14-16 rue Voltaire
94270 Kremlin Bicêtre

Benjamin DEVÈZE
Matthieu FOUQUIN
PROMOTION 2005
SCIA

QUELQUES ALGORITHMES DES SCIENCES COGNITIVES

Janvier 2004

Responsable de spécialité SCIA : **M. Akli Adjaoute**

Table des matières

Liste des Algorithmes

Chapitre 1

Introduction

Ce rapport a pour but de présenter quelques algorithmes fondamentaux utilisés dans le domaine des sciences cognitives. Il ne prétend bien-sûr pas à l'exhaustivité mais, a pour vocation de dresser un aperçu général de l'état de l'art en matière algorithimique, dans les différents domaines traités. Le lecteur pourra trouver la plupart des algorithmes abordés dans le corps du rapport. Notons également que nous avons volontairement choisi de ne pas écarter certains algorithmes naïfs et non exploitables par l'industrie afin de bien montrer les améliorations progressives qui ont eu cours. Notons d'ailleurs, qu'en général, la présentation des algorithmes vise à en faire ressortir l'évolution chronologique. Nous nous sommes souvent un peu plus étendu que ne l'exigeait le sujet, nous considérons en effet que certains algorithmes méritaient d'être développés et qu'une étude un peu plus précise ne pouvait être que bénéfique pour une consultation ultérieure de ce document.

Chapitre 2

Systèmes à base de connaissances & Techniques d'implémentation de RETE

2.1 Intérêt

Le monde des entreprises fait face actuellement à un important problème de gestion de son savoir, de son savoir-faire et de ses compétences. Ce problème se caractérise de plusieurs manières :

- Perte de savoir et de savoir-faire antérieur
- Méconnaissance des travaux effectués ailleurs
- Non communication entre les services
- Localisation monopolistique des connaissances et de l'expertise...

Constituer une mémoire vivante et productive de l'entreprise, faire vivre une base de connaissances reposent donc sur les trois thèmes principaux suivants :

- la gestion des experts et des expertises
- le retour d'expériences
- le transfert de connaissances et d'informations dans l'entreprise

Les systèmes à base de connaissances permettent de recueillir et d'exploiter le savoir acquis au cours des années, cette centralisation du savoir permet de répondre efficacement aux problématiques pouvant se présenter.

2.2 Systèmes à base de connaissances

Nous nous intéresserons ici successivement au raisonnement par cas et aux systèmes experts, qui font tous les deux parties des systèmes à base de connaissances, puisqu'ils exploitent tous les deux des connaissances acquises et stockées dans une base.

2.2.1 Raisonnement par cas

De manière générale, le raisonnement par cas est une approche de résolution de problèmes basée sur la réutilisation par analogie d'expériences passées appelées cas. Un cas représente notamment un problème et la solution qui a été appliquée (ou une méthode permettant de la générer). Le raisonnement se décompose habituellement en quatre phases principales :

- phase de recherche dont le but est de rechercher des cas ayant des similarités avec le problème courant
- phase de réutilisation permettant de construire une solution au problème courant en se basant sur les cas identifiés dans la phase précédente
- phase de révision de la solution qui permet de l'affiner grâce à son évaluation

– phase d'apprentissage chargée de mettre à jour les éléments du raisonnement en prenant en compte l'expérience qui vient d'être réalisée, et qui pourra ainsi être utilisée pour les raisonnements futurs.

Ainsi, un cas représente une expérience passée dont l'enseignement peut être utile lorsqu'un nouveau problème se présente. Généralement, un cas est indexé pour permettre de le retrouver suivant certaines caractéristiques pertinentes et discriminantes. Ces caractéristiques, aussi appelées indices, déterminent dans quelle situation (ou contexte) le cas peut être de nouveau réutilisé. La problématique de la phase de recherche est donc de permettre d'identifier un certain nombre de cas ayant des indices similaires au problème courant : il est en effet peu probable de retrouver un cas correspondant exactement au problème courant. Un système de raisonnement par cas doit permettre l'expression des indices pour les différents cas, et doit disposer de structures d'indexation ou index offrant une recherche efficace tout en utilisant des connaissances du domaine et/ou des connaissances induites à partir de son expérience. Dans ce sens, l'objectif de la phase de recherche dépasse les approches classiques des bases de données même si des techniques issues de ce domaine sont parfois utilisées. Voyons un peu plus en détail comment se déroulent ces processus.

Processus

La recherche
Cette phase permet de déterminer les cas de la base qui sont les plus similaires au problème à résoudre. La procédure de recherche est habituellement implémentée par une sélection des plus proches voisins (k-nearest-neighbors) ou par la construction d'une structure de partitionnement obtenue par induction. L'approche des plus proches voisins utilise des métriques de similarité pour mesurer la correspondance entre chaque cas et le nouveau problème à résoudre. Ces métriques peuvent varier d'un système à l'autre et peuvent être pondérées selon le problème à résoudre, ceci confère plus de flexibilité au système. L'approche par induction génère un arbre qui répartit les cas selon différents attributs et qui permet de guider le processus de recherche.

L'adaptation
Suite à la sélection de cas lors de la phase de recherche, le système CBR aide l'usager à modifier et à réutiliser les solutions de ces cas pour résoudre son problème courant. En général, on retrouve deux approches pour l'adaptation de cas. Par l'approche transformationnelle (ou structurelle), on obtient une nouvelle solution en modifiant des solutions antécédentes et en les réorientant afin de satisfaire le nouveau problème. Par l'approche générative (ou dérivationnelle), on garde, pour chaque cas passé, une trace des étapes qui ont permis de générer la solution. Pour un nouveau problème, une nouvelle solution est générée en appliquant l'une de ces suites d'étapes. Certains travaux visent également à unifier ces différentes approches d'adaptation. Peu de systèmes CBR font de l'adaptation complètement automatique. Pour la plupart des systèmes, une intervention humaine est nécessaire pour générer partiellement ou complètement une solution à partir d'exemples. Le degré d'intervention humaine dépend des bénéfices en terme de qualité de solution que peut apporter l'automatisation de la phase d'adaptation.

La maintenance
Durant le cycle de vie d'un système CBR, les concepteurs doivent préconiser certaines stratégies pour intégrer de nouvelles solutions dans la base de cas et pour modifier les structures du système CBR pour en optimiser les performances. Une stratégie simple est d'insérer tout nouveau cas dans la base. Mais d'autres stratégies visent à apporter des modifications à la structuration de la base de cas (e.g. indexation) pour en faciliter l'exploitation. On peut également altérer les cas en modifiant leurs attributs et leur importance relative.

La construction
Ce processus, en amont des activités de résolution de problèmes du système CBR, soutend la structuration initiale de la base de cas et des autres connaissances du système à partir de différentes

ressources tels des documents, bases de données ou transcriptions d'interviews avec des praticiens du domaine. Ce processus, souvent effectué manuellement par le concepteur du système, se prête moins bien à l'automatisation car il nécessite une connaissance du cadre applicatif pour guider, entre autre, la sélection du vocabulaire d'indexation et la définition des métriques de similarités.

La connaissance

Les différentes connaissances utilisées par un système CBR sont regroupées en quatre catégories (« knowledge containers ») :

- vocabulaire d'indexation : un ensemble d'attributs ou de traits (« features ») qui caractérisent la description de problèmes et de solutions du domaine. Ces attributs sont utilisés pour construire la base de cas et jouent un rôle important lors de la phase de recherche.
- base de cas : l'ensemble des expériences structurées qui seront exploitées par les phases de recherche, d'adaptation et de maintenance.
- mesures de similarité : des fonctions pour évaluer la similarité entre deux ou plusieurs cas. Ces mesures sont définies en fonction des traits et sont utilisées pour la recherche dans la base de cas.
- connaissances d'adaptation : des heuristiques du domaine, habituellement sous forme de règles, permettant de modifier les solutions et d'évaluer leur applicabilité à de nouvelles situations.

2.2.2 Systèmes experts

Rappelons brièvement le principe et l'organisation d'un système expert, cela étant nécessaire à la bonne compréhension des algorithmes utilisés dans le domaine.

Un système expert est un logiciel qui reproduit le comportement d'un expert humain accomplissant une tâche intellectuelle dans un domaine précis. On peut souligner les points suivants :

- les systèmes experts sont généralement conçus pour résoudre des problèmes de classification ou de décision (diagnostic médical, prescription thérapeutique, régulation d'échanges boursiers, ...)
- les systèmes experts sont des outils de l'intelligence artificielle, c'est-à-dire qu'on ne les utilise que lorsqu'aucune méthode algorithmique exacte n'est disponible ou praticable
- un système expert n'est concevable que pour les domaines dans lesquels il existe des experts humains. Un expert est quelqu'un qui connaît un domaine et qui est plus ou moins capable de transmettre ce qu'il sait

Un système expert est composé de deux parties indépendantes :

- une base de connaissances elle-même composée d'une base de règles qui modélise la connaissance du domaine considéré et d'une base de faits qui contient les informations concernant le cas que l'on est en train de traiter
- un moteur d'inférences capable de raisonner à partir des informations contenues dans la base de connaissance, de faire des déductions, etc.

Le rôle du cogniticien est de soutirer leurs connaissances aux experts du domaine et de traduire ces connaissances dans un formalisme se prêtant à un traitement automatique, c'est-à-dire en règles. Ces deux tâches sont aussi délicates l'une que l'autre. En effet, un expert est la plupart du temps inconscient de la majeure partie de son savoir ; et s'il arrive à en exprimer une partie, c'est souvent sous une forme qui ne se laisse pas facilement formaliser.

L'indépendance entre la base de connaissances et le moteur d'inférences est un élément essentiel des systèmes experts. Elle permet une représentation des connaissances sous forme purement déclarative, c'est-à-dire sans lien avec la manière dont ces connaissances sont utilisées. L'avantage de ce type d'architecture est qu'il est possible de faire évoluer les connaissances du système sans avoir à

agir sur le mécanisme de raisonnement.

Dans la réalité, les choses se passent de manière un peu moins idéale et il est souvent nécessaire d'organiser la base de connaissances, de réfléchir sur les stratégies d'utilisation des règles, etc.

Le système expert est souvent complété par des interfaces plus ou moins riches permettant un dialogue avec les utilisateurs, l'idéal étant une interface en langage naturel.

La représentation des connaissances

Les faits peuvent prendre des formes plus ou moins complexes. Un système expert qui n'utilise que des faits booléens est dit d'ordre 0. Un système expert qui utilise des faits symboliques ou réels, sans utiliser de variables, est d'ordre 0+. Un système utilisant toute la puissance de la logique du premier ordre est d'ordre 1.

Une règle est de la forme *Si conjonction de conditions alors conclusion*. Une base de règles est un ensemble de règles et sa signification logique est la conjonction de la signification logique de chacune des règles.

Un des plus grand problèmes que rencontre le cogniticien lorsqu'il tente de formaliser le savoir d'un expert, c'est que celui-ci est capable de raisonner sur des connaissances incertaines et qu'on ne dispose que de très peu d'outils pour rendre compte de cette capacité. C'est pourquoi des recherches sont en cours pour intégrer la logique floue, les logiques modales et non monotones dans la représentation des connaissances, ceci permettra sans doute de se rapprocher un peu plus de nos modes de raisonnement.

Le moteur d'inférence

Un moteur d'inférences est un mécanisme qui permet d'inférer des connaissances nouvelles à partir de la base de connaissances du système, composée de la base des faits et de la base de règles. Le moteur d'inférence va enchaîner les règles c'est à dire qu'il va effectuer un chaînage. On distingue essentiellement trois modes principaux de fonctionnement des moteurs d'inférence : le chaînage avant, le chaînage arrière, et le chaînage mixte. On remarquera que les moteurs d'inférence décrits ci-dessous le sont indépendamment de tout domaine d'application. Cette séparation entre connaissance et raisonnement est essentielle pour les systèmes experts.

Le chaînage avant

Le mécanisme du chaînage avant est très simple. On va analyser chaque fait et on va examiner toutes les règles où ce fait apparaît en prémisse. Pour les règles déclenchées, on va affecter les attributs en conclusion des valeurs qui leur correspondent. On dira que les faits ont été propagés. Ces attributs affectés feront partie du résultat final de l'expertise ; et, en même temps, ils seront eux-mêmes propagés. On fait cela jusqu'à l'épuisement des faits, et on communique les résultats à l'utilisateur. L'algorithme suivant calcule si *Fait* peut être déduit ou non de la base de connaissances.

Algorithm 1 Algorithme du chaînage avant

Ensure: retourne vrai si F peut être déduit faux sinon

```
1:  function ChainageAvant(BR, BF, F)
2:      while F n'est pas dans BF et qu'il existe dans BR une règle applicable do
3:          choisir une règle applicable R
4:          BR = BR - R
5:          BF = BF U Conclusion(R)
6:      end while
7:      if F appartient à BF then
8:          return vrai
9:      else
10:         return faux
11:     end if
12: end function
```

On remarque que l'algorithme précédent n'indique pas comment choisir une règle applicable. C'est à ce niveau que la métaconnaissance du domaine peut intervenir et permet de définir une stratégie de choix. Notons également que l'algorithme se termine toujours.

Cet algorithme présente les inconvénients suivants :
- Déclenche toutes les règles applicables même si aucun intérêt
- Base de faits doit contenir suffisamment de faits initiaux
- En cas d'échec, un seul fait pourrait permettre d'arriver au but, mais pas interactif
- Explosion combinatoire possible

Le chaînage arrière
Le mécanisme de chaînage arrière consiste à partir du fait que l'on souhaite établir, à rechercher toutes les règles qui concluent sur ce fait, à établir la liste des faits qu'il suffit de prouver pour qu'elles puissent se déclencher puis à appliquer récursivement le même mécanisme aux faits contenus dans ces listes.

L'exécution de l'algorithme de chaînage arrière peut être décrit par un arbre dont les noeuds sont étiquetés soit par un fait, soit par un des deux mots et, ou. On parle d'arbre et-ou.

Si les faits déjà examinés ne peuvent pas être mémorisés (par exemple parce qu'ils sont trop nombreux), l'algorithme de chaînage arrière peut boucler.

On peut enrichir l'algorithme de chaînage arrière en tenant compte du caractère demandable ou non d'un fait. Dans ce cas, lorsqu'un fait demandable n'a pas encore été établi, le système le demandera à l'utilisateur avant d'essayer de le déduire d'autres faits connus. Mais pour que ce mécanisme soit efficace (ce qui implique entre autres qu'il n'agace pas l'utilisateur en posant des questions stupides), il faut que le moteur d'inférences soit capable de déterminer quelles sont les questions pertinentes. Et ce problème est loin d'être simple. Ce système de questions posées à l'utilisateur rend le processus intéractif et réduit l'arbre de recherches.

Le chaînage mixte
L'algorithme de chaînage mixte combine, comme son nom l'indique, les algorithmes de chaînage avant et de chaînage arrière. Son principe est le suivant :

Algorithm 2 Algorithme du chaînage mixte

```
1: function ChainageMixte(F (à déduire))
2:     while F n'est pas déduit mais peut encore l'être do
3:         Saturer la base de faits par chaînage AVANT
4:         Chercher quels sont les faits encore éventuellement déductibles
5:         Déterminer une question pertinente à poser à l'utilisateur et ajouter sa réponse à la base
               de faits
6:     end while
7: end function
```

Résolution des conflits

Le choix de la ou les règles qui doivent effectivement être déclenchées est une source de conflits. Les stratégies de résolution de ces conflits sont variées, citons notamment :

- déclenchement de la règle dont la partie prémisse est la plus détaillée (conclusions plus précises)
- règle utilisant les informations les plus récemment acquises ou déduites
- règles amenant le plus grand nombre de conclusions
- règles fonction de l'intérêt des conclusions qu'elles apportent

2.3 Implémentation de RETE

2.3.1 Rappels

Présentation

Comme nous l'avons vu, les algorithmes classiques de chaînage avant présente une complexité de calcul trop importante pour être applicables à des systèmes d'envergure. L'algorithme de RETE (qui signifie réseau en latin) est un algorithme de chaînage avant qui exploite intelligemment les particularités des systèmes à base de règles à savoir :

- la ressemblance structurelle : de nombreuses prémisses de règles ont des clauses (pattern) en commun et donc le nombre de tests ainsi que la mémoire utilisée peuvent être réduits.
- la redondance temporelle : entre deux cycles du moteur d'inférence, la mémoire de travail diffère peu, il est donc avantageux de mémoriser les états antérieurs plutôt que de tout recalculer

Principe

L'algorithme de RETE compile la partie condition des règles sous forme d'un réseau de propagation différentielle. Les noeuds du réseau mémorisent et maintiennent par calcul différentiel des informations sur les résultats des tests. Le réseau prend en entrée les changements affectant la base de faits et calcule en sortie les changements correspondants de l'ensemble des conflits, qui n'est autre que l'ensemble des règles déclenchables à un instant t i.e. les règles qui ont été matchées par les faits.

Arbre de discrimination

Le réseau est partagé en deux parties distinctes, la première partie du réseau, appelée arbre de discrimination, effectue les tests de sélection sur les faits contenus dans la base de faits. La racine de l'arbre de discrimination est aussi le point d'entrée du réseau. L'arbre a autant de feuilles qu'il y a de littéraux distincts dans les parties conditions des règles, ces feuilles sont appelées noeuds alpha. Étant donnés un ensemble de règles, l un littéral figurant dans la partie condition d'une des règles de cet ensemble, et BF un état de la base de faits, le noeud alpha associé à l calcule l'ensemble des instances du littéral l dans BF. Le résultat du calcul d'un noeud alpha est mémorisé dans une mémoire (dite mémoire alpha). Une mémoire alpha contient donc un ensemble de faits de BF. L'arbre de discrimination contient aussi des noeuds internes qui permettent de partager des calculs communs à plusieurs

noeuds alpha.

Réseau de jointure
La deuxième partie du réseau (ou réseau de jointure) contient des noeuds qui effectuent des tests de jointuire entre les littéraux d'une même règle. Chaque noeud béta est associé à une mémoire dite mémoire béta dans laquelle est mémorisé l'ensemble des instances partielles calculées dans le noeud. Le réseau a autant de noeuds terminaux, aussi appelés noeuds règles, qu'il y a de règles dans la base de règles. Chaque noeud règle calcule l'ensemble des instances d'une règle.

Exécution des règles
L'algorithme d'exécution des règles calcule l'ensemble de conflits dans l'état initial de la base de faits. Puis il maintient cet ensemble de cycle en cycle en utilisant le réseau de propagation. Le calcul des changements de l'ensemble de conflits est entremélé avec l'exécution des actions. A chaque changement opéré sur la base de faits, le système génère un message. La procédure dite de propagation est exécutée dès qu'un message est généré. Cette procédure prend en entrée le réseau de propagation et un message ; elle calcule les modifications à apporter aux données mémorisées à la suite du changement survenu dans la base de faits. Initialement, les mémoires locales du réseau sont vides ainsi que l'ensemble de conflits. Au cours de la première phase, un message est généré et propagé pour chaque fait de la base de faits initiale. Le résultat de cette phase est le calcul de l'ensemble de conflits dans la base de faits initiale et la mémorisation des instances partielles dans les mémoires locales du réseau. Ensuite, l'algorithme exécute un cycle comportant deux phases : (i) choisir une instance de règle dans l'ensemble de conflits, (ii) exécuter chaque action spécifié par l'instance choisie en (i), générer le message correspondant et le propager. L'exécution s'arrête lorsque l'ensemble de conflits est vide. La complexité de la procédure de propagation est déterminante car cette procédure exécute la plus grande part du travail effectué par l'algorithme.

2.3.2 Techniques d'implémentation

Nous ne donnerons pas de pseudo-code pour l'algorithme de RETE ni pour TREAT car un code a déjà été vu en cours, d'autre part un code plus détaillé serait trop volumineux pour ce rapport. Nous réservons donc la partie plus technique du codage de RETE à une prochaine échéance puisque nous avons à l'implémenter.

Syntaxe des règles et des faits

Il convient tout d'abord de se fixer une syntaxe et une grammaire pour l'écriture des règles et des faits. Il faut déterminer si le moteur d'inférence pourra supporter les expressions arithmétiques, les négations, les variables, les littéraux d'arité quelconque... Dans la littérature on retrouve parfois des exemples d'implémentation où les règles sont sous forme de triplets de type *identifiant attribut valeur*, ce qui simplifie l'implémentation et n'est pas restrictif pour le système, puisque les autres formes peuvent être converties dans ce format. Il est toutefois préférable d'offrir la possibilité d'utiliser des n-uplets qui sont plus souples d'utilisation et qui restreignent le nombre de règles du système.

Implémentation du réseau alpha

On construit le réseau comme suit. Pour chaque condition, posons T1, ..., Tk ses constantes, on part de la racine et on construit un chemin de k noeuds correspondant aux constantes. En construisant ce chemin, on partage les noeuds existants quand cela est possible. Enfin, la mémoire alpha sera la sortie du noeud Tk. Il est possible de créer des noeuds internes qui portent sur la longueur des littéraux afin d'accélérer les recherches. Chaque noeud interne a une structure simple, il comprend la valeur de la constante à tester, sa position, une liste de noeuds fils. Évidemment cette solution n'est pas parfaite, car le système peut ralentir lorsqu'un noeud a beaucoup de fils, c'est pourquoi on peut mettre en place des tables de hachage pour se brancher tout de suite sur le bon fils. Notons également que si

l'on adopte une représentation des règles en k-uplets il est possible de mettre en place des tables de hachage exhaustives.

Implémentation des noeuds mémoires

Rappelons que les mémoires alpha stockent des ensembles de faits et que les noeuds bétas stockent des ensembles de tokens, chaque token représentant une séquence de faits, chaque token correspond alors à un match partiel qui satisfait les k premières conditions d'une règle. Il y a plusieurs façon d'implémenter les noeuds mémoires selon la façon de représenter les ensembles et les tokens. Pour stocker les ensembles on peut utiliser de simples listes, on peut gagner en performance en indexant les éléments dans des tables de hachage ou en utilisant des arbres. Pour ce qui est de la représentation des tokens, deux solutions principales sont envisageables, les tableaux et les listes. Les tableaux présentent des accès en temps constant mais ils nécessiteront plus de mémoire.

Un mémoire alpha sera donc représentée par une structure contenant une liste de faits et une liste de fils. Un token aura un pointeur vers son ascendant qui contient les k - 1 faits précédents et le k ième fait. Une mémoire béta sera une structure contenant une liste de tokens et une liste de fils.

Implémentation des jointures du réseau béta

Une jointure peut être activée par la droite lors de l'ajout d'un fait dans un noeud alpha ou bien par la gauche lorsqu'un token est ajouté à un noeud béta. Une jointure doit donc contenir un pointeur vers les deux mémoires alpha et béta qui la précèdent, une liste de fils, et une liste de tests à effectuer. Il convient ensuite d'écrire les procédures d'activation correspondantes. Notons que pour éviter les doublons de tokens il convient d'activer d'abord les descendants avant les ancêtres.

2.3.3 Variantes & Optimisations de RETE

Il existe un grand nombre de variantes et d'optimisations diverses pour l'algorithme de RETE. Certains favorisent la mémoire utilisée, d'autres le temps d'exécution et il convient donc d'adapter l'algorithme à ses besoins. Il est évident que l'algorithme doit être conçu différemment pour un système comptant plus de 100 000 règles que pour un système qui contiendra moins de 100 règles. Il faut également distinguer les systèmes qui doivent supporter les négations, les négations de conjonctions, les expressions arithmétiques, la suppression et l'ajout de règles, la modification de faits existants... Bref il existe autant de variantes que de cas d'utilisations.

Quelques variantes

- *Scaffolding* : utile quand les mêmes faits sont ajoutés et supprimés de façon répétitive, ils sont marqués actifs et inactifs plutôt que réellement supprimés
- dans les systèmes utilisant une forme de résolution des conflits, il est possible d'utiliser une version *paresseuse*. L'idée principale est de limiter le filtrage afin de ne pas construire des instances de règles qui ne seront jamais exécutées
- *Collection Rete* est un moyen de réduire le coût des jointures lorsque les mémoires de travail sont de tailles importantes. L'idée est de structurer le contenu des noeuds béta comme des ensembles de collections de tokens plutôt que comme des ensembles de tokens individuels
- Il est possible d'ajouter un algorithme de *cohérence d'arc* à Rete afin d'élaguer les combinaisons possibles
- *Rete UL* est un moyen de conserver un temps d'exécution raisonnable quelque soit le nombre de règles
- etc

TREAT

Nous ne nous étendrons pas sur l'algorithme de TREAT qui est une alternative très proche de RETE. TREAT n'utilise que l'arbre de discrimination et les noeuds règles, il n'utilise pas le réseau de jointure de RETE. De ce fait, la suppression d'un fait est rendu plus simple dans TREAT car il suffit de supprimer de l'ensemble des conflits tous les tokens faisant intervenir le fait à supprimer. Par contre l'insertion fait intervenir plus de calculs.

Chapitre 3

Algorithmes des Blackboards

3.1 Présentation des Blackboards

Les systèmes blackboards (tableau noir) ont été développés dans les années 1970 pour résoudre des problèmes complexes d'interprétation du signal. Depuis, l'approche des blackboards a été retenue pour aborder les problèmes difficiles et mal structurés, et ce dans des applications pour de nombreux secteurs.

Les blackboards constituent une technique multi-agents de résolution de problème. Le problème est décrit sur un tableau virtuel et chaque agent, en fonction de sa spécialité, en résout une partie en posant sur le tableau une solution ou un nouveau sous-problème.

Le système blackboard est fondé sur une recherche de solutions par l'intermédiaire d'une base de faits partagée.

Une architecture blackboard est constituée de 3 composants majeurs :
- Une mémoire organisée hiérarchiquement ou une base de données appelée blackboard dans laquelle les solutions sont sauvegardées
- Une collection de sources de connaissance qui génère des solutions sur le blackboard en utilisant systèmes experts, réseaux de neurones, analyse numérique...
- Un module de contrôle séparé qui passe en revue les sources de la connaissance et choisit la plus appropriée

3.2 Modèle HEARSAY-II (HSII)

3.2.1 Stratégie standard pour la résolution de problème standard

La stratégie standard pour blackboard de résolution de problème se réfère souvent à un « hypothesis and test »incrémental (ou évidence aggrégation). Cela implique de faire l'hypothèse d'une solution éventuellement partielle basée sur des données incomplètes et d'essayer de la vérifier sur des données supplémentaires pour la valider.

3.2.2 Auto-activation des sources de connaissances

Un aspect important du modèle HERSAY-II qui permet l'indépendance des sources de connaissance est que ces dernières procèdent par auto-activation. Chaque source de connaissance possède un format de précondition-action dans lequel la précondition lui permet de déterminer quand l'action est applicable à partir de l'état actuel du blackboard.

3.2.3 Agenda-based control mechanism

Comme les sources de connaissance dans le modèle HSII sont à la fois indépendantes et auto-activables, il n'y a, à priori, pas besoin de mécanisme de contrôle additionnel : une source de connaissance pourrait s'exécuter lorsqu'elle a déduit qu'elle était activable. Malgré cela, cette approche a deux sérieux problème :

- l'exécution des sources de connaissance doit être séquencée
- Sans contrôle, on se retrouve rapidement dans un problème d'explosion combinatoire

Pour résoudre ce problème, le modèle HSII utilise un mécanisme de contrôle d'agenda. Toutes les actions possibles sont placées dans un agenda et à chaque cycle celui qui possède l'évaluation la plus haute est choisi pour l'exécution.

Algorithm 3 Boucle de contrôle basique du modèle HSII

1: **repeat**
2: Identification des sources de connaissance à déclencher
3: Vérification des préconditions des sources de connaissances à déclencher
4: Mise à jour de l'agenda avec les instances représentant les sources de connaissance activées
5: Evaluation des instances et selection des KSI pour l'execution
6: Execution des KSI
7: **until** les critères de terminaison sont vérifiés

3.2.4 Contrôle du système BlackBoard

Contrôle Goal-Directed

Le contrôle Goal-Directed se réfère à un style spécifique de raisonnement de contrôle qui implique une réduction de problème (détermination de sous-objectif, backchaining des préconditions-actions, et planification). La détermination de sous-objectifs implique la réduction des buts abstraits de haut niveau en des buts de bas niveaux plus détaillés qui peuvent être résolus directement. Le backchaining des préconditions-actions nécessite d'identifier les actions permettant d'autres actions nécessaires pour satisfaire un objectif. Et la planification permet de rester concentrer sur les objectifs à plus long-terme intégrant les actions appropriées ou les éliminant, ce qui peut être nécessaire lorsque des actions effectuent des interactions destructives.

Problème de terminaison

Dans une résolution de problème blackboard, l'ensemble des buts du système se réfèrent souvent aux critères de terminaison (ces critères doivent être rencontrés pour que la résolution du problème se termine). Quand l'application blackboard est sous contrainte, la résolution du problème doit prendre en compte de quelle manière les hypothèses correspondent à l'ensemble du problème et quelle importance elles ont. Trouver une solution correspondant aux contraintes ne représente pas forcément une solution à l'intégralité du problème car les heuristiques de stratégies de recherche ne garantissent pas de trouver d'abord la meilleure réponse.

Les stratégies de résolution de problème

Si la résolution de problème de blackboard est considérée comme un processus de satisfaction de contrainte, l'avantage des stratégies appliquées aux blackboard est d'être optimisé aux contraintes et critères d'arrêt du système. Par conséquent, le contrôle du blackboard est plus souvent utilisé lorsque les hypothèses se concurrencent, coopèrent ou sont indépendantes. Quatre stratégies sont régulièrement utilisées dans ce sens :

Parcours en profondeur d'abord (depth first search)
Cet algorithme est présenté dans la rubrique sur les algorithmes forward. Il est utilisé lorsqu'il y a une solution partielle avec une grande crédibilité ou quand il n'y a pas d'alternative concurrente avec une évaluation similaire. Appliqué comme stratégie de résolution de problème dans les blackboard, le danger est que s'il est appliqué trop tôt, cela peut prendre trop de temps pour reconnaître qu'une branche est inutile. De plus, la valeur ultime d'une recherche dirigée dépend du critère de terminaison. Tandis que ce type de recherche peut réduire le coût pour compléter une solution particulière, il n'élimine pas le besoin de considérer les chemins de recherche alternatifs pour satisfaire les objectifs du système. Cependant, la création de solutions potentielles de haut niveau avec des évaluations plus sérieuses peut toujours réduire le temps de calcul nécessaire pour obtenir les différentes alternatives.

Parcours en largeur d'abord (breadth first search)
Cet algorithme est utilisé lorsqu'une solution partielle a une crédibilité faible ou instable ou lorsqu'il y a beaucoup de solutions partielles avec le même taux de crédibilité. L'avantage de cette approche est que cela permet de construire l'ensemble des contraintes qui peuvent être utilisées pour construire une hypothèse de haut niveau. De plus, il est parfois nécessaire d'effectuer un parcours complet.

Augmentation de l'espace de recherche incrémentalement
La particularité de cet algorithme est de permettre de l'appliquer aux contraintes les plus importantes en limitant l'espace de recherche. Cet algorithme est utilisé dans les architectures HERSAY-II et RESUN à travers l'affectation des hypothèses d'échec d'inférence.

Le diagnostic différentiel
Quand cette méthode est disponible, son utilisation permet de différencier directement les solutions concurrentes au lieu d'utiliser l'approche générer et tester. Quand l'hypothèse est incertaine due à l'existence d'hypothèses alternatives et concurrentes, le diagnostic différentiel permet au système d'essayer de trouver des contraintes consistantes avec une seule de ces alternatives. L'avantage d'utiliser des méthodes directes pour résoudre l'incertitude, est qu'elles peuvent atteindre plus rapidement des valeurs hautes répondant aux critères de terminaison que les méthodes indirectes.

3.2.5 Mécanismes de contrôle supplémentaires dans le modèle Hearsay-II

Predict and Verify

Predict and Verify est un mécanisme qui a été implémenté dans le système HERSAY-II pour étendre sa capacité de raisonnement dans la réalisation de buts. Cela permet à l'architecture HSII de réaliser des sous-objectifs. Predict effectue des prévisions sur les mots pouvant compléter une phrase. Verify confirme ou ignore la prédiction en regardant les données.

Large Grained KSs

HSII implémente plusieurs stratégies spécialisées à travers la Large Grained KSs. L'expérience a montré que le modèle avait besoin de stratégies spécifiques à un contexte. Les stratégies de bas niveau spécialisées traitant de façon uniformisée toutes les données entrées pouvant poser des problèmes de fiabilité. Ainsi, l'utilisation de stratégies de contrôle de haut niveau spécifique au contexte (aussi appelées stratégies de contrôle sophistiquées) sont utilisées dans les architectures BB1 et RESUN et permettent une représentation plus explicite de stratégie détaillée.

Stop Terminaison

Ce mécanisme examine les hypothèses alternatives existantes et élague celles qui ne sont pas capable de produire des réponses intéressantes. Raisonner sur la terminaison requiert une vue globale de la solution du problème. Ce mécanisme y parvient en utilisant une base de données de contrôle. Cela permet d'implémenter des stratégies de contrôle globales et sophistiquées qui déterminent les

hypothèses particulières qui suivent l'objectif et de supprimer les autres. Ce mécanisme se termine lorsque toutes les alternatives potentielles ont été retirées.

Generator and Policy KSs

Une des caractéristiques de l'implémentation d'HSII est l'utilisation de « generator and policy KSs ». Ce mécanisme est une synthèse de « large grained KSs »qui est capable de créer toutes les explications plausibles pour les hypothèses des différents niveaux. Cependant, au lieu de créer les hypothèses représentant toutes les explications, ce système peut être contrôlé pour ne s'appliquer qu'à une portion de ces hypothèses. Ce contrôle est apporté par le « policy KSs »qui spécifie combien d'hypothèse il faut créer et à quel endroit de l'espace de recherche. Les intérêts de l'approche « generator and policy KSs »est d'implémenter une stratégie s'étendant de manière incrémentale dans l'espace de recherche. Et il permet également d'apporter un mécanisme permettant une recherche plus globale avec une planification basique.

WORD-SEQ

Un autre « large grained KSs »utilisé dans HSII est WORD-SEQ (ou WOSEQ). Rechercher des réponses de haut niveau via des hypothèses de niveaux intermédiaires permet de regrouper les contraintes de façon incrémentale. Cela peut se produire dans un grand nombre de résolution de problème. Appliquer ces contraintes partielles peut diminuer les temps de calculs et éliminer un grand nombre d'hypothèses à considérer en appliquant l'ensemble des contraintes.

KSI Clusturing

Comme aucune technique de diagnostic différentiel explicite n'était utilisé dans HSII, le KSI Clustering (regroupement des instances de sources de connaissance) fut implementé afin de permettre un diagnostic différentiel limité. Cela implique de regrouper les hypothèses ayant des évaluations similaires. Cette technique est très utile lorsque ces hypothèses regroupées sont concurrentes, mais sans pour autant poser problème lorsqu'elles ne le sont pas. Le KSI Clustering a été développé parce que lorsque des hypothèses alternatives dont les évaluations similaires sont concurrentes, la stratégie « island driving »de HSII pose des problèmes en excluant les autres alternatives.

3.3 Autres Architectures de contrôle

3.3.1 HASP/SIAP : Contrôle basé sur les évènements

Contrairement au système basé sur un agenda de HSII, HASP utilise un mécanisme de contrôle se référant aux occurrences d'événements prédéfinis. Plutôt que de reporter de façon systématique les changements possibles, le modèle HASP spécifie les changements qui l'intéressent en définissant ses propres ensembles de type d'évènements. Ensuite, HASP spécifie une séquence d'évènements à exécuter pour chaque type d'événements.

Algorithm 4 Boucle de contrôle pour le modèle HASP.

1: **repeat**
2: Sélection des catégories d'évènements par le module de stratégie
3: Sélection des événements et identification des sources de connaissances appropriées
4: Exécution des sources de connaissances.
5: **until** les critères de terminaison sont vérifiés

3.3.2 CRYSALIS : Contrôle hiérarchique

Contrairement au modèle HSII, CRYSALIS utilise une hiérarchie de sources de connaissance à contrôler pour sélectionner la résolution de domaine ou problème à exécuter. CRYSALIS a deux

niveaux de contrôle, sur les stratégies et les tâches. La première phase permet de sélectionner une séquence de sources de connaissance à exécuter. La seconde sélectionne les événements intéressants et les séquences de domaine des sources de connaissance. Un des avantages de cette approche hiérarchique est que les actions sont directement identifiées ce qui rend le processus plus efficace.

3.3.3 Architecture Blackboard Goal-Directed

Un des handicaps posé par l'architecture présentée précédemment est que le format des règles ne permet pas un contrôle du raisonnement aussi explicite que dans les systèmes utilisant des fonctions numériques complexes d'évaluation. Cette architecture dont le contrôle est basé sur un agenda constitue une réponse à ce problème.

Algorithm 5 Boucle de contrôle pour l'architecture Goal-Directed

1: **repeat**
2: A partir des buts, on détermine des sous-objectifs quand cela est approprié, grâce au Goal-to-SubGoal Mapping
3: Identification des sources de connaissances déclenchés
4: Vérification des préconditions des sources de connaissances déclenchées
5: Mise à jour de l'agenda avec les KSI représentant les sources de connaissances actives.
6: Evaluation des KSI et sélection des KSI à exécuter
7: Exécution des KSI
8: Envoi des objectifs grâce au Hypothesis-to-Goal Mapping
9: **until** les critères de terminaison sont vérifiés

3.3.4 BB1

L'architecture BB1 est une extension de l'architecture de contrôle HSII permettant en plus un mécanisme de contrôle de planification. Dans BB1, le problème de contrôle est traité au sein même de la tâche de résolution de problème. Les problèmes de domaine et de contrôle sont résolus en utilisant une approche blackboard. Dans ce but, la structure BB1 intègre un contrôle de blackboard aux domaines du modèle HERSAY-II. La boucle de contrôle standard de BB1 est identique à celle de HSII.

3.3.5 Modèle basé sur une planification incrémentale

Cette architecture permet une planification incrémentale pour les systèmes d'interprétation basés sur les blackboards, intégrant les avantages d'une structure blackboard de résolutions de buts. Ce modèle spécifie les solutions potentielles dans l'espace de recherche, les relations entre les différentes solutions et la difficulté probable pour les construire. Toutes les données sont utilisées pour effectuer la planification. Ce modèle permet de développer des objectifs lointains de haut niveau correspondant aux critères de terminaison.

3.3.6 L'architecture channelized, parameterized

L'architecture blackboard channelized parameterized est une extension de l'architecture basée sur la réalisation de buts combinés avec une version modifiée de BB1. L'objectif de cette architecture était de trouver un système de résolution de problèmes dans certaines applications temps réel. Dans ce type de cas effectivement l'exécution de la boucle du blackboard doit être efficace et prévisible, alors que le nombre potentiel d'instance de sources de connaissance peut être énorme lorsque l'on considère des méthodes approximatives de traitement. En plus, il doit y avoir une représentation des buts actuels et futurs poursuivis.

3.3.7 ATOME : Contrôle hybride multiple

Dans la structure ATOME, la sélection des domaines de source de connaissance à exécuter est obtenue avec un mécanisme de contrôle dérivé de celui de CRYSALIS. L'objectif était d'augmenter l'efficacité du système blackboard sans sacrifier la flexibilité des architectures basées sur les mécanismes de contrôle de HSII. Effectivement, une architecture de contrôle hiérarchique permet généralement d'augmenter l'efficacité de la création du contrôle de décision, mais peut compromettre sa capacité à saisir des opportunités lorsque cela se révèle approprié. Au final, ATOME dérive de l'architecture CRYSALIS en ajoutant la possibilité de résoudre des sous-problèmes soit en identifiant directement des sources de connaissance à exécuter où en appliquant le mécanisme basé sur l'agenda pour sélectionner les sources de connaissance.

3.3.8 CASSANDRA : Contrôle Blackboard distribué

L'architecture CASSANDRA est une modification significative du modèle blackboard, et pas seulement une alternative de son architecture de contrôle. L'objectif était de répondre à une limitation du modèle blackboard : leur manque de modularité et de flexibilité pour un grand nombre de problèmes. Pour augmenter la modularité dans CASSANDRA, la base de données et les mécanismes de contrôle sont structurés de façon modulaire. Le principal composant de CASSANDRA est la gestion de niveau : LM (level manager). Chaque gestionnaire de niveau inclut sa propre base de données locale de solutions partielles, son propre ensemble de sources de connaissance et son propre mécanisme local de contrôle.

3.3.9 RESUN : Planification pour résoudre les sources d'incertitude

RESUN est une structure d'interprétation basée sur le blackboard, qui fait partie des plus récentes architectures de contrôle de blackboard. Son but premier est d'étendre le rang des méthodes que les systèmes d'interprétation peuvent utiliser pour résoudre les incertitudes en permettant l'implémentation de stratégies de contrôle sophistiquées. Ces stratégies impliquent une grande quantité de connaissances spécifiques au contexte étudié. Pour cela, la représentation d'hypothèses des blackboard conventionnel a été étendue et le mécanisme d'agenda a été abandonné pour un système de planification incrémental.

Pour utiliser les méthodes directes permettant de résoudre l'incertitude, le système doit être capable de comprendre pour quelles raisons ces hypothèses sont incertaines. Pour ce faire la représentation d'hypothèses de RESUN maintient des informations détaillées sur les raisons pour lesquelles une hypothèse est incertaine et sur les relations évidentes entre les différentes alternatives.

Chapitre 4

Algorithmes de programmation par contraintes

4.1 Présentation et notations

La notion de **Problèmes de Satisfaction de Contraintes (CSP)** désigne l'ensemble des problèmes définis par des contraintes et consistant à chercher une solution les respectant. Ils se modélisent sous la forme d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine. De façon plus formelle, on définira un CSP par un **triplet (X, D, C)** où X représente l'ensemble des variables, D est la fonction qui associe à chaque variable Xi son domaine D(Xi) i.e. l'ensemble des valeurs que peut prendre Xi et enfin C l'ensemble des contraintes. Chaque contrainte Cj est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables. On appelle **affectation**, noté A, le fait d'instancier certaines variables par des valeurs. Une affectation est dite **totale** si elle instancie toutes les variables du problème ; elle est dite **partielle** si elle n'en instancie qu'une partie. Une affectation (totale ou partielle) est **consistante** si elle ne viole aucune contrainte, et inconsistante si elle viole une ou plusieurs contraintes. Une solution est une **affectation totale consistante**, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte. Les bases étant posées passons à l'étude des algorithmes plus ou moins efficaces permettant de résoudre ces problèmes.

4.2 Algorithmes

4.2.1 Commentaires liminaires

Les algorithmes que nous allons étudier permettent de résoudre de façon générique n'importe quel CSP sur les domaines finis. Il existe d'autres algorithmes plus spécifiques qui tirent parti de connaissances sur les domaines et les types de contraintes pour résoudre des CSP. Par exemple, les CSP numériques linéaires sur les réels peuvent être résolus par l'algorithme du Simplex (bien connu en recherche opérationnelle) ; les CSP numériques linéaires sur les entiers peuvent être résolus en combinant l'algorithme du Simplex avec une stratégie de Séparation et Evaluation ; les CSP numériques non linéaires sur les réels peuvent être résolus en utilisant des techniques de propagation d'intervalles ; etc...

Nous aborderons principalement les algorithmes de recherche systématique qui nous le verrons peuvent être couplés avec des techniques d'analyse de consistance qui permettent de réduire les recherches. Nous verrons également qu'il existe aussi un certain nombre d'heuristiques et de techniques qui permettent de trouver des solutions non complètes ou pas toujours optimales aux CSP.

4.2.2 Génère et Teste - Generate and Test (GT)

La façon la plus simple (très naïve, peu efficace et inexploitable dans le monde industriel !) de résoudre un CSP sur les domaines finis consiste à explorer toutes les affectations totales possibles jusqu'à en trouver une qui satisfasse toutes les contraintes. L'algorithme est donné ci-après et ne nécessite pas de commentaires particuliers, il est de complexité $O(max(|Di|)^n)$, avec n le nombre de variables.

Algorithm 6 Génère et Teste (GT)

Require: (X,D,C) CSP sur domaines finis, A une affectation partielle pour (X,D,C)
Ensure: retourne vrai si l'affectation A peut être étendue en une solution pour (X,D,C), faux sinon

```
1:  function GenereTeste(A,(X,D,C))
2:      if toutes les variables de X sont affectées à une valeur dans A then   ▷ A affectation totale
3:          if A est consistante then                                          ▷ A est une solution
4:              return vrai
5:          else
6:              return faux
7:          end if
8:      else                                                                   ▷ A est une affectation partielle
9:          Choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
10:         for toute valeur Vi appartenant à D(Xi) do
11:             if GenereTeste(A U {(Xi, Vi)}, (X,D,C)) = vrai then
12:                 return vrai
13:             end if
14:         end for
15:         return false
16:     end if
17: end function
```

4.2.3 Simple retour arrière - Backtracking (BT)

Une première façon d'améliorer l'algorithme GT consiste à tester au fur et à mesure de la construction de l'affectation partielle sa consistance : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à la compléter. Dans ce cas, on « retourne en arrière »(backtrack) jusqu'à la plus récente instanciation partielle consistante que l'on peut étendre en affectant une autre valeur à la dernière variable affectée. Ce procédé permet donc d'explorer moins exhaustivement l'arbre des affectations possibles, mais il reste évidemment perfectible.

Algorithm 7 Simple retour arrière - Backtracking (BT)

Require: A = affectation partielle, (X,D,C) CSP sur domaines finis
Ensure: retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon

```
1:  function BackTrack(A,(X,D,C))
2:      if A n'est pas consistante then
3:          return faux
4:      end if
5:      if toutes les variables de X sont affectées à une valeur dans A then      ▷ A affectation totale
            consistante = solution
6:          return vrai
7:      else                                                                      ▷ A est une affectation partielle consistante
8:          choisir une varibale Xi de X qui n'est pas encore affectée à une valeur dans A
9:          for toute valeur Vi appartenant à D(Xi) do
10:             if BackTrack(A U {(Xi, Vi)}, (X,D,C)) = vrai then
11:                 return vrai
12:             end if
13:         end for
14:         return false
15:     end if
16: end function
```

4.2.4 Anticipation par noeud - Node Consistency (NC)

Pour améliorer l'algorithme simple retour-arrière, on peut tenter d'anticiper les conséquences de l'affectation partielle en cours de construction sur les domaines des variables qui ne sont pas encore affectées : si on se rend compte qu'une variable non affectée Xi n'a plus de valeur dans son domaine D(Xi) qui soit localement consistante avec l'affectation partielle en cours de construction, alors il n'est pas nécessaire de continuer à développer cette branche, et on peut tout de suite retourner en arrière pour explorer d'autres possibilités.

Pour mettre ce principe en oeuvre, on va, à chaque étape de la recherche, filtrer les domaines des variables non affectées en enlevant les valeurs localement inconsistantes, c'est-à-dire celles dont on peut inférer qu'elles n'appartiendront à aucune solution. On peut effectuer différents filtrages, correspondant à différents niveaux de consistances locales, qui vont réduire plus ou moins les domaines des variables, mais qui prendront aussi plus ou moins de temps à s'exécuter. Les algorithmes qui suivent respectent ce principe de manière plus ou moins complexe, avec plus ou moins d'anticipation.

C'est l'algorithme de ce type le plus simple. Le principe général de l'algorithme anticipation par noeud reprend celui de l'algorithme simple retour-arrière, en ajoutant simplement une étape de filtrage à chaque fois qu'une valeur est affectée à une variable. Le filtrage consiste, ici, à anticiper d'une étape l'énumération : pour chaque variable Xi non affectée dans A, on enlève de D(Xi) toute valeur v telle que l'affectation $AU\{(Xi, v)\}$ soit inconsistante. Voici l'algorithme intégré dans l'algorithme précédent.

Algorithm 8 Anticipation par noeud - Node Consistency (NC)

Require: A = affectation partielle consistante, (X,D,C) CSP sur domaines finis
Ensure: retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon

```
1:  function Nc(A,(X,D,C))
2:      if toutes les variables de X sont affectées à une valeur dans A then   ▷ A affectation totale
            consistante = solution
3:          return vrai
4:      else                                                                    ▷ A affectation partielle consistante
5:          choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
6:          for toute valeur Vi appartenant à D(Xi) do ▷ filtrage des domaines par rapport à A U {(Xi,Vi)}
7:              for toute variable Xj de X qui n'est pas encore affectée do
8:                  Dfiltré(Xj) ← Vj élément de D(Xj) / A U {(Xi,Vi), (Xj,Vj)} est consistante
9:                  if Dfiltré(Xj) est vide then
10:                     return faux
11:                 end if
12:             end for
13:             if Nc(A U {(Xi,Vi)}, (X,Dfiltré,C)) = vrai then
14:                 return vrai
15:             end if
16:         end for
17:         return faux
18:     end if
19: end function
```

4.2.5 Anticipation par arc - Arc Consistency (AC)

AC offre un filtrage plus poussé que l'algorithme précédent. Il teste la consistance des contraintes binaires entre des paires de variables. Il réduit ainsi la taille des domaines à explorer en supprimant les valeurs qui violent les contraintes binaires. L'idée peut-être représenté par un graphe des contraintes, l'arc (Xi, Xj) est un arc consistant si pour toute valeur vi de D(Xi) il y a une valeur vj de D(Xj) telle que Xi=vi et Xj=vj soit permise par les contraintes reliant Xi et Xj. Dès lors on peut supprimer les valeurs des domaines qui sont inconsistantes avec ces contraintes binaires.

Ceci a évidemment pour effet de réduire la taille des domaines à explorer et ce de façon plus radicale que NC. Si le domaine est vide à l'issue du test de consistance, le CSP n'a pas de solution et l'algorithme s'arrête. La fonction Revise illustre le procédé et sert de base à AC1, AC2 et AC3.

Algorithm 9 Anticipation par arc - Arc Consistency (AC)

```
1:  function Revise((Xi, Xj), (X,D,C))
2:      DELETE ← faux
3:      for tous les Vi appartenant à D(Xi) do
4:          if il n'y a pas de Vj dans D(Xj) qui satisfasse les contraintes binaires entre Xi et Xj then
5:              Supprimer Vi de D(Xi)
6:              DELETE ← vrai
7:          end if
8:      end for
9:      return DELETE
10: end function
```

AC1 (Mackworth)

AC1, AC2 et AC3 sont basés sur la répétition de la procédure Revise vu plus haut. Le CSP est rendu AC CSP par l'itération successive de la procédure Revise jusqu'à ce qu'il n'y ait plus de modification dans les domaines des variables. La différence entre ces 3 algorithmes est le choix des arcs sur lesquels va être relancée la procédure lorsqu'un domaine change. Si à l'issue de ces applications de l'algorithme le domaine devient vide, le problème n'a pas de solution.

AC1 est le plus simple. Il réapplique Revise sur chaque domaine à chaque fois qu'un domaine est changé. AC1 est donc coûteux car certains domaines se voient réappliqués Revise alors que cela n'est en fait pas nécessaire. Le problème majeur est que la révision réussie, de même un seul arc, à une itération, force tous les autres arcs à être revisité à la prochaine itération.

Algorithm 10 AC1

```
1:  function AC1((X,D,C))
2:      Q ← {(Xi, Xj) / il existe une contrainte entre Xi et Xj}
3:      repeat
4:          R ← false
5:          for Tous les (Xi, Xj) de Q do
6:              R ← (R ou Revise((Xi, Xj), (X,D,C)))
7:          end for
8:      until non R
9:      return (X,D,C)
10: end function
```

AC2 (Waltz) AC3 (Mackworth)

Ce sont des algorithmes plus efficaces qu'AC1 car ils ne réappliquent Revise que le nombre de fois nécessaires. AC3 est l'un des plus utilisés, il implémente une simple file d'arcs à étudier. Cet algorithme relance Revise seulement sur les arcs qui pourraient avoir été affectés par une modification antérieure.

Algorithm 11 AC3

```
1:  function AC3((X,D,C))
2:      Q ← {(Xi, Xj) / il existe une contrainte entre Xi et Xj}
3:      while Q ≠ ∅ do
4:          Q ← Q \ (Xi, Xj)
5:          if Revise((Xi, Xj), (X,D,C)) then
6:              Q ← Q U {(Xk, Xi) / il existe une contrainte entre Xk et Xi et Xk ≠ Xi et Xk ≠ Xj}
7:          end if
8:      end while
9:      return (X,D,C)
10: end function
```

AC4 (Mohr and Henderson)

AC4 conserve l'idée de réitérer un minimum de fois possible la routine de révision des domaines et ajoute des structures de données plus complexes pour contenir l'information des valeurs de chaque variable. En particulier, pour chaque valeur des variables il y a un compteur indiquant le nombre de valeurs satisfaisantes contenues dans le domaine D(Xj) associé. L'appel de la fonction Revise sera alors fait quand le nombre de ces valeurs satisfaisantes atteint 0. Grâce au maintien de ces structures, AC4 peut reduire le nombre d'appel à Revise, mais malheureusement la mise à jour de ces structures lourdes a un coût qui le handicape face aux autres algorithmes.

AC5 (Hentenryck, Deville and Teng)

AC5 est un algorithme générique de consistance des arcs qui peut rivaliser avec AC3 avec une bonne complexité en moyenne ainsi qu'avec AC4 avec une meilleur complexité au pire cas. De plus, cet algorithm peut exploiter des informations sémantiques durant les révisions, en particulier il apporte de bons resultats avec contraintes fonctionnelles ou monotes.

AC6 (Bessiere)

AC6 améliore à la fois la consommation de mémoire d'AC4 et le temps moyen d'exécution. Au lieu de garder l'ensemble complet des compteurs, AC6 mémorise seulement un compteur pour chaque valeur. Si le compteur est perdu par une réduction de domaine un autre est cherché. Ainsi, la complexité de l'initialisation d'AC4 est réduite et de larges structures de données sont inutiles.

AC7 (Bessiere, Freuder et Regin)

AC7 est une extension d'AC6 qui utilise la symétrie des contraintes : si la valeur v1 supporte une autre valeur v2 alors v2 supporte v1 également.

AC1 - AC7 sont des algorithmes de forte consistance des arcs (tous les arcs sont des arcs consistants). C'est pourquoi la majeur partie du temps d'exécution est passée à réitérer Revise. Une autre alternative permettant de réduire les temps est de considérer l'algorithme suivant de consistance faible des arcs.

Weak AC - Directional Arc Consistency (DAC)

DAC est plus faible qu'AC, les arcs sont consistants dans seulement une direction. DAC est un algorithme qui n'a pas besoin de rerévision. DAC ordonne les variables dans le graphe de contraintes en conservant la consistance des arcs (i, j) où i_j uniquement. DAC est donc plus efficace qu'un AC complet dans la construction du CSP consistant car chaque arc est révisé exactement une fois. DAC supprime cependant moins de valeurs qu'AC, il requiert moins de calculs qu'AC1-3 et moins d'espace qu'AC4.

DAC n'élimine pas complètement la nécessité de retour en arrière, mais en général il réduit considérablement l'espace de recherche.

4.2.6 Path Consistency (PC)

PC est une technique de consistance plus forte qu'AC, il en représente une extension naturelle. Au lieu de considérer les valeurs inconsistantes d'un arc entre une paire de variables, PC teste l'inconsistance des valeurs de toutes les paires de variables. En d'autres termes, la longueur du chemin considéré par AC est égale à 1 tandis que celle de PC est au moins égale à 2. Le chemin (V1, ..., Vn) est consistant si pour toute paire V1, Vn de valeurs consistantes, il existe des valeurs V2, ..., Vn-1 telles que toutes les contraintes Vi, Vi+1 soient satisfaites. Un CSP est consistant de chemin si tous ses chemins sont consistants. La plupart des algorithmes s'intéressent à la consistance des chemins de longueur 2. Comme AC, PC procède par itérations successives d'un algorithme de révision des domaines. Il y a plusieurs implémentations de PC, PC1 à PC5. PC1 se rapproche de AC1 sauf que l'on considère des chaînes de longueur 2 et non plus 1. Comme AC1 il réitère l'algorithme de révision de façon trop brutale ce qui le rend en pratique peu exploitable.

PC2 - PC3 (Mohr, Henderson)

PC2 et PC3 sont des algorithmes améliorés pour lesquels seulement les contraintes significatives sont visitées. Comme AC2 et AC3, PC répète la révision des domaines seulement sur les chemins affectés par une itération précédente, et non sur les 3 variables comme PC1.

PC4 (Han and Lee)

PC4 est une amélioration de PC3 qui ajoute des structures de données plus complexes (des listes) pour conserver les informations sur les supporters des variables (valeurs qui rendent la chaîne satisfiable). Ces informations aident à déterminer les chemins devant être revisités après un changement de domaine.

PC5 (Singh)

PC5 est une extension de PC4 qui utilise le même principe qu'AC6, un seul supporter est calculé et un nouveau supporter est recherché quand le supporter courant est perdu.

Bien que la consistance des chemins de longueur 2 soit strictement plus forte que la consistance des arcs, cette méthode est rarement utilisée en pratique, en effet PC souffre des problèmes suivants :

– PC élimine plus d'inconsistances qu'AC mais le rapport performance complexité est pire qu'AC

– PC consomme beaucoup de mémoire car il nécessite une représentation par extension des contraintes.

Weak PC - DPC

Directional Path Consistency (DPC) est plus faible que PC, comme DAC est plus faible qu'AC. DPC fait presque les mêmes opérations que PC sauf que DPC choisit et met à jour 3 variables en descendant.

RPC

Restricted Path Consistency (RPC) (Pierre Berlandier) est une combinaison des avantages d'AC et PC. RPC augmente la puissance d'AC4 en appliquant PC si il y a seulement un supporter dans une contrainte (il anticipera d'une étape). Si le supporter n'a pas de supporter alors RPC enlève la valeur du domaine. RPC supprime au moins le même nombre de valeurs inconsistantes qu'AC. RPC est donc plus fort que tous les autres algorithmes AC. Cependant, comme PC est appelé seulement sous la condition d'un seul supporter, RPC est plus faible qu'un PC complet.

4.2.7 Combinaison de recherche systématique et techniques de consistance

Nous avons présenté 2 approches différentes : la recherche systématique et les techniques de consistances. La combinaison de ces 2 méthodes augmente l'efficacité de recherche de solutions. Une façon simple de procéder est d'utiliser les techniques de consistance pour réduire la taille du problème et ensuite d'utiliser la recherche systématique pour trouver une solution. Cela permet évidemment de réduire l'espace de recherche, si l'ensemble des domaines est vide il est d'ailleurs évidemment inutile de lancer la recherche.

Une autre approche consiste à lancer des recherches de consistance durant l'exécution de la recherche systématique. Il y a 2 méthodes d'implémentation : Look Back et Look Ahead.

Look Back

Lorsque qu'un retour en arrière a lieu, l'algorithme peut identifier la source d'inconsistance. Ainsi il n'y a pas de travail redondant. Cependant, la détection tardive du conflit est un inconvénient. Backjumping (BJ) utilise les contraintes violées comme guide pour trouver la variable rentrant en conflit. Backmarking (BM) garde en mémoire les valeurs incompatibles avec la valeur récemment assignée. Tant que cette valeur est en cours d'étude, les valeurs incompatibles ne seront pas considérées. Backchecking (BC) est une amélioration de BM. Il réduit le nombre de tests de compatibilité en gardant en mémoire les inconsistances. De plus, il évite la répétition inutile de test de compatibilité qui ont déjà été effectué avec succès.

Look Ahead

Avant d'assigner une valeur à la prochaine variable, AC est appliqué pour réduire la taille du domaine de la prochaine variable. Si le domaine devient vide, la solution partielle courante est inconsistante. Avec cette méthode, lorsqu'une variable est assignée, toutes ses valeurs restantes sont garanties d'être consistantes avec les variables de la solution partielle. Les conflits sont donc anticipés.

Les différences entre Forward checking, Look Ahead partiel et Look Ahead complet viennent de la force de l'AC utilisé. Forward Checking (FC) utilise l'AC le plus faible avec seulement les contraintes de la variable courante et des futures variables. Partial Look Ahead (PLA) applique DAC avec les contraintes de la variable courante et des futures variables et celles des futures variables et de leurs futures variables. Full Look Ahead (FLA) accomplit un AC complet avec toutes les futures variables non encore instanciées. FLA detecte les valeurs inconsistantes plus tôt que PLA et FC. PLA detecte les inconsistances plus tôt que FC.

Évidemment tout cela a un coût. Dans certains cas FLA peut même être plus coûteux qu'un simple backtracking. C'est pourquoi FC et BC sont encore utilisés dans des applications.

4.2.8 Améliorations de la recherche

Choisir le bon ordre des variables et valeurs à teste peut améliorer l'efficacité de la recherche de solution d'un CSP. L'ordre peut être soit un ordre statique prédéfini à l'avance ou bien un ordre dynamique, dans lequel le choix de la prochaine variable à étudier dépend de l'état courant des recherches. Il existe un certain nombre d'heuristiques pour choisir un ordre.

Ordre des variables

Minimal Width Ordering (MWO)

Il s'agit d'une heuristique qui donne un ordre statique de choix de variables, les variables qui sont contraintes par le plus de variables seront étudiées en premières. En conséquence, moins de retour en arrière seront nécessaires.

Minimal bandwidth ordering (MBO)

MBO est une heuristique qui donne un ordre dynamique. L'ordre des variables est utilisé avant un retour arrière. La variable qui a le moins de distance (largeur de bande) est choisie pour le retour arrière. La largeur de bande d'un noeud V dans un graphe ordonné est la distance maximale entre V et tout noeud qui lui adjacent selon l'ordre. La bande passante d'un ordre h est la bande passante maximale de tous les noeuds du graphe et la bande passante d'un graphe est la bande passante minimale de tous les ordres du graphe.

The Fail First Principle (FFP)

FFP est une heuristique générale de recherche. Il s'agit d'accomplir d'abord les tâches les plus susceptibles d'échouer. La mesure de la probabilité déchec peut être faite en considérant les contraintes et les tailles des domaines des variables. La variable qui a le plus de contraintes ou le plus petit domaine a plus de change de mener à des inconsistances. L'implémentation de FFP peut être dynamique ou statique.

La méthode Search Rearrangement est une heuristique très puissante proposée par Bitner et Reingold souvent utilisé avec FC. Dans cette méthode, la variable qui offre le moins d'alternatives possibles

est choisie pour l'instanciation. L'ordre d'instanciation des variables est déterminé dynamiquement. Freuder présenta l'instanciation le plus tôt possible des variables participant au plus grand nombre de contraintes.

Fox, Sadeh et Baykan ont travaillé sur l'analyse structurale des caractéristiques du CSP à résoudre pour déterminer l'ordre des variables à choisir.

4.2.9 Ordre des valeurs

L'ordre des valeurs choisi peut avoir un impact substanciel sur le temps mis pour trouver la première solution. Quand la décision est prise d'instancier une variable, il se peut qu'elle est plusieurs valeurs possibles. Un ordre différent de choix des valeurs changera la structure arborescente des noeuds de l'arbre de recherche. Cela peut représenter un avantage si ce choix assure que la branche qui mène à une solution est choisie avant une branche qui ne mène à rien. Bien sûr, si l'on recherche tous les solutions, ou si le CSP n'a pas de solution l'ordre est indifférent.

Succeed first Principle

SFP est une stratégie qui choisit la valeur qui a le plus de chance de succès et le moins de chance de mener à un conflit. Une heuristique possible est de choisir préférentiellement les valeurs qui maximisent le nombre d'options disponibles. AC4 est adapté à cette heuristique puisqu'il compte le nombre de supporters. La valeur qui a le plus de supporters doit être choisie en première.

Pour des problèmes aléatoirement choisis, et probablement en général, le travail que cela implique ne vaut pas le bénéfice qu'il apporte, c'est à dire celui de choisir une valeur qui sera en moyenne plus susceptible de conduire à une solution qu'une autre. Pour certains types de problèmes cependant, il peut y avoir des informations permettant de choisir un ordre susceptible de conduire plus vite à une solution.

4.2.10 Résolution des MCSP

Parmi les CSP il y a des problèmes qui n'ont pas de solution complète. Résoudre ces problèmes par les algorithmes vu précédemment conduira à un échec. Une solution incomplète est une solution partielle qui peut être acceptée. Adopter une solution incomplète peut s'avérer utile dans le cadre des systèmes temps réel où une solution doit être trouvée dans un temps limité. Il convient alors soit d'assouplir les contraintes, soit de satisfaire le plus grand nombre de contraintes possibles. Assouplir les contraintes consiste par exemple à agrandir les domaines, supprimer une variable ou une contrainte. Trouver une valeur à toutes les variables, de telle sorte que le moins possible de contraintes soient violées est la résolution d'un MCSP (Maximum CSP). Il y a deux approches permettant de résoudre les MCSP, les méthodes exactes et approximatives. Les méthodes exactes basées sur Branch and Bound (BB) donnent la solution optimale. Les méthodes approximatives basées sur une recherche locale donne une solution incomplète qui n'est pas forcément optimale.

Méthode exacte

Branch and Bound Algorithm (BB)
BB a été développé pour résoudre les MCSP par Freuder & Wallace. L'algorithme parcourt tous les chemins, à travers un arbre de recherche, avec un coût qui ne décroît pas avec la longueur du chemin. La recherche à travers un chemin donné peut s'arrêter lorsque le coût de l'affectation partielle des valeurs des variables est au moins aussi grand que le plus petit coût déjà trouvé pour une affectation totale. BB étend une solution partielle dans chaque chemin de l'arbre et mémorise le chemin le plus long. Le chemin le plus long qui a le plus grand nombre de variables instanciées est le chemin optimal. L'efficacité de BB peut être améliorée par des choix d'ordonnancement de variables et de valeurs, à l'aide d'heuristique comme vu précédemment.

Méthode approximative

Les méthodes approximatives ne garantissent pas l'optimalité de la solution. Il peut y avoir une autre solution qui satisfait plus de contraintes, mais la solution approximative reste proche de la solution optimale.

Hill Climbing, Min-Conflicts (MC), Min-Conflicts Random Walk (MCRW), Steepest Descent Random Walk (SDRW) and Tabu list sont des algorithmes basés sur une idée commune basée sur la notion de recherche locale. Dans la recherche locale, une configuration initiale (valuation des variables) est générée et l'algorithme passe de cette configuration à une configuration voisine jusqu'à trouver une solution (problèmes de décision) ou une bonne solution (problèmes d'optimisation) ou encore jusqu'à ce que les ressources disponibles soient épuisées. Ils utilisent différentes heuristiques et algorithmes stochastiques pour orienter la recherche.

Les étapes de HC, MC, MCRW, SDRW et Tabu list sont les suivantes :

1. partir d'un état initial généré aléatoirement (assignement complet des valeurs des variables)
2. évaluer le nombre de contraintes violées dans l'état courant
3. essayer de passer à un état meilleur, état voisin qui diffère d'une variable
4. si l'état courant ne peut trouver un état voisin meilleur et que la solution n'est pas un optimum global, l'état est noté optimum local
5. Quitter l'optimum local
6. Répéter 2 à 5 jusqu'à trouver un optimum global.

La méthode de choix de l'état voisin et de fuite de l'optimum local sont les différences de HC, MC, MCRW, SDRW et Tabu list.

HC considère voisin, ce qui diffère dans la valeur d'une quelconque variable. Le voisinage d'HC est donc plutôt large. HC s'échappe des optima locaux en repartant d'une affectation aléatoire des variables.

MC considére voisin, tout ce qui diffère dans la valeur d'une quelconque variable en conflit. MC ne peut s'échapper des optima locaux.

RW considère voisin, tout état choisi aléatoirement. Il est difficile de trouver une solution avec cette technique car elle n'a pas
d'heuristique de recherche.

MCRW améliore MC afin de pouvoir quitter les optima locaux en ajoutant du bruit à l'algorithme. MCRW fait une combinaison entre heuristique et RW. Si la probabilité d'utiliser un chemin aléatoire est p, la probabilité d'utiliser l'heuristique est 1-p.

SDRW est la combinaison de RW avec HC, l'algorithme ne redemarrera pas à chaque fois qu'il est pris dans un optimum local. Comme pour MCRW, la probabilité d'utiliser un chemin aléatoire est p et la probabilité d'utiliser une heuristique 1-p.

Tabu mémorise une liste des variables changées et les valeurs de quelques-uns des derniers états visités. La liste contient les états interdits que le prochain état voisin ne pourra prendre. Cependant il y a certains critères qui permettent de passer aux états interdits quand cela conduit à une meilleur solution que celles obtenues jusqu'ici. Tabu list est une stratégie qui empêche l'algorithme de rester piéger dans un optimum local.

Chapitre 5

Algorithmes des systèmes multi-agents

5.1 Présentation des agents et des systèmes multi-agents

La résolution coopérative de problèmes prend une place prépondérante dans les recherches en intelligence artificielle distribuée (IAD). Les systèmes multi-agents (SMA) est un domaine de recherche dérivé de l'IAD. Les systèmes multi-agents se focalisent sur l'étude des comportements collectifs et sur la répartition de l'intelligence sur des agents plus ou moins autonomes, capables de s'organiser et d'interagir pour résoudre des problèmes.

L'intelligence artificielle distribuée s'intéresse à des comportements intelligents qui résultent de l'activité coopérative de plusieurs agents. Suite à la distribution de l'expertise sur un ensemble de composants qui communiquent pour atteindre un objectif global ou résoudre un problème, il est nécessaire de diviser le problème en sous-problèmes. Ainsi une extension des systèmes d'IAD est proposée : les composants doivent être capables de raisonner sur les connaissances et les capacités des autres dans le but d'une coopération effective. Pour ce faire, ils doivent être dotés de capacités de perception et d'action sur l'environnement et doivent posséder une certaine autonomie de comportement, on parle alors d'agents et par conséquent de système multi-agents.

Un agent est ainsi une entité qui perçoit son environnement et agit sur celui-ci. Cette entité, réelle ou abstraite, situé dans un environnement, agit d'une façon autonome pour atteindre les objectifs pour lesquels il a été conçu.

5.2 Algorithmes de contrôle

5.2.1 Agents Réactifs

L'exécution d'un agent réactif est directement liée à ses perceptions par une fonction réflexe (stimulus en fonction d'une réponse). Le comportement de l'agent correspond ainsi à un automate à états finis.

Algorithm 12 Cycle de base d'un agent réactif :

Require: rules : règles condition-action, percepts : ensemble de percepts
1: **repeat**
2: stat ← InterpretInput(percept)
3: rule ← match(state, rules)
4: execute(rule[action])
5: **until** l'agent est arreté

Le comportement réflexe est fondé sur des comportements, des interactions ou des situations

élémentaires. Cette modélisation ne comprend pas de représentation de l'environnement des autres agents ou de ses capacités. L'historique ou les plans d'actions ne sont pas pris en compte : les actions exécutées ne dépendent que des actions présentes.

5.2.2 Agents délibératifs

Les agents délibératifs utilisent des représentations explicites de l'environnement, des autres agents et de leurs capacités. Cela implique la gestion d'un historique et un contrôle délibératif :

- Interaction avec les autres par des communications sophistiquées
- Participation à des organisations sociales
- Systèmes constitués de peu d'agents, hétérogènes

Algorithm 13 Cycle de base d'un agent délibératif :

Require: s : état, eq : file d'événements
1: s ← initialise()
2: **repeat**
3: options ← option_generator(eq, s)
4: selected ← deliberate(options, s)
5: s ← update_state(selected, s)
6: execute(rule[action])
7: eq ← get_new_events()
8: **until** l'agent est arreté

5.2.3 Agents BDI

Une architecture BDI est conçue en partant du modèle « Croyance-Désir-Intention »(Belief-Desire-Intention), de la rationalité d'un agent intelligent.

Les croyances d'un agent sont les informations que l'agent possède sur l'environnement et sur d'autres agents qui existent dans le même environnement. Les croyances peuvent être incorrectes, incomplètes ou incertaines et, à cause de cela, elles sont différentes des connaissances de l'agent, qui sont des informations toujours vraies. Les désirs d'un agent représentent les états de l'environnement ou son propre état, tel qu'il aimerait les voir réalisés. Les intentions d'un agent sont les désirs que l'agent a décidé d'accomplir ou les actions qu'il a décidé de faire pour accomplir ses désirs.

Algorithm 14 Algorithme de contrôle d'agent BDI

Require: b : croyance, g : desirs, i : intentions, eq : file d'événements
1: (b, g, i) ← initialise()
2: **repeat**
3: options ← option_generator(eq, b, g, i)
4: selected ← deliberate(options, b, g, i)
5: i ← selected Union i
6: execute(rule[action])
7: eq ← get_new_events()
8: b ← update_beliefs(b, eq)
9: (g, i) ← drop_successful_attitudes(b, g, i)
10: (g, i) ← drop_impossible_attitudes(b, g, i)
11: **until** l'agent est arreté

5.3 Algorithmes de recherche dans les systèmes à agents

Les algorithmes de recherche sont utilisés pour résoudre deux types d'interactions entre les agents : la coopération pour résoudre les problèmes et la compétition dans le cas des jeux. Pour cela deux grandes classes d'algorithmes de recherche sont utilisées : les algorithmes non-informés (aveugles), qui réalisent une recherche exhaustive et les algorithmes informés, qui utilisent des sources d'information supplémentaires en parvenant ainsi à des performances meilleures. Ces méthodes étant décrites dans les autres sections de ce rapport (cf. section PPC, Elagage, Forward et Backward), nous ne les détaillerons pas dans cette partie.

5.4 La communication entre agents

5.4.1 KQML

« Knowledge Query and Manipulation Language »(KQML) est un langage extérieur de haut niveau pour les agents, orienté sur l'échange des messages, indépendant de la syntaxe et de l'ontologie du contenu des messages. Indépendant du transport et du langage utilisé, il permet de spécifier le format des messages échangés par les agents.

Le langage KQML spécifie le format des messages échangés par les agents. Un message KQML peut être vu comme un objet, défini par l'information clé, la performative (la classe) et un nombre variable d'attributs :
(ask-if // performatif
:sender A // Informations utiles pour le routage et l'interprétation du message
:receiver B
:language prolog
:ontology industrial
:reply-with id1
:content start(process, i) // contenu)

5.4.2 ACL-FIPA

Ayant une syntaxe similaire à KQML le langage de communication entre agents ACL-FIPA s'appuie sur la définition de deux ensembles :

1. un ensemble d'actes de communication primitifs, auquel s'ajoutent les autres actes de communication pouvant être obtenus par la composition de ces actes de base
2. un ensemble de messages prédéfinis que tous les agents peuvent comprendre ACL-FIPA possède 21 actes communicatifs, exprimés par des performatives, qui peuvent être groupés.
   - passage d'information
   - réquisition d'information
   - négociation
   - distribution de tâches (ou exécution d'une action)
   - manipulation des erreurs

En ACL-FIPA il n'existe pas de primitives de gestion ni de facilitation.

5.5 La négociation

5.5.1 Présentation

Dans un système multi-agents les agents interagissent en vue de réaliser des tâches ou d'atteindre des buts. L'interaction a lieu, d'habitude dans un environnement commun où les agents ont diverses zones d'influence, notamment diverses parties de l'environnement sur lesquelles ils peuvent agir. Ces

zones peuvent être disjointes mais, dans la plupart des cas, elles se superposent et l'environnement est partagé par les agents. En interagissant dans un environnement partagé, les agents doivent coordonner leurs actions et avoir des mécanismes pour la résolution des conflits. La coordination et la résolution des conflits sont surtout nécessaire dans le cas des agents egocentrés (des agents ayants leurs propres buts, désirs, préférences...) ou compétitifs mais aussi bien, parfois, dans le cas des agents coopératifs pour la communication des changements des plans ou l'allocation des tâches. Le mécanisme favori pour la résolution des conflits et la coordination, inspiré du modèle des humains, est la négociation.

5.5.2 Négociation aux enchères

Les enchères (auctions) sont des mécanismes d'interaction simples mais nécessitant une étude préalable d'un certain nombre de problème, concernant principalement le choix du protocole et de la stratégie à utiliser. Une enchère comprend habituellement un initiateur (actioneer), et plusieurs participants (bidders). Les offres des participants peuvent se faire une seule fois ou en plusieurs tours, en fonction du protocole d'enchère. A la fin, l'initiateur choisit le gagnant, les règles pour choisir le gagnant étant, de même, spécifiques au protocole.

Il y a beaucoup de protocoles d'enchère, nous nous contenterons donc de présenter les plus importants, en expliquant aussi qu'elle est la meilleure stratégie à choisir, lorsqu'une telle stratégie existe.

Enchère anglaise (premier-prix offre-publique)
L'initiateur commence l'enchère, d'habitude par l'annonce d'un prix de réservation. Chaque agent participant annonce de façon publique son offre, en plusieurs tours successifs. Quand aucun participant ne veut plus augmenter son offre, l'enchère s'arrête et le participant ayant fait la plus grande offre obtient l'objet au prix de son offre.

Dans les enchères à valeurs privées, la stratégie dominante est de faire une offre un peu plus grande que la dernière offre et de s'arrêter quand la valeur privée est atteinte. Dans les enchères à valeurs corrélées, il n'y a pas de stratégie dominante. Le participant augmente le prix d'une quantité constante ou d'une quantité qu'il considère justifiée.

Enchère première offre-cachée
L'initiateur commence l'enchère et chaque agent participant soumet une offre, dans un tour unique, sans connaître les offres des autres participants. Le participant qui a fait la plus grande offre gagne l'objet au prix de son offre. Dans ce protocole il n'y a pas de stratégie dominante, mais des algorithmes dépendant du contexte peuvent être réalisés pour évaluer la valeur attribuée par les autres participants à l'objet.

Enchère hollandaise (descendante)
L'initiateur commence par proposer un prix et, par des tours successifs, diminue ce prix jusqu'au moment où un des participants achète l'objet au prix proposé. Le protocole est équivalent à celui de l'enchère premier-prix offre-cachée et il n'y a donc pas de stratégie dominante, en général.

Enchère Vickery (deuxième-prix offre-cachée)
Chaque agent participant soumet une offre sans connaître les offres des autres, dans un seul tour. Jusqu'à ce moment le protocole est le même que celui de l'enchère premier-prix offre-cachée. La différence est que le participant qui a fait l'offre la plus grande gagne mais il doit payer le prix de la deuxième plus grande offre. La stratégie dominante d'un participant dans ce cas est de soumettre une offre avec sa valeur privée de l'objet. Cette particularité à permis à L'Enchère Vickery d'être la plus utilisée pour les agents logiciels.

5.5.3 Allocation des tâches par réseau contractuel

Nous avons présenté précédemment des protocoles de négociation entre agents egocentrés, c'est à dire entre agents ayant leurs propres buts. Le protocole réseau contractuel (Contract Net) est un protocole de négociation qui a été conçu en vue de la coordination d'agents coopératifs ayant les même buts et résolvant ensemble les problèmes. Ce protocole a été une des premières approches utilisées dans les systèmes multi-agents pour résoudre le problème d'allocation des tâches. Il s'appuie sur une métaphore organisationnelle : les agents coordonnent leurs activités grâce à l'établissement de contrats afin d'atteindre des buts spécifiques.

Dans le protocole réseau contractuel, les agents peuvent prendre deux rôles : gestionnaire et contractant. L'agent qui doit exécuter une tâche (le gestionnaire) commence par décomposer cette tâche en plusieurs sous-tâches. Le gestionnaire annonce chaque sous-tâche sur un réseau d'agents (les contractants). Les agents qui reçoivent une annonce de tâches à accomplir évaluent l'annonce. Les agents qui ont les ressources appropriées, l'expertise ou l'information requise pour accomplir la tâche, envoient au gestionnaire des soumissions qui indiquent leurs capacités à réaliser la tâche. Le gestionnaire rassemble toutes les propositions qu'il a reçues et alloue la tâche à l'agent qui a fait la meilleure proposition.

5.5.4 Allocation des tâches par redistribution

Ce type d'allocation utilise des domaines orientés tâches (task oriented domains). Un domaine orienté tâche est un triplet $< T, Ag, c >$ où :

- T est un ensemble de tâches
- Ag = {1, ... ,n} est un ensemble d'agents qui participent à la négociation
- c est une fonction coût qui définie les coûts nécessaires pour exécuter chaque sous-ensemble de tâches.

La fonction coût doit satisfaire deux contraintes : elle doit être monotone et le coût de ne pas exécuter une tâche doit être zéro.

Pour réaliser une meilleure allocation des tâches les agents utilisent un protocole appelé le protocole de concession monotone. Les règles du protocole sont comme suit :

- La négociation se déroule en une suite de tours.
- Au premier tour, les deux agents proposent simultanément une affaire de la série de négociation.
- Un accord est atteint si les deux agents proposent des affaires A1 et A2 telles que soit utilité1(A2) ≥ utilité1(A1) soit utilité2(A1) ≥ utilité2(A2).
- Si un accord est atteint : si les deux offres des agents égalent ou dépassent ceux de l'autre agent, alors une des propositions est choisie au hasard. Si seulement une proposition dépasse ou égale l'autre proposition, alors c'est elle qui est l'affaire sur laquelle les agents sont d'accord.
- Si aucun accord n'est atteint, alors la négociation continue pour un autre tour de propositions simultanées. Au tour u + 1, aucun agent n'a le droit de faire une proposition qui est moins préférée par l'autre agent que l'affaire qu'il a proposée au tour u.
- Si aucun agent ne fait de concession à un tour donné, alors la négociation est terminée et il y a conflit.

Le protocole de concession monotone garantit que la négociation se terminera avec ou sans accord, après un nombre fini de tours. Cependant le protocole ne certifie pas qu'un accord sera rapidement atteint. Il est concevable que la négociation continue pour un nombre de tours qui croît d'une manière exponentielle par rapport au nombre de tâches à allouer.

5.5.5 Négociation heuristique

La négociation heuristique concerne les agents égocentrés. Le handicape des protocoles présentés précédemment est que l'initiateur n'a aucun moyen de savoir si sa proposition est acceptable ou non, et si l'on est proche d'un accord.

Pour améliorer l'efficacité de la négociation, ce protocole permet aux agents de fournir des réactions plus utiles aux propositions qu'ils reçoivent. Ces réactions peuvent prendre la forme d'une critique ou d'une contre-proposition (proposition refusée ou modifiée). Une critique est un commentaire sur la partie de la proposition que l'agent accepte ou refuse. Une contre-proposition est une proposition alternative engendrée en réponse à une proposition. À partir de telles réactions, l'initiateur doit être capable d'engendrer une proposition qui est probablement plus apte à mener à un accord.

5.5.6 Négociation par argumentation

La négociation par argumentation permet aux agents d'essayer de changer le rejet ou la modification d'une proposition faite par un autre agent en utilisant des arguments. Ainsi, un agent peut essayer de persuader un autre agent de répondre favorablement à sa proposition en cherchant des arguments qui identifient de nouvelles occasions ou modifient les critères d'évaluation. En plus d'engendrer propositions, contre-propositions et critiques, un agent cherche à rendre la proposition plus attirante en fournissant une information supplémentaire sous forme d'arguments pour sa proposition. La nature et les types des arguments peuvent varier énormément :

- Mode logique (nature déductive)
- Mode émotif
- Mode viscéral (menace par exemple)
- Mode kisceral (intuition, religion...)

Chapitre 6

Algorithmes des réseaux de neurones

6.1 Présentation des réseaux de neurones

Un réseau de neurones est un système composé de plusieurs unités de calcul simples fonctionnant en parallèle, dont la fonction est déterminée par la structure du réseau, la solidité des connexions, et l'opération effectuée par les éléments ou noeuds.

Dans un réseau, chaque sous-groupe fait un traitement indépendant des autres et transmet le résultat de son analyse au sous-groupe suivant. L'information donnée au réseau va donc se propager couche par couche, de la couche d'entrée à la couche de sortie, en passant soit par aucune, une ou plusieurs couches intermédiaires (dites couches cachées). Il est à noter qu'en fonction de l'algorithme d'apprentissage, il est aussi possible d'avoir une propagation de l'information à reculons (back propagation). Habituellement, chaque neurone dans une couche est connecté à tous les neurones de la couche précédente et de la couche suivante, excepté pour les couches d'entrée et de sortie.

Les réseaux de neurones ont la capacité de stocker de la connaissance empirique et de la rendre disponible à l'usage. Les habiletés de traitement (et donc la connaissance) du réseau vont être stockées dans les poids synaptiques, obtenus par des processus d'adaptation ou d'apprentissage. En ce sens, les réseaux de neurones ressemblent donc au cerveau car non seulement, la connaissance est acquise au travers d'un apprentissage mais de plus, cette connaissance est stockée dans les connexions entre les entités, soit dans les poids synaptiques

6.2 Les réseaux feed-forward

Appelés aussi réseaux de type Perceptron, ce sont des réseaux dans lesquels l'information se propage de couche en couche sans retour en arrière possible.

6.2.1 Perceptron simple (ou monocouche)

Le Perceptron est un réseau simple, puisqu'il ne se compose que d'une couche d'entrée et d'une couche de sortie. Le principe de base de sa règle d'apprentissage est d'utiliser l'erreur pour modifier les poids des connexions et diminuer, petit à petit, l'erreur globale du système. Si on considère y comme étant la sortie calculée par le réseau, et d la sortie désirée, le principe de cette règle est d'utiliser l'erreur (d-y), afin de modifier les connexions et de diminuer ainsi l'erreur globale du système. Le réseau va donc s'adapter jusqu'à ce que y soit égal à d.

Algorithm 15 Algorithme d'apprentissage du Perceptron

1: Initialisation des poids et du seuil à de petites valeurs aléatoires
2: Présenter un vecteur d'entrée x^{μ} et calculer sa sortie
3: Mettre à jour les poids en utilisant : $w_t(t+1) = w_j(t) + \eta(d-y)x_j$ ▷ Avec d la sortie désirée

6.2.2 Rétro-Propagation (back propagation)

Contrairement au Perceptron qui ne peut apprendre que dans les cas dans lesquels les catégories à apprendre sont linéairement séparables, cet algorithme permet de résoudre ce problème. Il a fallut lui ajouter une couche. À l'aide de cette couche centrale, il devient alors facile de faire apprendre une telle fonction au réseau.

L'algorithme consiste dans un premier temps à propager vers l'avant les entrées jusqu'à obtenir une entrée calculée par le réseau. La seconde étape compare la sortie calculée à la sortie réelle connue. On modifie alors les poids de telle sorte qu'à la prochaine itération, l'erreur commise entre la sortie calculée et connue soit minimisée. On rétro-propage alors l'erreur commise vers l'arrière jusqu'à la couche d'entrée tout en modifiant la pondération.

Algorithm 16 Algorithme de Rétro-Propagation

1: Initialisation des poids à des petites valeurs aléatoires
2: Choisir, aléatoirement, un pattern d'entrée x^{μ}
3: Propager l'information (en-avant) dans le réseau
4: Calculer δ_i^{λ} sur la couche de sortie ($O_i = Y_i^{\lambda}$ $\delta_i^{\lambda} = dg(h_i^{\lambda})(d_i^{\mu} - y_i^{\lambda})$) : avec ${h_i}^{\lambda}$ l'entrée sur la ième cellule dans la λ^{e}couche. et dg est la dérivée de la fonction d'activation g.
5: Calculer les deltas de la couche précédente par propagation arrière de l'erreur : Pour l de $\lambda - 1$ à 1 faire $\delta_i^{\lambda} = dg(h_i^{\lambda}) \sum_j w_{ij}^{i+1} \delta_j^{i+1}$
6: Mettre à jour les poids en utilisant : $\Delta_{ji}^{\lambda} = \eta \delta_j^i y_j^{i-1}$
7: Retourner en 2 et répeter pour l'entrée suivante, jusqu'à ce que l'erreur en sortie soit inférieure à la limite fixée ou que le nombre maximum d'itérations soit atteint

6.2.3 Adaline

Le réseau Adaline a été développé par Widrow. Il est constitué d'un unique neurone effectuant la combinaison linéaire de ses entrées. Il s'agit en fait d'un Perceptron sans saturation des sorties.

La règle d'apprentissage de ce réseau consiste à minimiser l'erreur quadratique en sortie du réseau de neurone. La règle d'apprentissage est identique à celle du Perceptron, à la différence près que ce sont les entrées non-saturées qui sont prises en compte.

6.2.4 Le perceptron multicouches

C'est une extension du précédent, avec une ou plusieurs couches cachées entre l'entrée et la sortie. Chaque neurone dans une couche est connecté à tous les neurones de la couche précédente et de la couche suivante (excepté pour les couches d'entrée et de sortie) et il n'y a pas de connexions entre les cellules d'une même couche. Les fonctions d'activation utilisées dans ce type de réseaux sont principalement les fonctions à seuil ou sigmoïdes. Il peut résoudre des problèmes non-linéairement séparables et des problèmes logiques plus compliqués, et notamment le fameux problème du XOR. Il suit aussi un apprentissage supervisé selon la règle de correction de l'erreur.

6.2.5 Analyse de discriminants linéaires

Cet algorithme repose sur le postulat de Hebb établi à partir d'observations d'expériences de neurobiologie : si des neurones, de part et d'autre d'une synapse, sont activés de manière synchrone et répétée, la force de la connexion synaptique va aller croissant.

L'une des propriétés remarquables de cette règle est qu'elle exprime que l'apprentissage se fait localement c'est-à-dire que la modification ne dépend que de l'activité des cellules. Cette approche simplifie ainsi de manière significative la complexité d'un circuit d'apprentissage. Un seul neurone

entraîné par la règle de Hebb s'oriente de façon sélective. L'orientation est déduite à l'aide d'une distribution gaussienne et utilisés pour entraîner le neurone. Le vecteur de poids est initialisé, puis au cours de l'apprentissage, le vecteur évolue.

6.3 Les réseaux feed-back

Appelés aussi réseaux récurrents, ce sont des réseaux dans lesquels il y à retour en arrière de l'information.

6.3.1 Apprentissage de Boltzmann

Les réseaux de Boltzmann sont des réseaux symétriques récurrents. Ils possèdent deux sous-groupes de cellules, le premier étant relié à l'environnement (cellules dites visibles) et le second ne l'étant pas (cellules dites cachées). Cette règle d'apprentissage est de type stochastique (relève partiellement du hasard) et elle consiste à ajuster les poids des connexions, de telle sorte que l'état des cellules visibles satisfasse une distribution probabiliste souhaitée.

6.3.2 Cartes Auto-Organisatrices de Kohonen (SOM)

Ce sont des réseaux à apprentissage non-supervisé qui établissent une carte discrète, ordonnée topologiquement, en fonction de patterns d'entrée. Le réseau forme ainsi une sorte de treillis dont chaque noeud est un neurone associé à un vecteur de poids. La correspondance entre chaque vecteur de poids est calculée pour chaque entrée. Par la suite, le vecteur de poids ayant la meilleure corrélation, ainsi que certains de ses voisins, vont être modifiés afin d'augmenter encore cette corrélation. Ainsi, chaque cellule calcule la distance euclidienne entre le vecteur patron et le vecteur de poids associé à la cellule dans le tableau.

6.3.3 Les réseaux de Hopfield

Les réseaux de Hopfield sont des réseaux récurrents et entièrement connectés. Dans ce type de réseau, chaque neurone est connecté à chaque autre neurone et il n'y a aucune différenciation entre les neurones d'entrée et de sortie.

La règle d'apprentissage proposée par Hopfield est basée sur la règle de Hebb. La règle de Hebb consiste à forcer les poids des liaisons entre les neurones actifs au même moment. Par contre, les poids seront diminués si les neurones sont dans des états contraires. Dans le cas de Hopfield, cette règle est légèrement étendue si l'on considère que deux neurones dans l'état -1 sont actifs.

Hopfield a utilisé une fonction d'énergie associée au réseau comme outil pour définir des réseaux récurrents et pour comprendre leurs dynamiques. Ils fonctionnent comme une mémoire associative non-linéaire et sont capables de trouver un objet stocké en fonction de représentations partielles ou bruitées. L'application principale des réseaux de Hopfield est l'entrepôt de connaissances mais aussi la résolution de problèmes d'optimisation. Le mode d'apprentissage utilisé ici est le mode non-supervisé.

6.3.4 Le Réseau de Anderson (Brain in a Box)

Ce réseau a été conçu par James Anderson sous le nom de Brain in a Box afin d'étudier les fonctionnalités du cerveau humain. En d'autres termes, ce réseau essaye de modéliser un comportement psychologique.

Pour réaliser la phase d'apprentissage, on initialise d'abord les poids à des valeurs faibles. On présente alors les vecteurs d'exemples en entrée. On propage ainsi la valeur vers la couche intermédiaire. La propagation finie, on calcule la différence entre la valeur réelle de l'exemple en sortie

et la valeur calculée. On obtient alors la valeur de correction des poids. On ré-injecte alors cette valeur calculée en entrée de la couche intermédiaire et on répète le même processus. Cette phase d'apprentissage est répétée un certain nombre de fois, fixe et déterminé à l'avance. Ce paramètre joue un rôle très important dans l'erreur d'apprentissage.

6.3.5 Les modèles de Résonance Adaptative

Il s'agit de résoudre le dilemme de stabilité-plasticité. (Comment apprendre nouvelles choses (plasticité) tout en gardant une stabilité garante d'une connaissance ni supprimée ni abîmée). Les modèles développés par Carpenter et Grossberg (ART-1, ART-2, ARTMap) dans le cadre de la théorie de résonance adaptative (ART) essaient de résoudre ce dilemme. Le réseau possède un réservoir de cellules de sortie qui ne sont utilisées que si nécessaire.

L'algorithme d'apprentissage met à jour les vecteurs prototypes stockés uniquement s'ils sont suffisamment proches du patron fourni en entrée au réseau. Lorsqu'un patron n'est pas assez proche des vecteurs prototypes déjà présents dans le réseau, une nouvelle catégorie est créée et une cellule libre y est assignée avec comme vecteur prototype le patron correspondant.

6.4 Les algorithmes d'apprentissage par compétition

6.4.1 Winner Take All (WTA)

A la différence de la règle de Hebb dans laquelle plusieurs neurones peuvent être activés en sortie, cet apprentissage n'active qu'un seul neurone. On parle de WTA (winner-take-all), phénomène a été mis en évidence dans le cas de réseau biologique. Ce type d'apprentissage regroupe les données en catégories, les patrons similaires sont rangés dans la même classe et représentés par un unique neurone, en se fondant sur les corrélations des données.

Le WTA simule les mécanismes de compétition existant entre neurones ou populations de neurones. Le modèle courant utilise des groupes de neurones formels dont l'apprentissage est fixé par la règle de Hebb. L'ajout de liaisons inhibitrices latérales permet de simuler le processus de compétition. Après convergence, seul le neurone ayant la plus grande activité reste actif et inhibe tous les autres.

6.4.2 LVQ

L'algorithme LVQ est un algorithme d'apprentissage très utilisé pour la compression de données, dans le cadre du traitement de la parole, du stockage d'images, de la transmission et de la modélisation. Il s'agit de représenter un ensemble ou une distribution de vecteurs à l'aide d'un nombre restreint de vecteurs prototypes ou d'un livre de codes. Une fois que le livre de codes a été construit et agréé par le transmetteur et le récepteur, il ne reste alors qu'à transmettre ou stocker l'index du vecteur prototype correspondant au vecteur de données. Étant donné un vecteur de données, son vecteur prototype peut être trouvé en cherchant le vecteur prototype le plus voisin dans le livre des codes.

6.4.3 Les ART

Les réseaux ART (Adaptative Resonnance Theorie) sont des réseaux à apprentissage par compétition. Le problème majeur qui se pose dans ce type de réseaux est le dilemme stabilité/plasticité. En effet, dans un apprentissage par compétition, rien ne garantit que les catégories formées vont rester stables. La seule possibilité, pour assurer la stabilité, serait que le coefficient d'apprentissage tende vers zéro, mais le réseau perdrait alors sa plasticité. Les ART ont été conçus spécifiquement pour contourner ce problème. Dans ce genre de réseau, les vecteurs de poids ne seront adaptés que si l'entrée fournie est suffisamment proche, d'un prototype déjà connu par le réseau. On parlera alors de résonance. A l'inverse, si l'entrée s'éloigne trop des prototypes existants, une nouvelle catégorie va alors se créer, avec pour prototype, l'entrée qui a engendré sa création. Il est à noter qu'il existe deux principaux

types de réseaux ART : les ART-1 pour des entrées binaires et les ART-2 pour des entrées continues. Le mode d'apprentissage des ART peut être supervisé ou non.

6.4.4 Réseau à fonction radiale

Les réseaux à fonction radiale (RBF) qui possèdent deux couches forment une classe particulière de réseaux multi-couches. Chaque cellule de la couche cachée utilise une fonction noyau telle que la Gaussienne en tant que fonction d'activation. Cette fonction est centrée au point spécifié par le vecteur de poids associé à la cellule. La position et la largeur de ces courbes sont apprises à partir des patrons. Il y a, en général, beaucoup moins de fonctions noyaux dans un réseau RBF que de patrons d'entrée. Chaque cellule de sortie implémente une combinaison linéaire de ces fonctions, l'idée étant d'approximer une fonction par un ensemble de fonctions. De ce point de vue, les cellules cachées fournissent un ensemble de fonctions qui forment une base représentant les patrons d'entrées dans l'espace couvert par les cellules cachées.

Chapitre 7

Forward Algorithms

7.1 Algorithmes forward standards de recherche

7.1.1 Recherche en largeur d'abord (Breadth First)

L'algorithme de recherche en largeur d'abord étudie tous les états à une profondeur donnée avant d'étudier les états qui sont à un niveau plus profond dans l'arbre de recherche. Cet algorithme utilise généralement une file initialisée avec un élément : l'état initial. L'état est retiré en avant de la file et on regarde s'il s'agit du but recherché. Si c'est le cas, la recherche se termine, sinon l'état est développé et les états résultants sont ajoutés à la file.

Algorithm 17 Algorithme de recherche en largeur d'abord

```
1:  Q ← Etat initial
2:  repeat
3:      if Q est vide then
4:          Retourner erreur
5:      else
6:          C ← Retirer(Q)
7:          if C est un but  then
8:              Retourne C
9:          else
10:             for chaque N ← Successeur(C) do
11:                 Ajouter(Q, N)
12:             end for
13:         end if
14:     end if
15: until forever
```

7.1.2 Recherche en profondeur d'abord (Depth First)

L'algorithme de recherche en profondeur d'abord traverse l'espace de recherche en développant d'abord l'état le plus profond dans l'arbre de recherche. L'algorithme de base utilise une pile qui est initialisée avec une seule valeur : l'état initial. Il se termine lorsque le but recherché est trouvé. En développant un état, les noeuds successeurs sont empilés.

Algorithm 18 Algorithme de recherche en profondeur d'abord

```
1: Empiler(S, Etat Initial)
2: repeat
3:     if S est vide then
4:         Retourner erreur
5:     else
6:         C ← Depiler(S)
7:         if C est un but then
8:             Retourne C
9:         else
10:            for chaque N ← Successeur(C) do
11:                Empiler(Q, N)
12:            end for
13:        end if
14:    end if
15: until forever
```

7.1.3 Recherche limitée en profondeur d'abord (Depth First)

L'algorithme de recherche limitée en profondeur est identique à l'algorithme précédent, à la différence qu'une limite de profondeur est imposée sur les états à étudier. Généralement cette recherche est utilisée lorsque l'on sait que le but est à une certaine distance de l'état initial, ou lorsqu'un but trop éloigné n'a pas d'intérêt.

Algorithm 19 Algorithme de recherche limité en profondeur

```
1: Empiler(S, Etat Initial)
2: repeat
3:     if S est vide then
4:         Retourner erreur
5:     else
6:         C ← Depiler(S)
7:         if C est un but then
8:             Retourne C
9:         else
10:            if Profondeur(C) < Limite then
11:                for chaque N ← Successeur(C) do
12:                    Empiler(Q, N)
13:                end for
14:            end if
15:        end if
16:    end if
17: until forever
```

7.1.4 Algorithme de Dijkstra

L'algorithme de Dijkstra résout un problème du plus court chemin pour un graphe G(V,E) orienté et connexe dont les poids liés aux arcs sont positifs (≥ 0).

Le coût du chemin entre deux noeuds est la somme des coûts des arcs du chemin. Le coût d'un arc peut être vu comme une généralisation de la distance entre ces deux noeuds. Pour une paire donnée de noeuds s,t dans l'ensemble des noeuds du graphe, l'algorithme trouve le chemin depuis s vers t de moindre coût (c'est à dire le plus court chemin). L'algorithme fonctionne en construisant

un sous-graphe S tel que la distance entre un noeud v' (dans S) depuis s est connue pour être un minimum dans G. Initialement S contient simplement le noeud s isolé, et la distance de s à lui-même vaut zéro. Des arcs sont ajoutés à S à chaque étape :

– en identifiant tous les arcs ei = (vi1,vi2) dans G-S tel que vi1 est dans S et vi2 est dans G.
– puis en choisissant les arcs ej = (vj1,vj2) dans G-S qui donne la distance minimum de ce noeud vj2 (dans G) depuis s depuis tous les arcs ei. L'algorithme se termine soit quand S devient un arbre couvrant de G, soit quand tous les noeuds d'intérêt sont dans S. La procédure pour ajouter un arc ej à S conserve la propriété suivante : les distances de tous les noeuds dans S depuis s sont des minimums connus.

Algorithm 20 Algorithme de Dijkstra

```
1: InitialiserSourceSimple(G, s)
2: S ← ensemble vide
3: Q ← ensemble de tous les noeuds
4: while Q n'est pas un ensemble vide do
5:     u ← ExtraireMinimum(Q)
6:     S ← S ⋃ u
7:     for chaque noeud v voisin de u do
8:         Relax(u, v, w)
9:     end for
10: end while
```

L'algorithme de Dijkstra peut-être mis en oeuvre efficacement en stockant le graphe sous forme de listes adjacentes et en utilisant une pile comme une file à priorités pour réaliser la fonction Extract-Min.

7.1.5 A* (A-Star)

L'algorithme A* est un des algorithmes les plus utilisés dans la programmation de jeux. Il reprend l'algorithme de Disjktra mais en ajoutant une analyse d'orientation de la recherche. Au lieu de placer les points dans la file en fonction de leur vrai poids, ils sont placés en fonction de leur poids plus une estimation de la distance pour atteindre le point de destination.

La formule utilisée est la suivante : f(n) = g(n) + h(n) où :
– f(n) est le score du point (c'est lui qui va déterminer sa position dans la file)
– g(n) est le poids du point
– (n) est une estimation du coût pour atteindre le point de destination.

Ceci permet à l'algorithme de se concentrer sur les points qui ont le plus de chances d'aboutir.

7.1.6 Recherche du meilleur d'abord (Best First)

L'algorithme de recherche du meilleur d'abord utilise une fonction d'évaluation et choisi toujours l'état qui a obtenu le meilleur score. Pourtant le parcours des états de l'arbre de recherche est exhaustif et l'algorithme pourra potentiellement étudier tous les cas possibles. Il utilise un agenda comme dans les recherches en largeur/profondeur d'abord, mais au lieu d'enlever le premier noeud et de générer ses successeurs, il va enlever le meilleur noeud, c'est à dire celui qui aura le meilleur score. Les successeurs de ce noeud seront évalués et ajoutés à la liste.

7.1.7 Profondeur itératif (Iterative Deepening)

La recherche en profondeur itérative combine les avantages de la recherche en profondeur d'abord et ceux de la recherche en largeur d'abord. Cet algorithme utilise la même quantité de mémoire que celui de la recherche en profondeur pour les même entrées, est complet et optimisé sous certaines

conditions.

La recherche commence par une recherche en profondeur limitée, et si l'objectif n'est pas trouvé, on incrémente cette limite (cette incrémentation pouvant être supérieur à 1). On boucle ensuite jusqu'à ce qu'on trouve le but.

7.2 Algorithmes forward dérivés du backtracking

7.2.1 Le Forward Checking

L'algorithme du forward checking a pour but de répondre au même problématique que les algorithmes de backtracking (cf. section backward). Contrairement a ces derniers dont il est dérivé, il exécute les tests de consistance à la descente. Alors que les algorithmes de backtrack effectuent les vérifications de contraintes entre la variable courante et les variables passées, le forward checking effectue les tests de consistances entre la variable courante et les variables restantes qui ne sont pas encore instanciées. A chaque niveau dans l'arbre de recherche, le domaine des futures variables est filtré afin que les variables inconsistantes avec l'instantiation courante soient retirées. Le forward checking est très efficace grâce à sa capacité à déceler les inconstitances très tôt dans la recherche. Cela dit, il est possible que ces tests soient plus nombreux que ceux des algorithmes de backtracking.

Algorithm 21 Algorithme du Forward Checking

```
1:  procedure ForwardChecking(entier u, echec)
2:      if courant > N then
3:          solution()
4:          retourne (N)
5:      end if
6:      for i ← 0 à K do
7:          if domaine[courant][i] then
8:              continue
9:          end if
10:         v[courant] ← i
11:         echec ← consistent(courant)
12:         if fail = 0 then
13:             ForwardChecking(courant + 1)
14:         end if
15:         Restaurer(current)
16:     end for
17:     Retourne(current - 1)
18: end procedure
```

7.2.2 Algorithmes hybrides du Forward Checking (FC-BJ et FC-CBJ)

Le Forward Checking and Backjumping (FC-BJ) et le forward checking and Conflict-Directed Backjumping (FC-CBJ) intègrent le Backjumping au sein de l'algorithme de Forward Checking.

Contrairement au forward checking qui retourne en arrière de façon chronologique, les dérivés du forward checking sauvegarde l'information sur les variables qui ont causé une inconsistance. Ensuite, l'information est utilisée pour déterminer à quel point on effectue le retour-arrière.

De plus des structures héritées du forward checking, ces algorithmes hybrides utilisent les structures de données des algorithmes de backward checking. FC-BJ utilise le vecteur max_test de BJ, tandis que FC-CBJ utilise le conf_set de CBJ. Les fonctions consistent et restore sont identiques à celles

du forward checking.

Les dérivés du forward checking essayent de combiner les avantages du forward checking et du backjumping. Cependant l'algorithme résultant et complexe et plus difficile à mettre en oeuvre.

Chapitre 8

Backward Algorithms

8.1 Simple Backtracking (BT)

Le simple retour arrière, est le plus simple algorithme backward. Plutôt que d'explorer de façon exhaustive l'arbre des affectations possibles, cette méthode consiste à tester à chaque assignation un certain nombre de contraintes. Si ces contraintes sont satisfaites, on continue l'exploration et dans le cas contraire, on élague de l'arbre de recherche le noeud courant et ceux qui sont en dessous de lui puisqu'ils ne correspondent pas aux critères, enfin on retourne en arrière jusqu'à la prochaine conformation cohérente. L'algorithme est présenté dans la section 4.2.3 sur la programmation par contrainte.

8.2 Backjumping (backtracking intelligent)

La différence entre le backtracking et le backjumping (BJ) est que le simple retour arrière effectue son retour à la variable la plus récemment instanciée. Au contraire, dans l'algorithme du backjumping, on retourne à la plus haute variable en conflit avec la variable courante. On utilise pour cela un vecteur max_test[i] qui sauvegarde la dernière variable vérifiée à l'instantiation courante de Xi. Il suffit alors de sauter jusqu'à la variable max_test[current].

Algorithm 22 Algorithme du Backjumping

```
1:  procedure BACKJUMPING(entier i, saut)
2:      if courant > N then
3:          solution()
4:          retourne (N)
5:      end if
6:      max_test[courant] ← 0
7:      for i ← 0 à K do
8:          v[courant] ← i
9:          if consistent(courant) then
10:             saut ← Backjumping(courant + 1)
11:             if saut ≠ courant then
12:                 retourne saut
13:             end if
14:         end if
15:     end for
16:     Retourne(max_test[courant])
17: end procedure
```

8.3 Conflict-Directed Backjumping (CBJ)

L'algorithme de Conflict-Directed Backjumping est une extension du backjumping vu dans la section précédente. Il utilise l'information sur les conflits entre l'instanciation courante et les futures variables. Chaque variable a son propre ensemble de conflits contenant les variables passées et échouant aux tests de l'instantiation courante. Ainsi, le Conflict-Directed Backjumping effectue un saut à la plus haute variable de l'ensemble de conflits. Il est également possible d'effectuer plusieurs sauts, afin qu'après le premier, il soit possible de continuer un backjumping des conflits. Cela permet potentiellement une réduction significative du parcours.

Algorithm 23 Algorithme du Conflict-Directed Backjumping

```
1:  procedure CBJ(entier h, i, saut)
2:      if courant > N then
3:          solution()
4:          retourne(N)
5:      end if
6:      vider(conf_set[courant])
7:      max_test[courant] ← 0
8:      for i ← 0 à K do
9:          v[courant] ← i
10:         if consistent(courant) then
11:             saut ← CBJ(courant + 1)
12:             if saut ≠ courant then
13:                 retourne saut
14:             end if
15:         end if
16:     end for
17:     h ← max(conf_set[current])
18:     Fusionne(conf_set[h], conf_set[current])
19:     Retourne(h)
20: end procedure
```

8.4 Graph-Based Backjumping (GBJ)

Comme, le précédent algorithme, le Graph-Based Backjumping est une extension du backjumping, et il utilise la connaissance sur le graphe de contrainte. L'algorithme retourne en arrière jusqu'à la plus haute variable connectée au noeud courant. Cela signifie que le saut s'effectue jusqu'à la plus haute variable qui est connectée par une contrainte non triviale.

Par contre, cet algorithme n'est utile que dans les cas où le graphe est relativement dispersé. Si le graphe est presque complet il sera préférable d'implémenter un simple backtracking.

Algorithm 24 Algorithme du Graph-Based Backjumping

```
1:  procedure GBJ(entier h, i, saut)
2:      if courant > N then
3:          solution()
4:          retourne(N)
5:      end if
6:      vider(conf_set[courant])
7:      max_test[courant] ← 0
8:      for i ← 0 à K do
9:          v[courant] ← i
10:         if consistent(courant) then
11:             saut ← GBJ(courant + 1)
12:             if saut ≠ courant then
13:                 retourne saut
14:             end if
15:         end if
16:     end for
17:     Fusionne(P, parents(current))
18:     Supprimer(h, P)
19:     Retourne(h)
20: end procedure
```

8.5 Backmarking

L'objectif de l'algorithme du backmarking est de réduire le nombre de test de consistance. Il parait intuitif qu'il y ait beaucoup de redondance dans les tests effectués par l'algorithme du backtracking, et un grand nombre peuvent être éliminés. Les combinaisons de marquage sont basées sur les observations suivantes :

– Si à un noeud récent, une instanciation donnée a été vérifiée et que cette instantiation a échoué à cause d'un conflit avec une variable précédente n'ayant pas changé, alors il échouera systématiquement par la suite. Par conséquent, tous ces tests de consistance peuvent être évités

– De même si à un noeud récent, une instanciation a été testée avec succès dans ces même conditions, alors il n'est pas nécessaire de vérifier l'instanciation qu'avec les instantiations les plus récentes qui ont changé.

Algorithm 25 Algorithme du Backmarking

```
1:  procedure BM(entier h, i)
2:      if courant > N then
3:          solution()
4:          retourne(N)
5:      end if
6:      vider(conf_set[courant])
7:      max_test[courant] ← 0
8:      for i ← 0 à K do
9:          v[courant] ← i
10:         if consistent(courant) then
11:             BM(courant + 1)
12:         end if
13:     end forState h ← courant-1 State mbl[courant] ← h
14:     for i ← h+1 à N do
15:         mbl[i] ← min([i], h)
16:     end for
17:     Retourne(h)
18: end procedure
```

8.6 Algorithmes hybrides du Backmarking (BM-BJ, BM-CBJ, BM-GBJ, BMJ2, BM-CBJ...)

Il existe deux algorithmes principaux utilisant le système du backjumping dans l'algorithme du backmarking. Ce sont les algorithmes Backmarking and Backjumping (BM-BJ ou BMJ) et Backmarking and Conflict-Directed Backjumping (BM-CBJ). Ces algorithmes sont similaires à celui du backmarking mais ils intègrent l'information du marquage pour décider qu'elle est la meilleure variable à laquelle il faut effectuer un saut.

D'autres algorithmes hybrides existent par exemple BM-GBJ qui combine le Backmarking et le Graph-Based Backjumping. Nous ne le verrons pas en détail ici, cet algorithme étant relativement complexe sans être pour autant plus efficace que ceux étudié précédemment.

L' efficacité de ces algorithmes hybrides est d'ailleurs souvent relativement problématique. L'algorithme de Backmarking and Backjumping par exemple n'atteint pas la performance de chacun des algorithmes de base en terme de tests de consistance. Ainsi pour résoudre ce problème, un Backmarking and Backjumping modifié a été mis au point (BMJ2). Cet algorithme résout ce problème en utilisant un vecteur de deux dimensions plutôt qu'une dimension. Le nouveau vecteur de taille n x m, où n représente le nombre de variable et m la taille du plus large domaine. La taille mémoire est raisonnable puisque le BMJ utilise déjà un vecteur n x m.

Une modification analogue a été effectuée sur le BM-CBJ, ce qui a permis de produire BM-CBJ2, utilisant également un vecteur de dimension 2.

Chapitre 9

Algorithmes d'élagage

Élaguer : *Dépouiller un arbre des branches inutiles, retrancher les parties inutiles de.*

Nous avons déjà vu dans le corps de ce rapport un grand nombre de techniques d'élagage, qu'il s'agisse d'heuristiques ou d'algorithmes, nous invitons le lecteur à reconsulter notamment le chapitre sur la programmation par contraintes qui est un bon exemple de techniques d'élagage en domaines finis. Nous allons présenter ici quelques algorithmes classiques supplémentaires.

9.1 Algorithmes de la théorie des jeux

Introduction

Les jeux à deux joueurs font partie des applications classiques en programmation symbolique. C'est un bon exemple de résolution de problèmes pour au moins deux raisons :

– le nombre de solutions à analyser pour obtenir le meilleur coup nécessite des méthodes autres que la force brute : aux échecs le nombre de coups possibles est en moyenne de 30 et une partie se joue en une quarantaine de coups pour chaque joueur ; ce qui donne quelques 30^{80} positions pour explorer l'arbre complet d'une partie !

– la qualité d'une solution est facilement vérifiable, en particulier il est possible de tester la qualité de la solution proposée par un programme en utilisant un autre programme.

Supposons que l'on puisse utiliser une exploration totale de toutes les parties possibles à partir d'une position légale du jeu. Un tel programme a besoin d'une fonction de génération des coups légaux à partir de cette position et d'une fonction d'évaluation donnant un score à la position donnée. La fonction d'évaluation donne un score maximal à une position gagnante et un score minimal à une position perdante. À partir de la position initiale, on peut donc construire l'arbre de toutes les variantes de la partie, où chaque noeud correspond à une position, ses fils aux positions suivantes obtenues en ayant joué un coup et les feuilles aux positions gagnantes, perdantes ou nulles. Une fois cet arbre construit, son exploration permet de déterminer s'il existe un chemin menant à la victoire, ou à une position nulle le cas échéant. Le chemin de plus petite longueur peut alors être choisi pour amener au résultat voulu.

Comme la taille d'un tel arbre est en règle générale trop grande pour être envisageable, il est nécessaire d'en limiter sa construction. La première possibilité est de limiter la profondeur de recherche, c'est-à-dire le nombre de coups et de réponses à évaluer. On réduit ainsi la profondeur de l'arbre donc sa taille. Dans ce cas on atteint rarement des feuilles à moins d'être en fin de partie.

D'autre part, nous pouvons essayer de limiter le nombre de coups engendrés pour pouvoir les évaluer plus finement. Pour cela, nous tentons de n'engendrer que les coups semblant les plus favorables et de les examiner en commençant par les meilleurs. Cela permet ainsi d'élaguer rapidement des branches entières de l'arbre.

Minimax

L'algorithme minimax, dû à Von Neumann, est un algorithme de recherche en profondeur, avec une profondeur limitée. Il nécessite d'utiliser :

- une fonction de génération des coups légaux à partir d'une position
- une fonction d'évaluation d'une position de jeu

À partir d'une position du jeu, l'algorithme explore l'arbre de tous les coups légaux jusqu'à la profondeur demandée. Les scores des feuilles de l'arbre sont alors calculés par la fonction d'évaluation. Un score positif indique une bonne position pour le joueur A, un score négatif une mauvaise position pour le joueur A donc une bonne position pour le joueur B. Selon le joueur qui joue, le passage d'une position à une autre est maximisante (pour le joueur A) ou minimisante (pour le joueur B). Les joueurs essaient de jouer les coups les plus profitables pour eux-mêmes. En cherchant le meilleur coup pour le joueur A, la recherche en profondeur 1 cherchera à déterminer le coup immédiat qui maximise le score de la nouvelle position.

L'exploration en profondeur 1 est en règle générale insuffisante, car on ne tient pas compte de la réponse de l'adversaire. Cela produit des programmes cherchant le gain immédiat de matériel (comme la prise d'une reine aux échecs), sans s'apercevoir que les pièces sont protégées ou que la position devient perdante (gambit de la reine pour faire mat). Une exploration de profondeur 2 permet de s'apercevoir du contrecoup.

Dans la plupart des jeux, il est possible de faire lanterner son adversaire, en le faisant jouer à coups forcés, dans le but d'embrouiller la situation en espérant qu'il commette une faute. Pour cela la recherche de profondeur 2 est très insuffisante pour le côté tactique du jeu. Le côté stratégique est rarement bien exploité par un programme car il n'a pas la vision de la probable évolution de la position en fin de partie. La difficulté de profondeur plus grande provient de l'explosion combinatoire. Par exemple aux échecs, l'exploration de 2 profondeurs supplémentaires apporte un facteur d'environ mille fois plus de combinaisons (30 * 30). Donc si on cherche à calculer une profondeur de 10, on obtiendra environ 5^{14} positions, ce qui est bien entendu trop. Pour cela on essaie d'élaguer l'arbre de recherche.

La fonction d'évalutation d'une position du jeux est une heuristique qu'il convient bien sûr d'adapter à chaque type de jeux. L'algorithme donné ci-après repose donc sur une exploration partielle de l'espace des coups possibles. Cette exploration se fait à une profondeur donnée. Évidemment plus la profondeur est importante, plus le coup sera bon et plus long sera le calcul. On associe généralement à un noeud de fin de partie une valeur positive pour une victore de A, 0 pour un match nul et une valeur négative pour une victoire de B. Pour les noeuds terminaux qui ne sont pas des noeuds de fin de partie, on fait intervenir notre fonction d'évaluation qui doit être soigneusement choisie pour refléter au mieux les caractéristiques du jeux. La construction de minimax se fait donc de façon ascendante en remontant à partir des feuilles. Cela repose sur l'idée que l'adversaire joue également de façon à maximiser sa propre fonction d'évaluation et donc à minimiser celle de l'adversaire.

Algorithm 26 Minimax

```
1:  function MINIMAX(Configuration S)
2:      if S est une feuille then
3:          return evaluation(S)
4:      end if
5:      if S est un noeud MAX then
6:          return Max(Minimax(S'))                    ▷ S' les fils de S
7:      end if
8:      if S est un noeud MIN then
9:          return Min(Minimax(S'))                    ▷ S' les fils de S
10:     end if
11: end function
```

Negmax

Il s'agit d'une variante de Minimax qui a été développé par Donald Knuth. Le principe de cet algorithme consiste à éviter de traiter différement les noeuds MIN et MAX. L'idée en est très simple et s'exprime ainsi :

Algorithm 27 Negmax

```
1: function NEGMAX(Configuration S)
2:     if S est une feuille then
3:         return evaluation(S)
4:     else
5:         return Max(-Negmax(S1), ..., -Negmax(Sn))   ▷ S1 ... Sn les fils de S
6:     end if
7: end function
```

AlphaBeta

L'algorithme minimax effectue une exploration complète de l'arbre de recherche jusqu'à un niveau donné, alors qu'une exploration partielle de l'arbre pourrait suffire. Il suffit en effet, dans l'exploration en profondeur d'abord et de gauche à droite, d'éviter d'examiner des sous-arbres qui conduiront à des configurations dont la valeur ne contribuera sûrement pas au calcul du gain à la racine de l'arbre. L'algorithme alpha-beta est donc une optimisation de MiniMax, qui coupe des sous-arbres dès que leur valeur devient inintéressante aux fins du calcul de la valeur MiniMax du jeu. On s'intéressera donc, sur chaque noeud, en plus de la valeur, à deux autres quantités, nommées alpha et beta, qui seront utilisées pour calculer la valeur du noeud.

alpha d'un noeud :
C'est une approximation par le bas de la vraie valeur du noeud. Elle est égale à la valeur sur les feuilles, et est initialisée à -infini ailleurs. Ensuite, sur les noeuds joueur elle est maintenue égale à la plus grande valeur obtenue sur les fils visités jusque là, et elle est égale à la valeur alpha de son prédécesseur sur les noeuds opposant.

beta d'un noeud :
C'est une approximation par le haut de la vraie valeur du noeud. Elle est égale à la valeur sur les feuilles, et est initialisée à +infini ailleurs. Ensuite, sur les noeuds opposant elle est maintenue égale à la plus petite valeur obtenue sur les fils visités jusque là, et elle est égale à la valeur beta de son prédécesseur sur les noeuds joueur.

L'algorithme AlphaBeta peut être décrit par le pseudo-code suivant :

Algorithm 28 AlphaBeta

```
Require: ici A est toujours inférieur à B
function AlphaBeta(P, A, B)
    if P est une feuille then
        return evaluation(P)
    else
        Initialiser Alpha de P à -Infini et Beta de P à +Infini
        if P est un noeud Min then
            for Tous les fils Pi de P do
                Val = AlphaBeta(Pi, A, Min(B, Beta de P))
                Beta de P = Min(Beta de P, Val)
                if A ≥ Beta de P then                    ▷ Ceci est la coupure alpha
                    return Beta de P
                end if
            end for
            return Beta de P
        else
            for Tous les fils Pi de P do
                Val = AlphaBeta(Pi, Max(A, Alpha de P), B)
                Alpha de P = Max(Alpha de P, Val)
                if Alpha de P ≥ B then                   ▷ Ceci est la coupure beta
                    return Alpha de P
                end if
                return Alpha de P
            end for
        end if
    end if
end function
```

On sait que la véritable valeur MiniMax v d'un noeud est encadrée par alpha et beta (i.e. alpha $\leq v \leq$beta), et si on appelle la fonction AlphaBeta avec les valeurs (P,-infini,+infini) on obtient précisément Minimax(P). AlphaBeta permets assez souvent de doubler la profondeur d'exploration d'un arbre à parité de ressources, par rapport à Minimax. Contrairement à minimax, le calcul des valeurs de AlphaBeta se fait de façon à la fois ascendante et descendante.

SSS*

Il s'agit d'un algorithme relativement peu connu, en tous cas nettement moins connu que l'algorithme AlphaBeta. Il a pourtant été démontré qu'il lui est théoriquement supérieur, dans le sens où il n'évaluera pas un noeud si AlphaBeta ne l'examine pas, tout en élaguant éventuellement quelques branches explorées par AlphaBeta. Cette qualité supplémentaire se paie, SSS étant un gros consommateur de mémoire.

Définissons rapidement une stratégie et une stratégie partielle. Étant donné un arbre de jeu J, on appelle stratégie pour le joueur Max, un sous-arbre de J qui contient la racine de J, dont chaque noeud Max a exactement un fils, dont chaque noeud Min a tous ses fils. Étant donné un arbre de jeu J, on appelle stratégie partielle pour le joueur Max, un sous-arbre de J qui contient la racine de J, dont chaque noeud Max a au plus un fils.

Une stratégie indique au joueur Max ce qu'il doit jouer dans tous les cas. Si Max respecte une

stratégie, il est assuré d'aboutir à une des feuilles de stratégie. La valeur d'une stratégie pour Max est le minimum des valeurs des feuilles de cette stratégie, gain assuré contre toute défense de Min. Le but de SSS* est d'exhiber la stratégie de valeur maximum pour Max.

L'algorithme SSS* explore un espace d'état dont chaque noeud est une stratégie partielle, en utilisant une approche meilleur d'abord avec une heuristique minorante qui sera la valeur des stratégies partielles et qui garantit l'optimalité. À partir du moment où une sous-stratégie optimale est établie, la stratégie optimale est marquée, et toutes les sous-stratégies sous-optimales supprimées.

On dira qu'un noeud est résolu, si la stratégie complète issue de ce noeud a été déterminée. Un noeud qui n'est pas résolu est vivant. Un état sera donc un triplet (noeud, type, valeur). La liste G des états générés non développés sera triée par odre décroissant des valeurs. Nous ne donnons pas le pseudo-code ici mais il se trouve dans [23].

SCOUT

Nous décrirons rapidement l'algorithme de Scout, élaboré par J. Pearl comme outil théorique. Son efficacité est, en général, inférieure à celle d'AlphaBeta, pour une consommation mémoire du même ordre. Il peut toutefois lui être supérieur. Scout repose sur idée fort simple : si l'on disposait d'un moyen efficace pour comparer (sans nécessairement la déterminer) la valeur minimax d'un noeud à une valeur donnée, une quantité importante de recherche pourrait être évitée. Considérons par exemple un noeud Max n, ayant deux fils : f_1, dont la valeur v est connue, et f_2. Si l'on sait que la valeur de f_2 est inférieure à v, il est inutile d'explorer la branche de f_2.

Scout s'appuie donc sur deux procédures simples : la première, appelée *Test* permet de vérifier si la valeur d'un noeud n est strictement supérieure (ou supérieure ou égale) à une valeur donnée v. Nous désignerons par h la fonction heuristique.

La seconde, *Eval*, utilise *Test* et applique le principe donné plus haut pour calculer la valeur minimax d'un arbre de jeu. Elle prend en paramètre un noeud n. Il pourrait sembler que, du fait de la redondance éventuelle des évaluations, lorsque *Test* ne permet pas la coupure, Scout devrait être très inférieur à AlphaBeta, voire même à minimax. Une étude mathématique du comportement asymptotique de Scout montre un comportement identique à AlphaBeta pour des profondeurs élevées. Le pseudo-code se trouve également dans [23].

9.2 A*

A* est le nom d'un algorithme générique utilisé dans le cadre des problèmes combinatoires et d'ordonnancement. Il s'agit d'une extension du célèbre algorithme de Djikstra. A la base, l algorithme A* était utilisé pour résoudre les problèmes de puzzle (résolution à base de matrices). Créé en 1968, il a depuis évolué en de nombreuses variantes, et est maintenant utilisé aussi bien pour résoudre des labyrinthes complexes, le cheminement d un robot dans un terrain inconnu, les jeux sur échiquiers (échecs, dames, jeu de go), les jeux de carte (FreeCell), que pour résoudre les problèmes de PathFinding dans les jeux 2D et 3D.

Il reprend l'algorithme de Dijkstra mais en ajoutant une analyse d'orientation de la recherche. Au lieu de placer les points dans la file en fonction de leur vrai poids, ils sont placés en fonction de leur poids plus une estimation de la distance pour atteindre le point de destination suivant la formule f(n) = g(n) + a.h(n). Où f(n) est le score du point (c'est lui qui va déterminer sa position dans la file), g(n) est le poids du point, h(n) est une estimation du coût pour atteindre le point de destination, et a une constante donnant l'importance de h(n). Cela permet à l'algorithme de se concentrer sur les points qui ont le plus de chance d'aboutir. La recherche se fait vers le point de destination tout en conservant une approche optimale. Pour h(n), plusieurs méthodes existent comme celle de calculer la

distance exacte ou d'utiliser la distance de Manhattan. C'est une fonction f(n) bien choisie qui permet à l'élagage d'être performant.

Globalement , il est certain que pour être sur d'obtenir le meilleur chemin, la méthode de l'A* est nettement la meilleure. Après, il existe plusieurs façons de l'implémenter en variant la formule d'évaluation, la structure contenant les noeuds explorés (pile de priorité, arbre binaire...).

9.3 Programmation linéaire

Nous ne nous étendrons pas sur le sujet mais en quelques mots disons que la forme classique d'un problème de programmation linéaire est la suivante :

- maximiser une forme linéaire de n variables $x_1 ... x_n$
- les variables sont soumises à m contraintes linéaires de la forme $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$; i = 1, ..., m
- les variables sont soumises à n contraintes de non négativité : $x_j \geq 0$

Le problème peut également être résolu dans le cas où il faut minimiser la forme linéaire ou que les variables sont contraintes à être non positives. D'autres problèmes peuvent être mis sous cette forme standard.

Les n-uplets qui satisfont les contraintes s'appellent solutions réalisables du problème. Ce sont les coordonnées des points intérieurs au polyèdre des contraintes.

9.3.1 L'algorithme du Simplexe

L'idée de l'algorithme du Simplexe est de passer itérativement d'un sommet du polyèdre des contraintes à un sommet adjacent de façon à augmenter la valeur de la fonction à optimiser jusqu'à trouver un sommet où le maximum est atteint. Il s'agit ici d'une forme d'élagage de domaine. Son fonctionnement est assez proche d'une élimination de Gauss applicable à des inégalités.

Bibliographie

[1] *Iterative Deepening Search*, 2004. url : http://ai.squeakydolphin.com/wiki.php?pagename=AIAWiki.IterativeDeepeningSearch.

[2] *MutiAgent Systems*, 2004. url : http://www.multiagent.com/.

[3] *UMBC AgentWeb*, 2004. url : http://www.csee.umbc.edu/aw/.

[4] Richard Baron. *Réseaux de Neurones Artificiels*, 1997. url : http://www.univ-st-etienne.fr/creuset/personnel/baron/.

[5] Yoshua Bengio. *Algorithmes d'apprentissage*, 2000. url : http://www.iro.umontreal.ca/~bengioy/ift6266/.

[6] Pierre Berlandier. Etude et réalisation d'un ensemble de primitives pour la satisfaction de contraintes en domaines finis, Janvier 1991.

[7] Olivier Boissier. *Cours DEA - Systèmes Multi-Agent*, 2000. url : http://www.emse.fr/~boissier/enseignement/sma/.

[8] Jean Bénech. *Les agents*, 1999. url : http://jbenech.free.fr/old_site/agents/.

[9] Michèle Soria Christine Froidevaux, Marie-Claude Gaudel. *Types de données et Algorithmes*. McGraw-Hill, 1992.

[10] Daniel D. Corkill. *Blackboard Systems*, 1991. url : http://www.bbtech.com/papers/ai-expert.pdf.

[11] Leiserson Cormen. *Algorithme de Dijkstra*, 2002. url : http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra.

[12] Marc-Michel Corsini. *Réseaux de neurones artificiels : une introduction*, 1998. url : http://www.scico.u-bordeaux2.fr/~corsini/Pedagogie/ANN/main/main.html.

[13] Patrice Dargenton. *Configuration d'un réseau de neurones avec un méta-réseau de neurones*, 2001. url : http://patrice.dargenton.free.fr/ia/ialab/rnautoconfigurant.html.

[14] François Denis. *Systèmes experts*, 2002. url : http://www.grappa.univ-lille3.fr/polys/se/index.html.

[15] Robert B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University PA, January 1995.

[16] Françoise Fabret. *Optimisation du Calcul Incrémentiel dans les Langages de Règles pour Bases de Données*. PhD thesis, Thèse de Doctorat de l'Université de Versailles, 1994.

[17] Jacques Ferber. *Les systèmes multi-agents*. INTEREDITIONS, 1996.

[18] Emmanuel Fougeras. *Pathfinding : algorithme de parcours*, 2003. url : http://www.vieartificielle.com/article/index.php?action=article&id=178.

[19] S. Garlatti. Les systèmes à base de règles.

[20] GBBopen. *Annotated Blackboard-System Bibliography*, 2003. url : http://www.bbtech.com.

[21] Terry H. *Blackboard Technology*, 2002. url : http://www.pcai.com/web/ai_info/blackboard_technology.html.

[22] Michel Jaczinski. *Le raisonnement à partir de cas*, 2001. url : http://www-sop.inria.fr/axis/people/Michel.Jaczynski/rapc-fra.htm.

[23] Thomas Schiex Jean-Marc Alliot. *Intelligence Artificielle et Informatique Théorique*, 1993.

[24] Grzegorz Kondrak. *A Theoretical Evaluation of Selected Backtracking Algorithms*, 1994. url : `http://citeseer.nj.nec.com/21227.html`.

[25] Rachid LADJADJ. *Les réseaux de neurones*, 2003. url : `http://etudiant.univ-mlv.fr/~rladjadj/`.

[26] René Lalement. *Cours de programmation*, 1999. url : `http://cermics.enpc.fr/polys/info1/main/`.

[27] Luc Lamontagne Guy Lapalme. Raisonnement à base de cas textuels : état de l'art et perspectives. *Revue de l'intelligence artificielle*, X :1 à X, 2002.

[28] Steven M. LaValle. *Planning Algorithms*, 2004. url : `http://msl.cs.uiuc.edu/planning/`.

[29] Neil Madden. Optimising rete for low-memory, multi-agent systems, 2003.

[30] neural nets. *FAQ comp.ai.neural-nets*, 2002. url : `ftp://ftp.sas.com/pub/neural/FAQ.html`.

[31] Emmanuel CHAILLOUX Pascal MANOURY Bruno PAGANO. *Développement d'applications avec OCAML*, 2003. url : `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/book-ora161.html`.

[32] PMSI. *Réseaux de neurones : formation avancée*, 2002. url : `http://www.pmsi.fr/neurini2.htm`.

[33] A. Revel. *Ingénierie de la cognition*, 2001. url : `http://www.etis.ensea.fr/~revel/html/cours_IA/`.

[34] Brugger Rolf. *Multi-Agents Architecture*, 1996. url : `http://www-iiuf.unifr.ch/~brugger/papers/95_cidre/cidre/node26.html`.

[35] Richard Sinn. *Blackboard Technology*. url : `http://www.openloop.com/softwareEngineering/patterns/architecturePattern/arch_Blackboard.htm`.

[36] Christine Solnon. *Programmation par contraintes*, 2003. url : `http://www710.univ-lyon1.fr/~csolnon/Site-PPC/e-miage-ppc-som.htm`.

[37] Amrudee Sukpan. *A Survey on Constraint Satisfaction Problems*, 2002. url : `http://www2.cs.uregina.ca/~sukpan1a/csp/csp.htm`.

[38] Ronald L. Rivest Clifford Stein Thomas H. Cormen Charles E. Leiserson. *Introduction à l'algorithmique - 2eme édition*. Dunod, 2002.

[39] Marc Torrens. *Constraint Satisfaction Problems*, 1997. url : `http://liawww.epfl.ch/~torrens/Project/project/node8.html`.