness :	vecteurs aléatoires de L bit à rentrer dans le réseau N_L.
	Le nombre de vecteurs est déterminé par le GA
Fitness brute	Nombre de Points de fitness
d'un réseau	où le réseau produit la sortie correcte
Fitness normalisée	Fitness brute divisée par le nombre de point de fitness
d'un réseau.	Elle est comprise entre 0 et 1
Fitness normalisée	La somme des fitness normalisées des réseaux développés. Elle est comprise entre 0 et 10.
Paramètres:	$M = 8092$, $t_m = 0.005$, $l_0 = 30$, $l_{max} = 5$
Prédicat de succès:	La fitness normalisée vaut 10, c'est à dire les 10 premiers réseaux sont développés et leur
	fitness est maximum,

Tableau 9.1: Tableau décrivant la recherche génétique d'une famille de réseaux de neurones pour résoudre le problème du décodeur jusqu'à 10 entrées. Au lieu du temps T, on reporte la taille de la population M qui, cette fois, est fixée dans cette expérience.

On essaie de trouver une famille de réseaux de neurones N_L, telle que N_L calcule la fonction décodeur de taille L. N_L doit avoir L unités d'entrée et 2^L unités de sortie. Si le code binaire d'un entier $i < 2^L$ est appliqué sur les unités d'entrée, l'unité de sortie de numéro i doit avoir une activité égale à 1, et toutes les autres doivent avoir une activité égale à 0. Ce problème est plus complexe que le problème de la parité ou que le problème de la symétrie (si on doit le résoudre en utilisant le codage cellulaire). Dans le paragraphe précédent, on a présenté un exemple de code pour un réseau de neurones calculant la fonction décodeur. Ce code utilise 3 arbres grammaire. Comparativement, la symétrie ou la parité peuvent se coder avec un seul arbre grammaire.

On va utiliser tous les résultats des chapitres précédents pour résoudre ce problème plus complexe. Ces améliorations sont: l'apprentissage développemental, l'apprentissage switch de base, le GA parallèle sur une machine parallèle iPSC860, comportant 64 noeuds. Dans le GA parallèle, on commence le développement des réseaux de neurones à partir de $N_{L_{min}}$,au lieu de N_1, en utilisant la règle de développement décrite au paragraphe 5.2. L_{min} varie régulièrement sur les 64 processeurs entre 1 et 3. Les processeurs pour lesquels $L_{min} = 3$ commencent directement par développer N_3 alors que ceux pour lesquels $L_{min} = 1$ commencent par N_1. Il y a trois fonctions de fitness différentes, selon que l'on commence le développement à partir de N_1, N_2 ou N_3. Chacune représente un aspect différent du problème. Ainsi chaque processeur développe une population en cherchant à générer les blocs de base neuronaux correspondant à un trait particulier, et la solution finale peut être trouvée à partir de la recombinaison de ces blocs, lorsque les individus sont échangés entre les processeurs. La diversité génétique est maintenue jusqu'à ce qu'un code qui développe à la fois N_1, N_2 et N_3 correctement, soit obtenue. Mais un tel code va probablement développer correctement N_L pour $L > 3$, parce que la récurrence est apprise. Ainsi, la diversité génétique peut être gardée jusqu'à ce qu'une solution soit trouvée.

Les arbres grammaire sont ordonnés. La mutation, ainsi que la génération d'arbres aléatoires, (pour la population initiale) sont implantés de façon à respecter une hiérarchie entre les arbres grammaire. La hiérarchie sur les arbres grammaire se traduit aussi par une hiérarchie sur les sous réseaux codés par chaque arbre grammaire. Un sous réseau donné ne peut inclure que des sous réseaux qui lui sont inférieurs ou égaux, relativement à cette hiérarchie. Cela empêche qu'il y ait des bouclages entre des arbres grammaire différents.

La fitness des réseaux de neurones n'est plus basée sur la théorie de l'information, mais simplement sur le nombre de calculs corrects effectués par le réseau. Cela est cohérent avec l'apprentissage switch, qui optimise aussi le nombre de calculs corrects.

Les génomes recombinés par le GA comprennent deux arbres grammaire `ADSN0` et `ADSN1`. L'arbre grammaire `main` est prédéfini comme étant `SEQ (ADSN1 )( WAIT ( WAIT( ADSN0)))`. Ce code indique la structure générale du réseau. Il est composé de deux sous réseaux `ADSN1` et `ADSN0` connectés séquentiellement. Le développement de `ADSN0` est retardé deux fois, parce que `ADSN0` a besoin des neurones de `ADSN1` pour commencer son développement. On a défini un sous-réseau appelé `CONST1` comprenant un seul neurone qui génére la constante 1, ou bien qui calcule la fonction identité, suivant la valeur du registre de lien. Le code de `CONST1` est: `BLIFE2(END )(DECBIAS (VAL- (CUT ( END))))` Le tableau 9.1 décrit toutes les caractéristiques du problème de la génération d'un réseau de neurones calculant la fonction décodeur. La figure 9.2 montre le génome trouvé par le GA. Le réseau de neurones correspondant de numéro L calcule le décodeur de L entrées avec $4 * 2^L - 2$ unités cachées et $15 * 2^{L-1} - 6$ liens. Il est clair que c'est une combinaison de $2^L - 1$ sous réseaux. On a testé N_L jusqu'à $L = 10$. Bien qu'il n'y ait aucune preuve, il paraît évident que le réseau N_L calcule le décodeur de L entrées pour un L arbitrairement grand, parce que le GA a appris la récurrence. La taille d'une sous population d'un processeur est 128. La taille totale de la population répartie sur les 64 processeurs est donc 8192. Le nombre de générations moyenné sur les processeurs est de 130. Donc le nombre total d'individus traités est approximativement un million. Le temps machine utilisé est une heure. Cela équivaut à peu près à 4 jours de sun4, (station de travail sparc2). Le GA utilisé est donc très puissant.

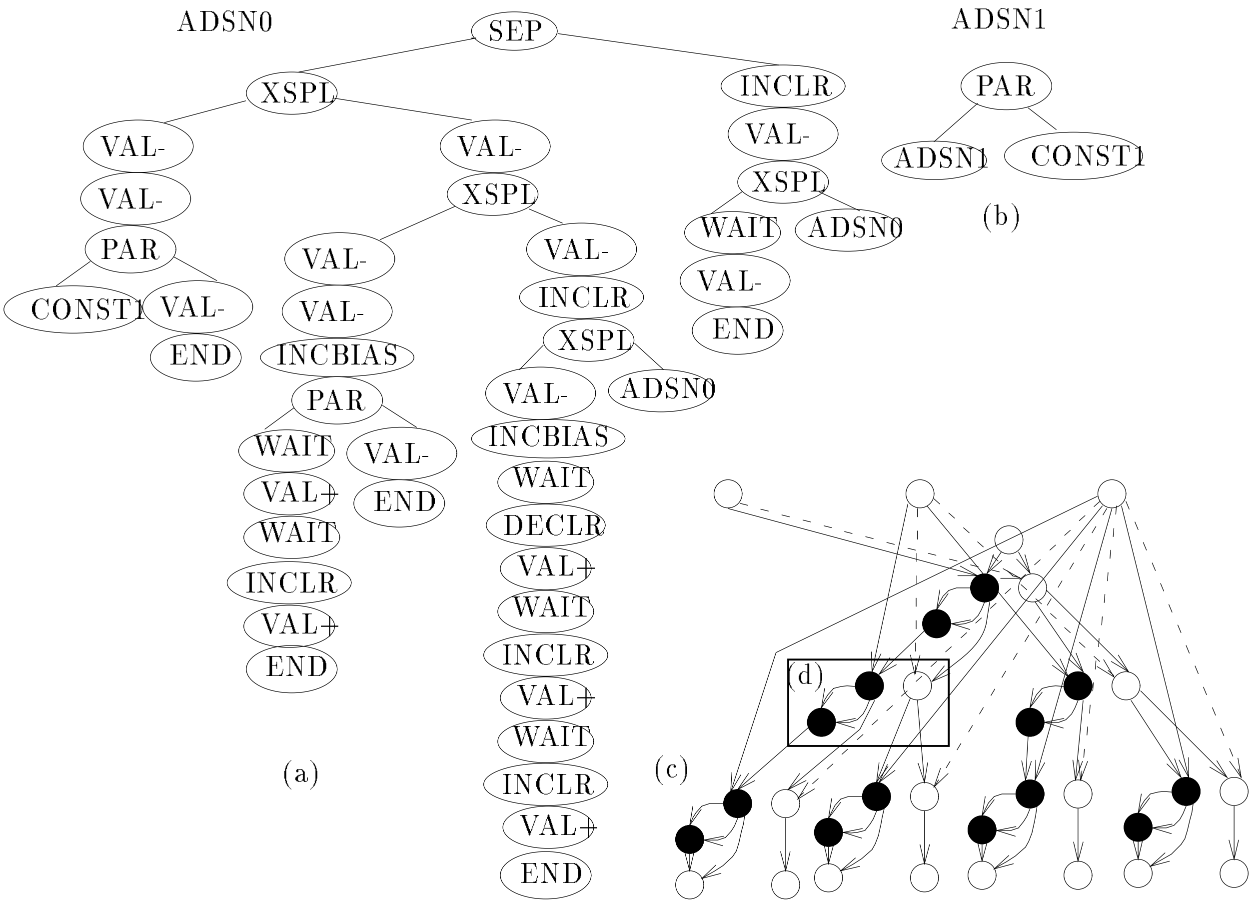

Figure 9.2: (a) et (b) le génome obtenu par le GA. (c) le réseau de neurones pour $L = 3$. (d) bloc de base neuronal recopié en plusieurs endroits.

9.4 Conclusion

Dans ce chapitre, on a précisé comment implanter la modularité dans le codage cellulaire pour pouvoir l'utiliser avec l'algorithme génétique. Cette amélioration permet de coder les réseaux de neurones d'une façon modulaire, suivant la définition de la modularité donnée au paragraphe 3.4. On a montré comment développer un réseau de neurones booléen pour la fonction booléenne décodeur, en utilisant deux sous réseaux de neurones. Les expériences montrent que le GA réussit à faire évoluer simultanément les deux sous-réseaux, et à générer un réseau de neurones calculant la fonction décodeur de 10 entrées et 1024 sorties. Cependant, le GA a dû être aidé avec deux morceaux de code définis à la main. Le premier décrit comment combiner les sous réseaux entre eux, le deuxième code un sous réseau bloc de base, très simple, qui peut être utilisé par le GA. Cette façon d'ajouter de l'information peut être vue comme une technique, et elle est intéressante en elle même. Le décodeur est une fonction booléenne de complexité supérieure aux fonctions parité et symétrie déjà étudiées. L'architecture correspondante, bien que régulière, est plus compliquée. Pour la parité et la symétrie, les architectures décrites à la figure 5.9 et 5.10 sont organisées en couches, le nombre de liens sortant d'un neurone est borné par 2, la complexité est linéaire. Pour le réseau de neurones calculant le décodeur montré à la figure 9.3, les liens sont arrangés en une structure d'arbre binaire, le nombre de liens sortant d'un neurone n'est pas borné, la complexité est exponentielle. C'est à cause de cette complexité que nous avons dû utiliser un GA particuliérement étudié, préservant la diversité génétique. Le GA comporte toutes les améliorations définies dans les chapitres précédents.

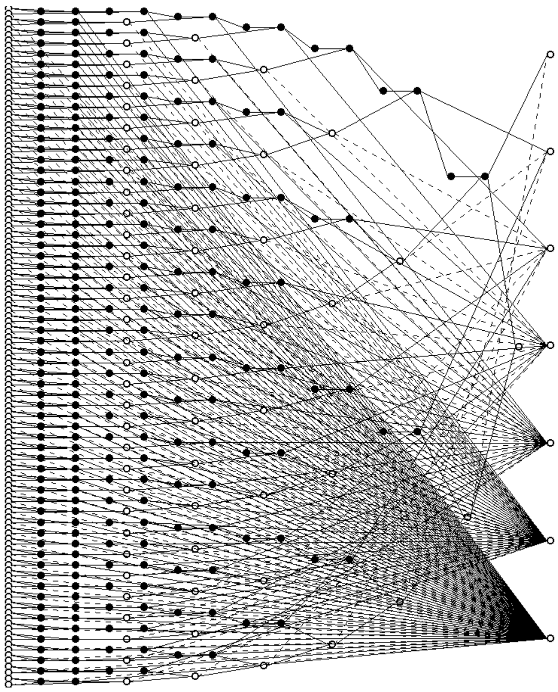

Figure 9.3: Le réseau de neurones trouvé par le GA pour calculer la fonction décodeur de 7 unités d'entrées et 128 unités de sortie.

Chapitre 10

Conclusion: GA + CE = programmation génétique de réseaux de neurones

A travers cette thèse, nous avons défini et étudié le codage cellulaire (CE). C'est une méthode pour coder des familles de réseaux de neurones pareillement structurés, pouvant calculer une famille de fonctions booléennes. L'objet qui est codé est une grammaire de graphe parallèle. La grammaire est codée par un ensemble d'arbres (appelés *arbres-grammaires*), au lieu d'un ensemble de règles. Le codage cellulaire peut décrire un réseau de façon concise et claire. De plus l'algorithme génétique permet de manipuler les arbres grammaires avec efficacité. Plusieurs propriétés du codage cellulaire ont été formalisées et prouvées. Le GA a été utilisé pour conduire une recherche dans l'espace des arbres-grammaires. Le GA peut produire un code cellulaire qui fournit à la fois l'architecture et les poids ± 1 spécifiant un réseau de neurones particulier pour résoudre la parité, la symétrie et le décodeur. De plus les arbres grammaire sont des codages récursifs, capables de générer des familles entières de réseaux de neurones, pour calculer la parité, la symétrie ou le décodeur. Enfin, une fois résolus pour de petites tailles, les problèmes concernant ces trois fonctions, les grammaires résolvent automatiquement les problèmes de tailles arbitrairement grandes.

Koza a montré que la structure d'arbre est un codage efficace par rapport au GA. Les arbres peuvent être recombinés en utilisant des opérateurs génétiques. La "*programmation génétique*" a été définie par Koza comme une façon de faire évoluer des programmes d'ordinateurs avec le GA [17]. Dans le paradigme de la programmation génétique, les individus sont des S-expressions LISP, lesquelles peuvent être décrites graphiquement comme des arbres étiquetés ayant une racine et dont les branches sont ordonnées. Comme le code cellulaire a exactement la même structure d'arbre étiqueté, la même approche est utilisée pour générer et recombiner des codes cellulaires avec le GA. Dans la conclusion de cette thèse, nous allons présenter une dernière contribution de cette thèse qui généralise la programmation génétique. Nous l'avons mise dans la conclusion, car elle représente un peu l'aboutissement, la synthèse, des travaux décrits dans les chapitres précédents. Voici le résumé de cette contribution:

On va montrer que la recherche génétique menée dans l'espace des codes cellulaires est réellement une approche du type "programmation génétique", et pas seulement parce que la représentation génétique utilise la même structure d'arbre. Tout d'abord, le codage cellulaire peut être considéré comme un langage de programmation de réseaux de neurones de bas niveau. Il existe en effet un compilateur (JaNNeT) qui, à partir d'un programme Pascal, produit le code cellulaire d'un réseau de neurones, celui-ci calculant ce que le programme Pascal spécifie. On effectue un pas de plus pour comparer le codage cellulaire et LISP. Les deux utilisent le concept de programmation génétique. LISP et le codage cellulaire sont deux langages de programmation génétique distincts. On défini un critère pour classer les langages de programmation génétiques de complexité croissante. Selon ce critère, LISP se révéle être un langage plus complexe que le codage cellulaire. Nous proposons une hiérarchie de langages de programmation génétiques de complexité croissante. Cette hiérarchie va du langage de très bas niveau utilisé pour microcoder les opérateurs, jusqu'au langage évolué: LISP, équipé de bibliothéques de fonctions. On considère la question de savoir quel est le meilleur langage génétique. Nous montrons que la programmation génétique de réseaux de neurones utilisant le codage cellulaire est totalement différente de celle où l'on utilise LISP avec un opérateur d'encapsulation, proposée par Koza et Rice. Les avantages apportés par le codage cellulaire sont décrits. Tous les résultats avancés par Koza montrent que LISP est très efficace quand le problème à résoudre met en jeu la manipulation de symboles. En revanche, quand le problème consiste à construire des réseaux de neurones, le codage cellulaire paraît mieux adapté. Une méthode pour l'évolution du langage génétique lui-même est brièvement décrite. Dans cette méthode un langage génétique apparaît comme une hiérarchie de bloc de base (la traduction anglaise: "building blocks" est un terme employé dans l'étude des GA, il signifie: un morceau de code génétique intrinsèquement intéressant). Le dernier paragraphe résume les paradigmes nouveaux de la programmation génétique des réseaux de neurones.

10.1 Généralisation de la programmation génétique

Dans ce paragraphe, on donne une définition de ce qu'est un langage de programmation génétique, un critère pour classifier les langages génétiques, et une liste de langages de complexité croissante.

Définition 4 *Un langage de programmation génétique $\mathcal{L}_S$ est un ensemble S de symboles d'arités différentes. Un programme génétique, sur un tel langage, est un ensemble ordonné d'arbres étiquetés par des symboles de S. Pour chaque noeud, le nombre de noeuds fils est égal à l'arité du symbole étiquetant le noeud.*

Des contraintes syntaxiques peuvent éventuellement limiter le champ des programmes génétiques possibles. Koza distingue entre les symboles d'arité 0, qu'il appelle des terminaux et les symboles d'arité supérieure à 0 qu'il appelle des fonctions. Ici de telles appellations ne sont pas appropriées, car les symboles de fonctions ne correspondent pas à des fonctions. La définition de la programmation génétique donne un squelette théorique. Son but est de capter la propriété la plus utile d'un langage de programmation génétique, à savoir la modularité. Chaque arbre t représente un bout de code, codant une fonction particulière f. Ce code peut être appelé à partir d'autres arbres en utilisant le numéro associé a t. Avec le codage cellulaire, l'appel d'une fonction f équivaut à l'inclusion du réseau de neurones qui calcule f. Cet appel-inclusion est réalisé en utilisant l'opérateur `JMP` vu au paragraphe 4.5. L'opérateur `JMP` a un argument. Quand une cellule exécute `JMP x` elle place sa tête de lecture sur l'arbre de numéro `x`.

Au lieu d'utiliser les numéros d'arbres pour les adresses des arbres, il est possible d'utiliser un mécanisme basé sur la mise en correspondance de motifs binaires complémentaires, comme celui utilisé par Koza (voir le chapitre sur la self replication spontanée). Le paragraphe 3.4 décrit comment cela peut être fait. Bien que ce système d'adressage soit intéressant d'un point de vue génétique, la structure générale du programme génétique reste la même; elle peut toujours être décomposée en un ensemble d'arbres. La différence est seulement dans l'implantation. Considérons un type spécial de grammaires d'arbres:

Définition 5 *Une GEN-grammaire est une grammaire d'arbre, utilisée pour réécrire un programme génétique écrit dans un langage $\mathcal{L}_S$, en un programme génétique écrit dans un langage $\mathcal{L}_{S'}$. Une règle de cette grammaire doit vérifier les conditions suivantes: Le membre gauche est un symbole de S d'arité r. Le membre droit est un arbre étiqueté par des symboles de S' et des symboles spéciaux $T_i, i = 1, \ldots, r$. Le membre droit contient une et une seule occurrence de chacun de ces symboles spéciaux. Quand s est réécrit par r, ces symboles spéciaux spécifient où insérer les sous-arbres du noeud étiquetés par s.*

Une GEN-grammaire est une grammaire d'arbre très simple. En fait elle décrit un morphisme d'arbres. Lors de la réécriture d'un programme génétique, chaque noeud est simplement remplacé par le sous-arbre étiqueté avec des symboles de S'. Ainsi la taille du nouveau programme génétique obtenu après réécriture est bornée par une constante K multipliée par la taille du programme qui est réécrit. K peut être choisi comme le nombre maximum de noeuds des sous-arbres "membres-droit" des règles de la grammaire.

Au chapitre 4 on a présenté un compilateur appelé JaNNeT qui à partir d'un programme Pascal génére un réseau calculant ce que le programme Pascal spécifie. Le compilateur commence par modifier et compléter l'arbre syntaxique du programme Pascal. L'arbre syntaxique en résultant est plus complexe qu'un arbre syntaxique usuel. Il comporte davantage de sortes de noeuds. Entre autres, il comporte des noeuds pour la définition des variables. Il n'y a par contre pas beaucoup de différences entre ces arbres syntaxiques, et les programmes LISP écrits avec un jeu d'instructions réduit. En effet un programme LISP est un arbre étiqueté qui ressemble à un arbre syntaxique, comportant de plus, des noeuds pour la déclaration des variables. La deuxième étape du compilateur consiste à réécrire l'arbre syntaxique modifié en un code cellulaire, en utilisant une GEN-grammaire. A la troisième étape, le compilateur développe le code cellulaire, après avoir choisi une valeur L de la vie initiale. On produira ainsi un réseau de neurones qui calcule ce qui est spécifié par le programme Pascal, pourvu qu'il n'y ait pas plus de L appels récursifs encapsulés, effectués pendant l'exécution. En dehors du module développant le code cellulaire, qui est un module séparé, le logiciel du compilateur consiste uniquement en les 27 règles de réécriture de la GEN-grammaire qui réécrit l'arbre syntaxique en un code cellulaire.

On propose une classification de langages génétiques basée sur la relation d'ordre suivante:

Définition 6 *Soit deux langages de programmation génétiques $\mathcal{L}_1$ et $\mathcal{L}_2$, $\mathcal{L}_1$ est dit supérieur en complexité à $\mathcal{L}_2$ ($\mathcal{L}_1 \succ \mathcal{L}_2$) s'il existe une GEN-grammaire capable de réécrire tout programme génétique P écrit avec $\mathcal{L}_1$ en un programme génétique P' écrit avec $\mathcal{L}_2$, qui effectue le même calcul.*

L'existence du compilateur JaNNeT prouve que les arbres syntaxiques Pascal sont supérieurs en complexité au codage cellulaire, En utilisant LISP avec un jeu d'instructions réduit on a aussi LISP $\succ$ codage cellulaire, puisque les arbres syntaxiques que l'on utilise sont similaires à des S-expressions LISP qui incluent la définition de variables.

On propose maintenant d'autres langages de programmation génétiques que l'on classe en utilisant la relation $\succ$. Dans le paragraphe 4.4 on montre que les opérateurs du codage cellulaire peuvent être micro-codés. Ce micro-codage peut être réalisé en utilisant une GEN-grammaire. Le membre gauche d'une règle est le nom d'un opérateur, le membre droit est son micro-code, disposé sous la forme d'un arbre. Pour écrire le logiciel JaNNeT, on a défini des arbres grammaires de blocs de base neuronaux, qui sont appelés par le code compilé. Ces arbres-grammaires forment une bibliothéque, et peuvent être considérés comme des macro-opérateurs écrits en code cellulaire. Le paragraphe D.8.3 énumère les codes cellulaires de ces macros-opérateurs. Le langage qui comprend le code cellulaire plus les noms de ces macros, peut être appelé macro codage cellulaire. Il est possible de réécrire un macro-code cellulaire en utilisant une GEN-grammaire. Le membre gauche est le nom de la macro-fonction, le membre droit est son code cellulaire. Finalement, et de la même façon, on peut utiliser une bibliothéque de fonction LISP et le langage résultant peut s'appeler macro-LISP. Le membre gauche est le nom de la fonction LISP, le membre droit est le code de la fonction LISP. On peut résumer ces faits avec la formule:

$$\text{Macro LISP} \succ \text{LISP} \succ \text{Macro CE} \succ \text{CE} \succ \text{Micro CE}.$$

Voici maintenant une illustration concrète de cette hiérarchie: Considèrons le programme Pascal suivant:

```
program p; var a: integer; begin   write (a) end.
```

Comme les variables sont initialisées à zéro par défaut, ce programme va se compiler en un seul neurone de sortie ayant un biais 0, et qui calcule 0. L'arbre syntaxique modifié du programme Pascal est:

PROGRAM(DECL("a")(TYPE-SIMPLE)(WRITE(IDF-LEC("a"))))

La traduction en macro-code cellulaire effectuée par le compilateur JaNNeT suit. Le nom des variables doit être remplacé par un numéro correspondant à leur rang d'apparition. Donc l'arbre syntaxique devrait contenir le numéro des variables plutôt que leur nom. Mais cela est un détail de l'implantation technique de JaNNeT. Cela ne remet pas en question la hiérarchie. Il est possible de modifier le codage cellulaire utilisé, de façon à garder les noms des variables.

```
IPAR(1)(START-)(SEQ(CPFO(1(SWITCH(1)(IPAR(2)(VAR(LR(1)(VAL(0)(EVER(2))))(PUTL
(2)(BLOC(PAR(CHOPO(1)(SEQ(CPFO(1)(CUTO(1)(CHOP(3)(BPN(1)(END)(MRG(PN-(2))))))
)(BLOC(JMP WRITE-MACRO))))))))(CUT(1)(CUTO(1)))))))))(BLOC(MRG(1)(CUTO(1)))))
```

La traduction en code cellulaire se fait en remplaçant chaque appel à une macro (ici WRITE-MACRO) par le code cellulaire de la macro. On produit alors:

```
IPAR(1)(START-)(SEQ(CPFO(1(SWITCH(1)(IPAR(2)(VAR(LR(1)(VAL(0)(EVER(2))))(PUTL
(2)(BLOC(PAR(CHOPO(1)(SEQ(CPFO(1)(CUTO(1)(CHOP(3)(BPN(1)(END)(MRG(PN-(2)))))))
(BLOC((LABEL(-29)(BPN(1)(END)(MRG(1)(SPLIT(JMP(-29)))))))))(CUT(1)(CUTO(1)))))
))))(BLOC(MRG(1)(CUTO(1)))))
```

Comme on a pris un programme très simple, la taille de l'arbre augmente peu entre le macro-code cellulaire et le code cellulaire. La traduction en microcode cellulaire est faite en remplaçant chaque opérateur par son micro-code. On produit:

```
DIV(m(<)(s(d(=)(m(>))))))(1)(EXE(e)(0))(DIV(s)(0)(-(TOP(m(<)(d(^(#))(m(>=))))(1))
(TOP(m(<)(m($)(m((>$)(m(=))))))(1)(DIV(m(<)(s(d(=)(m(>))))))(2)(AFF(var)(0)(CEL1
(1)(LNK(v)(0)(CEL(e)(2))))))(TOP(m(<)(m(>)(m(=))))(2)(EXE(q)(0)(DIV(d(*)(R(*)))(0
)(-(TOP(m(=))(1))(DIV(s)(0)(-(TOP(m(<)(d(^(#))(m(>=))))))(1)(-(TOP(m(<)(m(>)))(1))
(TOP(m(=))(3)(EXE(p)(1)(EXE(e)(0))(TOP(m(<)(d(^(*))(m(>))))(1)(CEL(e)(2))))))(EXE(
q)(0)(EXE(y)(-29)(EXE(p)(1)(EXE(e)(0))(TOP(m(<)(d(^(*))(m(>))))(1)(HOM(0)(0)(EXE(
j)(-29)))))))(TOP(m(<)(m(>)))(1)(TOP(m(<)(m(>)))(1)))))))))(EXE(q)(0)(TOP(m(<)(
d(^(*))(m(>))))(1)(-(TOP(m(<)(m(>)))(1)))))
```

On peut observer que la taille finale du programme génétique est bien plus grande que la taille de l'arbre syntaxique initial. Mais le nombre d'étiquettes nécessaires a diminué. Il est possible de coder l'arbre ci-dessus en utilisant seulement 8 sortes d'étiquettes.

10.2 Comparaison du Codage cellulaire avec le LISP

Koza a utilisé avec succés le langage génétique LISP, dans des domaines d'applications très variés. Cependant les résultats qu'il obtient sont surtout remarquables pour les problèmes mettant en jeu la manipulation de symboles. Par symboles, on n'entend pas uniquement les symboles mathématiques, mais toute entité dotée d'une signification de haut niveau. Ce succés doit beaucoup au fait que LISP est un langage permettant aisément de manipuler des symboles. En revanche, Koza et Rice [16] ont tenté de mener une recherche dans l'espace des réseaux de neurones en utilisant LISP, et ont obtenu des résultats moins impressionnants. Ils ont trouvé un réseau de neurones pour le XOR et un autre pour l'additionneur de 1 bit.

La combinaison entre programmation génétique et réseaux de neurones qui utilise le codage cellulaire, est totalement différente de la combinaison proposée par Koza et Rice. Ici ce n'est pas un programme qui simule un réseau de neurones, mais un réseau de neurones qui se comporte comme un programme. Nous avons en effet montré qu'un programme peut être compilé en un réseau de neurones. Moyennant cette nouvelle forme de combinaison, la programmation génétique de réseaux de neurones permet de combiner une recherche symbolique menée par le GA avec une recherche de type connexionniste obtenue par un apprentissage des poids sur une architecture fixée. On a en effet combiné la recherche génétique avec un apprentissage appelé Switch Learning, et un mode de combinaison de l'apprentissage appelé apprentissage développemental. C'est une solution naturelle du problème fondamental de la construction d'un réseau de neurones pouvant manipuler des symboles sans utiliser d'*associations explicites* entre symboles et neurones ou états de neurones. Ici, la méthode exploite toute la magie de la programmation génétique qui construit une sorte de programme différent, sans utiliser une *association explicite* du problème à résoudre avec un algorithme.

Un autre intérêt du codage cellulaire est d'incorporer la modularité dans les réseaux de neurones. La modularité est une propriété essentielle de la programmation. Considérée relativement au codage de réseaux de neurones, la propriété de modularité a été définie au paragraphe 3.4. Le chapitre précédent montre que le codage cellulaire posséde la propriété de modularité, et décrit une expérience où la modularité est utilisée pour résoudre un problème difficile à aborder autrement. Un autre avantage du codage cellulaire par rapport au LISP est qu'il peut inférer une récurrence où le nombre d'entrées et de sorties croît. Par exemple, le codage cellulaire a pu trouver un code génétique pour une famille de réseaux de neurones qui calculent la parité de n entrées, n étant arbitrairement grand. Cela semble impossible a faire avec LISP d'une façon naturelle, parce que les entrées du réseau doivent être spécifiées comme des terminaux qui sont en nombre fini.

D'après le livre GPII de John Koza [18], il semble que le point le plus intéressant dans la programmation génétique est celui où l'on propose une méthode pour la décomposition automatique de problèmes. Koza fait évoluer une liste de S-expressions LISP où chaque S-expression code une fonction; et les fonctions s'appellent les unes les autres de façon hiérarchique. De la même façon nous faisons évoluer un ensemble d'arbres grammaires où chaque arbre-grammaire code un sous-réseau de neurones et où les sous-réseaux peuvent s'inclurent les uns les autres d'une façon hiérarchique, Koza écrit:

> "For a variety of different problems, from a variety of different fields, automatic function definition enables genetic programing to automatically and dynamically decompose a problem into subproblems, to automatically and dynamically discover a solution to the sub-problems, to automatically and dynamically discover a way to assemble the solutions to the subproblems into a solution to the overall problem."

Nous écrivons: pour une variété de problèmes booléens différents, le codage cellulaire a permis à la programmation génétique de décomposer automatiquement et dynamiquement un problème en sous-problèmes, de découvrir un sous-réseau de neurones qui résoud le sous-problème, puis enfin de réassembler des copies du sous-réseau en un réseau global qui effectue le bon calcul. Pour utiliser la définition automatique de fonctions, Koza doit définir avant de lancer le GA, le nombre de fonctions qu'il va utiliser, le nombre d'arguments qu'a chacune de ces fonctions, et l'ensemble des allèles pour chaque fonction.

Le codage cellulaire n'impose aucun de ses prétraitements. Dans le chapitre 9 on a utilisé deux arbres grammaires pour deux réseaux de neurones définis automatiquement. Dans la programmation génétique qui utilise le LISP, les paramètres doivent être transmis explicitement entre la fonction qui appelle et la fonction appelée. C'est la raison pour laquelle le nombre de paramètres à transmettre doit être connu à l'avance. Au contraire, avec le codage cellulaire, les paramètres sont transmis implicitement, par l'intermédiaire des cellules voisines de la cellule qui commence à développer le sous-réseau à inclure. Il n'est pas nécessaire de définir un nombre d'arguments affectés à un sous-réseau. Il n'est pas non plus nécessaire de définir un ensemble d'allèles particuliers pour les arbres grammaires, cependant on peut le faire pour aider le GA.

La conclusion générale de ce paragraphe est que si LISP est meilleur pour faire de la manipulation de symboles, le codage cellulaire est mieux adapté à la synthèse de réseaux de neurones. Nous pensons que la synthése de réseaux de neurones est un problème plus fondamental que des problèmes concernant la manipulation de symboles. Un des problèmes principaux de l'intelligence artificielle est le problème de l'ancrage des symboles. Il est difficile d'établir un pont entre les symboles et le monde réel. Il y a toujours un fossé qui sépare le monde des symboles du monde réel qui est bruité, flou, et très vaste. Si la recherche génétique est menée au niveau des symboles, elle ne pourra pas résoudre le problème de l'ancrage des symboles. Le codage cellulaire permet de mener la recherche génétique à un niveau de complexité moindre, comme nous l'avons vu dans le paragraphe précédent. On peut imaginer la définition de blocs de base neuronaux qui se répétent, utilisés dans le traitement des images pour extraire les bords et les angles des contours telles que la rétine ou les colonnes corticales. Ces blocs de bases ont un certain rapport avec le monde réel mais ne sont pas directement connectés avec des symboles humains. Un exemple concret est le réseau trouvé par Beer et Gallagher qui résoud le problème de la locomotion à 6 pattes dans [2]. Ils ont codé un gros réseau avec peu de bits, en utilisant les symétries du problème. Ici, la structure est représentée par les symétries, et les symétries peuvent être representées par le codage cellulaire, en utilisant un code modulaire. Si notre but final est de générer des structures cognitives complexes comme le cerveau, plutôt que de résoudre des problèmes ponctuels en manipulant des symboles, il est probablement préférable d'utiliser un langage de plus bas niveau que le LISP.

10.3 Faire évoluer le langage génétique lui-même

Le codage cellulaire est une méthode pour coder des familles de réseaux de neurones booléens, calculant des familles de fonctions booléennes paramétrées. Nous avons montré que la recherche génétique de réseaux de neurones, en utilisant le codage cellulaire, équivalait à la micro programmation génétique de réseaux de neurones, et que le codage cellulaire est semblable à un langage de micro programmation de réseaux de neurones. Pour construire un cadre théorique permettant la généralisation, nous avons proposé une définition d'un langage de programmation génétique avec un critère permettant de classer leur complexité. Ce critère utilise une sorte de grammaires d'arbre très simple, appelée GEN-grammaire. Cinq langages génétiques ont été classés: Macro LISP $\succ$ LISP $\succ$ Macro CE $\succ$ CE $\succ$ Micro CE. Cette classification montre qu'une décomposition est possible, depuis une description abstraite basée sur un langage de programmation de haut niveau, jusqu'à un microcode très simple qui spécifie des transformations locales de graphes élémentaires: le mouvement ou la duplication d'une suite de liens contigus. Il n'y a aucune preuve permettant de dire lequel de ces langages est le plus efficace pour faire de la recherche génétique. On argumente qu'un langage génétique de haut niveau tel que LISP est plus adapté au traitement de symboles, et un langage de bas niveau tel que le codage cellulaire est plus adapté à la synthése de réseaux de neurones. En fait, le langage génétique lui-même peut être obtenu par algorithme génétique. La population initiale de programmes doit être construite en utilisant le langage génétique le plus simple qui est le micro codage cellulaire. Un bloc de base est un morceau de code qui en moyenne augmente la fitness du code dans son ensemble. Le symbole d'un langage au niveau l est un morceau de code au niveau $l-1$ qui a une signification intrinsèque, et peut donc être utile en général, ceci se traduit par une augmentation moyenne de la fitness. Ainsi, chaque symbole d'un langage au niveau l apparaît naturellement comme un bloc de base relativement aux structures générées par le langage au niveau $l-1$. Dans ce paradigme, l'évolution est un processus par lequel des langages génétiques de complexité croissante sont acquis et stockés en utilisant des GEN-grammaires. Les règles des GEN-grammaires sont des bloc de bases; elles sont codées par des arbres, dans un chromosome représentant un ensemble d'arbres. Le paradigme nécessite un mécanisme d'adressage, un mécanisme pour décider quand augmenter le nombre d'arbres de n vers $n+1$, et un algorithme pour initialiser la $(n+1)^{ieme}$ composante. Le premier mécanisme peut être l'adressage par motifs complémentaires où la suite de bits complémentaire est recherchée sur des arbres de rang inférieur ou égal à celui de l'arbre appelant. Le deuxième mécanisme peut augmenter le nombre d'arbres lorsqu'un changement survient dans l'environnement, reflété par un changement dans la fitness moyenne. L'initialisation de la $(n+1)^{ieme}$ composante peut être simplement aléatoire. Un langage génétique produit avec cette méthode serait simplement une hiérarchie de blocs de base. Si la population initiale est générée avec des langages de plus haut niveau que le micro codage cellulaire, la méthode est une méthode d'évolution de réseaux de neurones modulaires, où chaque bloc de base neuronal au niveau $l-1$ est un symbole au niveau l.

10.4 Un mariage réussi qui a des perspectives d'avenir

On oppose toujours les réseaux de neurones formels aux ordinateurs classiques basés sur la machine de Turing. Ils sont considérés comme deux modèles de calculs aux antipodes. Un ordinateur classique se programme avec un algorithme déterministe; on sait prévoir combien de temps il va mettre pour résoudre un problème, ainsi que l'espace mémoire nécessaire. Un réseau de neurones ne se programme pas, il apprend tout seul. On ne sait pas s'il va réussir à apprendre, encore moins combien de temps il peut mettre, et le nombre de neurones nécessaire. Dans cette thèse on marie ensemble ces deux approches, par l'intermédiaire de l'algorithme génétique. Ce mariage résulte en un concept nouveau: "programmation génétique de réseaux de neurones". Il consiste d'abord à utiliser un langage du type "programmation procédurale" pour décrire des réseaux de neurones; puis à mener une recherche indirecte dans l'espace des programmes, plutôt qu'une recherche directe dans l'espace des réseaux de neurones. Ce mariage est basé sur les principes suivants:

Nécessité de la modularité Il est impossible de générer automatiquement de gros réseaux de neurones permettant de résoudre des problèmes complexes, sans leur donner une structure modulaire et hiérarchique reflétant les régularités du problème à résoudre

Nécessité d'un codage On ne peut définir formellement "modulaire " et "hiérarchique " que par rapport à un codage de réseaux de neurones.

Nécessité d'une méthode de recherche des codes La génération de réseaux de neurones modulaires et hiérarchiques nécessite qu'une recherche soit menée dans l'ensemble des codes.

Langage de programmation de réseaux Un codage de structures modulaires et hiérarchiques doit être l' équivalent d'un langage de programmation, pour la description de réseaux de neurones. Ceux-ci sont des graphes étiquetés, d'un point de vue structure de données.

Réécriture de graphe Pour cela il doit être basé sur la réécriture de graphes, seul outil puissant permettant de décrire des graphes étiquetés, à notre connaissance.

Différentes granularités possibles Le codage peut avoir une granularité plus au moins fine suivant la grammaire de graphe sous-jacente. Les granularités possibles sont les suivantes, dans l'ordre de complexité décroissant :

- Réécrire le noeud d'un graphe en plusieurs noeuds.
- Réécrire le noeud d'un graphe en moins de deux noeuds
- Déplacer ou dupliquer des suites de connexions

Changement du cahier des charges de l'apprentissage L'apprentissage classique dans les réseaux de neurones opère sur une architecture fixée et est basé sur une méthode de descente de gradient. Il doit à présent être vu comme une aide à la méthode de recherche des codes. Ce changement d'optique entraîne une conception différente de l'apprentissage, et ouvre un domaine de recherche vierge.

Dans cette thèse on a illustré ces paradigmes nouveaux comme suit:

Modularité L'expérience montre qu'il est possible de générer automatiquement des réseaux de neurones modulaires et hiérarchiques arbitrairement grands, pour résoudre des problèmes booléens de taille arbitrairement grande.

Codage La définition formelle d'un codage modulaire a été donnée. Il a été montré que le codage cellulaire est modulaire.

Méthode de recherche des codes L'algorithme génétique est une méthode adaptée à la recherche de codes cellulaires, cela est montré par les résultats expérimentaux. De plus, l'hypothèse des blocs de base qui conditionne le succès du GA peut s'interpréter. Un sous-réseau étant codé sur un sous-arbre du code cellulaire, le sous-arbre sera un bloc de base, si le sous-réseau présente des caractéristiques intrinséquement intéressantes; de telle sorte que lorsque le sous-arbre se trouve inclus dans un code, la fitness du code dans son ensemble soit meilleure en moyenne.

Langage de programmation de réseaux Les propriétés du codage, ainsi que l'existence du compilateur neuronal JaNNeT montrent que le codage cellulaire est comme une sorte de langage de programmation de réseaux de neurones.

Réécriture de graphes Il a été montré que le codage cellulaire est un système de réécriture parallèle de graphes.

Différentes granularités Les macro-opérateurs introduits pour la compilation, les opérateurs du codage cellulaire et le microcodage illustrent les différentes granularités possibles.

Apprentissage On a défini un apprentissage nouveau, appelé apprentissage Switch, ainsi qu'un nouveau moyen de combiner apprentissage et recherche génétique, appelé apprentissage développemental. Ces nouveautés ont permis d'accélérer la recherche génétique d'un facteur 13.

Le mariage de la programmation génétique et des réseaux de neurones présente ceux-ci sous un jour nouveau. Par exemple, on n'est plus obligé de se limiter à des neurones dotés d'une sigmoïde traditionnelle, on peut utiliser des neurones faisant le produit de leurs entrées. L'algorithme génétique peut générer n'importe quel type de neurones. Plutôt que de faire systématiquement de l'apprentissage, on peut concevoir des réseaux où seulement une petite partie fait de l'apprentissage, l'autre étant générée à la main; ceci en utilisant le compilateur JaNNeT, ou directement avec le codage cellulaire. Le cahier des charges d'un algorithme d'apprentissage a changé, il ne doit pas systématiquement converger, il doit simplement aider l'algorithme génétique. Il devient moins important de parallèliser l'apprentissage dans les réseaux de neurones. Le parallélisme est bien plus naturel et puissant dans son utilisation pour la recherche des codes qu'employé à l'apprentissage dans les réseaux de neurones. Ainsi l'algorithme génétique présente une accélération super-linéaire. C'est l'évaluation de la fitness du code qui est coûteuse en temps, et les fitness peuvent être évaluées en parallèle.

L'idée de rechercher et d'optimiser les réseaux de neurones indirectement à partir d'un code représente un changement de paradigme encore plus radical, qui va, nous en sommes persuadés, tôt ou tard bouleverser le monde du connexionisme. Le vrai problème, que ce soit en Intelligence artificielle, ou dans le connexionisme, est le problème d'échelle. Les méthodes fonctionnent sur une petite échelle, mais ont une complexité exponentielle, et prennent trop de temps machine dès que la taille du problème devient grande. La seule façon de résoudre le problème d'échelle est de concevoir un algorithme qui permette automatiquement de décomposer un problème en sous-problèmes, de résoudre les sous-problèmes, et de recomposer la solution des sous-problèmes en une solution globale au problème tout entier. La solution du problème trouvée avec un tel algorithme a, de façon obligée, une structure qui refléte celle du problème. John Koza [18] démontre sur une quantité d'exemples que la programmation génétique peut réaliser cette décomposition et recomposition automatique, en utilisant ce qu'il appelle la définition automatique de fonctions (ADF). Le GA décompose automatiquement le problème en sous-problèmes, définit des fonctions (ADF) qui résolvent ces sous-problèmes, et réutilise ces fonctions plusieurs fois, pour composer une solution au problème global. Le livre GPII de Koza représente à nos yeux une percée. Cependant Koza n'a pas pensé que son concept de décomposition automatique puisse s'appliquer aux réseaux de neurones. Koza écrit:

> "Conceivebly the subtask performed by the neuron might be useful elsewhere in the neural network. However, existing paradigms for neural networks do not provide a way to re-use the set of weights discovered in one part of the network in other part of the network where a similar subtask must be performed on a different set of inputs"

C'est precisément ce que le GA, combiné avec le codage cellulaire, peut faire. Il peut réutiliser non seulement un neurone avec son jeu de poids, mais même un sous-réseau, de la même façon que la programmation génétique permet de réutiliser une procédure. Pour une variété de problèmes différents, le codage cellulaire a permis à l'algorithme génétique de décomposer automatiquement et dynamiquement un problème en sous-problèmes, de construire un sous-réseau pour le sous-problème, et de découvrir un moyen d'assembler des copies de ce sous-réseau en un réseau de neurones résolvant le problème tout entier. Dans les trois exemples proposés: la parité, la symétrie et le décodeur, la structure du réseau refléte exactement la structure du problème.

Un travail énorme reste à faire, pour appliquer la méthode proposée dans cette thèse à des problèmes plus naturels que les fonctions booléennes. Koza a montré que la décomposition automatique est intéressante lorsque le problème à résoudre présente un degré de régularité au delà d'un certain "break-even point". N'importe quelle application où les réseaux présentent un certain degré de régularité peut constituer un champ d'expériences. Nous avons en particulier deux propositions: Dans [2], Beer et Gallagher ont généré par GA un réseau de neurones pour la locomotion d'une bête à 6 pattes. Ils ont utilisé les symétries du problème pour obtenir une représentation compacte. Le réseau est composé de 6 sous-réseaux identiques $N_i, i = 1, .., 6$, chacun responsable d'une patte. Les connections entre ces sous-réseaux dessinent un huit. Elles sont aussi identiques. Ainsi sur le chromosome, Beer et Gallagher ont codés seulement N_1 et les connections entre N_1 et N_2. Grâce à cette représentation très compacte, le problème a pu être résolu. Le codage cellulaire peut représenter les symétries de ce problème. On peut coder le réseau total d'une façon compacte. Une partie du code représente N_1 et l'autre partie assemble 6 copies de N_1. Avec le codage cellulaire, le GA peut automatiquement décomposer le problème en 6 sous-problèmes identiques, générer le sous-réseau commun à ces 6 sous-problèmes, et assembler six copies de ce sous-réseau. Ainsi le problème de la locomotion à 6 pattes peut être résolu entièrement automatiquement, sans utiliser la connaissance que l'on a des symétries du problème. L'autre application concerne les réseaux invariants par translation, pour le traitement d'images de bas niveau. Ici, un même sous réseau est répété autant de fois qu'il y a de pixels, il constitue un bloc de base codé séparément comme une procédure.

Cette thèse présente des idées nécessitant de nombreux paramètres pour être concrétisées. Nous avouons nous-mêmes ne plus nous y retrouver dans les fichiers de paramètres qui dépassent la centaine. La question qui se pose toujours en informatique: "Comment faire pour aller plus vite ", doit être étudiée de façon systématique. Aller plus vite, c'est la motivation constante de toutes les simulations que nous avons effectuées. Nous avons toujours cherché à améliorer les performances de l'algorithme génétique, pour résoudre des problèmes de complexité croissante. Nous avons cherché un apprentissage optimal, uniquement dans ce but. Dans le premier chapitre de la seconde partie, le chapitre 6, le GA tout seul met à peu près 800 secondes pour résoudre le problème de la parité. C'est le temps qui a été mentionné dans la première publication sur le codage cellulaire, à COGANN92. Ce temps est réduit à 1.2 seconde, dans le chapitre 9, en utilisant l'apprentissage Switch, le mode de combinaison entre GA et apprentissage appelé apprentissage développemental et le GA parallèle mixte sur 64 noeuds de l'IPSC860. Cela représente une facteur de gain de vitesse de 660. Nous sommes persuadés qu'en continuant les recherches, et en utilisant les machines parallèles actuelles, il est possible d'aller encore 1000 fois plus vite. Cette course contre la montre n'est pas un jeu mais une nécessité quand on utilise le GA, et qu'on doit à l'issue une expérience d'une heure, générer, apprendre et tester un million de réseaux de neurones, chacun ayant plusieurs dizaines de neurones (cf expérience avec le décodeur). C'est la seule façon d'obtenir une méthode robuste de génération automatique de réseaux de neurones ayant une taille importante.

Bibliographie

[1] D.H. Ackley and M. Littman. Interactions between learning and evolution. In *the 2nd Conf. on Artificial Life*. Addison-Wesley, 1991.

[2] R. Beer and J. Gallagher Evolving dynamical neural networks for adaptive behavior *Adaptive Behavior* 1:92–122, 1993

[3] R.K. Belew. When both individuals and populations search: Adding simple learning to the genetic algorithm. In D. Schaffer, editor, *3th Intern. Conf. on Genetic Algorithms*. Morgan Kaufmann, 1989.

[4] R. Collins and D. Jefferson. Selection in massively parallel genetic algorithm. In *4th Intern. Conf. on Genetic Algorithms*, 1991.

[5] G.E. Hinton D.H. Ackley and T.J. Sejnowski. A learning algorithm for boltzman machines. *Cognitive Science*, 9:147–169, 1985.

[6] H. Ehrig, M. Korf, and M. Lowe. A tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *LNCS 532*, 1990.

[7] Max Garzon and Stan Franklin. Neural computability ii (extended abstract). In *IJCNN 89*.

[8] M. B. Gordon, P. Peretto, and D. Berchier. Learning algorithms for perceptrons from statistical physics. *Journal de Physique*, january 1993.

[9] S. Harp, T. Samad, and A. Guha. Toward the genetic synthesis of neural networks. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 360–369, 1989.

[10] G.E. Hinton and S.J. Nowlan. How learning can guide evolution. *Complex Systems*, 4:495–502, 1987.

[11] John Holland. *Adaptation in natural and artificial systems*. MIT press, 1993.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[13] David Hubel. *Eye Brain and Vision*. Scientific American Library, 1988.

[14] J.M.Baldwin. A new factor in evolution. *American naturalist*, 30:441–451, 1896.

[15] H. Kitano. Designing neural network using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476, 1990.

[16] J. R. Koza and J. P. Rice. Genetic generation of both the weigths and architecture for a neural network. In *IJCNN92, International Joint Conference on Neural Networks*, pages 397–404, 1991.

[17] John R. Koza. *Genetic programming:On the programming of computers by mean of natural selection.* MIT press, 1992.

[18] John R. Koza. *Genetic programming II:Automatic Discovery of Reusable Subprograms.* MIT press, 1994, to appear.

[19] Tsu-Chang Lee. *Structure level adaptation for artificial neural networks.* Kluwer Academics Pubishers, 1991.

[20] A. Linden and C. Tietz. Combining multiple neural network paradigms and applications using sesame. In *International Joint Conference onNeural Netwroks, Baltimore*, 1992.

[21] A. Lindenmmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–299, 1968.

[22] S. Lucas. An algebraic approach to learning in syntactic neural networks. In *IJCNN92*, 1992.

[23] Eric Mjolness, David Sharp, and Bradley Alpert. Scaling, machine learning and genetic neural nets. La-ur-88-142, Los Alamos National Laboratory, 1988.

[24] H. Muehlenbein. Limitations of multi-layer perceptrons networks - steps towards genetic neural networks. *Parallel Computing*, 14:249–260, 1990.

[25] H. Muehlenbein. Darwin's continent cycle theory and its simulation by the prisonners dilemma. *Complex System*, pages 459–478, 1991.

[26] H. Muehlenbein, M. Schomish, and J.Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[27] Hans P.Zima, Peter Brezany, Barbara Chapman, and Jan Hulman. Automatic parallelization for distributed-memory systems: Experiences and current research. In *European Informatics Congress Systems Architecture*, 1993.

[28] Thomas Ray. Is it alive or is it ga ? In Richard Belew, editor, *4th Intern. Conf. on Genetic Algorithms*, pages 527–534, 1991.

[29] J.D. Schaffer and D. Whitley . *Combination of Genetic Algorithms and Neural Networks.* IEEE Computer Society press, 1992.

[30] Hava Siegelman and Eduardo D. Sontag. Neural nets are universal computing devices. Sycon-91-08, Rutger University, 1991.

[31] R. Tanese. Distributed genetic algorithm. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 434 – 440, 1989.

[32] D. Whitley. The genitor algorithm and selection pressure why rank based allocation of reproductive trials is best. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 116–121, 1989.

[33] D. Whitley, T. Stakweather, and C. Bogart. Genetic algorithms and neural networks, optimizing connection and connectivity. *Parallel Computing*, 14:347–361, 1990.

Appendix A

Bibliographie personnelle

A.1 Articles avec comité de lecture

- Adding learning to the the cellular developmental process: a comparative study. *Evolutionnary Computation*, 1993. V1N3. En collaboration avec Darell Whitley
- Genetic Microprograming of Neural Networks Contribution au livre *Advance in Genetic Programing* edited by K. Kinnear, MIT press 1993, à paraître.
- Genetic programing of Neural Networks, theory and practice Contribution au livre *Intelligent hybrid systems* edited by S. Goonatilake and S. Khebbal, John Whiley, 1993 à paraître.

A.2 Brevet

Dépôt d'un brevet intitulé: procédé de traduction et de conception de réseau de neurones à partir d'une description logique du problème à résoudre.

A.3 Conférence dont les actes sont publiés

- A 2D toroidal systolic array for the knapsack problem, In *Algorithms and Parallel VLSI Architectures II* 1991. Elsevier En collaboration avec Rumen Andonov
- A modular systolic 2D torus for the general knapsack problem. In *Application Specific Array Processors* 1991 IEE Computer Society Press. En collaboration avec Rumen Andonov
- Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *Combination of Genetic Algorithms and Neural Networks*, 1992. IEEE Computer Society press
- Genetic synthesis of modular neural networks. In *5th International conference on Genetic ALgorithm*, 1993. Morgan Kaufman Publisher
- Grammatical inference with genetic search using cellular encoding. In *Grammatical Inference: Theory,applications and alternatives*, 1993. Digest No 1993/092
- A learning and pruning algorithm for genetic neural networks. In *European Symposium on Artificial Neural Network*, 1993.

- The mixed parallel genetic algorithm. In *PARCO93 (Parallel Computing)*, 1993.

A.4 Conférence Nationale dont les actes sont publiés

Un outil pour combiner approches symboliques et connexionistes. In *Emergence des symboles dans les modéles de la cognition*, 1993. Editor: Bernard Amy, Alain Grumbach Jean-Jacques Ducret

A.5 Rapport de Recherche

- Cellular encoding of genetic neural network. research report 92.21, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1992.
- Adding learning to the the cellular developmental process: a comparative study. Research report rr93-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1993. En collaboration avec D. Whitley.
- Compilateur Pascal de Réseaux de Neurones Rapport de stage de projet de fin d'année Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquée de Grenoble. En collaboration avec Jean-Yves Ratajszcak and Gilles Wiber.

A.6 Conférences sans actes

- A neural Compiler using Graph Grammars. oral presentation Journees ASMICS , Schloss Dagstuhl, Octobre 1993. (Algebraic and Syntactical Methods in Computer Science)
- Genetic algorithm for designing neural network. Journees Modélisation et Cognition, Autrans, Pole Rhone Alpes de Cogniscience, 1992.
- Equivalence between cellular encoding and graph grammar. oral presentation Journees ASMICS , Bordeaux, 1993. (Algebraic and Syntactical Methods in Computer Science)
- Parallel GA applied to neural network synthesis. Journees Modélisation et Cognition, Paris, 1993.
- A parallel genetic algorithm. oral presentation at Journées CAPA, Grenoble, 1993. (Conception et Analyse des Algorithmes Parallèles)
- Participation to the summer school "Mathematical methods for artificial neural networks" at the Kurt Bösch Institute in Sion (Switzerland) 1992.

Appendix B

Lexique

Algorithme Méthode décrite pas à pas pour résoudre un problème.

Arbre syntaxique Arbre obtenu après l'analyse syntaxique d'un programme écrit dans un langage de programmation.

Arbre-grammaire Structure de données d'arbre étiqueté par des opérateurs cellulaires. Il représente un code cellulaire.

Blocage Une cellule de lecture est dite bloquée si elle doit attendre que toutes ses voisines en entrées soient devenues des neurones avant de pouvoir continuer à se développer.

Cellules Une cellule possède des connexions rentrantes et sortantes numérotées et orientées, vers d'autres cellules. Une cellule est donc un noeud d'un réseau appelé réseau de cellules. Une cellule posséde des registres. Des premiers registres qui lui servent pour se développer, des seconds registres qui stockent les attributs d'un neurone. Une cellule de lecture posséde une copie du code cellulaire, et une tête de lecture pour lire le code cellulaire à un endroit particulier. Elle peut exécuter des opérateurs appelés opérateurs cellulaires, lus par sa tête de lecture sur le code cellulaire Elle peut aussi exécuter des macro-opérateurs cellulaires. Un neurone peut être considéré comme une cellule particulière qui a perdu sa tête de lecture.

Codage Cellulaire Méthode pour coder une famille de réseaux de neurones structurés de la même façon.

Environnement L'environnement, en compilation, désigne l'ensemble des variables accessibles, en un point donné du programme.

Intelligence Artificielle Dans cette thèse, on désigne par intelligence artificielle des systèmes experts, qui fonctionnent à partir d'une base de règles et un moteur d'inférence. Les règles comportent un prédicat à vérifier, et une conclusion qui est vraie si le prédicat est vrai. Par exemple, *si x est un homme, alors x est mortel.* La base de règles traduit un ensemble de connaissances concernant un domaine bien délimité, et le moteur d'inférence combine les règles entre elles, pour manipuler des symboles et faire des déductions logiques. Par exemple *Socrate est un homme, donc Socrate est mortel.*

Langage de programmation Langage informatique utilisé pour décrire un algorithme

Macro-opérateur Un macro-opérateur remplace la cellule c qui l'exécute par un graphe d de cellules et de neurones. Ce graphe est rattaché au graphe global de la cellule c par l'intermédiaire des voisines de c. Voisines désigne deux cellules connectées entre elles, directement ou par l'intermédiaire de cellules pointeurs. Un macro-opérateur peut être associé à chaque étiquette de chaque noeud de l'arbre syntaxique d'un programme Pascal. Dans ce cas, le graphe d doit indiquer quelles cellules vont exécuter les opérateurs correspondant aux mots Pascal étiquetant les racines des sous-arbres du noeud étiqueté. Un macro-opérateur peut être interprété comme une règle de réécriture. Pour cette raison on dira qu'un macro-opérateur réécrit la cellule c en le graphe d. Le graphe d est appelé le membre droit du macro-opérateur.

Contexte de réécriture Le contexte de réécriture d'un macro-opérateur est la séquence ordonnée de ces cellules voisines, étiquetées ou non. L'étiquette désigne la fonction de la cellule quand la cellule a une fonction particulière, comme par exemple, neurone "pointeur entrée", cellule "start", cellule "next".

Motif invariant Supposons que l'environnement contienne n variables. Le contexte de réécriture d'une cellule par un macro-opérateur vérifie l'invariant suivant: La cellule a $n+2$ liens en entrée et 2 liens en sortie. Le premier lien en entrée et en sortie pointe vers la cellule pointeur entrée, et la cellule pointeur sortie. Le deuxième lien en entrée pointe vers une cellule appelée cellule "start", le deuxième lien en sortie pointe vers une cellule appelée cellule "next" chargée de faire la liaison entre le graphe généré par le macro-opérateur et le reste du réseau de neurone en amont. Cette cellule peut être un neurone (macro-opérateur du type expression) ou une cellule de lecture (macro-opérateur qui modifie l'environnement).

Neurone Un neurone est connecté en entrée et en sortie vers d'autres neurones. Il est donc inclus dans un graphe de neurones appelé réseau de neurones. Un neurone fait un calcul très simple à partir de ses entrées, puis envoie le résultat vers les neurones voisins en sortie. Ce calcul consiste à faire la somme pondérée des entrées, puis à appliquer une fonction appelée sigmoïde. Un neurone étendu peut aussi faire le produit ou le quotient de ses entrées.

neurones pointeurs Ce sont des neurones utilisés uniquement pour structurer des données en arbres. Ces neurones se trouvent aux noeuds de l'arbre; sur les feuilles de l'arbre se trouvent les données. On peut ainsi représenter des tableaux Pascal à plusieurs dimensions, ou des structures (mot clé Pascal record).

Neurone Start Ces neurones sont utilisés dans les réseaux compilés. Ils servent à lancer l'activité du réseau, et à gérer le renouvellement du flux d'activité à l'intérieur des réseaux de neurones récurrents. D'un point de vue pratique, il est traité comme une variable de l'environnement.

Opérateur cellulaire Un opérateur cellulaire peut diviser, modifier les registres de la cellule qui l'exécute, ou gérer l'ordre d'éxécution des cellules.

Réécriture Réécrire une structure informatique consiste à remplacer un bout de la structure par un autre, en spécifiant comment les bouts se recollent entre eux. On peut réécrire des mots, des arbres, des graphes.

Réseau de cellules Graphe orienté dont les connections partant où arrivant à un sommet sont ordonnées. A chaque noeud se trouve une cellule.

Réseau de neurones Graphe orienté, à chaque noeud se trouve un neurone.

Tête de lecture Un registre particulier de la cellule. Ce registre contient un pointeur vers un arbre-grammaire, et permet à la cellule de lire sur l'arbre grammaire le prochain opérateur qu'elle aura à exécuter.

Appendix C

Le codage cellulaire est une grammaire de graphes

C.1 Introduction

In this chapter, Cellular encoding is discussed in relation to graph grammars. Cellular Encoding is specified as a system of rewriting rules. The objects that are rewritten are *cells*, where a cell is a node of a directed graph, with **ordered** connections. If the rewriting rules are interpreted recursively i times, they develop the i_{th} neural-net of the family, that should compute the parity of i inputs, if parity is the target problem. In order to unify cellular encoding with the concept of usual graph grammar, it is shown that a cellular encoding can be translated into a graph grammar of the kind used in the Berlin algebraic approach [6]. The process of encoding a grammar into a set of trees is described. Trees are an interesting data structure, because the genetic algorithm can be efficiently used to recombine and mutate them. Experimental results obtained on the symmetry, the parity and the decoder boolean function, in the second part of this thesis, support this claim.

Searching a cellular encoding with the GA is a grammatical inference process. It differs from usual grammatical inference approach because the searched grammar does not have to generate a language specified by positive and negative examples. Here the language is specified indirectly. The language of neural networks must be such that the i^{th} neural net computes the mapping for the target problem of size i, and must be of reasonable size.

Our approach is different from the "syntactic neural network" of Lucas [22], where the language is itself the learning task, and therefore, it is known from positive and negative examples. In Lucas's approach, there seems to be an "isomorphism" between the grammar and the syntactic neural network, i.e, a grammar is mapped on a neural network. Syntactic neural networks have always the same type of architectures, whereas cellular encoding does not impose a prior constraint on the architecture, neither structural constraint nor complexity constraint.

C.2 Cellular Encoding presented as a graph grammar

Cellular encoding can be seen as a system of rewriting rules. These rules operate on *cells*. A cell is a node of a directed *network graph* with ordered connections. Each cell is labeled with a symbol of the grammar alphabet, either terminal or non-terminal. There is a single terminal celll which is labeled by n. Terminal cells are finished neurons.

Cells and connections have attributes, some of which are the thresholds and weights of the final neural net, while others are used by the rewriting rules. For example, the *link-register* is used to refer to one of possibly several fan-in connections into a cell. The *life* attribute is used to bound the number of recursive interpretations of the grammar.

A rule r is denoted $A \stackrel{F}{\rightarrow} \alpha$ rewrites one cell into k cells, where k is the lenght of the string α. A is a non terminal symbol that must label the *parent cell* on which r is applied, α is a string of k symbols, terminal or non terminal that label the k *child cells* created by the rule, and F refers to a program-symbol for computing the links and attributes for each of the k child cells. Here we consider only the case $k \leq 2$. If $k = 2$, the number of cells of the network-graph augments by one. The parent cell is said to *divide* into two child cells. In this case, the attributes of the two child cells are inherited from the parent cell. If $k = 1$, the rewriting rule *modifies* the attributes of the parent cell. If $k = 0$, the cell is deleted.

The axiom of the grammar is a network graph that consists of a single cell called the *ancestor cell* connected to an input *pointer cell* and an output *pointer cell.* Consider the starting network and the encoding depicted in figure C.1. At the starting step, the attributes of the ancestor cell are initialized with default values. The threshold is 0, the link-register is 1, the life is L, if we want to develop the L^{th} network. As this cell repeatedly divides, it gives birth to all the other cells that will eventually make up the neural network. The input pointer cell and the output pointer cell are not rewritten. They are used to point to the input and output unit of the network. For example, in figure C.1 (a), the XOR neural net has two input units labeled 1 and 2, and one output unit labeled 1. There are many kinds of cell division and cell modification, depending on the particular program-symbol used to compute the attributes and links of the child cells. We illustrate an example in figure C.1. The grammar used is $\{\$ \stackrel{SEQ}{\rightarrow} AB,\ A \stackrel{PAR}{\rightarrow} \$n,\ B \stackrel{SEQ}{\rightarrow} CD,\ C \stackrel{PAR}{\rightarrow} En,\ E \stackrel{INCBIAS}{\rightarrow} n,\ D \stackrel{VAL-}{\rightarrow} n\}$. The program-symbol `SEQ, PAR, INCBIAS, VAL-` have been defined in the preceding chapter.

The sequence in which cells are rewritten is determined as follows: once a cell is rewritten, it enters a First In First Out (FIFO) queue. The next cell to be rewritten is the head of the FIFO queue. If the cell divides, the child which reads the first letter of the right member enters the FIFO queue first. This order of execution tries to model what would happen if cells were rewritten in parallel. It ensures the following invariant:

> **Invariant 1:** A cell cannot be rewritten twice while another cell has not been rewritten at all.

By imposing a rewriting order we ensure that a grammar produces a single network for a given initial life value L. The handling of L is done as follows: each cell manages a stack of characters, used to record the history of its rewriting. Each time a rewriting rule like $\$ \stackrel{SEQ}{\rightarrow} AB$ is applied on a cell, the cell executes a 3-steps algorithm. Call s the symbol in the left member, here $s = \$$. First the cell checks whether s appears on its stack. If it does, the cell decrements its life number and empties its stack. Second, the cell pushes s on its stack. Third, the cell checks whether its life is strictly positive. If $L > 0$ the rule $\$ \stackrel{SEQ}{\rightarrow} AB$ is used, else the rule $\$ \stackrel{WAIT}{\rightarrow} n$ is used, and the cell becomes a finished neuron.

C.3 Encoding a Grammar into a set of trees

In this section is described how to encode a "clean" grammar into a set of trees. A clean grammar means a deterministic grammar where all the symbols are reached by the productions. First,

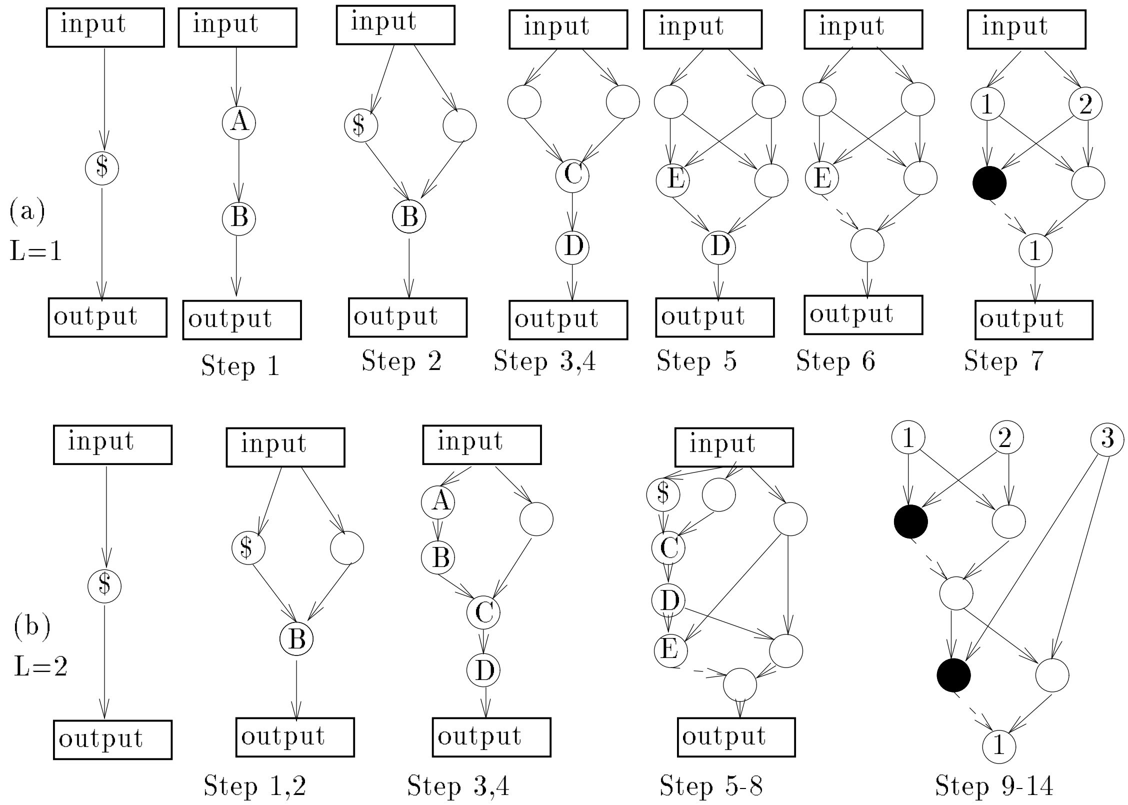

Figure C.1: The derivation of a parity network. The box labeled input represents the input pointer cell, The box labeled output represents the output pointer cell; Circles represent other cells. If the threshold is 0, the circle is empty, while if the threshold is 1 the circle is filled with black. A continuous line indicates a weight 1, and a dashed line a weight -1. The first seven network graphs (a) show the derivation for L=1, the network computes the XOR. In the second sequence of derivation (b), starting with L=2, the network for the three inputs parity is derived.

a directed graph $\mathcal{G}$ that represents the grammar is built in a constructive manner. Initially $\mathcal{G}$ contains one node for each non-terminal, labeled by the name of the corresponding non-terminal. For each rule $A \xrightarrow{F} \alpha$ where α includes the non terminal B, an edge is drawn from the node A to the node B. If α includes the terminal n, a new node labeled n is created and an edge is drawn from the node A to this node. The final graph has therefore as many nodes labeled with n as there are occurrences of n in right members of the rules. Since the grammar is clean, it is possible to embed a tree $\mathcal{T}$ in $\mathcal{G}$ such that the root of the tree is labeled by the axiom, and $\mathcal{T}$ contains all the nodes of $\mathcal{G}$. We now want to decompose the graph $\mathcal{G}$ into a set of trees. Branches of $\mathcal{T}$ are pruned. They become a tree once they are pruned, and are given a name which is a lower case. The initial tree $\mathcal{T}$ is a particular branch named with the symbol \$. The pruning of branches is done by processing $\mathcal{T}$ in a bread first manner. Each time a node N of $\mathcal{T}$ has some k input links in $\mathcal{G} \setminus \mathcal{T}$, these k links are redirected towards k newly created node of arity 0. There are two possible cases for the labeling of these created nodes. If N is the root of $\mathcal{T}$, the k nodes are labeled with the name of $\mathcal{T}$. Else, the branch of $\mathcal{T}$ beginning at N is pruned, and given a name x, and the k added nodes are labeled by x. In the last case, node N in $\mathcal{T}$ is replaced by a 0-arity node also labeled by x. Once $\mathcal{T}$ is fully processed, the pruning procedure is recursively applied to the pruned branches. The recursive provess will end, because each time it is applied, the number of nodes of the branches to be processed, strictly diminishes. When the process is ended we obtain a set of branches or trees. In this resulting set of trees, each non zero-arity node is labeled with a non terminal. We replace this non terminal by the name of the program-symbol of the rule that rewrites it. In this way, the final set of trees can be interpreted without any references to the initial grammar.

Each lower case x is to be interpreted as a `JMP x` program symbol, that we have seen in chapter 4. A cell that executes the program-symbol x just places its reading-head on the root of the grammar-tree which name is x. The program-symbol `REC` used in section 2.2 is equivalent to a program-symbol x where x is the name of the tree that is curently read by the cell. The register life of the cell L, must be decremented, not only in case of a `REC` program-symbol, but also to prevent an infinte looping between two sub-tree that call each other. This is the reason why each cell manages a stack of characters, used to record the history of its rewriting. The letter "n" must be interpreted as an `END` program symbol.

Figure C.2 shows how to transform the grammar for the parity network into a set of 1 tree. The grammar $\{\$ \overset{SEQ}{\rightarrow} AB, A \overset{PAR}{\rightarrow} An, B \overset{PAR}{\rightarrow} BB\}$ is also translated into a set of 3 trees. This last grammar derives a 2 layers network with L input units and 2^L output units. Each input unit is connected to all the output units. Since there is no need to code names of non-terminal, the encoding of a grammar as a set of trees is more compact if there are few trees. In the case of the parity grammar, the encoding as a set of rules needs $\lceil \ln(6^{4*3+2*2} * 4^6) \rceil = 55$ bits, whereas the encoding as a tree need only $\lceil \ln(6^{11}) \rceil = 29$ bits, that is about the half. Encoding a grammar in a single tree ensures that the grammar that are generated are always clean. It is a good closure property for the application of the GA. More and more people in the GA-community tends to put information into the structures manipulated by the GA, to ensure that these structures are similarly kept "clean".

C.4 Equivalence with the algebraic approach to graph grammar

In this section, we show that a cellular encoding grammar can be translated into a graph grammar of the kind proposed by Ehrig, Körff and Löwe so-called "Berlin algebraic approach". In [6] they write:

> The rewriting of a graph G, via some rule p is done by first deleting some part DEL and then adding a new part ADD finally resulting into a new derived graph H. A rule p shall only be applicable if G additionally contains some context $K_L$which is essentially to be kept. Rewriting means to replace some part $L = DEL \cup K_L$ by another one $R = K_R \cup ADD$

A simplified model is enough for our purpose. The graph DEL will always be a single node n, $K_R = K_L$ will be all the neighbor nodes of n. Therefore, the representation of a production by the pair (L, R) will be easy to understand. Moreover, ADD will most of the time include only one or two nodes.

In the network-graph used in cellular encoding the links are ordered. A network graph can be represented as an ordinary directed-graph where the links are not ordered, as indicated in figure C.3. Each cell corresponds to a node represented by a circle, each link l connecting a cell c_i to a cell c_o, corresponds to two squares s_i and s_o and one oval. Cell c_i (resp. c_o) uses the square s_i (resp. s_o) to encode the link l considered as an output link of c_i (resp. an input link of c_o). The oval

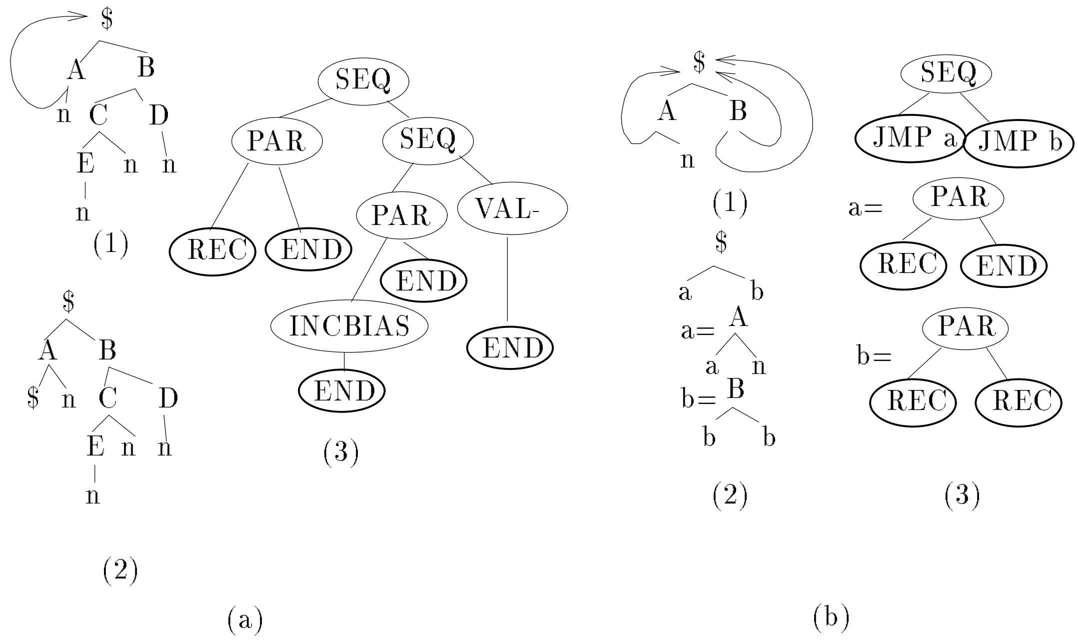

Figure C.2: Transforming a grammar into a set of trees. (a) A grammar for the parity (b) A grammar for a 2 layers network with L input units and 2^L output units. (1) Shows the graph $\mathcal{G}$ and the embeded tree $\mathcal{T}$. (2) Shows the results of the pruning. (3) Presents the final set of trees, labeled with the names of the program-symbols.

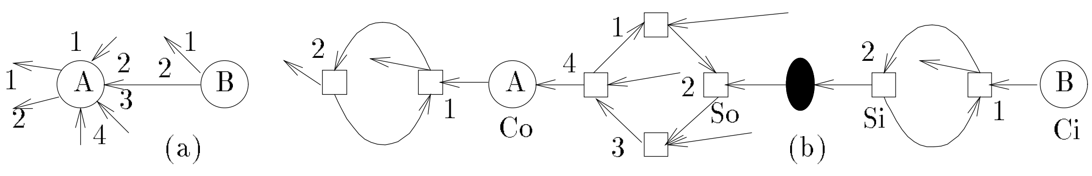

Figure C.3: How to represent a network-graph as a directed graph. The link-register of the cell labeled A points to the 4^{th} input link.

connects s_i to s_o. The squares representing the list of input-links, (or output links) of a given cell are organized in a 1-D torus. The circle representing the cell is connected to a particular square of the input list. This square corresponds to the link pointed by the link-register. The fan-in and fan-out of this directed graph is bounded by 2. The edges that fan-in or fan-out from a node need not be ordered, as was the case with the network-graph representation. The circles are labeled with non-terminal of the grammar. Ovals and squares need not be labeled. The edges may be labeled with arrows that indicate the next node to be rewritten, as will be later explained. Each production of the grammar will be translated into a corresponding production on the associated directed graph, called G.

The sequential division in figure C.4(a) can be easily translated, because its effect remains local on G. However, the parallel division is not obvious. We write the rule assuming that the cells have more than three inputs and outputs. The case where the linked list of squares has one or two squares are special cases. We do not write the rules for these special cases because they are similar to the rules in figure C.4.

The first rule of (b) duplicates the circle-node representing the cell. The next three rules 2-4 of figure C.4(b) are used to duplicate the squares corresponding to the the links of the parent cell, The 5^{th} rule describes how to duplicate the ovals. The following two rules 6-7 show how to duplicate the squares corresponding to the 1-D torus of squares of the neighbors. The neighbor cell might also have done a parallel division in this case the oval will be replicated in four, as specified by the 8^{th} rule. The oval must not be rewritten before knowing whether the cells linked through this oval duplicate or not. Edges are labeled with arrows used for this purpose. The arrows circulate through the 1-D torus of squares, two round tours for the simulation of one rewriting step. Rule 9,10,11 account for the beginning, the middle, the end of the first tour. After the first tour, the edges connecting the ovals are labeled with arrows oriented towards the ovals. Rules 9,10,11 need not be used in the case of parallel division, where the arrow on edge connecting the oval are set during the square duplications. Rules 9,10,11 are used for the sequential division (with rules 0). As shown by rule 5 or rule 8, an oval can be rewritten only when all its connections are labeled with incoming arrows. This ensures that the effect of all neighbor division has been taken into account. Rules 11,12,13 specify the beginning, middle and the end of the second tour. After the second circulation of arrows, the 1-D ring of squares is updated, and ready to start simulation of a new rewriting. The cell can apply the next rewriting rule only when both the list of squares that fans-in and fans-out have been updated. When it is the case, there are two arrows oriented towards the circle representing the cell. The circulation of arrows has another interest. It ensures that the rewriting order will be the same in the original grammar and in the graph grammar by allowing to check Invariant 2. This invariant is not exactly the same as invariant 1, but it ensures at least that both grammars derive the same network.

> **Invariant 2**: A cell cannot be rewritten twice while one of its neighbor cell has not been rewritten at all.

We now translate the rules that modify cell attributes. It is not difficult to increment or decrement the link-register. We just need to rotate the ring of squares, one square forward or one square backward. Rule 1 of figure C.5 describes how to increment the link register. Rule 2 specifies how to set the weight of the link pointed by the link register to -1. Rule 3 shows how to cut the link pointed by the link register. Rule 4,5 make the oval disappear. Rule 6,7 simplify the neighbor's linked list of squares.

Rules 4,5,6,7 figure C.5 (b) handle the flow of arrows. They do not perfectly model the previous order of execution used to develop a cellular encoding. In order to match the new rules, this order of execution must be changed for the sequential division. The new definition must be the following: If the cell divides in a sequential way, the child which reads the first letter of the right member enters the FIFO queue last (instead of first). It comes from the fact that a cell can modify its input links, but not its output links. Consider two cells c_1 and c_2 linked through a link l from c_1 to c_2. Assume c_1 makes a parallel division which makes the link l duplicate into two links numbered l and $l+1$. Let c_2 cut (or modify the weight of) l. Then translate the cellular encoding into a graph grammar. How will the network-graph be rewritten by the graph grammar? The links l and $l+1$ of c_2 will be cut (or have their weight modified). With the modification brought to sequential division, the order of rewriting of the cell will be the closure of an order induced by the directed acyclic graph of cells, where a cell is rewriten after another one if it is linked to it.

Until now we have translated everything except the managing of the life attribute. It is possible fit a more classical frame of rewriting system, by copying the approach proposed by Lindenmeyer's L-system [21]. In L-systems, the rewriting rules are applied in parallel. Let us also apply the rewriting rule of a grammar resulting from the translation of a cellular encoding, in parallel. Write the cellular encoding as an ordered set of trees. Assume that a tree numbered i can only refer to other trees numbered j, where $i \leq j$. Add a number of waiting operator so that the same number of rewriting steps C is needed to go through the different trees. The derivation for a given L value can be reproduced by the parallel application of the rules. The rewriting must be stopped at time $C * (L + i)$ for the tree i, and it is no more needed to handle a life attribute.

C.5 Conclusion

Cellular Encoding is a method for encoding neural network families into a set of named trees. This chapter shows that cellular encoding encodes a graph grammar. More precisely, we make precise how to translate a cellular encoding into a set of graph grammar rewriting rules of the kind used in the "Berlin algebraic approach to graph rewriting". This translation is done for a restricted alphabet of the cellular encoding code, and still need to be done for the whole alphabet listed in appendix D.5. We believe that this should not involve conceptual difficulties. The equivalence between cellular encoding and graph grammar brings a new light on genetic neural network. The genetic search of neural networks via cellular encoding appears as a grammatical inference process where the language is implicitly specified, not explicitly by positive and negative examples. This results in three new perspectives: We will use a method to search cellular encoding (second part of this thesis) that is, to encode a grammar into a set of trees and evolve sets of trees with the genetic algorithm. This chapter shows that this method can be used to infer any kind of grammars. Conversely, result from the research in the grammatical inference field can be used to improve cellular encoding. Last, the study of cellular encoding as a grammar can be helpful to classify cellular encoding into the complexity classes that already exist among rewriting grammars. It seems that cellular encoding is a context sensitive, parallel graph grammar.

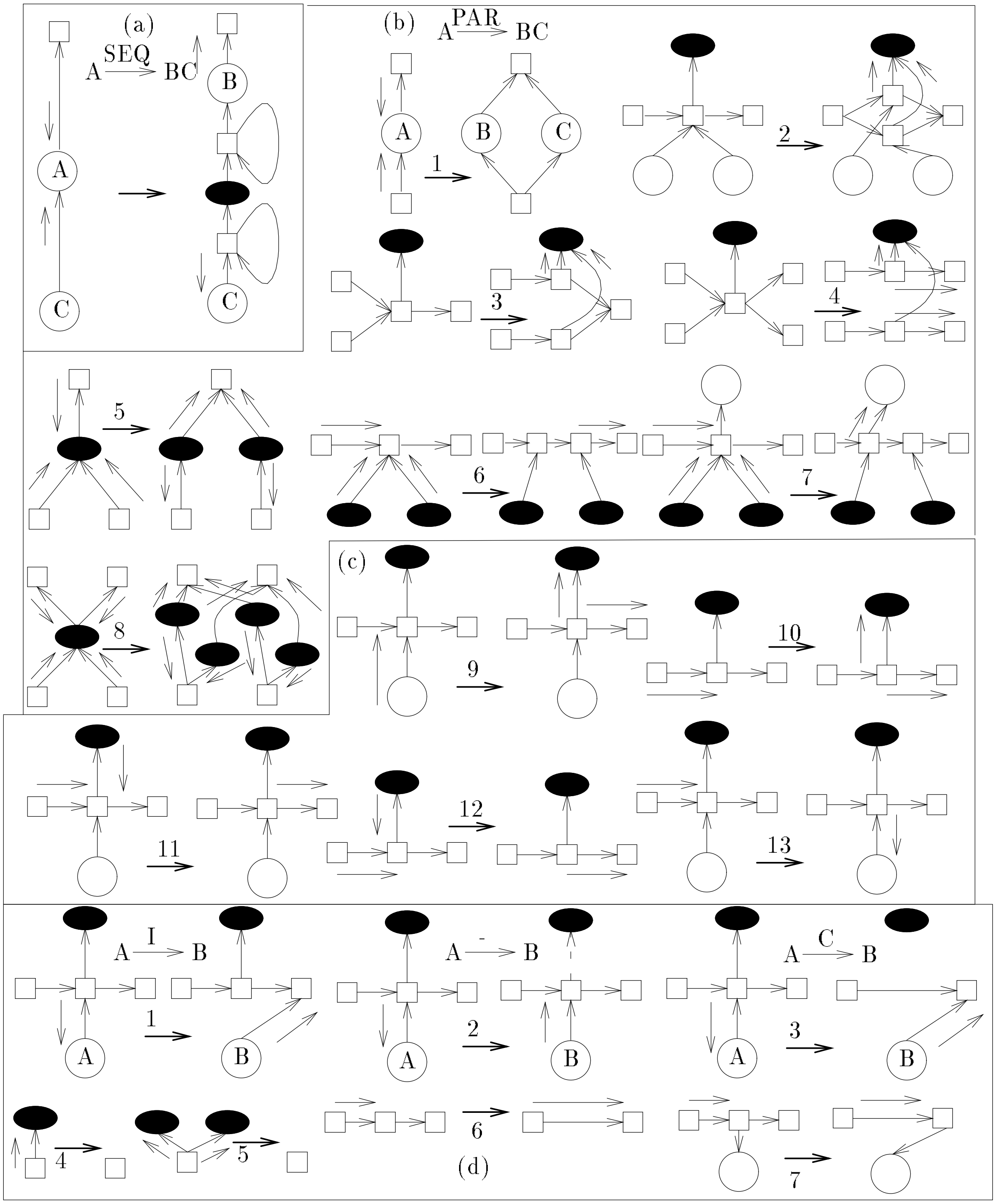

Figure C.4: Translation of the rule for cell division. (a) The sequential division (b) The parallel division (c) Managing of the edge labels (arrows).

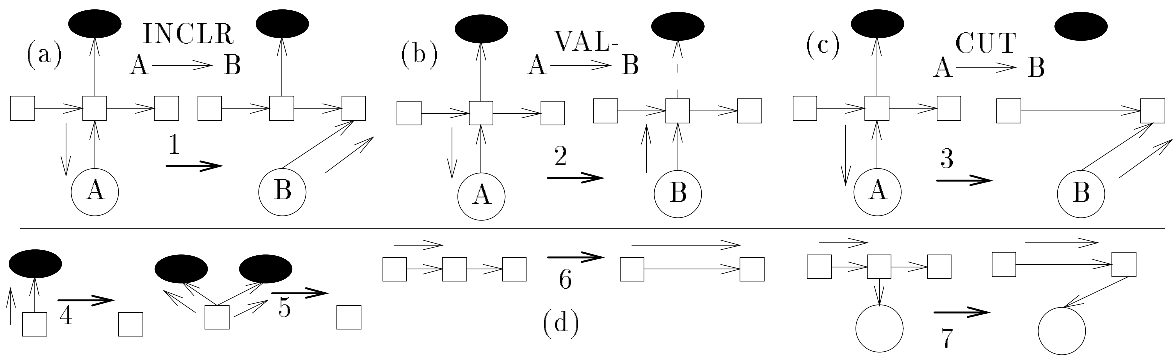

Figure C.5: Translation of the rules for cell modification

Appendix D

Compléments techniques

D.1 Petit exemple de compilation

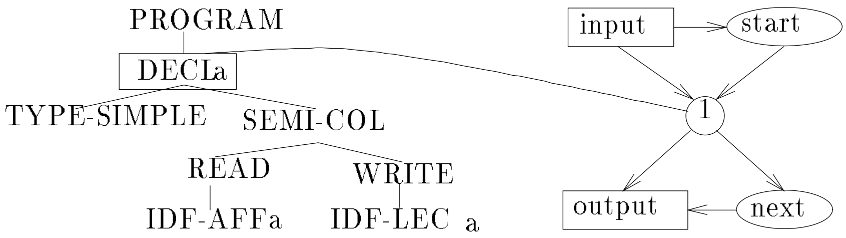

Figure D.1: Exécution à l'étape 1 du macro-opérateur PROGRAM. La cellule ancêtre a donné naissance à deux autres cellules. Une cellules étiquetée "start" et une autre étiquetée "next". La cellule "start" est utilisée pour démarrer le réseau, et pour gérer la propagation des activités. La cellule "next" n'a pas fini son développement. Elle attend que toutes ses cellules voisines en entrée soient devenues des neurones (c'est à dire qu'elles aient perdu leur tête de lecture) pour continuer. A présent, la cellule ancêtre va lire le noeud étiqueté DECL.

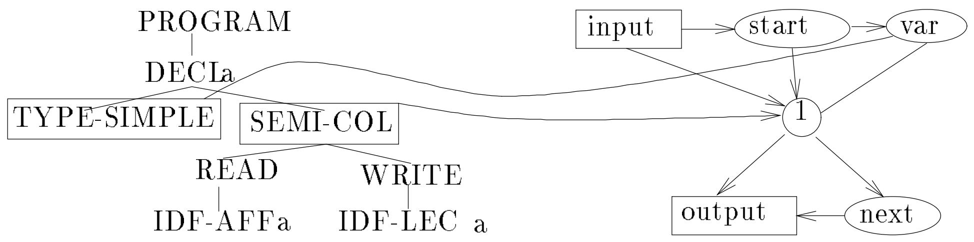

Figure D.2: Exécution à l'étape 2 du macro-opérateur DECL. La cellule ancêtre donne naissance à une autre cellule étiquetée "var". Cette cellule représente la variable "a". Les connexions entrantes et sortantes de la cellule ancêtre sont à présent complètes et représentent une sorte d'invariant topologique vérifié avant la compilation neuronale de chaque instruction. le premier lien entrant pointe vers la cellule "pointeur entrée ", le second lien entrant pointe vers la cellule "start", les liens entrées suivants pointent vers les neurones représentant les variables de l'environnement. Le premier lien en sortie pointe vers la cellule "pointeur sortie ", le deuxième lien pointe vers une cellule "next" qui sert à établir les connexions vers l'amont.

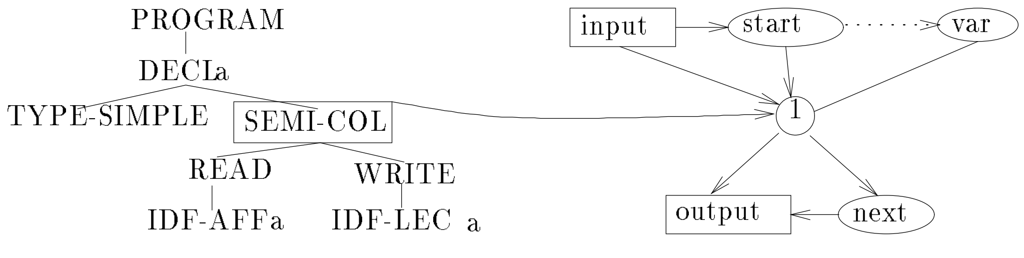

Figure D.3: Exécution à l'étape 3 du macro opérateur TYPE-SIMPLE Le macro-opérateur TYPE-SIMPLE a pour simple effet de positionner sur 0 (ligne pointillée) le poids reliant le neurone représentant la variable "a". Ceci n'est pas fait par le macro-opérateur DECL, car on ne peut pas savoir à l'avance si "a" ne sera pas de type composé.

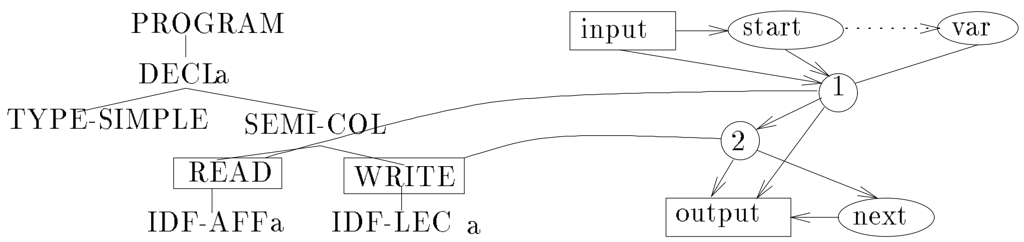

Figure D.4: Exécution à l'étape 4 du macro opérateur SEMI-COL Cet opérateur sert à séquencer deux instructions. La cellule "1" donne naissance à une autre cellule étiquetée "2". La cellule "1" va exécuter les macro-opérateurs correspondants à la première instruction `read a`. Pendant ce temps, la cellule "2" est bloquée sur l'instruction d'après, qui est `write a`.

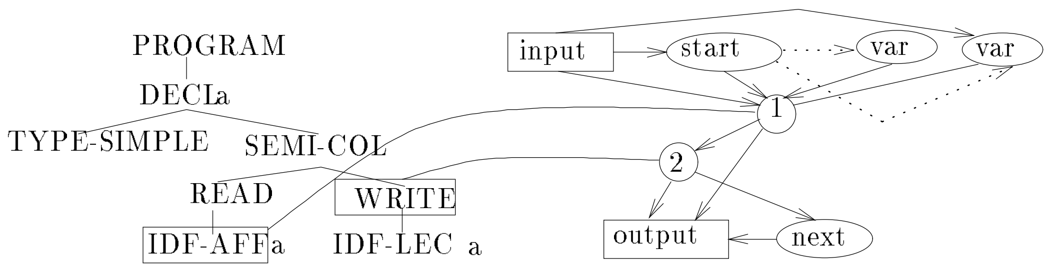

Figure D.5: Exécution à l'étape 5 du macro opérateur READ. Un neurone "var" est créé, il contient la nouvelle valeur de la variable a. Il est relié a la cellule pointeur entrée. Cela signifie que la nouvelle valeur de a sera rentrée de l'extérieur.

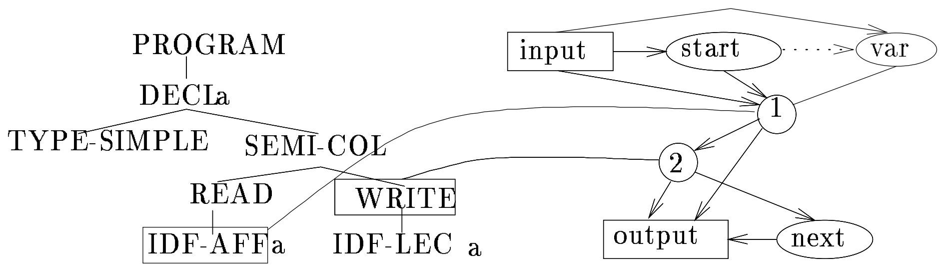

Figure D.6: Exécution à l'étape 6 du macro opérateur IDF-AFF. Le neurone contenant l'ancienne valeur de a est éliminé car il n'a plus de liens en sortie. La cellule "2" est débloquée, elle continue son développement

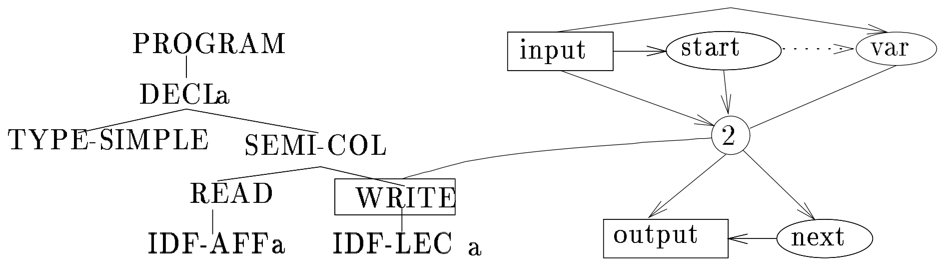

Figure D.7: Déblocage à l'étape 7 de la cellule "2". Avant de continuer la compilation du WRITE, la cellule "2" exécute un petit code cellulaire. Elle récupère les liens d'entrée de la cellule "1". La cellule "1" disparaît, on a restauré l'invariant, la compilation de l'instruction suivante peut commencer.

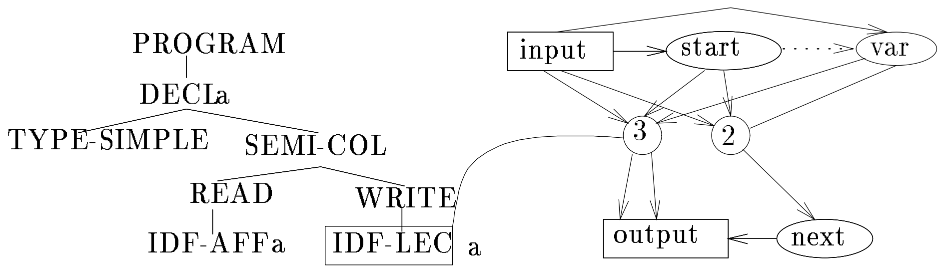

Figure D.8: Exécution à l'étape 8 du Macro opérateur WRITE. La cellule "2" donne naissance à une autre cellule "3" connectée à la cellule "pointeur sortie ". Cette cellule "3" possède touts les liens d'entrées de la cellule "2" mère. La cellule "2" devient neurone.

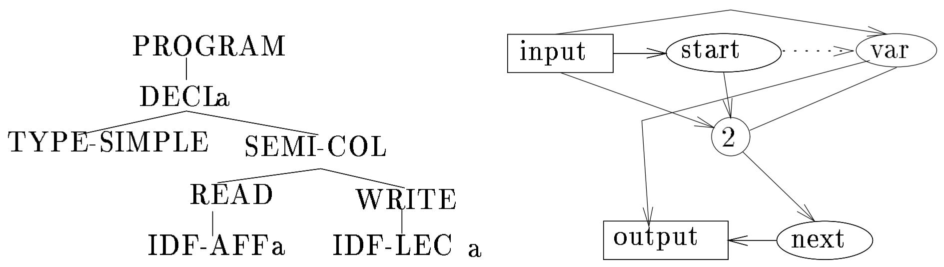

Figure D.9: Exécution à l'étape 9 du macro opérateur IDF-LECT. La cellule "3" sélectionne l'entrée correspondant à la variable "a", pour la connecter à la cellule pointeur sortie. Puis la cellule "3" disparaît.

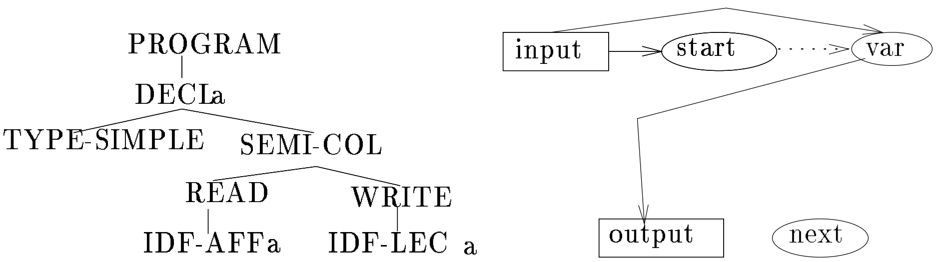

Figure D.10: La cellule "next" se débloque, et disparaît, sa supression entraîne la suppression de plusieurs autres neurones. Le réseau de neurones final comprend deux neurones d'entrée: la cellule "start" et la variable "a" et un neurone de sortie: la variable "a". Il lit "a" puis l'écrit, traduisant ce qui est spécifié dans le programme Pascal

D.2 Les autres macro-opérateurs

On représente ici l'effet des macro-opérateurs non-encore illustrés sur l'exemple simple précédent.

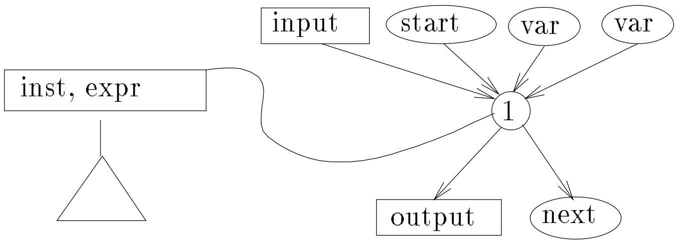

Figure D.11: Motif invariant. On montre ici le graphe de cellules courant, après la déclaration de deux variables scalaires. On voit bien les deux neurones correspondants étiquetés "var"". Ces neurones "var"" avec le neurone "start"constituent la première couche du futur réseau de neurones. Ce graphe sera choisi comme configuration initiale pour chacun des prochains macro-opérateurs décrits, pour éviter la répétition, et bien montrer la propagation du même motif invariant représenté par cette configuration initiale.

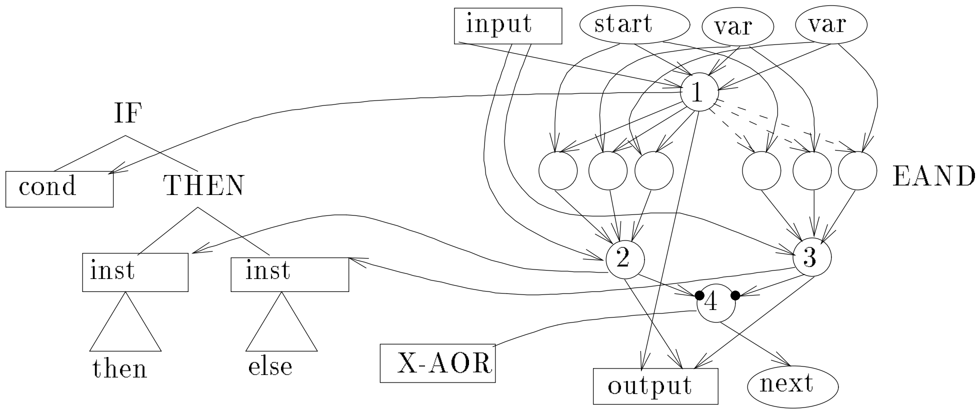

Figure D.12: Exécution des macro-opérateurs IF et THEN par la cellule ancêtre du motif invariant. Pour la concision, on a représenté l'action combinée des deux macro-opérateurs IF et THEN. Il est facile de le redécomposer. Quatre cellules de lecture et une couche de 6 neurones EAND sont créés. La cellule "1" va développer un sous-réseau permettant de calculer la valeur de la condition du IF. La valeur “vraie” est codée sur 1, la valeur faux est codée sur -1. L'opération logique NOT est réalisée à l'aide d'un poids -1 représenté en pointillés. Suivant la valeur de la condition, le flot d'activités représentant l'environnement courant est envoyé à droite ou à gauche. Ce sont les neurones EAND qui jouent le rôle d'aiguillage. Un neurone EAND est passant si sa première entrée est à 1, il est bloquant sinon. Les cellules "2" et "3" développent respectivement le corps du THEN et le corps du ELSE. La cellule "4" posséde deux sous-sites de sortie, shématisés par deux disques noirs. Elle est bloquée sur un code cellulaire X-AOR. Elle est destinée à créer une couche de neurones qui rassemble les lignes de sortie du THEN et du ELSE.

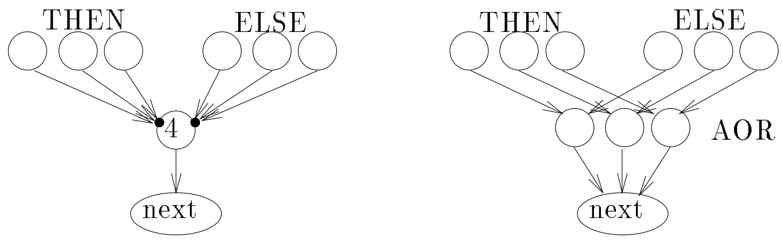

Figure D.13: Exécution du code cellulaire X-AOR par la cellule "4" de la figure précédente. La cellule "4" possède deux sites d'entrée représentés par deux disques noirs, qui lui permettent de répartir ses liens d'entrée en deux listes. Le code cellulaire permet ensuite de traiter ces deux listes globalement. Le résultat est la production d'une couche de neurones AOR. Cette couche de neurones récupère soit l'environnement de sortie du corps du THEN, soit celui du corps du ELSE, pour le transmettre à l'instruction qui suit.

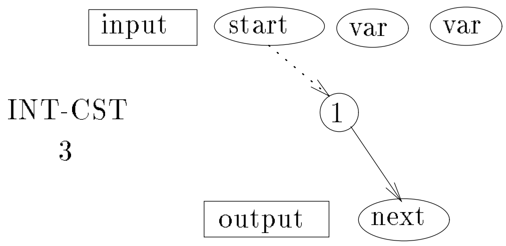

Figure D.14: Exécution du macro-opérateur INT-CST par la cellule ancêtre du motif invariant. Ce macro-opérateur est paramétré par la valeur de la constante. INT-CST 3 a pour effet de positionner le biais de la cellule ancêtre à la valeur 3, de supprimer tout les liens excepté le lien vers le neurone "start" et le neurone "next", de mettre le poids du lien vers le neurone "start" à la valeur 0, et la sigmoïde de l'ancêtre à la fonction identité. Lorsque la constante est nécessitée, le neurone "start" sert à activer le neurone 1 qui délivre alors la constante codée dans son biais.

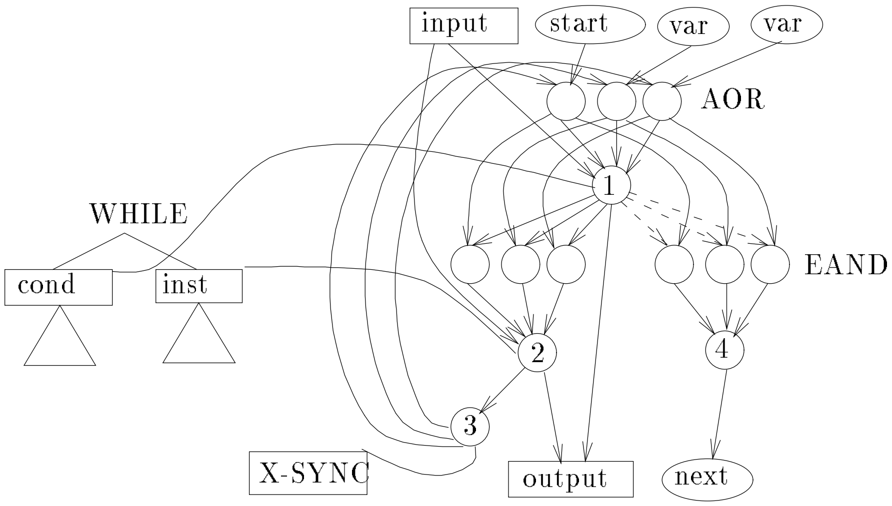

Figure D.15: Exécution du macro-opérateur WHILE par la cellule ancêtre du motif invariant. Sont créés: trois cellules de lecture, 3 neurones AOR, 6 neurones EAND, et un neurone étiqueté 4, utilisé pour transmettre l'environnement à l'instruction suivant le WHILE. La couche de neurones AOR sert de mémoire tampon pour stocker l'environnement entre deux passages dans la boucle. La dynamique AOR est utilisée, parce que le flot d'activités arrive soit d'en haut (premier passage), soit d'en bas (passage suivant), mais jamais des deux en même temps. La cellule "1" développe un réseau qui calcule la condition du while. La cellule "2" (à gauche) développe le corps du WHILE. Si la condition du WHILE est vraie, le flot d'activités peut passer dans le corps de la boucle (à gauche), car les 3 EAND de gauche sont passants. Si la condition du WHILE est fausse, ce sont les trois EAND de droite qui sont passants, et le flot d'activité passe par la droite et sort de la boucle. La cellule "3" développe une couche de neurones intermédiaires dans la boucle, dont le rôle est de synchroniser le flot d'activité pour éviter des collisions.

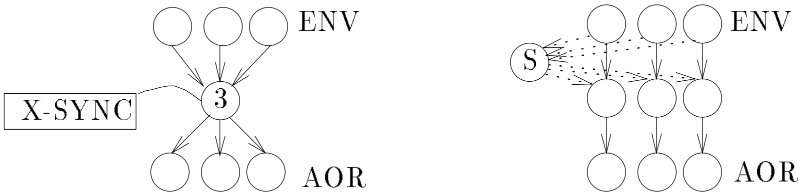

Figure D.16: Exécution du code cellulaire X-SYNC par la cellule "3" de la figure précédente. La cellule "3" donne naissance à une couche de 3 neurones plus un neurone "S" dont la dynamique est normale (dynamique $\mathcal{D}_0$), intercalée entre la couche de neurones stockant l'environnement sortant du WHILE étiquetée ENV, et la couche de neurones tampon AOR qui se trouve en entrée de boucle. Le neurone "S" étant connecté en entrée à tout l'environnement, ne devient actif que lorsque tout l'environnement est disponible. Le neurone "S" est relié à la couche de neurones normaux par des poids 0, qui servent uniquement à bloquer le passage du flot, tant que le neurone "S" n'est pas actif. Ainsi toutes les activités rebouclent en même temps, et le flot est synchronisé.

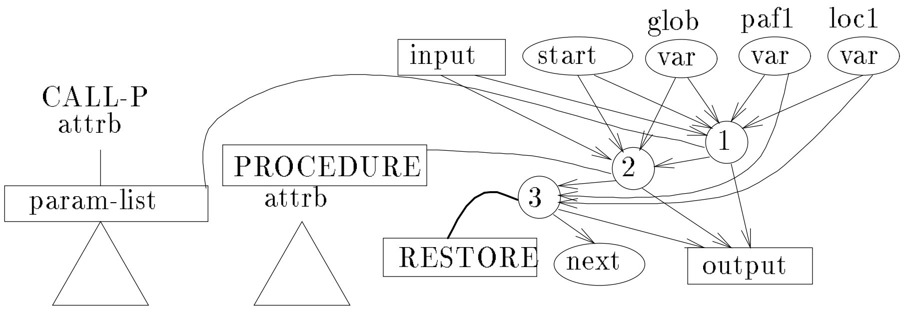

Figure D.17: Exécution du macro-opérateur CALL-P par la cellule ancêtre du motif invariant à trois variables. Ce macro-opérateur est paramètré par le nom de la procédure appelée. Trois cellules sont créées. La cellule "1" est chargée de développer plusieurs sous-réseaux de neurones: un sous-réseau pour calculer la valeur de chaque paramètre d'appel. La cellule "2" va développer les instructions qui constituent le corps de la procédure appelée. Elle place sa tête de lecture sur la racine de l'arbre syntaxique de la procédure. On remarque que la cellule "2" n'est pas reliée aux neurones contenant les variables locales de la procédure appelante "pef1" et "loc1". Ces variables ne sont en effet pas accessibles depuis "proc2". La cellule "3" est bloquée sur un code cellulaire RESTORE dont l'objet est de restaurer l'environnement de la procédure appelante, lorsque la procédure appelée est terminée. Pour ce faire, la cellule "3" garde des connexions vers les variables "pef1" et "loc1" qui sont des variables locales dans la procédure appelante

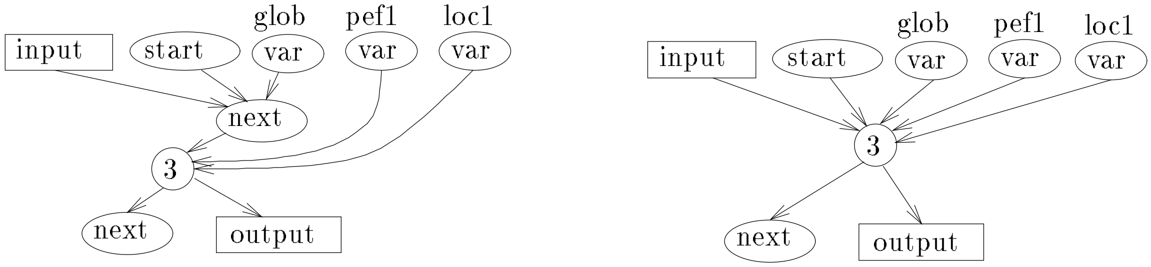

Figure D.18: Exécution du code cellulaire RESTORE par la cellule "3" de la figure précédente. Lorsque la cellule "3" se débloque, le corps de la procédure proc2 est développé. La cellule "3" est reliée à une cellule "next" elle-même reliée aux variables Pascal globales éventuellement modifiées par la procédure appelée. L'exécution de RESTORE réunit les nouvelles variables globales, et les anciennes variables locales de la procédure appelante.

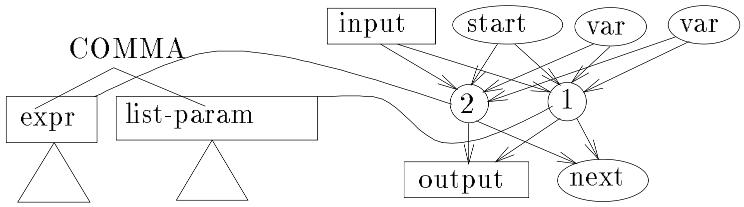

Figure D.19: Exécution du macro-opérateur COMMA par la cellule ancêtre du motif invariant. Ce macro-opérateur crée deux cellules. La cellule "2" va développer un sous-réseau pour calculer la valeur d'un paramètre d'appel, la cellule "1" va continuer à développer les autres paramètres. La cellule "next" est celle qui va exécuter le corps de la procédure.

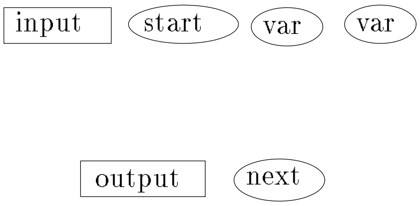

Figure D.20: Exécution du macro-opérateur NO-PARAM par la cellule ancêtre du motif invariant. Ce macro-opérateur permet simplement un retour en arriére. La cellule 1 de la figure précédente allait développer un autre paramètre. Elle est arrivée en fait, au bout de la liste des paramètres. Ce bout est marqué par NO-PARAM. En exécutant le macro-opérateur NO-PARAM, la cellule "1" s'élimine.

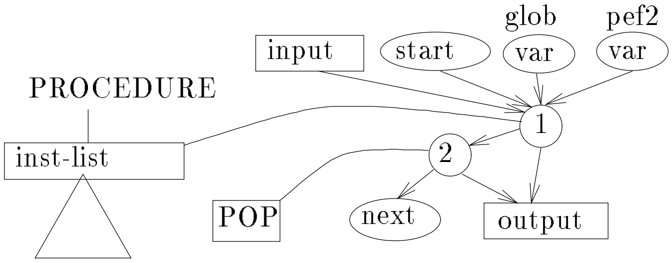

Figure D.21: Exécution du macro-opérateur PROCEDURE par la cellule ancêtre du motif invariant. Ce macro-opérateur rajoute une cellule "2" intermédiaire bloquée sur un code cellulaire POP. Cette cellule se développera lorsque le corps de la procédure sera développé. Elle sert à couper les connexions vers les variables locales de la procédure appelée, de même que lorsqu'une sous-routine écrite en langage machine rend la main, elle dépile les variables locales. La cellule "1" n'a pas encore le neurone "loc2" dans son environnement, celui-ci est généré par les déclarations internes à la procédure appelée.

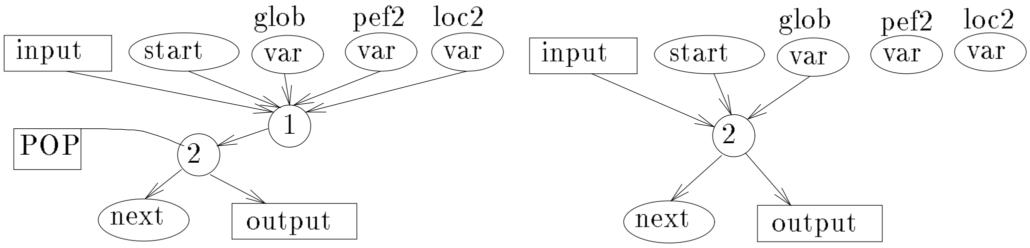

Figure D.22: Exécution du code-cellulaire POP par la cellule "2" de la figure précédente. Ce code permet de récupérer seulement la partie de l'environnement qui est globale. Les variables locales ne sont plus reliées au neurone "2" après son exécution. Ce macro-opérateur est équivalent à l'instruction "dépiler" en langage machine, utilisée pour le retour des sous-programmes.

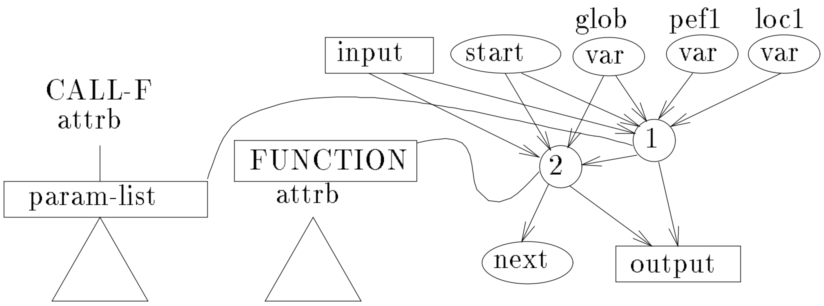

Figure D.23: Exécution du macro-opérateur CALL-F par la cellule ancêtre du motif invariant. L'environnement contient deux variables locales. Le macro-opérateur CALL-F a le même effet que CALL-P, sauf qu'il ne crée pas la cellule "3" car il n'est pas nécessaire de restaurer l'environnement.

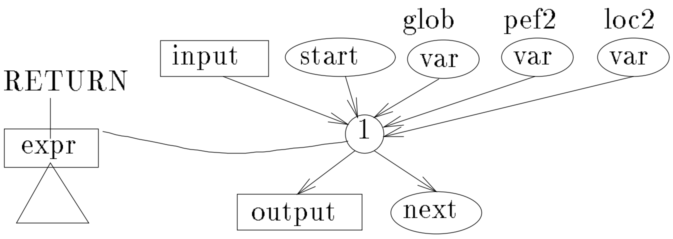

Figure D.24: Exécution du macro-opérateur RETURN par la cellule ancêtre du motif invariant. Ce macro-opérateur a un effet nul.

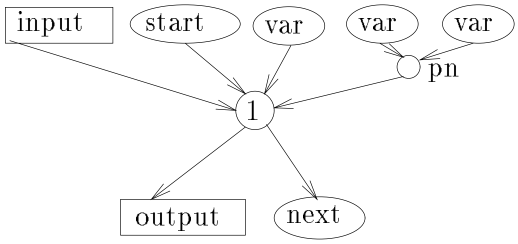

Figure D.25: Motif invariant étendu. L'environnement contient une variable scalaire et un tableau à deux éléments.

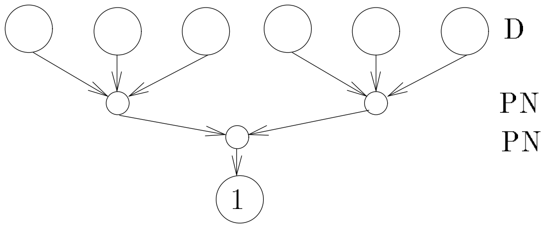

Figure D.26: Représentation d'un tableau à deux dimensions en utilisant un arbre de neurones pointeur, de profondeur 2. "PN" est mis pour "neurones pointeurs". La couche de neurones marquée "D" contient les données du tableau.

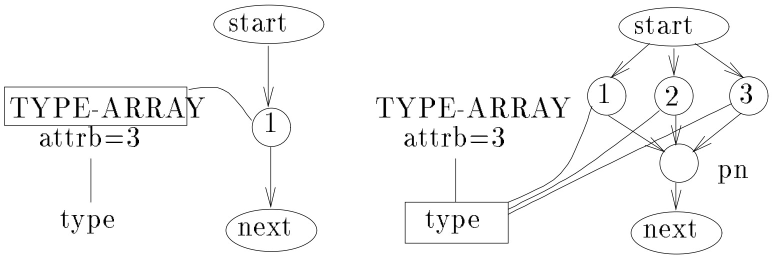

Figure D.27: Exécution du macro-opérateur TYPE-ARRAY. Ce macro-opérateur permet de développer la structure de neurones pointeurs, nécessaire pour coder un tableau. Il faut d opérateurs de ce type pour coder un tableau de d dimensions. Ce macro-opérateur est paramétré par un entier qui indique le nombre de colonnes de la dimension courante. C'est le seul exemple de macro-opérateur pour lequel plusieurs cellules vont lire le même noeud de l'arbre syntaxique.

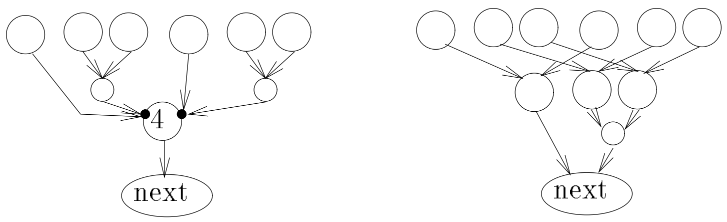

Figure D.28: Exécution du code cellulaire X-AOR par la cellule "4" de la figure décrivant le macro-opérateur IF. Ce macro-opérateur produit une couche de neurones AOR, et recopie les neurones pointeurs structurant les données. Cette couche de neurones récupère soit l'environnement de sortie du corps du THEN, soit celui du corps du ELSE, pour le transmettre à l'instruction qui suit.

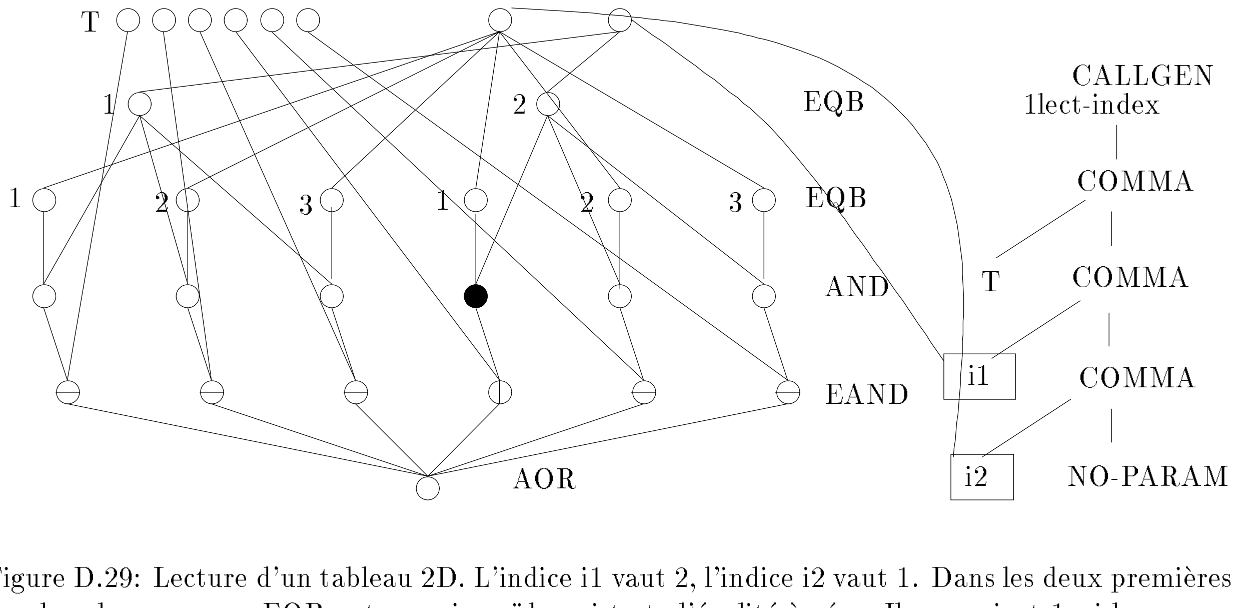

Figure D.29: Lecture d'un tableau 2D. L'indice i1 vaut 2, l'indice i2 vaut 1. Dans les deux premières couches, les neurones EQB ont une sigmoïde qui teste l'égalité à zéro. Ils renvoient 1, si la somme pondérée des entrées moins le seuil vaut 0, et -1 sinon. Ces neurones ont tous des seuils différents qui sont indiqués. La troisième couche de neurones effectue des AND logiques. Sur cette couche, un seul neurone renvoie 1 (le disque noir), celui qui correspond à l'élément que l'on veut lire. Tout les autres sont à -1. La quatrième couche de neurones est une couche de neurones EAND, un seul des neurones de cette couche est passant, il laisse passer l'élement du tableau que l'on désire lire. Enfin un neurone AOR est chargé de collecter la donnée que l'on récupère à la sortie. L'instruction CALLGEN est à peu près comme CALL-F, sauf que tous les liens vers les neurones contenant les valeurs de variables globales, sont coupés. CALL-GEN est décrit deux figures plus loin.

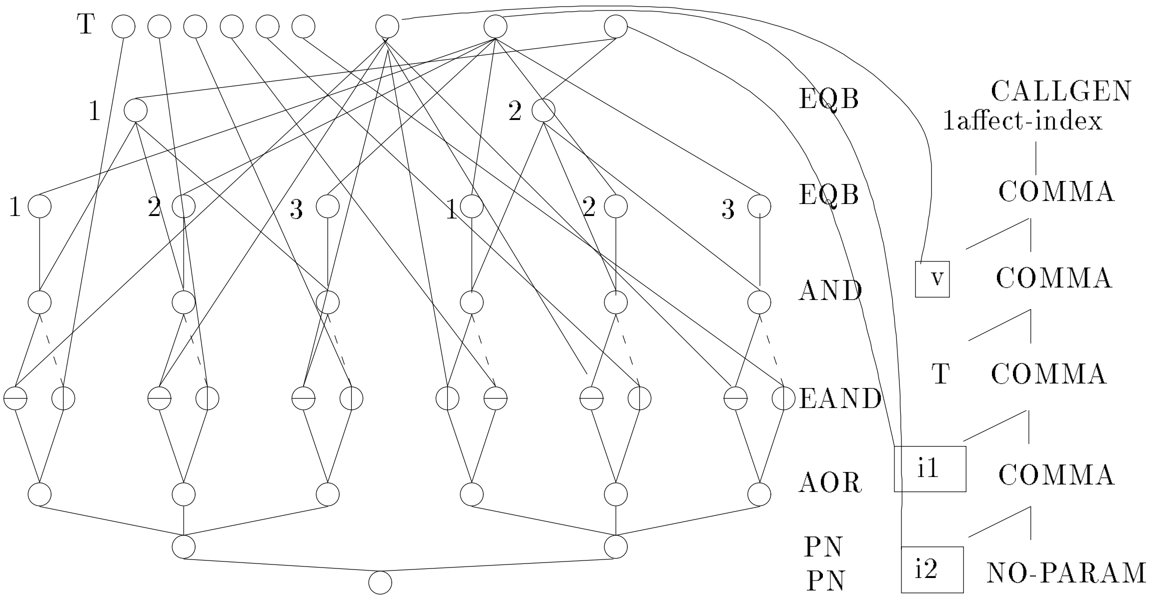

Figure D.30: Ecriture d'un tableau 2D. l'indice i1 vaut 2, l'indice i2 vaut 1. Les trois premières couches de neurones sont les mêmes que dans la figure précédente. La quatrième couche contient deux fois plus de neurones EAND. Ces neurones EAND sont disposés par paires. Dans chaque paire, un des deux neurones est passant, l'autre est bloquant. Le neurone de gauche est toujours bloquant, excepté dans un seul cas: le cas où la paire correspond aux indices indiquant l'endroit où l'on veut écrire. La couche de neurones suivante est une couche de neurones AOR. Chaque neurone AOR transmet un élément du tableau, il récupère soit l'ancienne valeur de cet élement, soit la nouvelle valeur v. Vient en dernier une structure d'arbre de neurones pointeurs, dont le rôle est de conserver la structure de tableau 2D.

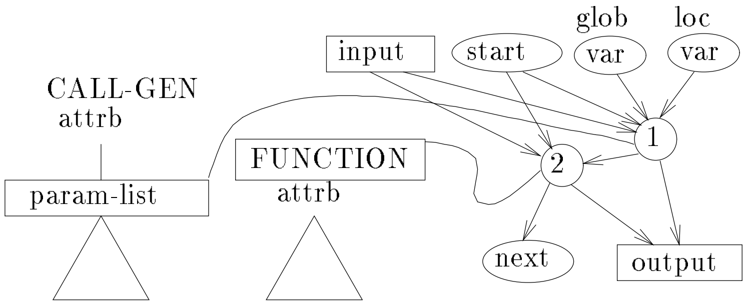

Figure D.31: Exécution du macro-opérateur CALLGEN par la cellule ancêtre du motif invariant. CALLGEN a le même effet que CALL-F sauf que la cellule "2" devant exécuter le corps de la fonction écrite en code cellulaire, ne possède plus les connexions vers les variables globales du langage Pascal

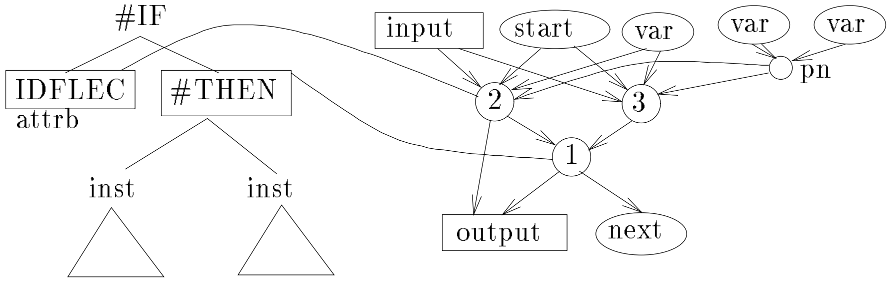

Figure D.32: Exécution du macro-opérateur #IF par la cellule ancêtre du motif invariant. Trois cellules sont créées. La cellule "2" va lire un opérateur IDF-LEC lui permettant de sélectionner la variables tableau de l'environnement à tester. Le test achevé, la cellule "3" transmettra l'environnement à la cellule "1".

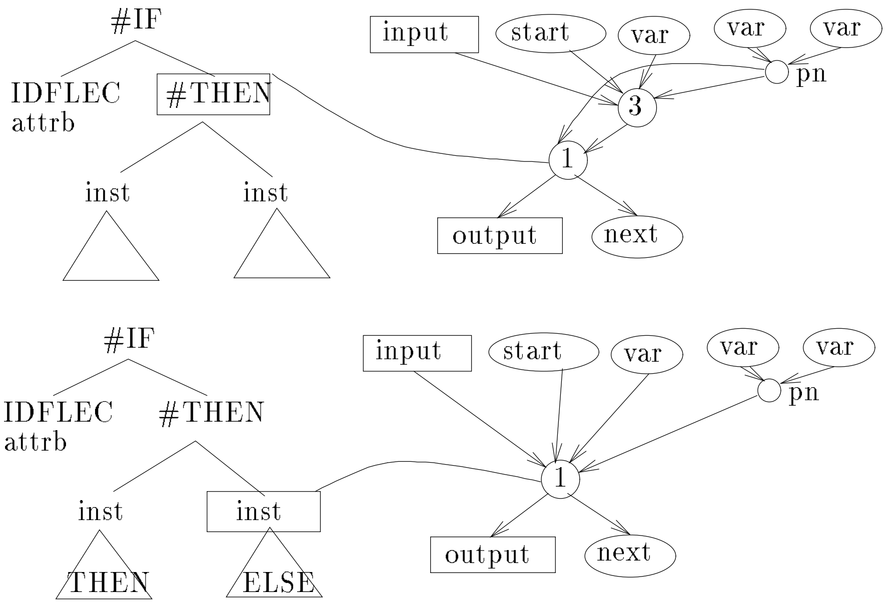

Figure D.33: Exécution du macro-opérateur #THEN par la cellule "1" de la figure précédente. Après exécution du macro-opérateur IDF-LEC, la cellule "1" s'est débloquée et a recopié les liens d'entrée de sa voisine d'entrée numéro 1. La situation est décrite en haut de la figure. Le macro-opérateur teste le nombre de liens entrants de son premier voisin. Ici il vaut 2, car le tableau a deux éléments. Si ce nombre est supérieur à 1, la cellule "1" positionne sa tête de lecture sur le sous-arbre droit, comme c'est le cas ici. Sinon elle va lire le sous-arbre gauche. Le sous-arbre gauche contient le corps du #THEN, et le sous-arbre droit contient le corps du #ELSE.

D.3 Les registres des cellules

Nom	Utilisation
Tête de lecture	Lire à un endroit du code cellulaire
Vie	Compter le nombre de récursions
Biais	Stocker le biais
Sigmo	Stocker la sigmoïde
Dyn	Stocker la dynamique
nu-lien	Indiquer un numéro de lien
simplif	Indiquer le niveau de simplifiabilité
x,y,dx,dy	Repérer l'endroit où afficher la cellule
Type	Indiquer s'il s'agit d'un neurone ou d'une cellule de lecture

Tableau D.1: Les registres d'une cellule

Par ailleurs les liens ont trois registres, pour stocker le poids `weight`, l'état `state` et la marque de début de sous-site `begin-sub-site`. Les sites ont un registre appelé `divisable`.

D.4 Syntaxe du micro-codage des opérateurs cellulaires

La grammaire ci-après génére des microcodes d'opérateurs.

```
 <microcode> ::= <top> | <div> |  <reg> | <exe> | <aff>
<top> ::= -TOP<segments> | TOP<segments>
<div> ::= <div1> | <div2>
<div1>::= -DIV<segments> | DIV<segments>
&lt;segments&gt;::=&lt;segment&gt;&lt;segments&gt; | &lt;segment&gt;
&lt;segment&gt;::=<operateur><operande> |<operateur>^<operande> | s
<operateur>::=  m | M | r | R | d | D
<operande> ::= * | > | >= | < | <= | = | $ | #
<div2> :: =HOM<chiffre>
<reg> ::= CELL<caractere> | SIT<caractere> | LNK<caractere>
<exe> ::= EXE<caractere> | BRA<caractere>
<aff> ::= AFF<caractere>
```

Le nom terminal <chiffre> se réécrit en un chiffre, le nom terminal <caractere> se réécrit en un caractère. Ce caractère est: pour le nom terminal CEL, SIT, LNK le nom abrégé d'un registre de la cellule, du site ou du lien, pour EXE il rappelle quelle sorte de mouvement la tête de lecture fait, pour BRA, il indique le type de condition testée pour décider le branchement, pour AFF il indique quelle type de neurone on a.

D.5 Listes des opérateurs et de leur microcode

La lettre x indique que l'opérateur utilise un argument entier. i est le nombre de liens d'entrée, o est le nombre de liens de sortie, r est la valeur du registre de lien.

```
******************Local Topological Transformation***************************
  CUTL x   TOPm-<m->    Cuts the input links i-x
  CUTRI x  TOPm<        Cuts the right input links starting from link x
  CUT x    TOPm<m>      Cuts input link  x
  CLIP     TOPm_<m_>    Cuts input link  r
  CUTO x   -TOPm<m>     Cuts output link  x
  MRG x    TOPm<d^*m>   Merges the input link of input neighbour x
  MG x     TOPm_<d_^*m_>Merges the input link of input neigbour r
  CPFO x  -TOPm<d^#m>=  Copies  first output of output cell x
  CHOPI x  TOPm=        Cuts all the input links except link x
  CHOPO x  -TOPm=       Cuts all the output links except link x
  CHHL     TOPm|>       Cuts the first input links from 1 until i/2
  CHHR     TOPm|<=      Cuts the last input links from i/2+1 until i.
  PUTF x   TOPm=m<m>    Puts link x in first position
  PUTL x   TOPm<m>m=    Puts link x in last position
  SWITCH x TOPm<m$m>$m= Makes a link permutation between  last link and link x
  KILL     TOP          Deletes the neurone
  CYC      TOPs         Creates a recurrent link
********Cell division into two childs which will execute a separate   code******
  PAR      DIVd*R*      Parallel  division
  SEQ      DIVs         Sequential division
  XSPL x   DIVm<=s      Gradual division
  SEP      DIVd*M|>     Separation division
  ADL x    DIVm<sm>     Adds a cell on link x
  AD  x    DIVm_<sm_>   Adds a cell on link r
```

```
  IPAR x   DIVm<sd=m>   Duplicates link x, Add a cell on link x
  SPLFI x  DIVd<=s      Duplicates first x inputs
  SPLFIL x DIVd-<=s     Duplicates first i-x inputs
  SPLLI x  DIVsd->=     Duplicates last input starting at link x
  SPLLIL x DIVsd>=      Duplicates last input starting at link i-x
******Cell division into more than two childs which will execute the same code***
  CLONE x  OLC          Clones into x childs, x is the argument
  SPLIT    HOM0         Splits into y childs, y is computed from the topology
  TAB      HOM1         Same has split, but child i as its bias set to i
  SPLITN   HOM2         Same as split, but the sub-sites are merged
************************Modification of  a register******************************
  SBIAS x  CELb         Sets the bias to the argument
  SBIAS0   CELb 0       Sets the bias to 0
  SBIAS1   CELb 1       Sets the bias to 1
  INCBIAS  CEL+b 1      Increments the bias
  DECBIAS  CEL+b -1     Decrements the bias
  SSIGMO x CELs         Sets the type of sigmoid
  SDYN x   CELd         Sets the dymamic
  EVER x   CELe         Sets the level of simplifiability
  LR x     CELl         Sets the link register
  INCLR    CEL+l 1      Increments the link register
  DECLR    CEL+l -1     Decrements the link register
  SITE     SITD         Sets the divisability of the sub sites to the argument
  SITI     SITd         Sets the divisability of the input sub-sites to the argument
  SITO    -SITd        Sets the divisability of the output sub-sites to the argument
  DELSITE  SITO         Merges all the input sites and all the output sites
  DELSITI  SITo         Merges all the input sites
  DELSITO  -SITo        Merges all the output sites
  MULTSIT  SITm         Creates one input site for each input link
  VAL x    LNKv         Sets the value of the input weight r to x
  VALO x   -LNKv        Sets the value of the output weight r to x
  VAL+     LNKv 1       Sets the value of the input weight r to 1
  VAL-     LNKv -2      Sets the value of the input weight r to -1
  INC      LNK+v 1      Increments the value of the weight on the input link r
  DEC      LNK+v -1     Decrements the value of the weight on the input link r
  MULT2    LNK*v 2      Doubles the value of the weight on the input link r
  DIV2     LNK/v 2      Halves the value of the weight on the input link r
  RAND x   LNK?         Sets weight x to a random value in {-1, 0, 1}
  ON       LNKs 1       Sets the state of the link to on
  OFF      LNKs 0       Sets the state of the link to off
************************Managing of the execution order*************************
  WAIT x   EXEw         Waits the specified number of steps
  WAIT     EXEf         Waits one step if no arguments are supplied
  JMP  x   EXEj         Includes a subnetwork  which number is the argument
  JMP      EXEj         Uses a pattern matching mechanism if no arguments are supplied
  REC      EXEr         Jumps to the currently read subtree
  END      EXEe         Becomes a finished neuron
  BLOC     EXEq         Waits for the neigbour to loose their reading head
  NOP_0    EXEw         Is a label
  NOP_1    EXEw         Is a label
  DEF      EXEw         Starts a new branch that encodes a subnetwork
  BLIFE    BRAf         Tests if the value of the life register is one
  BLIF2    BRAg         Tests if the value of the life register is the initial value
  BUN x    BRAu         Tests if the neigbour has x input links
  BPN      BRAp         Tests if the neighbour is a pointer neuron
  BLR x    BRAl         Tests if the value of the link register equals the argument
************************Enhancing display of the graph***************************
  PN       AFFpn        Draws a pointer neuron
```

```
 VAR     AFFvar       Draws a neuron that contains the value of a variable
************************Operators for labeling*********************************
 DINS    EXEf         It indicates insertion of the right sub-tree
 GINS    EXEf         It indicates insertion of the left sub-tree
 PN_     CELe         It is a pointer neuron
 VAR_    EXEe         It contains the initial value of a variable
 EAND_   EXEe         It has a dynamic EAND
 AOR_    EXEe         It has a dynamic AOR
 INF_    EXEe         It compares the 2 inputs
 SUP_    EXEe         It compares the 2 inputs
 INFEQ_  EXEe         It compares the 2 inputs
 SUPEQ_  EXEe         It compares the 2 inputs
 PLUS_   EXEe         It adds the 2 inputs
 MOINS_  EXEe         It substracts the first input from the other
 MULT_   EXEe         It multiplies the 2 inputs
 QUO_    EXEe         It divides the first input by the other
 NOT_    EXEe         It does a logical NOT
 OR_     EXEe         It does a logical OR
 AND_    EXEe         It does a logical AND
 NEQ_    EXEe         It tests if the two inputs are different
 EQ_     EXEe         It tests if the two inputs are equal
 EQB_    EXEe         It tests if the input is equal to bias
 START_  EXEe         It is a Start neuron
```

D.6 Syntaxe des arbres syntaxique utilisés

La grammaire ci-après génère des expressions parenthésées qui sont interprétables par des arbres syntaxiques corrects. Un arbre syntaxique peut être celui d'un programme, celui d'une procédure ou celui d'une fonction. Le compilateur doit générer un ensemble d'arbres. Un arbre pour le programme principal et un arbre pour chaque fonction ou procédure. La représentation parenthésée utilisée ici, est inhabituelle. On omet de mettre des parenthéses lorsqu'il y a un seul sous arbre, on se contente de mettre un blanc. Lorsqu'il y a deux sous-arbres, on considère qu'il y a un tronc qui part à droite et une branche qui part à gauche. Seule la branche est mise entre parenthèses. Cette représentation spéciale permet de décrire les arbres syntaxiques avec une grammaire simple. Les lettres majuscules et les parenthèses sont des terminaux. Les non-terminaux sont écrits avec des lettres minuscules entre crochets.

```
<parse_tree> ::= <program> | <procedure> | <function>
   <program> ::= PROGRAM <decl_list> <inst_list>
 <procedure> ::= PROCEDURE <decl_list> <inst_list>
  <function> ::= FUNCTION <decl_list> <inst_f>
 <decl_list> ::= DECL attrb (<type>)<decl_list> | DECL attrb <type>
      <type> ::= TYPE_ARRAY attrb <type> | TYPE_SIMPLE
 <inst_list> ::= SEMI_COLON (inst) <inst_list> | <inst>
    <inst_f> ::= SEMI_COLON (<inst>) <inst_f> |  RETURN <expr>
      <inst> ::= <write> | <read> | <assign> | <if> | <#if> |
                 <while> | <repeat> |<call_p> | <callgen>
    <assign> ::= ASSIGN ( <idf_aff> ) <expr>
      <read> ::= READ <idf_aff>
   <idf_aff> ::= IDF_AFF attrb
     <write> ::= WRITE <expr>
```

```
          <if> ::= IF(<expr>) <then>
        <then> ::= THEN (<inst_list>)<inst_list>
         <#if> ::= #IF(<idf_lect>) <#then>
       <#then> ::= #THEN (<inst_list>)<inst_list>
       <while> ::= WHILE (<expr>) <inst_list>
      <repeat> ::= REPEAT (<expr>) <inst_list>
      <call_p> ::= CALL_P attrb <param_list>
  <param_list> ::= COMA (<expr>)<param_list> | NO_PARAM
     <callgen> ::= CALLGEN attrb0 <param_list>
        <expr> ::= <read_array> | <write_array> | <idf_lect> |
                   <const> | <bin_op> | <un_op> | <call_f>
  <read_array> ::= CALLGEN attrb1  <param_list>
 <write_array> ::= CALLGEN attrb2 <param_list>
    <idf_lect> ::= IDF_LEC attrb
       <const> ::= INT_CST attrb
      <bin_op> ::= BINOP attrb (<expr>)<expr>
      <bin_op> ::= UNOP attrb <expr>
      <call_f> ::= CALL_F attrb <param_list>
```

Cette grammaire génére des arbres syntaxiques avec des noeuds ayant parfois des attributs. Ci-dessous, on indique la nature de chaque attribut.

Nom	rôle de l'attribut
DECL	nom de la variable declarée
TYPE-ARRAY	dimension du tableau
IDF-AFF	nom de la variable à affecter
IDF-LEC	nom de la variable à lire
INT-CST	valeur de la constante entiére
CALL-P	nom de la procédure à appeler
CALL-F	nom de la fonction à appeler
CALL-GEN	attrb0 est le nom d'une procédure précodée
CALL-GEN	attrb1 est le nom d'une procédure précodée qui permet de lire un tableau
CALL-GEN	l'attribut attrb2 est le nom d'une procédure précodée qui permet d'ecrire un tableau
BINOP	nom de l'opérateur binaire
UNOP	nom de l'opérateur unaire

Tableau D.2: Les attributs de l'arbre syntaxique

D.7 Mode d'emploi du compilateur

Le compilateur s'appuie sur une base de régles et de prédéfinitions qui sont reportées dans les paragraphes suivants. Cela facilite la mise au point, car pour modifier le compilateur, il suffit de modifier ces deux bases, et il n'est pas nécessaire de recompiler les sources. L'exécutable s'appelle JaNNet, il a plusieurs options:

- l'option "-r <nom du fichier de règles>" (changement du fichier de base de règles à utiliser)

- l'option "-p <nom du fichier de prédéfinition>" (changement du fichier des prédéfinitions à utiliser)
- l'option "-m" permet la création d'un fichier de suffixe ".lib" contenant les codes cellulaires des fonctions déclarées dans le programme. Ces codes cellulaires peuvent être ensuite appelés directement avec CALLGEN. Cette option permet ainsi de faire de la compilation modulaire

De la même façon qu'un compilateur de langage évolué génére du langage machine, le compilateur génére un code cellulaire, qui est développé par un autre logiciel indépendant. Celui-ci permet non seulement de développer le réseau codé, mais aussi de voir le réseau compilé ainsi que les étapes de développement, sur une fenêtre X11. De nombreux boutons servent à piloter le dévelopement, à gérer les entrées/sorties du réseau compilé, à voir les contenus des différents registres des cellules, et à agrandir avec une loupe une partie du dessin du réseau.

D.8 Les prédéfinitions

Ces prédéfinitions sont des codes cellulaire élémentaires permettant de factoriser les codes cellulaires membres droit des règles de réécritures de la grammaire d'arbre.

D.8.1 Opérateurs arithmétiques

Ont énumére ici des codes élémentaires qui réalisent des opérateurs arithmétiques sur des neurones, ainsi que les dynamiques AOR et EAND. Les sigmoïdes sont, par ordre de numéro croissant: Identité (1), Pi-Unit(3), marche-escalier-L(5), marche-escalier-R(6), égalité à zéro (7) Div-unit (8)

OR OR SBIAS 2 (SSIGMO 5 *MARCHE-ESCALIER-L* (OR-))

AND AND SSIGMO 5 *MARCHE-ESCALIER-L* (AND-)

NEG NEG LR 1 (VAL -1 (NOT-))

INFEQ INFEQ LR 1 (VAL -1 (SSIGMO 6 *MARCHE-ESCALIER-R* (INFEQ-)))

SUPEQ SUPEQ LR 2 (VAL -1 (SSIGMO 6 *MARCHE-ESCALIER-R* (SUPEQ-)))

INF INF LR 1 (VAL -1 (SSIGMO 5 *MARCHE-ESCALIER-L* (INF-)))

SUP SUP LR 2 (VAL -1 (SSIGMO 5 *MARCHE-ESCALIER–L* (SUP-)))

EQ EQ LR 1 (VAL -1 (SSIGMO 7 *EGALITÉ-À-ZÉRO* (EQ-)))

NEQ NEQ SEQ (JMP EQ) (JMP NEG)

PLUS PLUS SSIGMO 1 *IDENTITÉ* (PLUS-)

MOINS MOINS LR 2 (VAL -1 (SSIGMO 1 *IDENTITÉ* (MOINS-)))

MULT SSIGMO 3 *PI Unit* (MULT-)

QUO SSIGMO 8 *DIV Unit* (QUO-)

EOR AOR SDYN 2 (AOR-)

EAND EAND SDYN 3 (EAND-)

D.8.2 Lecture et écriture dans un tableau

On présente les codes cellulaires de fonctions permettant de lire et d'écrire dans un tableau à plusieurs dimensions. Chacune de ces fonctions nécessite trois codes élémentaires qui s'appellent chacun eux mêmes récursivement.

1AFFECT-INDEX SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -2 (SPLIT (JMP 1AFFECT-SUIT)))) (SITE 0 (BLOC (PN- 2)))))))

CREAT-EQB TAB (VAL -1 (SSIGMO 7 (EQB-)))

1AFFECT-SUIT BUN 3 (JMP 2AFFECT-SUIT) (SEQ (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -2 (SPLIT (JMP 1AFFECT-SUIT)))) (SITE 0 (BLOC (PN- 2))))

2AFFECT-SUIT BPN 2 (SITE 0 (JMP 3AFFECT-SUIT)) (MRG 3 (MRG 2 (SEQ (SPLIT (JMP 2AFFECT-SUIT)) (SITE 0 (BLOC (PN- 2))))))

3AFFECT-SUIT SEQ (PAR (CUT 3 (LR 1 (VAL -1 (JMP EAND)))) (CUT 2 (JMP EAND))) (JMP AOR)

1INDEX-LEC SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -1 (SPLIT (JMP 1INDEX-SUIT)))) (BLOC (JMP 2INDEX-SUIT))))))

1INDEX-SUIT BUN 2 (JMP SUPER-EAND) (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -1 (SPLIT (JMP 1INDEX-SUIT))))

2INDEX-SUIT SPLIT (BPN 1 (JMP AOR) (SEQ (LR -1 (JMP SUPER-MRG)) (SITE 0 (BLOC (PN- 2)))))

D.8.3 Macro-codes récursifs pour manipuler tout l'environnement

On énumére ci-joint des codes élémentaires qui sont utilisés à plusieurs endroits dans le code cellulaire. Ces codes appliquent des opérateurs récursivement sur tout l'environnement.

SUPER-MRG BLR 0 (MG (ADDLR -1 (JMP SUPER-MRG))) (JMP 2INDEX-SUIT)

la macro SUPER-EAND SUPER-EAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (SEQ (JMP SUPER-EAND) (SITE 0 (BLOC (PN- 2))))))

la macro SUPER-AOR SUPER-AOR SEQ (MRG 2 (MRG 1 (SPLIT (BPN 1 (JMP AOR) (JMP SUPER-AOR))))) (SITE 0 (BLOC (PN- 2)))

la macro SUPER-DYN SUPER-DYN SPLIT (BPN 1 (EVER 1 (SDYN 2)) (SEQ (MRG 1 (JMP SUPER-DYN)) (SITE 0 (BLOC (PN- 2)))))

la macro SYNCHRO SYNCHRO IPAR 1 (VAL 0 (EVER 0)) (SITE 0 (BLOC (DELSITO (SPLITN (EVER 1)))))

la macro REPEAND REPEAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (JMP REPEAND)))

la macro WHILEND WHILEND SPLIT (BPN 1 (END) (MRG 1 (JMP WHILEND)))

la macro READ-SUIT READ-SUIT BPN 3 (CUT 3 (LR 2 (VAL 0))) (SEQ (MRG 3 (SPLIT (JMP READ-SUIT))) (SITE 0 (BLOC (PN- 2))))

la macro WRITE-SUIT WRITE-SUIT BPN 1 (END) (MRG 1 (SPLIT (JMP WRITE-SUIT)))

D.9 Règles de reécriture des noeuds de l'arbre syntaxique

IF SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (RINS)))) (SEQ (SITE 0 (SPLFI 2 (CUT 2 (PAR (WAIT 2 (SITE 1 (JMP SUPER-EAND))) (LR 1 *NOT SIMPLIFIE* (VAL -1 (SITE 1 (JMP SUPER-EAND)))))) (CUT 1 (CPFO 1 (LINS)))))) (BLOC (CUTO 1 (JMP SUPER-AOR)))))

THEN PAR (CUT 3 (BLOC (DELSIT (RINS)))) (CUT 2 (BLOC (DELSIT (LINS))))

REPEAT SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP SUPER-DYN)) (SPLFI 1 (CPFO 1 (BLOC (DELSIT (RINS)))) (BLOC (MRG 2 (SITE 1 (SPLLIL 2 (CPFO 1 (SITE 0 (DELSIT (LINS)))) (PAR (CUTO 1 (CUTO 1 (LR 1 (VAL -1 (SEQ (WAIT 2 (JMP REPEAND)) (JMP SYNCHRO)))))) (CHOPO 2 (SEQ (JMP SUPER-EAND) (SITE 0 (BLOC (END))))))))))))))

WHILE SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP SUPER-DYN)) (BLOC (SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (LINS)))) (PAR (CUTO 2 (SPLFI 2 (CUT 2 (JMP SUPER-EAND)) (SITE 0 (CUT 1 (BLOC (SEQ (CPFO 1 (RINS)) (BLOC (SITE 1 (CUTO 1 (SEQ (MRG 1 (WAIT (JMP WHILEND))) (JMP SYNCHRO))))))))))) (CHOPO 2 (LR 1 (VAL -1 (SEQ (CUT 2 (JMP SUPER-EAND)) (SITE 0 (BLOC (END)))))))))))))))

PROGRAM IPAR 1 (START-) (SEQ (CPFO 1 (SWITCH 1 (LINS))) (BLOC (MRG 1 (CUTO 1)))))

NO-INST CUTO 1 (CUT 1)

DECL IPAR 2 (RINS) (PUTL 2 (BLOC (LINS)))

TYPE-ARRAY SEQ (CLONE ATTR (LINS)) (BLOC (EVER 2 (PN)))

TYPE-SIMPLE VAR (LR 1 (VAL 0 (EVER 2)))

SEMICOL SPLFI 1 (CPFO 1 (LINS)) (BLOC (MRG 2 (RINS)))

ASSIGN SPLFIL 1 (CPFO 1 (RINS)) (BLOC (LINS))

IDF-AFF SWITCH attr (CUTL 1 (CUT 1 (CUTO 1)))

IDF-LEC CUTO 1 (CHOP attr (BPN 1 (END) (MRG 1 (PN- 2))))

COMMA PAR (LINS) (RINS)

UNOP SEQ (CPFO 1 (LINS)) (CUTO 1 (JMP attr))

BINOP SEQ (CPFO 1 (PAR (LINS) (RINS))) (CUTO 1 (JMP ATTR))

INT-CST CHOP 2 (CUTO 1 (SBIAS ATTR (LR 1 (VAL 0 (EVER 2)))))

CALL-F SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0))))))

NO-PARAM KILL ()

FUNCTION ou RETURN LINS ()

CALL-P SPLLI LOCAL (CPFO 1 (SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (END))))) (BLOC (MRG 1 (CUTO 1)))

PROCEDURE SEQ (CPFO 1 (LINS)) (BLOC (MRG 1 (CUTRI GLOBAL (CUTO 1))))

READ SPLFIL 1 (SPLFI 2 (CHOP ATTR) (MRG 3 (SITE 1 (MULTSIT (JMP READ-SUIT))))) (BLOC (RINS))

WRITE PAR (CHOPO 1 (SEQ (CPFO 1 (RINS)) (BLOC (JMP WRITE-SUIT])))) (CUT 1 (CUTO 1))

#IF SEQ (CPFO 1 (PAR (CUTO 1) (RINS))) (BLOC (MRG 2 (LINS)))

#THEN BUN ATTR (CHOP 1 (MRG 1 (RINS))) (CHOP 1 (MRG 1 (LINS)))

CALLGEN SPLFI 2 (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0))))))

D.10 Bibliothéque de fonctions pour manipuler les tableaux

gauche CUTLE 3 (CUTO 1 (MRG 1 (CHHR 1 (PN- 2))))

droite CUTLE 3 (CUTO 1 (MRG 1 (CHHL 1 (PN- 2))))

concat CUTLE 3 (CUTO 1 (MRG 2 (MRG 1 (PN- 2))))

int-to-array CUTLE 3 (CUTO 1 (PN- 2))

array-to-int CUTLE 3 (CUTO 1 (MRG 1 (BPN 1 (END) (MRG 1 (PN- 2)))))

randomise CUTLE 3 (CUTO 1 (JMP rand))

rand BPN 1 (LR 1 (VAL 0 (RAND))) (SEQ (MRG 1 (SPLIT (JMP rand))) (PN- 2))

D.11 Réseaux de neurones pré-appris

Ces réseaux de neurones sont inclus dans le programme Pascal bête.p.

retine CUTLE 3 (CUTO 1 (SEQ (MRG 1 (SPLIT)) (SEQ (CLO 10 (EVER 0)) (BLOC (PN- 2)))))

position-objet CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 2 (EVER 0))) (CLO 2 (EVER 0))) (EVER 0)))

predateur CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 3 (EVER 0))) (CLO 3 (EVER 0))) (EVER 0)))

moteur CUTLE 3 (CUTO 1 (SEQ (EVER 0) (SEQ (CLO 2 (EVER 0)) (SEQ (CLO 4 (EVER 0)) (PN- 2)))))

D.12 Modifications syntaxiques et sémantiques

- L'instruction RETURN doit être la derniére instruction du corps d'une fonction.
- On ne peut définir de procédures ou de fonctions à l'intérieur d'une procédure ou d'une fonction (comme dans le langage C).
- On ne peut pas passer de paramètres par variables. Tout les paramètres sont passés par valeur.
- On ne peut pas faire d'allocation dynamique de mémoire.
- On peut compiler la récursivité, jusqu'à une certaine profondeur, qui doit être spécifiée à l'avance. À moins qu'il soit possible de connaître la profondeur nécessaire avant la compilation. C'est le cas pour les algorithmes du type "diviser pour régner".
- les fonctions peuvent retourner une valeur qui est un tableau.
- Le pascal est complété par les instructions: CALLGEN #include #IF #THEN #ELSE.

D.13 Modifications de l'arbre syntaxique

Les arbres syntaxiques que l'on utilise ont étés obtenus à partir d'un arbre abstrait classique en faisant les modifications suivantes:

- Remplacer un noeud IDF par IDF-LEC et IDF-AFF suivant que la variable est lue où écrite.
- Créer des noeuds particuliers READ et WRITE pour les appels de ces fonctions.
- Implanter les écriture et les lectures dans un tableau, à l'aide de fonctions particuliéres: index-lec1 et affect-index1.
- créer des noeuds pour la déclaration de variables, avec leur type codé par une chaîne d'opérateurs TYPE-ARRAY terminée par un opérateur TYPE-SIMPLE.

D.14 Simplifications du réseau de neurones

Lors du dévelopement du réseau, les neurones peuvent se simplifier. Pour cela, le voisinage de chaque neurone est examiné. A partir du voisinage, on peut détérminer si le neurone peut être supprimé. Après la supression du neurone, on rétablit les connexions nécessaires pour que le réseau fonctionne normalement. Les conditions de supression sont les suivantes:

- Le neurone n'a pas de lien en sortie
- Le neurone n'a pas de lien en entrée
- Le neurone a une seule entrée et sa sigmoïde est l'identité.
- Le neurone est un EAND toujours passant ou toujours bloquant
- Le neurone a une sortie constante, et on peut le supprimer en ajoutant un biais à ses voisins de sortie.

- le neurone est un AOR connecté à deux EAND, eux même connectés à la même paire de neurones. Le premier neurone de cette paire contient un booléen, le deuxième contient une valeur réelle ou entière. On peut tout simplifier de la même façon que l'on simplifie la formule logique `(A AND B) OR (A AND (NOT B)) = A`.

Certains neurones spéciaux ne supportent pas ces simplifications. Les pointeurs neurones ne se simplifient pas, même s'ils calculent l'identité et ont un seul lien en entrée. Les neurones d'entrée ne se simplifient pas même s'ils n'ont pas de liens en entrée, les neurones de sorties ne se simplifient pas même s'ils n'ont pas de liens en sortie. Pour gérer les simplifications, on positionne leur registre `simplif` sur une valeur élevée. A chaque simplification est attribué un numéro. Un neurone est simplifié, seulement si le numéro de la simplification qu'on veut lui appliquer est inférieur à la valeur de son registre `simplif`.

Il est surprenant de voir à quel point les simplifications permettent de réduire la taille des réseaux de neurones. Le compilateur génére en effet beaucoup de neurones inutiles. Fréquemment, aprés les simplifications, le nombre de neurones est divisée par 3.

D.15 L'apprentissage Switch étendu

```
Let c be the neuron currently processed;
oni:=the old net input of c;
For each neuron n that fans into c through link l
      If (l's weight is not null)
            Try to modify n's activity;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if(Small(oni)) r_n += r_c;
               else r_n += r_c * p1
               w_n += w_n + w_c;
      Case + : r_n += w_c * p2;
      Case - : w_n += w_c * p3 * d; r_n += r_c * p4 * d;
  If (n is an input unit) P5=1; Else P5=1/p5;
  If (n's activity is not null)
    For all v in {-1,0,1}, v not equal to l's initial weight
      Try to set l's weight to v;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if (Small(oni)) r_l[v] += r_c * P5;
               else r_l[v] += r_c * p1 * P5
               w_l[v] += w_c;
      Case + : r_l[v] += w_c * p2 * P5;
      Case - : w_l[v] += w_c * p3 * d * P5; r_l[v] += r_c * p4 * d * P5;
  Elseif n is not input unit
    For all v in {-1,0,1}, v not equal to l's initial weight
      Try to both set l's weight to v and modifyn's activity;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if(Small(oni)) r_n += r_c * p6; r_l[v] += r_c * P5 * p6;
           else r_n += r_c * p1 * p6; r_l[v] += r_c * p1 * P5 * p6;
           w_n += w_c * p6; w_l[v] += w_c * p6;
      Case + : r_n += w_c * p2 * p6; r_l[v] += w_c * p2 * P5 * p6;
      Case - : w_n += w_c * p3 * d * p6; w_l[v] += w_c * p3 * d * P5 * p6;
```

```
r_n += r_c * p4 * d * p6; r_l[v] += r_c * p4 * d * P5 * p6;
```

Cet algorithme décrit comment calculer les valeurs r_n, w_n, $w_l[v]$, $r_l[v]$, $v \in \{-1, 0, 1\}$, en commençant par les unités de sortie, puis en remontant jusqu'aux unités d'entrée, comme dans le cas de la rétro propagation du gradient. L'indice n indique le neurone en cours de traitement, l'indice l indique le lien en cours de traitement, Les valeurs r_n, w_n, $w_l[v]$, $r_l[v]$, $v \in \{-1, 0, 1\}$ sont utilisées pour sélectionner le poids que l'on va modifier. La fonction `Comp` renvoie 0 si le champ change de signe, et sinon elle retourne le signe de la différence entre son premier argument et son deuxième argument. La fonction `Small` renvoie 1 si son argument est 0 ou 1, sinon elle renvoie 0. L'opérateur `+=` incrémente la variable du membre gauche par la quantité définie dans le membre droit. Les entiers positifs $p_i, i = 1..6$ sont des paramètres de l'algorithme d'apprentissage.