LABORATOIRE
EISTI
LAπPI
EN PROCESSUS INTELLIGENTS

Maria MALEK

INTELLIGENCE ARTIFICIELLE

EISTI, 2006-07

1

RECHERCHE DANS UN ESPACE D'ÉTATS

1.1 Introduction

L'objectif de ce chapitre est d'introduire les principes de la résolution d'un problème donné en utilisant un espace de recherche. Cet espace de recherche permet de recenser les états d'un système donné et de trouver parmi ces états une ou plusieurs solutions. Le passage d'un état à un autre se fait par l'application d'une action donnée. Nous verrons rapidement que le problème de recherche de l'état solution dans l'espace nous ramènera à développer *un arbre de recherche* et à définir une stratégie de recherche sur cet arbre. Le but de la recherche dans un tel arbre serait la diminution du temps de recherche en trouvant une stratégie qui converge rapidement vers la solution. Nous allons dans un premier temps présenter les algorithmes de recherche

classique : en profondeur et largeur qu'on appellera la recherche aveugle. Ce seront les algorithmes les plus coûteux évidemment en temps ou en espace car ils ont tendance à développer l'arbre *complet* pour aboutir à une solution où à l'ensemble de solutions. Ensuite, nous étudions l'algorithme A* qui intègre une heuristique courant le développement de l'arbre de recherche ; cette heuristique permettra d'élaguer un ensemble de branches de l'arbre et de converger rapidement vers une solution si cette solution existe. Dans la suite, nous définissons les notions utilisées à savoir : *les états et les actions* . Ensuite, nous montrons les techniques de résolution en effectuant une recherche dans un espace d'états. L'application la plus directe à ces méthodes sera les problèmes de planification pour lesquels on ne connaît pas d'algorithme général ; mais on sait qu'étant donnés, un état initial, un état final et un ensemble d'actions élémentaires valides nous permettant de passer d'un état à un autre, on doit trouver une séquence d'action à executer pour résoudre le problème.

Il existe d'autres types d'application des méthodes de recherche dans un espace comme la démonstration de théorèmes, les jeux et la résolution de contraintes .

1.2 Notions utilisées

Nous modélisons dans cette section un problème général de planification. Un problème de planification est modélisé par le quadruplet (S, E_0, F, T) où :

S est l'ensemble de tous les états.

E_0 est l'état initial, $E_0 \in S$

F est l'ensemble des états finaux $F \subset S$

T est la fonction de transition qui associe à chaque état E_i dans S un ensemble de couples (A_{ij}, E_{ij}) tels que A_j soit une action élémentaire permettant de passer de l'état E_i à l'états E_{ij}.

1.2.1 Comment résoudre un problème de planification ?

Résoudre un problème de planification signifie trouver une séquence $E_0, E_1, .., E_j, ..E_n$ tel que $\exists A_1, ..A_j, ..A_n$ tels que $(A_j, E_j) \in T(E_{j-1})$ et $E_n \in F$. Pour effectuer une recherche, un arbre de recherche est construit ; la racine de l'arbre correspond à l'état initial, un état E_{ij} est fils d'un autre état E_i s'il existe une action qui permet d'obtenir E_{ij} à partir de E_i. Si une des

feuilles de l'arbre correspond à un état final, la solution est trouvée par la stratégie de la recherche.

1.3 Recherche aveugle

La recherche aveugle de la solution peut s'effectuer en profondeur ou en largeur, le parcours en profondeur signifie le développement entière d'une branche entière avant de parcourir le reste de l'arbre en effectuant du "backtracking". Ce développement peut ramener à trois situations différentes :

- La solution est trouvée et dans ce cas le développement de la branche s'arrête avec une possibilité de "backtraking" si d'autres solutions sont encore sollicitées.
- La solution n'est pas trouvée et un état d'échec est détecté (c'est un état qui n'engendre pas d'autres états) et dans ce cas le "backtraking" est appliqué pour poursuivre la recherche.
- Une branche infinie est à explorer et dans ce cas, un test d'arrêt sur une profondeur maximale doit être intégré dans l'algorithme.

Nous remarquons que les performances de la recherche en profondeur sont liées à l'ordre d'exploration des branches de l'arbre. Autrement dit, si la solution se trouve dans la première branche à explorer cette stratégie devient optimale.

Par contre la recherche en largeur signifie que les états doivent être visités en parcourant l'arbre niveau par niveau ; autrement dit, tous les successeurs d'un état donné sont visités l'un après l'autre avant le passage au niveau suivant. Pour effectuer le parcours en largeur, une file est utilisée. Le parcours s'arrête quand un état final est trouvé ou quand une profondeur maximale est atteinte. Ce parcours est très cher en temps et en espace mais il garantit de trouver la solution si elle existe ; tandis que le parcours en profondeur peut ne pas converger même si la solution existe à cause d'un branche infinie.

```
FONCTION ParcoursProf (E_i, DejaVu, N) : BOOLEEN
VAR res : BOOLEEN
  SI E_i ∈ F ALORS
    res ← VRAI
  SINON
    SI N = 0 ALORS
      res ← FAUX
    SINON
      POUR TOUT (A_j, E_j) ∈ T(E_i) ET NON (E_j ∈ DejaVu) FAIRE
        SI ParcoursProf(E_j, DejaVu ∪ E_j, N − 1) = VRAI ALORS
          AfficherA_j, E_j
          res ← VRAI
        FIN SI
      FIN POUR
    FIN SI
  FIN SI
RETOURNER res
```

Algorithme 1: Recherche en profondeur

Remarquer bien que le premier appel à cette fonction sera : $ParcoursProf(E_0, [E_0], d)$ ou d est la profondeur maximale tolérée.

FONCTION ParcoursLarg(E_0) : Liste

VAR F : FILE

$F \leftarrow fileVide$

$L \leftarrow listeVide$

$Ajouter(F, E_0)$

TANTQUE NON vide(F) **FAIRE**

inserer(L,premier(F))

SI NON $premier(F) \in F$ **ALORS**

POUR TOUT $(A_j, E_j) \in T(premier(F))$ **FAIRE**

$ajouter(F, E_j)$

FIN POUR

supprimer(F)

SINON

$F \leftarrow fileVide$

FIN SI

FIN TANTQUE

Algorithme 2: Recherche en largeur

1.4 Les différents problèmes de planification

Nous pouvons distinguer trois types de problèmes de planification :

(1) Le problème de la recherche d'une *solution quelconque*. On peut utiliser les algorithmes de recherche en profondeur d'abord pour effectuer ce type de recherche.

(2) Le problème de la recherche de *toutes les solutions*. Nous pouvons dans ce cas-là construire l'arbre en largeur d'abord en introduisant une stratégie qui n'explore pas les branches ne menant pas à une bonne solution (à condition de les détecter le plus rapidement possible).

(3) Le problème de la recherche de la *meilleure solution* selon un critère donné. Souvent ce critère est formalisé par un coût qui sera associé à l'ensemble des *actions* formalisant le problème. Nous pouvons égale-

ment explorer l'arbre en largeur avec une stratégie concernant la non exploration de certaines branches.

1.5 Introduction d'un coût

On suppose qu'on associe à chaque action A_i un coût (réel positif). Le coût associé à une suite d'états est la somme des coûts associés aux actions exécutées.

Une solution est *optimale* s'il n'existe pas de solution de coût strictement inférieur. Une méthode est dite *admissible* si chaque fois qu'il existe une solution optimale, elle est trouvée en un temps fini. Dans la suite nous adoptons les notations suivantes :

$k(E_i, E_j)$ Le coût de l'action la moins chère pour aller de E_i à E_j si elle existe.

$k^*(E_i, E_j)$ Le coût de la séquence d'actions la moins chère pour aller de E_i à E_j.

$g^*(E_i)$ $g^*(E_i) = k^*(E_0, E_i)$

$h^*(E_i)$ $h^*(E_i) = \min k^*(E_i, E_j)$ avec $E_j \in F$, autrement dit $h^*(E_i)$ représente le coût minimal pour atteindre l'objectif si ce chemin existe.

$f^*(E_i)$ $f^*(E_i) = g^*(E_i) + h^*(E_i)$ Autrement dit $f^*(E_i)$ représente le coût minimal d'une solution passant par E_i si elle existe.

Nous utilisons dans la suite la notion de successeur d'états qui est définie par : $Succ(E_i) = \{E_j : (A_j, E_j) \in T(E_i)\}$

1.6 Recherche guidée

Nous allons montrer comment introduire une *heuristique* qui permet de guider le parcours en largeur en recherchant une solution dans l'arbre de recherche. Cette heuristique tente de diriger le parcours en coupant certaines branches et en allant en profondeur dans d'autres. Une heuristique est une mesure associée à un état donné qu'on notera $h(E_i)$.

1.6.1 Propriétés d'une heuristique

Une heuristique $h(E_i)$ est dit *coïncidente* si elle reconnaît les états de l'objectif : $\forall E_j \in F, h(E_j) = 0$.

Elle est *presque parfaite* si elle donne sans erreur la branche à explorer : $h(E_j) < h(E_i)$ où E_j est un état qui suit E_i.

Elle est dite *consistante* si pour toute paires d'états nous avons : $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$.

Elle est *monotone* si $\forall (E_i, E_{ij}), (A_{ij}, E_{ij}) \in T(E_i), h(E_i) - h(E_{ij}) \leq k(E_i, E_{ij})$.

Une heuristique est dite minorante si elle sous-estime systématiquement le coût du chemin restant à parcourir : $h(E_i) \leq h^*(E_i)$

THÉORÈME 1.6.1 *h est monotone ssi h est consistante*

DÉMONSTRATION. Supposons que h est consistante , dons pour toute paire d'états nous avons $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$ et plus particulièrement si $E_j \in Succ(E_i)$ alors $h(E_i) - h(E_j) \leq k^*(E_i, E_j)$ et par définition nous avons $k^*(E_i, E_j) \leq k(E_i, E_j)$.

Maintenant supposons que la stratégie est monotone Soient (E_0, E_n) une paire d'états pour laquelle il y a un chemin optimal $E_1, ..E_{n-1}, E_n$, nous avons $\forall i (h(E_{i-1}) - h(Ei) \leq k(E_{i-1}, Ei))$ En sommant nous obtenons : $h(E_0) - h(E_n) \leq \sum_{i=1}^{n} k(E_{i-1}, Ei)$ et donc : $h(E_0) - h(E_n) \leq k^*(E_0, E_n)$ ∎

THÉORÈME 1.6.2 *Si h est monotone et coïncidente, alors h est minorante.*

DÉMONSTRATION. Par hypothèse de monotonie et sur un chemin optimal, on obtient que $h(E_0) - h(E_n) \leq k^*(E_0, E_n)$ et puisque le chemin est optimal alors $h(E_0) - h(E_n) \leq h^*(E_0)$ et puisque l'heuristique est coïncidente alors $h(E_0) \leq h^*(E_0)$ ∎

1.6.2 Présentation de l'algorithme A*

L'algorithme A* effectue un parcours en largeur guidé par une heuristique définie préalablement. Deux files sont utilisées pour stocker les états visités et à visiter ; la file *Actif* contient les états à visiter tandis que la file *Inactif* contient les états déjà visités. Parmi tous les successeurs d'un état donné, sont ajoutés à la file *Actif*, les états non visités ou les états déjà visités mais dont le coût du passage actuel est moins élevé qu'avant. Une mesure $f(e) = g(e) + h(e)$ est associée à chaque état ; cette mesure combine l'heuristique $h(e)$ avec le coût actuel du passage par l'état $g(e)$. La file *Actif* est triée par ordre de f croissantes. Autrement dit, l'état dont le coût estimé est le moins élevé sera sélectionné.

PROCEDURE A*()

VAR E,E' : ETAT,

Actif, Inactif : FILE

$Actif \leftarrow [E_0]$

$Inactif \leftarrow []$

$g(E_0) \leftarrow 0$

$E \leftarrow E_0$

TANTQUE non fileVide(Actif) ET non $E \in F$ **FAIRE**

supprimer(Actif)

inserer(Inactif, e)

POUR TOUT $e' \in Succ(e)$ **FAIRE**

SI non ($e' \in Actif$ ET $e' \in Inactif$) OU g(e') > g(e)+k(e,e') **ALORS**

$g(e') \leftarrow g(e) + k(e,e')$

$f(e') \leftarrow g(e') + h(e')$

$pere(e') \leftarrow e$

ajouterTrier(Actif,e')

FIN SI

FIN POUR

SI non(fileVide(Actif)) **ALORS**

$e \leftarrow premier(Actif)$

FIN SI

FIN TANTQUE

Algorithme 3: l'algorithme A*

THÉORÈME 1.6.3 *Si tout chemin de longueur infinie a un coût infini, si sur un chemin optimal, la valeur de l'heuristique est bornée et si chaque état a un nombre fini de successeurs alors l'algorithme A* termine.*

THÉORÈME 1.6.4 *Si les conditions de terminaisons sont réalisées et si l'heuristique est minorante alors l'algorithme A* est admissible.*

Selon ce théorème, toute heuristique qui sous-estime la réalité est une heuristique qui permet de trouver le chemin optimal. Nous disons que h2 est *plus informée* que h1 si toutes les deux sont minorantes et si $h2(e) > h1(e)$, $\forall e \in S$. En effet h2 permet de trouver le chemin optimal plus rapidement parcequ'elle est plus proche de $h^*(e)$.

1.7 Implémentation en Prolog

1.7.1 Le parcours en Profondeur d'abord

Nous présentons un programme prolog qui effectue la recherche en profondeur d'abord. Ce programme est basé sur la récursivité. Le prédicat solve permet de retrouver l'état final à partir d'un état donné. Les états déjà visités sont stockés dans une liste. Le prédicat *move* associe des actions à des états de départs. Le prédicat *update* change l'état en appliquant une action. Finalement le prédicat *legal* teste si un état existe ou non. Le programme en prolog est donné par :

PROGRAMME 1.7.1

```
solve(State,History,[]) :- etat_final(State).
solve(State,History,[Move|Moves]):-
move(State,Move),
update(State,Move,State1),
legal(State1),
not member(State1,History),
solve(State1,[State1|History],Moves).

test(Moves) :- etat_initial(State), solve(State,[State],Moves).
```

1.7.2 Le parcours en Largeur d'abord

Ce programme est composé de trois prédicats :

solveB effectue l'exploration en largeur et s'arrête dès qu'il trouve une solution ;

update-set met à jour la liste des noeuds à explorer en remplaçant le premier élément par ses fils (ces fils sont insérés à la fin) ;

insertF insert un élément à la fin d'un liste donnée si jamais cet élément correspond à un noeud non visité auparavant.

PROGRAMME 1.7.2

```
solveB([state(State,Path)|Reste],_,Moves):-
etat\_final(State),
reverse(Path,Moves).
solveB([state(State,Path)|Reste],History,Finalpath):-
not(etat\_final(State)),
findall(M,move(State,M),Moves),
update-set(Moves,State,Path,History,Reste,Reste1),
solveB(Reste1,[State|History],Finalpath).
update-set([M|Ms],State,Path,History,R,R1):-
update(State,M,State1),
insertF(state(State1,[M|Path]),History,R,R2),
update-set(Ms,State,Path,History,R2,R1).
update-set([],S,P,H,R,R).
insertF(state(State,_),History,[],[]):-
    member(State,History),!.
    insertF(X,History,[],[X]).
insertF(X,History,[T|Reste],[T|Reste1]):-
insertF(X,History,Reste,Reste1).

test(Moves):-etat\_initial(State),
    solveB([state(State,[])],[State],Moves).
```

1.7.3 La modélisation du problème du taquin

Nous modélisons le problème du taquin en utilisant quatre prédicats qui sont :

etat_initial décrit l'état initial du taquin en le lisant de haut en bas et de gauche à droite ;

etat_final décrit l'état final du taquin ;

move décrit un action possible appliquée à un état donné ;

update décrit l'état résultant de l'application d'une action à un état donné.

PROGRAMME 1.7.3

```
etat_initial([1,b,2,3]).
etat_final([2,1,3,b]).
move([b,X,Y,Z],droite):-X\==b,Y\==b,Z\==b.
move([b,X,Y,Z],bas):-X\==b,Y\==b,Z\==b.
move([X,b,Y,Z],gauche):-X\==b,Y\==b,Z\==b.
move([X,b,Y,Z],bas):-X\==b,Y\==b,Z\==b.
move([X,Y,b,Z],droite):-X\==b,Y\==b,Z\==b.
move([X,Y,b,Z],haut):-X\==b,Y\==b,Z\==b.
move([X,Y,Z,b],gauche):-X\==b,Y\==b,Z\==b.
move([X,Y,Z,b],haut):-X\==b,Y\==b,Z\==b.
update([b,X,Y,Z],bas,[Y,X,b,Z]).
update([b,X,Y,Z],droite,[X,b,Y,Z]).
update([X,b,Y,Z],bas,[X,Z,Y,b]).
update([X,b,Y,Z],gauche,[b,X,Y,Z]).
update([X,Y,b,Z],haut,[b,Y,X,Z]).
update([X,Y,b,Z],droite,[X,Y,Z,b]).
update([X,Y,Z,b],haut,[X,b,Z,Y]).
update([X,Y,Z,b],gauche,[X,Y,b,Z]).
```

2

APPRENTISSAGE SYMBOLIQUE DE CONCEPTS

2.1 Introduction

Étant donnée une tâche T à effectuer par un programme informatique donné et une mesure de performance par rapport à cette tâche P, on dit que ce programme *apprend* à partir d'une expérience E si la valeur de la mesure P augmente avec l'expérience.

Par exemple dans le domaine de la *Reconnaissance de manuscrits* : T est la tâche de reconnaissance et de classement des lettres manuscrites données en entrée, P est le pourcentage des mots classés correctement par le programme, E est une base de données contenant des mots avec une fonction qui permet d'effectuer la classification.

Pour mettre en oeuvre un système d'apprentissage automatique, il faut choisir l'expérience qui permet au système d'apprendre ainsi que la fonc-

tion cible à apprendre. L'expérience est souvent exprimée en terme d'un ensemble d'exemples qui peuvent, dans certains domaines, être appuyés par une théorie qui décrit quelques aspects du domaine en question. La fonction cible est apprise à partir de l'ensemble d'exemples, chaque exemple serait représenté par le couple (b,f(b)), b étant les données de l'exemple et f(b) est la valeur de la fonction cible sur cette donnée.

Nous présentons dans la suite, un cas particulier d'un système d'apprentissage dans lequel la fonction cible est l'appartenance ou non d'un exemple donné à un concept donné.

En d'autres mots, l'objet de ce chapitre est de présenter deux algorithmes (Find-S et Candidate-Elimination) qui permettent de décrire des concepts appris à partir d'un ensemble contenant d'exemples positifs et négatifs. Cette description sera formalisée en utilisant la terminologie des espaces d'hypothèses et de versions. Ce chapitre permettra à l'élève d'avoir un premier aperçu de l'apprentissage automatique à partir de données en utilisant un langage simple de description de concept qu'est celui des hypothèses.

2.2 L'apprentissage de concepts

La plupart des systèmes d'apprentissage ont comme objectif d'apprendre de concepts généraux à partir d'exemples spécifiques. Chaque concept peut être représenté (par extension) par le sous-ensemble d'exemples qu'il représente, ou (par intension) par une fonction booléenne qui décrit l'appartenance au concept en fonction de certaines caractéristiques.

Par conséquent, on peut dire que l'apprentissage de concept consiste à inférer une fonction booléenne à partir d'un ensemble d'exemples contenant chacun les entrées et la sortie correspondante.

2.2.1 Notations

Nous adoptons dans la suite la terminologie suivante qui nous permettra de décrir la tâche de l'apprentissage de concept.

Une instance Étant donné un ensemble d'attribut, chacun ayant un domaine de valeurs, une *instance* est une valuation possible de cet ensemble d'attributs dans leurs domaines respectifs.

La fonction concept Soit X un ensemble donné d'instances, la *fonction concept* : $c(X) \rightarrow \{0,1\}$ est une fonction qui permet d'affecter une valeur d'appartenance à une instance $x \in X$ au concept en cours d'apprentissage.

L'ensemble d'apprentissage C'est un ensemble d'instances munis de leurs valeurs d'appartenance au concept.

L'espace d'hypothèses Une *hypothèse* est définie comme étant une conjonction de contraintes sur la liste d'attributs : <val1,..,valN>. Une contrainte sur un attribut peut être une valeur appartenant au domaine de l'attribut ou, peut prendre la valeur " ?" ou "∅". La valeur " ?" signifie que l'attribue en question peut prendre n'importe quelle valeur tandis que la valeur "∅" signifie que l'attribut ne peut prendre aucune valeur dans son domaine. L'espace d'hypothèses comprend toutes les hypothèses possibles pour décrire une domaine donné.

Par exemple, la table suivante nous montre un ensemble d'apprentissage. Une instance est décrite par les attributs : $\{CIEL, TEMP, HUMI, VENT\}$ définis respectivement sur les domaines suivants :
$D(CIEL) = \{ensoleille, couvert, pluvieux\}$,
$D(TEMP) = \{eleve, moyenne, basse\}$,
$D(HUMI) = \{forte, normale, moyenne\}$,$D(VENT) = \{oui, non\}$.
Cette table comporte deux exemples positifs (ayant c(x)=1) et deux exemples négatifs (avec c(x)=0).

NUM	CIEL	TEMP.	HUMI.	VENT	CLASSE
1	ensoleillé	élevé	forte	non	N
2	ensoleillé	élevé	forte	oui	N
3	couvert	élevé	forte	non	O
4	pluvieux	moyenne	forte	non	O

TAB. 2.1 – Description des conditions météorologiques et de la classe JouerFoot

Une hypothèse possible serait h=<ensoleillé, élevé,forte, ?> qui couvre les deux exemples négatifs dans l'ensemble d'apprentissage.

L'hypothèse < ?, ?, ?, ?> est appelée l'hypothèse la plus générale car elle représente tous les exemples possibles tandis que toute hypothèse contenant la valeur ∅ représente l'ensemble vide, autrement dit elle ne satisfait[1] aucun exemple.

[1] Un hypothèse h satisfait un exemple x ssi $h(x) = 1$

2.2.2 Schéma de description d'une tâche

La tâche à apprendre par le système peut être décrite par le schéma suivant : étant donné un ensemble d'instances X, muni de la fonction cible $c(X) \rightarrow \{0,1\}$, appelé *ensemble d'apprentissage* et contenant des exemples positifs et négatifs, il s'agit de déterminer $h \in H$ où H est l'espace d'hypothèses possible tel que h(x)=c(x) $\forall x \in X$

Autrement dit, il faut que l'hypothèse inférée classe correctement les exemples positifs et rejette les exemples négatifs.

Ce type d'apprentissage appartient à l'apprentissage par induction qui exprime l'hypothèse suivante : *une hypothèse trouvée à partir d'un ensemble d'exemples contenant des exemples suffisamment représentatifs du domaine doit classer assez correctement les nouveaux exemples*.

2.2.3 Relation d'ordre sur l'espace d'hypothèses

Il existe une relation d'ordre partiel sur l'espace d'hypothèses qui nous permettra dans la suite de trouver un treillis d'hypothèses consistantes. Cette relation est basée sur la relation PGE (Plus Général ou Égal) qu'on définira ainsi : Soit h_j,h_k deux hypothèses définies sur X. h_j est PGE que h_k : $(h_j \geq h_k)$ ssi $(\forall x \in X)[h_k(x) = 1 \rightarrow h_j(x) = 1]$.

De même, h_j est PG (plus général strictement) que h_k $(h_j > h_k)$ ssi $(h_j \geq h_k) \wedge (h_k \ngeq h_j)$.

2.3 L'algorithme Find-S

Nous présentons dans cette section l'algorithme Find-S dont l'objectif est de trouver l'hypothèse la plus spécifique qui satisfait à tous les exemples positifs dans l'ensemble d'apprentissage.

(1) Initialiser h à $< \varnothing, .., \varnothing >$, l'hypothèse la plus spécifique.

(2) Pour chaque exemple positif x faire
   - Pour toute valeur a_i de contrainte dans h
     - Si a_i est satisfait par x, alors (Ne rien faire)
     - Sinon, remplacer a_i dans h par la contrainte suivante la plus générale qui satisfait x

(3) Retourner h

Par exemple, en appliquant Find-S sur les quatre exemples présentés ci-dessus, nous initialisons h à $h =< \varnothing, \varnothing, \varnothing, \varnothing >$, après lecture de

l'exemple numéro 3, nous aurions : $h =< couvert, eleve, forte, non >$ et après lecture de l'exemple 4, nous aurions : $h =<?, ?, forte, non >$.

L'inconvénient majeur de cet algorithme est qu'il ignore complètement les exemples négatifs. Par conséquence, il est possible d'avoir une hypothèse qui satisfait tous les exemples positifs mais qui ne rejette pas obligatoirement tous les exemples négatifs car aucune vérification n'est effectuée dans l'algorithme. Autrement dit, l'algorithme Find-S est incapable dans ce cas de retourner une hypothèse vide.

D'un autre côté, Find-S trouve l'hypothèse la plus spécifique qui couvre les exemples positifs sachant qu'il peut exister d'autres hypothèses plus générales qui seront consistantes avec l'ensemble d'apprentissage. Nous présentons dans la suite un algorithme permettant de construire l'espace des hypothèses consistantes avec un ensemble donné d'exemples. Cet espace est appelé *espace de version*.

2.4 L'espace de version

La construction de l'espace de version est fondée sur la recherche des *hypothèses consistantes* avec l'ensemble d'apprentissage D dans H. Une hypothèse h est dite *consistante* avec un ensemble d'apprentissage donné ssi $h(x) = c(x)$ pour chaque exemple (x,c(x)) dans D. L'espace de version est donné par

$$VS_{H,D} = \{h \in H \mid Consistant(h, D)\}$$

Nous définissons dans la suite la *limite générale* et *la limite spécifique* d'un espace d'hypothèses H par rapport à un ensemble d'apprentissage D, notées respectivement par G et S.

La limite générale G est l'ensemble des hypothèses les plus générales qui seront consistantes avec D et elle est donnée par

$$G = \left\{ g \in H \mid Consistant(g, D) \wedge (\nexists g' \in H) \left[(g' > g) \wedge Consistant(g', D) \right] \right\}$$

De même, la limite spécifique S est l'ensemble des hypothèses les plus spécifiques qui seront consistantes avec D et elle est donnée par

$$S = \left\{ s \in H \mid Consistant(s, D) \wedge (\nexists s' \in H) [(s' < s) \wedge Consistant(s', D)] \right\}$$

THÉORÈME 2.4.1 *L'espace de version est donnée par :*

$$VS_{H,D} = \{h \in H \mid (\exists s \in S)(\exists g \in G) \mid (s \leq h \leq g)\}$$

DÉMONSTRATION. Soit h une hypothèse appartenant à l'ensemble $VS_{H,D}$ et soit $x \in X$ un exemple positif. Par définition s satisfait x, et puisque $h > s$ nous avons $h(x) = 1$. De même, soit $y \in X$ un exemple négatif, g ne satisfait pas y : $g(y) = 0$, par conséquence h(y) ne peut prendre que la valeur 0 car si $h(y) = 1$ alors $g(y) = 1$ (car g est plus général que h)

Nous avons démontré que toute hypothèse appartenant à $VS_{H,D}$ est consistante avec l'ensemble d'apprentissage D. Il reste à démontrer que toute hypothèse consistante appartient à l'ensemble $VS_{H,D}$; cette démonstration est laissée à la charge du lecteur intéressé ! ∎

2.5 L'algorithme Candidate-Elimination

L'algorithme Candidate-Elimination calcule l'espace de version en trouvant les deux limites G et S. Les deux limites sont initialisées aux hypothèses $<?,..,?>$ et $<\varnothing,..,\varnothing>$ respectivement. À chaque lecture d'un exemple positif x, la limite S est généralisée d'une façon minimale pour que l'exemple positif soit consistant avec les éléments de S ; de même, les éléments de G non consistants avec x sont supprimés. Un comportement symétrique par rapport à (G,S) est effectué lors de la lecture d'un exemple négatif.

Nous donnons dans la suite l'algorithme détaillé :

(1) Initialiser G à $\{<?,\ldots,?>\}$

(2) Initialiser S à $\{<$
varnothing$,\ldots,\varnothing>\}$

(3) Si d est un exemple positif
   - Supprimer de G les hypothèses inconsistantes avec d
   - Pour chaque hypothèse $s \in S$ qui n'est pas consistante avec d faire
     - Supprimer s de S
     - Ajouter à S toutes le généralisations minimales h de s tel que h soit consistante avec d en vérifiant qu'il existe une hypothèse $g \in G$ plus générale que h.

  - supprimer de S chaque hypothèse plus générale que d'autre hypothèse contenue également dans S.

(4) Si d est un exemple négatif
  - Supprimer de S les hypothèses inconsistantes avec d
  - Pour chaque hypothèse $g \in G$ qui n'est pas consistante avec d faire
    - Supprimer g de G
    - Ajouter à G toutes le spécialisations minimales h de g tel que h soit consistante avec d en vérifiant qu'il existe une hypothèse $s \in S$ plus spécifique que h.
    - supprimer de G chaque hypothèse plus spécifique qu'autre hypothèse contenue également dans G.

2.6 Exercices

EXERCICE 2.1 En analysant des actifs financiers côtés en bourse, nous voulons apprendre le concept-cible *l'actif distribue des dividendes.* Pour ce faire nous avons les exemples du tableau suivant :

No	Prix	Bénéfices	Secteur	Actif	Bourse	Croissance	Dividende
1	Élevé	Élevés	Manufacture	Action	NYSE	Forte	1
2	Élevé	Élevés	Services	Action	NYSE	Forte	1
3	Faible	Faibles	Services	Action	NYSE	Faible	0
2	Élevé	Élevés	Services	Action	Nasdaq	Faible	1

(1) Appliquer les algorithmes FIND-S et CANDIDAT-ELIMINATION et discuter les étapes de chaque algorithme.

(2) Calculer le nombre d'éléments de l'espace d'hypothèses et de l'espace d'instances.

(3) Donner ces mêmes nombres si on rajoute un attribut nouveau qui a k valeurs différentes.

(4) Ranger les exemples dans un autre tableau en ordre inverse et appliquer de nouveau l'algorithme CANDIDAT-ELIMINATION. Expliquer pour quelles raisons l'espace des versions final est le même dans les deux cas.

(5) Pouvez-vous envisager une stratégie pour ordonner les exemples, afin de réduire le nombre d'opérations nécessaires lors de l'apprentissage du concept ?

EXERCICE 2.2 Nous voulons apprendre le concept *voiture familiale japonaise*. Pour cela nous avons les exemples suivants :

Pays	Constructeur	Couleur	Année	Type	Exemple
Japon	Honda	Bleue	2000	Familiale	1
Japon	Toyota	Verte	1997	Sportive	0
Japon	Toyota	Bleue	1999	Familiale	1
Étas-Unis	Chrysler	Rouge	2000	Familiale	0
Japon	Honda	Blanche	2000	Familiale	1

Appliquer les algorithmes FIND-S et CANDIDAT-ELIMINATION et discuter les résultats.

Ajouter les deux nouveaux exemples suivants :

Pays	Constructeur	Couleur	Année	Type	Exemple
Japon	Toyota	Verte	2000	familiale	1
Japon	Honda	Rouge	1999	Familiale	0

En appliquant l'algorithme CANDIDAT-ELIMINATION évaluer l'évolution des ensembles $S(X)$ et $G(X)$.

EXERCICE 2.3 Considérons comme espace des exemples X les points $x = (x_1, x_2) \in \{0,1,2,\ldots,10\} \times \{0,1,2,\ldots,10\} = [10] \times [10]$, où par $[10]$ on note l'ensemble des entiers $\{0,1,2,\ldots,10\}$. On cherche à apprendre le concept rectangle qui est l'ensemble $((a,b),(c,d)) \in [10]^2 \times [10]^2$, avec (a,b) les coordonnées de l'angle gauche supérieure et (c,d) les coordonnées de l'angle coordonnées de l'angle droit inférieure.

(1) Considérons l'ensemble X des exemples

No	Coordonnées	Appartenance
1	(0, 5)	0
2	(4, 5)	1
3	(2, 2)	0
4	(9, 5)	0
5	(6, 3)	1
6	(5, 6)	1
7	(7, 0)	0
8	(5, 8)	0

qui indiquent si un point de coordonnées connues appartient ou non au rectangle.

Donner les hypothèses qui forment les ensembles $S(X)$ et $G(X)$.

(2) Supposons que nous n'avons pas d'ensemble X d'exemples mais seulement un point initial qui appartient au rectangle et la possibilité de poser

des questions concernant les coordonnées des différents points et recevoir comme réponse leur appartenance ou non au rectangle.

Si le concept-cible est le rectangle $((4,\ 6),(6,\ 4))$ et le point initial a comme coordonnées $(5,\ 5)$, élaborer une stratégie qui permet d'apprendre le concept-cible en utilisant un nombre d'exemples minimal.

EXERCICE 2.4 *(D'après [2]).* Considérons les exemples suivants qui décrivent le concept-cible « couple de gens qui vivent dans la même maison ».

No	Couple	Ensemble
1	(homme, chatain, grand, USA) (femme, noire, petite, USA)	1
2	(homme, chatain, petit, France) (femme, noire, petite, USA)	1
3	(femme, chatain, grande, Allemagne) (femme, noire, petite, Inde)	0
4	(homme, chatain, grand, Irlande) (femme, chatain, petite, Irlande)	1

(1) Appliquer l'algorithme CANDIDAT-ELIMINATION.

(2) Considérons l'exemple suivants :

No	Couple	Ensemble
	(homme, noir, petit, Portugal) (femme, blonde, grande, Inde)	1

Indiquer l'évolution des ensembles $S(X)$ et $G(X)$.

3

MÉTHODES DE RECHERCHE POUR LES JEUX

Une activité, certes ludique, pour laquelle les êtres humains déploient leur intelligence depuis la nuit des temps est le jeu. Il est donc naturel de chercher de créer des programmes de jeux en utilisant des méthodes de l'intelligence artificielle. On pourrait même évaluer les progrès de l'intelligence artificielle avec les progrès des programmes de jeux.

Les jeux offrent en effet un paradigme exceptionnel pour les méthodes de l'intelligence artificielle. Ils ont un nombre de règles limité et des situations parfois très complexes. De plus les décisions, lors du déroulement d'un jeu, peuvent être évaluées de façon exacte, ce qui permet d'avoir une idée précise sur la valeur du programme de jeu.

Nous considérons des jeux à deux joueurs avec information parfaite, i.e. chaque joueur est au courant de ce que l'autre joueur a fait ou peut en

faire[1]. Le joueur qui commence s'appelle le joueur MAX (on dit aussi le *joueur A*) et l'autre joueur s'appelle MIN (ou le *joueur B*). Ceci parce que

- celui qui commence cherche à trouver, parmi toutes les situations qui a sa disposition, une situation qui lui permet de maximiser ses gains ;
- celui qui répond doit trouver, à partir de toutes les situations qui conduisent à la victoire du premier joueur, la situation qui minimise les gains de ce joueur.

Dans la suite nous présentons les principaux algorithmes de recherche pour les jeux. Il y a bien évidemment un ensemble impréssionant d'algorithmes pour les jeux et il n'est possible que de travailler sur un très petit nombre d'algorithmes. Les élèves interessés par le domaine sont priés de se reporter à la bibliographie.

3.1 Définition générale d'un jeu

La représentation des jeux se fera à l'aide d'un graphe. Les sommets du graphe représentent les différentes situations du jeu. Ils sont symbolisés par des carrés si c'est au tour du joueur MAX de jouer sur ce sommet, sinon ils sont notés par un cercle.

Un jeu est défini par :

(1) Un ensemble X, dont tous les éléments sont appelés *positions* (*situations*) du jeu.

(2) Une application multivoque $\Gamma : X \to X$, appelée *règle du jeu*.

(3) Une application univoque $\theta : X \to \{0,1,2\}$ appelée *le trait*. Si $\theta(x) = 1$ c'est au joueur MAX de jouer et si $\theta(x) = 2$ c'est au joueur MIN de jouer. Nous avons $\theta(x) = 0$ si et seulement si $\Gamma(x) = \emptyset$.

(4) Deux fonctions réelles ϕ_A et ϕ_B définies sur X et bornées, appelées *les fonctions de préférence* des joueurs A et B.

La partie s'effectue de la façon suivante : D'une position initiale $x_0 \in X$ avec $\theta(x_0) = 1$, le joueur A choisit une position $x_1 \in \Gamma(x_0)$ avec $\theta(x_0) = 2$. Le joueur B choisit une position $x_2 \in \Gamma(x_1)$ avec $\theta(x_1) = 1$, et ainsi de suite. Si un joueur choisit une position x telle que $\Gamma(x) = \emptyset$, alors la partie s'arrête.

Si $S \subset X$ est l'ensemble des positions rencontrées au cours d'une partie, les gains des joueurs sont :

[1] D'une certaine façon les problèmes examinés au chapitre 1 peuvent être vus comme des jeux à un joueur.

– pour le joueur A : $q_A(S) = \sup_{x \in X} q_A(x)$;

– pour le joueur B : $q_B(S) = \inf_{x \in X} q_b(x)$.

On posera $X_A = \{x \in X \ / \ \theta(x) = 1\}$, $X_B = \{x \in X \ / \ \theta(x) = 2\}$. Un jeu est donc un graphe $G = (X, \Gamma)$ valué en chaque sommet x par le vecteur $[\theta(x), q_A(x), q_B(x)]$ et par conséquent peut être représenter par le quintuplet $(X, \Gamma, \theta, q_A, q_B)$.

À partir d'un graphe $G = (X, \Gamma)$ qui représente un jeu, nous pouvons définir des sous-jeux et des jeux partiels

DÉFINITION 3.1.1 *Un sous-jeu est un sous-graphe (Y, Γ_Y) du graphe (X, Γ).*

Un jeu partiel est un graphe partiel (X, Δ) du graphe (X, Γ) avec $\Delta(x) \subset \Gamma(x)$ $\forall x \in X$.

Nous pouvons aussi définir des nouveaux jeux à l'aide des opérations sur les graphes.

DÉFINITION 3.1.2 *Soient n graphes $G_1 = (X_1, \Gamma_1), \ldots, G_n = (X_n, \Gamma_n)$. On définit*

– *le graphe-somme $G = (X, \Gamma)$ des graphes, avec $X = X_1 \times \cdots \times X_n$ et*

$$\begin{aligned}\Gamma(x_1, \ldots, x_n) = {} & (\Gamma_1(x_1) \cup \{x_2\} \cup \cdots \cup \{x_n\}) \cup \\ & (\{x_1\} \cup \Gamma_2(x_2) \cup \cdots \cup \{x_n\}) \cup \cdots \cup \\ & (\{x_1\} \cup \{x_2\} \cup \cdots \cup \Gamma(x_n))\end{aligned}$$

– *le graphe-produit $G = (X, \Gamma)$ des graphes, avec $X = X_1 \times \cdots \times X_n$ et $\Gamma(x_1, \ldots, x_n) = \Gamma_1(x_1) \times \Gamma_2(x_2) \times \cdots \times \Gamma(x_n)$.*

Le graphe-somme sera noté $G = G_1 + \cdots + G_n$ et le graphe-produit $G = G_1 \times \cdots \times G_n$.

3.1.1 Arbres et stratégies de jeux

Un arbre de jeu est issu du graphe du jeu et il représente de manière explicite toutes les façons de jouer le jeu. La racine est la situation initiale du jeu, ses successeurs sont les situations que le joueur A peut atteindre en un seul coup, et ainsi de suite. Les feuilles représentent les situations finales du jeu et elles peuvent être étiquetées de trois façons : G pour gagnant, P pour perdant et N pour la nullité[2].

Il y a manifestement des similitudes entre les arbres de jeu et l'arbre de recherche $\mathbf{G} = (X, \alpha, \Omega, \Gamma)$ que nous avons étudié au chapitre précédent.

[2]On rappelle que les définitions sont données pour le joueur A ou joueur MAX.

Nous avons que Ω est l'ensemble des feuilles et Γ représente les stratégies des joueurs. La stratégie du joueur A consiste à trouver un chemin qui mène de la racine à une feuille étiquetée G et ceci indépendamment des mouvements du joueur B. La stratégie du joueur B consiste à trouver un chemin qui aboutit à une feuille étiquetée P. Chaque chemin qui mène de la racine à une feuille constitue une partie que les joueurs pourraient jouer.

Il va de soi que pour la plupart de jeux il n'est pas envisageable de développer la totalité de l'arbre de jeu. On commence, comme avec l'arbre de recherche, par la racine et nous développons les sommets successifs au fur et à mesure que le jeu avance. La méthode pour choisir une position dépend de la nature du jeu : pour certains types de jeux le choix est le résultat d'un calcul. Pour d'autres – les plus nombreux – se fera de manière heuristique.

3.1.2 Exemples

EXEMPLE 3.1.1 ***Jeu des poursuites***. *Sur une surface S (la mer) considérons deux bateaux dont l'un poursuit l'autre.*

Si x_1 est le point de S où se trouve le bateau poursuivant et x_2 le point du bateau poursuivi et si i est le trait ($i = 1,2$), alors la position du jeu est $\mathbf{x} = [x_1, x_2, i]$. Si $B_1(x_i, r_i) \subset S$ est le cercle de centre x_i et de rayon r_i, où r_i est la distance que le bateau i peut parcourir en un coup, on a :

- $\Gamma(x_1, x_2, 1) = B_1(x_1, r_1) \times \{x_2\} \times \{2\}$;
- $\Gamma(x_1, x_2, 2) = \{x_1\} \times B_2(x_2, r_2) \times \{1\}$;

si on suppose que les joueurs jouent alternativement.

Si le but du poursuivant est de rattraper le poursuivi, alors :

$$q_A(x_1, x_2, 1) = \begin{cases} 1 \text{ si } x_1 = x_2, \\ 0 \text{ sinon.} \end{cases}$$

$$q_B(x_1, x_2, 2) = \begin{cases} 1 \text{ si } x_1 \neq x_2, \\ 0 \text{ sinon.} \end{cases}$$

EXEMPLE 3.1.2 ***Jeu d'échecs***. *Le joueur A joue avec les blanc et le B avec les noirs. Une position est donnée par le vecteur $\mathbf{x} = [x_1, x_2, \ldots; i]$ où x_k représente une case sur laquelle il y a une pièce et i est le trait. Nous avons :*

$$q_L(\mathbf{x}) = \begin{cases} 1 \text{ si } \mathbf{x} \text{ est une position de mat pour l'adversaire,} \\ 0 \text{ sinon.} \end{cases}$$

3.2 Les graphes infinis

Le jeu d'échecs nous introduit à une catégorie des graphes qui se distingue par le très grand nombre d'arcs et de chemins. Pour généraliser, nous pouvons même envisager des graphes qui ne sont pas finis. Néanmoins, leur traitement il faut qu'il soit fini.

D'abord, qu'appelle-t-on un graphe fini ? Il s'agit d'un graphe dont le nombre de sommets est fini, i.e. si $G = (X, \Gamma)$ est le graphe, alors G est fini si $| X | < \infty$. Mais cette définition ne dit pas tout. En effet, tout programmeur sait qu'il ne suffit pas d'avoir un nombre de lignes de code fini, pour que l'exécution du programme correspondant se termine à temps fini. De même le parcours d'un graphe fini peut ne pas se faire à temps fini s'il existe des circuits[3] dans ce graphe. Étant donné que ce qui nous intéresse pour les jeux c'est le parcours du graphe, nous pouvons à la limite s'en accommoder des graphes infinis, à condition que le parcours dans ces graphes, pour trouver une solution, peut se faire en temps fini. Nous voyons ainsi que l'effort de la réflexion doit se faire sur la nature de Γ que sur le cardinal de X.

3.2.1 Les graphes progressivement finis

Nous commençons par quelques définitions :

DÉFINITION 3.2.1 *Un graphe $G = (X, \Gamma)$ est localement fini si $| \Gamma(x) | < \infty$ et $| \Gamma^{-1}(x) | < \infty, \forall x \in X$.*

Le graphe est borné s'il existe un naturel m tel que $| \Gamma(x) | < m, \forall x \in X$

.

Il est évident qu'un graphe fini est localement fini.

DÉFINITION 3.2.2 *Un graphe $G = (X, \Gamma)$ est progressivement fini en $x \in X$ s'il n'existe pas de chemins de longueur infinie qui commencent en x.*

Un graphe est dit progressivement fini s'il est progressivement fini en chacun de ses sommets.

Un graphe est progressivement borné en $x \in X$ s'il existe un naturel m tel que tout chemin qui commence par x soit de longueur non supérieure à m.

Un graphe est progressivement borné s'il est progressivement borné en chacun de ses sommets.

Un graphe progressivement borné est aussi progressivement fini, mais la réciproque n'est pas toujours vraie.

Nous avons les théorèmes suivants :

[3] Le circuit dans un graphe est un chemin dont le dernier sommet se confond avec le premier. De ce fait nous pouvons parcourir le circuit autant de fois que l'on souhaite.

THÉORÈME 3.2.1 *Si un graphe est fini, les propriétés progressivement fini, progressivement borné et sans circuits sont équivalentes.*

THÉORÈME 3.2.2 *Si un graphe est localement fini et connexe*[4]*, on peut évaluer l'ensemble de ses sommets et de ses chaînes.*

3.2.2 Fonction ordinale

Pour travailler avec des graphes $G = (X, \Gamma)$ qui ne sont pas finis, nous devons utiliser les nombres ordinaux qui sont définis sur les graphes de la façon suivante :

$$\begin{aligned}
&X(0) = \{x \in X \ / \ \Gamma(x) = \emptyset\} \\
&X(1) = \{x \in X \ / \ \Gamma(x) \subset X(0)\} \\
&X(2) = \{x \in X \ / \ \Gamma(x) \subset X(1)\} \\
&\vdots \\
&X(k) = \{x \in X \ / \ \Gamma(x) \subset X(k-1)\} \\
&\vdots \\
&X(\omega) = \bigcup_{\alpha<\omega} X(\alpha) \\
&X(\omega+1) = \{x \in X \ / \ \Gamma(x) \subset X(\omega)\} \\
&\vdots
\end{aligned}$$

Nous pouvons, bien évidement, prolonger cette définition indéfiniment. Si α est un ordinal non limite, on a :

$$X(\alpha) = \{x \in X \ / \ \Gamma(x) \subset X(\alpha-1)\}$$

et si α est un ordinal limite, on a :

$$X(\alpha) = \bigcup_{\beta<\alpha} X(\beta)$$

Nous avons $X(\beta) \subset X(\alpha)$ si $\beta < \alpha$.

DÉFINITION 3.2.3 *L'ordre $o(x)$ d'un sommet x est le plus petit ordinal α tel que*

$$\begin{aligned}
&x \in X(\alpha) \\
&x \notin X(\beta) \ \textit{pour tout } \beta < \alpha.
\end{aligned}$$

La fonction $o(x)$ est appelée la fonction ordinale du graphe.

[4]Un graphe est connexe si entre deux sommets quelconques, il y au moins une chaîne qui les relie.
Notons que la chaîne est l'équivalent de la notion du circuit quand on travaille avec des arêtes et pas avec des arcs, c'est-à-dire quand la notion de l'orientation d'une liaison n'est pas prise en compte.

Il est évident que les sommets qui sont sur un circuit n'admettent pas d'ordre. Donc la fonction ordinale n'est pas définie obligatoirement sur tout X.

Pour l'existence d'une fonction ordinale sur un graphe, nous avons le théorème suivant :

THÉORÈME 3.2.3 *Un graphe $G=(X,\Gamma)$ a une fonction ordinale définie sur X si, et seulement si, le graphe est progressivement fini.*

3.3 Fonctions de Grundy

Soient un graphe fini $G=(X,\Gamma)$ et une fonction $g: X \to \mathbb{N}$.

DÉFINITION 3.3.1 *La fonction g est une fonction de Grundy sur le graphe si pour tout sommet $x \in X$, la valeur $g(x)$ est le plus petit naturel qui n'est pas dans l'ensemble $g(\Gamma(x)) = \{g(y) \ / \ y \in \Gamma(x)\}$, ou, encore $g(\Gamma(x)) = \min_{y\in\Gamma(x)} \{\mathbb{N} - g(y)\}$.*

Nous pouvons envisager une extension de la fonction de Grundy dans le cas des graphes infinis. En effet nous pouvons dire que $g(x)$ est le plus petit ordinal qui n'est pas dans l'ensemble $\{g(y) \ / \ y \in \Gamma(x)\}$.

Si x est une feuille, i.e. si $\Gamma(x)=\emptyset$, alors $g(x)=0$.

Notons qu'un graphe peut ne pas avoir une fonction de Grundy (par exemple s'il a une boucle), comme il peut en avoir plusieurs.

Nous avons les deux théorèmes suivants :

THÉORÈME 3.3.1 *Si un graphe $G=(X,\Gamma)$ est progressivement fini, admet une, et une seule, fonction de Grundy. Cette fonction a la propriété*

$$g(x) \leq o(x) \ ; \ \forall x \in X$$

THÉORÈME 3.3.2 *Si $|\Gamma(x)| < \infty$, alors $g(x) \leq |\Gamma(x)|$.*

Considérons maintenant des graphes-somme et des graphes-produit. La question qui se pose concerne l'existence d'une méthode qui permet de propager les fonctions de Grundy des graphes qui composent les graphes-somme et les graphes-produits sur ces graphes-ci. Pour ce faire, on introduira la somme digitale de nombres naturels.

Soit c un nombre naturel. En base 2, ce nombre s'écrirait $c_{(2)} = c^k c^{k-1} \cdots c^1 c^0$ avec $c^l \in \{0,1\}$. Le développement binaire de ce nombre est $c = 2^0 c^0 + 2^1 c^1 + \cdots + 2^k c^k$, que l'on peut noter $\left(c^0, c^1, \ldots, c^k\right)$.

DÉFINITION 3.3.2 *La somme digitale de nombres naturels $c_1, \ldots, c_n$ avec développement binaire $\left(c_1^0, c_1^1, \ldots, c_1^{k_1}\right), \ldots, \left(c_n^0, c_n^1, \ldots, c_n^{k_n}\right)$ est un naturel $c = c_1 \dot{+} c_2 \dot{+} \cdots \dot{+} c_k$ avec développement binaire*

$$\left(\left[\sum_{i=1}^{n} c_i^0\right]_{(2)}, \left[\sum_{i=1}^{n} c_i^1\right]_{(2)}, \cdots, \left[\sum_{i=1}^{n} c_i^k\right]_{(2)} \right)$$

où $k = \max(k_1, \ldots, k_n)$ et où le développement binaire pour un nombre c_i avec $k_i < k$ est complété par des zéros.

Nous pouvons aussi, pour calculer la somme digitale, utiliser la représentation binaire des nombres $c_1, \ldots, c_n$, et on a :

$$c = c_1 \dot{+} c_2 \dot{+} \cdots \dot{+} c_k = 2^0 \times \left[\sum_{i=1}^{n} c_i^0\right]_{(2)} + 2^1 \times \left[\sum_{i=1}^{n} c_i^1\right]_{(2)} + \cdots + 2^k \times \left[\sum_{i=1}^{n} c_i^k\right]_{(2)}$$

Nous avons les théorèmes suivants :

THÉORÈME 3.3.3 *L'ensemble des nombres naturels muni de l'opération de la somme digitale est un groupe abélien, i.e. la somme digitale appliquée aux nombres naturels*

- *est associative : $a \dot{+} (b \dot{+} c) = (a \dot{+} b) \dot{+} c$;*
- *a un élément neutre 0, vérifiant $a \dot{+} 0 = a$;*
- *pour chaque naturel a il existe un naturel b tel que $a \dot{+} b = 0\dot{+}$;*
- *est commutative : $a \dot{+} b = b \dot{+} a$.*

THÉORÈME 3.3.4 *Si les graphes $G_1, \ldots, G_2$ admettent des fonctions de Grundy $g_1, \ldots, g_n$, alors le graphe-somme $G = G_1 + \cdots + G_n$ admet aussi une fonction de Grundy g, dont la valeur en $x = (x_1, \ldots, x_n)$ est donnée par la somme digitale :*

$$g(x_1, \ldots, x_n) = g_1(x_1) \dot{+} \cdots \dot{+} g_n(x_n)$$

3.4 Graphes stables

Soient un graphe $G = (X, \Gamma)$ et un sous-ensemble de sommets $S \subset X$. Pour S on peut définir deux types de stabilité : interne et externe.

DÉFINITION 3.4.1 *L'ensemble de sommets S est stable intérieurement si deux sommets quelconques de S ne sont pas adjacent, c'est-à-dire si*

$$\Gamma(S) \cap S = \emptyset$$

On note par $\mathcal{S}(G)$ la famille des ensembles intérieurement stables du graphe G.

Le nombre de stabilité interne du graphe G est

$$\alpha(G) = \max_{S \in \mathcal{S}(G)} | S |$$

Nous avons $\emptyset \in \mathcal{S}(G)$ et si $S \in \mathcal{S}(G)$, $A \subset S$, alors $A \in \mathcal{S}(G)$.

LEMME 3.4.1 *Soient deux graphes G_1, G_2. On a*

$$\alpha(G_1 \times G_2) \geq \alpha(G_1)\alpha(G_2)$$

DÉFINITION 3.4.2 *Soient le graphe $G = (X, \Gamma)$ et une application surjective $\sigma : X \to X$. Cette application est dite préservante si*

$$\forall x, y \in X \text{ avec } y \neq x,\ y \notin \Gamma(x),\ \text{alors } \sigma(x) \neq \sigma(y),\ \sigma(y) \notin \Gamma(\sigma(x))$$

c'est-à-dire cette application préserve pour tout couple de sommets la propriété d'être non adjacents et distincts.

LEMME 3.4.2 *Soit $S \subset X$ sous-ensemble intérieurement stable et σ une application préservante. Alors $\sigma(S)$ est aussi intérieurement stable et on a*

$$| S | = | \sigma(S) |$$

LEMME 3.4.3 *Soit le graphe $G = (X, \Gamma)$. Si l'ensemble $\sigma(X)$ est intérieurement stable, alors*

$$\alpha(G) = | \sigma(X) |$$

Il n'y a pas d'algorithme simple qui permet de déterminer, pour un graphe donné, un ensemble stable intérieurement avec un nombre maximal d'éléments.

Passons maintenant à la stabilité externe. Nous avons la définition :

DÉFINITION 3.4.3 *Soient un graphe $G = (X, \Gamma)$ et un sous-ensemble de sommets $T \subset X$. L'ensemble T est stable extérieurement si pour tout sommet $x \notin T$, on a $\Gamma(x) \cap T = \emptyset$; autrement dit si l'on a $X - T \subset \Gamma^{-1}(T)$.*

On note par $\mathcal{T}(G)$ la famille des ensembles extérieurement stables du graphe G.

Le nombre de stabilité externe du graphe G est

$$\beta(G) = \min_{T \in \mathcal{T}(G)} |T|$$

Nous avons $X \in \mathcal{T}(G)$ et si $T \in \mathcal{T}(G)$, $T \subset A$, alors $A \in \mathcal{T}(G)$.

Bien qu'il existe un algorithme simple pour déterminer un ensemble extérieurement stable et ayant un nombre minimal d'éléments, nous ne le présentons pas car il en dehors de l'objectif de ce cours. L'élève interessé pourra consulter [3, pp.42-43].

3.5 Noyaux d'un graphe

Le noyau d'un graphe est très utile pour déterminer la solution d'un type particulier de jeux. Bien qu'il s'agit d'une notion introduite dans le cadre de la théorie d'utilité par von Neumann et Morgenstern (cf. [6]) sous le nom de *solution*, nous pouvons utiliser cette notion de façon indépendante.

DÉFINITION 3.5.1 *Soit un graphe $G = (X, \Gamma)$. Un ensemble $S \subset X$ de sommets est un noyau du graphe si S est stable intérieurement et extérieurement, c'est-à-dire si*

(1) $\forall x \in S$ *on a* $\Gamma(x) \cap S = \emptyset$;

(2) $\forall x \notin S$ *on a* $\Gamma(x) \cap S \neq \emptyset$.

Donc le noyau S n'admet pas de boucles et contient tout sommet x pour lequel on a $\Gamma(x) = \emptyset$. De plus $\emptyset$ n'est pas un noyau.

Nous avons les résultats suivants :

THÉORÈME 3.5.1 *Le noyau S d'un graphe $G = (X, \Gamma)$ est l'ensemble intérieurement stable avec nombre d'éléments maximal, i.e.*

$$|S| = \max_{S' \in \mathcal{S}(G)} |S'|$$

et inversement si G est un graphe non orienté, sans boucles l'ensemble intérieurement stable maximal est le noyau du graphe.

COROLLAIRE 3.5.1 *Un graphe non orienté, symétrique*[5]*, sans boucles admet un noyau.*

THÉORÈME 3.5.2 *Soient un graphe $G=(X,\Gamma)$ et un sous-ensemble $S\subset X$. La condition nécésessaire et suffisante pour que S soit noyau du graphe est que sa fonction caractéristique vérifie*

$$\chi_S(x)=1-\max_{y\in\Gamma(x)}\chi_S(y)$$

Si $\Gamma(x)=\emptyset$, on posera par convention $\max_{y\in\Gamma(x)}\chi_S(y)=0$.

THÉORÈME 3.5.3 *Un graphe progressivement fini admet un noyau.*

THÉORÈME 3.5.4 *Si un graphe localement fini n'a pa de circuits de longueur impaire, il admet un noyau.*

Il est important de pouvoir calculer le noyau d'un graphe s'il existe. Nous donnons un algorithme pour la construction du noyau du graphe $G=(X,\Gamma)$.

0 Données : Graphe $G=(X,\Gamma)$ avec $\mid X\mid=n$. Tableau d'incidence du graphe $A(i,j)\leftarrow\begin{cases}0,\text{ si } x_j\notin\Gamma(x_i)\\1,\text{ si } x_j\in\Gamma(x_i)\end{cases}; i,j=1,\ldots,n$. Tableau de valeurs de Grundy : $R(i,j)\leftarrow+\infty; i,j=1,\ldots,n$. Fonction de Grundy $g(\bullet)$. Listes `OUVERT` et `MARQUÉ` initialisées à vide.

1 Soit $k^*\in]n]$[6] tel que $\sum_{j=1}^{n}A(k^*,j)=\min_{k\in]n]}\sum_{j=1}^{n}A(k,j)$ et $x_{k^*}\notin$ `MARQUÉ`.

On pose

- $g(x_{k^*})\leftarrow 0$
- `OUVERT` $\leftarrow\{x_{k^*}\}$, `MARQUÉ` $\leftarrow\{x_{k^*}\}$
- $\forall i\in]n]$ tel que $A(i,k^*)\neq 0: R(i,k^*)\leftarrow g(x_{k^*})$

2 Si `OUVERT` = $\emptyset$, alors sortir en échec.

Sinon

Soit x_{k^*} le premier élément d'`OUVERT` $x_{k^*}\leftarrow$ `car(OUVERT)`

On supprime d'`OUVERT` l'élément x_{k^*} : `OUVERT` $\leftarrow$ `OUVERT` - $\{x_{k^*}\}$

3 On forme l'ensemble des sommets M$=\left\{x\in\left(X-\text{MARQUÉ}\right)\ /\ A(x_i,x_{k^*})=1\right\}$

[5] Un graphe $G=(X,\Gamma)$ est symétrique si $\forall x,y\in X$ avec $y\in\Gamma(x)$, on a $x\in\Gamma(y)$.

[6] $]n]=\{1,\ldots,n\}$

4 Si M $= \emptyset$ allez en 2.

Sinon

Soit $l^* \in]n]$ tel que $\sum_{j=1}^{n} A(l^*, j) = \min_{x_l \in M} \sum_{j=1}^{n} A(l, j)$

On pose

- M$\leftarrow$ M $- \{x_{l^*}\}$
- $g(x_{l^*}) \leftarrow \min\{q \in \mathbb{N} \ / \ q \neq R(x_{l^*}, x_j) \ \forall j \in]n]\}$ si $x_{l^*} \notin$ MARQUÉ
- On place dans OUVERT à la fin de la liste $\{x_{l^*}\}$: OUVERT $\leftarrow$ OUVERT $\cup \{x_{l^*}\}$
- $\forall i \in]n]$ tel que $A(i, l^*) \neq 0 : R(i, l^*) \leftarrow g(x_{l^*})$
- MARQUÉ $\leftarrow$ MARQUÉ $\cup \{x_{l^*}\}$

5 Si MARQUÉ = X, alors sortie en succès.

Sinon retour au 3.

3.5.1 Application aux fonctions de Grundy

Avant d'examiner l'utilisation des noyaux à l'élaboration d'une solution dans un jeu, nous allons établir les relations qu'elles existent entre noyaux et fonctions de Grundy ce qui permettra d'établir ceux-là à l'aide des celles-ci. Nous avons les deux théorèmes suivants :

THÉORÈME 3.5.5 *Si tout sous-graphe de $G = (X, \Gamma)$ admet un noyau, le graphe $G = (X, \Gamma)$ admet une fonction de Grundy.*

THÉORÈME 3.5.6 *Un graphe symétrique admet une fonction de Grundy si et seulement s'il est sans boucles.*

Considérons, pour terminer, des graphes $G_1, G_2, \ldots, G_n$ qui admettent chacun un noyau. Pour savoir si le graphe-somme admet un noyau, il suffit d'associer le théorème 3.3.4 avec le théorème 3.5.5. En effet, grâce à ces deux théorèmes, nous pouvons constater que si tout sous-graphe du graphe G_i admet un noyau et ceci pour tout $i = 1, 2, \ldots, n$, alors le graphe-somme $G = G_1 + G_2 + \cdots + G_n$ admet une fonction de Grundy g, et par conséquent un noyau $S = \{x \in X_1 \times X_2 \times \cdots \times X_n \ / \ g(x) = 0\}$.

3.6 Stratégies

Pour un jeu $(X, \Gamma, \theta, q_A, q_B)$ une stratégie du joueur A est une application $\sigma : X_A \to X$ telle que $\sigma(x) \in \Gamma(x)$ pour tout $x \in X_A$. On dira que le

joueur A adopte la stratégie σ s'il décide a priori que chaque fois qu'il se trouve à la situation x, il choisira la situation $\sigma(x)$. On notera l'ensemble de stratégies du joueur A par Σ_A.

Si la position initiale $x_0 \in X$ est fixée et si les stratégies σ et τ des joueurs A et B respectivement sont fixées, la partie est entièrement déterminée et l'on désigne par $[x_0;\sigma,\tau]$ l'ensemble des positions rencontrées lors du déroulement du jeu. Le gain de A est :

$$q_A(x_0;\sigma,\tau) = \sup\left\{ q_A(x) \ / \ x \in [x_0;\sigma,\tau] \right\}$$

Le gain de B est

$$q_B(x_0;\sigma,\tau) = \inf\left\{ q_B(x) \ / \ x \in [x_0;\sigma,\tau] \right\}$$

DÉFINITION 3.6.1 *Un couple des stratégies (σ^*,τ^*) est un équilibre relativement à la position $x \in X$ si*

$$\forall \sigma \in \Sigma_A : q_A(x;\sigma,\tau^*) \leq q_A(x;\sigma^*,\tau^*)$$
$$\forall \tau \in \Sigma_B : q_B(x;\sigma^*,\tau) \leq q_B(x;\sigma^*,\tau^*)$$

Un couple qui est en équilibre pour tout sommet $x \in X$, est appelé un équilibre absolu du jeu.

THÉORÈME 3.6.1 (de Zermelo-von Neumann) *Si le graphe $G = (X,\Gamma)$ du jeu est progressivement fini, et si les ensembles $q_A X$ et $q_B(X)$ sont finis, le jeu admet un équilibre absolu (σ^*,τ^*).*

THÉORÈME 3.6.2 *Si le graphe $G = (X,\Gamma)$ du jeu est progressivement fini, il existe pour tout $\varepsilon > 0$ des stratégies σ^* et τ^* telles que :*

$$q_A(x;\sigma,\tau^*) - \varepsilon \leq q_A(x;\sigma^*,\tau^*) \leq q_A(x;\sigma^*,\tau) + \varepsilon \ ; \quad \forall \sigma \in \Sigma_A \text{ et } \forall \tau \in \Sigma_B$$

Ce théorème montre que le joueur A, en utilisant la stratégie σ^* peut garantir un gain d'au minimum $q_A(x;\sigma^*,\tau^*)$ et qu'aucune autre stratégie ne lui permettra de dépasser cette valeur.

DÉFINITION 3.6.2 *Un jeu à deux personnes pour lequel les gains de deux joueurs sont égaux en valeur absolue et de signes contraires, on dira que nous avons un jeu à somme nulle ou un duel.*

La valeur $q_A(\sigma,\tau) = -q_B(\sigma,\tau)$ est appelé le résultat du jeu.

Dans le cadre d'une partie le joueur A cherche à obtenir le plus grand gain et le joueur B à avoir la plus petite perte. Ainsi le meilleur résultat que le joueur A peut obtenir au moyen d'une stratégie σ est donné par

$$\alpha_0 = \sup_{\sigma\in\Sigma_A} \inf_{\tau\in\Sigma_B} q_A(\sigma,\tau)$$

Le meilleur résultat que le joueur B peut obtenir au moyen d'une startégie τ est donné par

$$\beta_0 = \inf_{\tau\in\Sigma_B} \sup_{\sigma\in\Sigma_A} q_A(\sigma,\tau)$$

Nous avons toujours $\alpha_0 \leq \beta_0$. Pour les jeux à somme nulle nous avons le

THÉORÈME 3.6.3 *Soit un jeu à somme nulle. Si le graphe du jeu* $G = (X,\Gamma)$ *est progressivement fini, on a*

$$\alpha_0 = \beta_0$$

3.7 Jeu de nim

Le *jeu de nim* est un jeu de mat à deux joueurs, alternatif et à somme nulle. C'est un jeu très ancien, probablement d'origine chinoise, qui a été appelé ainsi par le mathématicien anglais Charles Leonard Bouton en 1901 d'après le nom allemand *nimm* qui signifie prendre. Le nom chinois du jeu est fan-tan. Pour jouer à ce jeu il faut mettre un certain nombre de pièces de monnaie, d'allumettes ou de jetons en plusieurs rangées ne contenant pas nécessairement le même nombre d'objets. À tour de rôle, chaque joueur enlève une ou plusieurs pièces prises sur une même rangée. Le gagnant est celui qui prend la dernière pièce ou, selon une variante, celui qui ne prend pas la dernière pièce. Cette variante est connue sous le nom de jeu de Marienbad d'après le film d'Alain Resnais *L'année dernière à Marienbad* (1961) où l'acteur Sacha Pitoëff jouait à ce jeu et il prononçait même cette phrase : *Je puis perdre, mais je gagne toujours...* . Nous allons voir qu'il n'en était en rien excessif. Notons qu'aujourd'hui nous avons plusieurs jeux qui sont du type du jeu de nim et par conséquent nous pouvons appliquer à ces jeux les résultats concernant le jeu de nim.

Pour qu'une partie du jeu de nim se termine il faut que le graphe du jeu soit progressivement fini. Nous considérons donc le graphe $G = (X,\Gamma)$ du jeu comme étant progressivement fini. Le problème qui se pose est de

caractériser les positions gagnantes, i.e. les sommets que l'on doit choisir pour gagner la partie quelles que soient les réponses de l'adversaire. Nous avons le théorème suivant :

THÉORÈME 3.7.1 *Si pour un jeu de nim le graphe admet un noyau S et si un joueur choisit un sommet dans S, ce choix lui assure le gain ou la nullité.*

D'après ce théorème pour bien jouer il faut calculer une fonction de Grundy $g(x)$ pour chaque sommet du graphe. Si une telle fonction existe, elle permet de calculer le noyau $S = \{x \in X \ / \ g(x) = 0\}$ pour le graphe du jeu. Soit x_0 la position initiale qui, pour les jeux de nim, est unique. Si $g(x_0) \neq 0$, le joueur A s'assurera le gain en choisissant un sommet $x_1 \in \Gamma(x_0)$ tel que $g(x_1) = 0$. Par contre si $g(x_0) = 0$, le joueur A ne peut espérer de gagner que si son adversaire fasse des erreurs. Nous voyons donc ce qui constitue le fondement de la phrase de S.Pitoëff. Plus précisement nous avons le corollaire :

COROLLAIRE 3.7.1 *Si le graphe d'un jeu est progressivement fini, il existe une fonction de Grundy $g(x)$ et une seule. Tout choix x tel que $g(x) = 0$ est gagnant et tout choix x tel que $g(x) \neq 0$ est perdant.*

Dans le cas où le graphe du jeu est un graphe-somme, nous avons les

THÉORÈME 3.7.2 *Considérons les jeux $G_1, \ldots, G_2$ et supposons qu'ils admettent des fonctions de Grundy $g_1, \ldots, g_n$. Si on joue avec leur somme, on assurera le gain ou la nullité en choisissant les positions $x = (x_1, \ldots, x_n)$ pour lesquelles :*

$$g(x) = g_1(x_1) \dot{+} \cdots \dot{+} g_n(x_n) = 0$$

THÉORÈME 3.7.3 *(Théorème de P.M.Grundy) Si le graphe $G = (X, \Gamma)$ du jeu est progressivement fini et s'il est muni d'une opération $\dot{+}$ vérifiant*

$$\Gamma(x \dot{+} y) = (\Gamma(x) \dot{+} y) \cup (x \dot{+} \Gamma(y))$$

sa fonction de Grundy g vérifie en $z = x \dot{+} y$, la formule

$$g(z) = g(x \dot{+} y) = g(x) \dot{+} g(y)$$

3.8 Résolution des jeux

Après les méthodes de calcul qui permettent d'élaborer une stratégie, nous allons examiner la possibilité d'utiliser pour résoudre un problème de jeu, les méthodes présentées dans le chapitre 1.

Comme nous avons déjà indiqué les feuilles d'un arbre de jeu sont étiquetées en G, P ou N. Ces étiquettes sont les états des feuilles. En fonction de son étiquette une feuille a une valeur – un gain ou une perte – que nous pouvons assimiler au coût du parcours de l'arbre $f(p)$, afin de pouvoir utiliser l'approche du chapitre précédent. Ainsi si on dit que l'état d'une feuille x est $etat(x) = G$, ceci signifie ipso facto que e.g. $f(x) = +1$. De même si $etat(x) = P$, alors $f(x) = -1$ et si $etat(x) = N$, alors $f(x) = 0$. Nous pouvons calculer l'état des autres sommets selon la méthode suivante :

Considérons un sommet $x \in X$ avec $\Gamma(x) \neq \emptyset$.

Si $x \in X_A$

$$etat(x) = \begin{cases} G, & \text{s'il existe un } y \in \Gamma(x) \text{ tel que } etat(y) = G \\ P, & \text{si pour tout } y \in \Gamma(x) \text{ on a } etat(y) = P \\ N, & \text{s'il existe un } y \in \Gamma(x) \text{ tel que } etat(y) = N \\ & \text{et il n'existe pas un } z \in \Gamma(x) \text{ tel que } etat(z) = P \end{cases}$$

Si $x \in X_A$

$$etat(x) = \begin{cases} G, & \text{si pour tout } y \in \Gamma(x) \text{ on a } etat(y) = G \\ P, & \text{s'il existe un } y \in \Gamma(x) \text{ tel que } etat(y) = P \\ N, & \text{s'il existe un } y \in \Gamma(x) \text{ tel que } etat(y) = N \\ & \text{et il n'existe pas un } z \in \Gamma(x) \text{ tel que } etat(z) = P \end{cases}$$

Résoudre un arbre de jeu signifie pouvoir étiqueter sa racine.

Une stratégie pour le joueur A est un sous-arbre $G^+ = (X^+, \Gamma)$ de $G = (X, \Gamma)$ tel que

– $\forall x \in X_A$ avec $\Gamma(x) \neq \emptyset : |\Gamma(x) \cap X^+| = 1$

– $\forall x \in X_B$ avec $\Gamma(x) \subset X^+$

On voit ainsi qu'une stratégie pour le joueur A est un sous-arbre G^+ de G qui contient la racine et qui contient pour chaque sommet MAX non terminal un successeur et tous les successeurs pour un sommet MIN non terminal.

Une stratégie pour le joueur B est un sous-arbre $G^- = (X^-, \Gamma)$ de $G = (X, \Gamma)$ tel que

– $\forall x \in X_B$ avec $\Gamma(x) \neq \emptyset : |\Gamma(x) \cap X^-| = 1$

– $\forall x \in X_A$ avec $\Gamma(x) \subset X^-$

Comme pour le joueur A, une stratégie pour le joueur B est un sous-arbre G^- de G qui contient la racine et qui contient pour chaque sommet MIN non terminal un successeur et tous les successeurs pour un sommet MAX non terminal.

3.9 Stratégie de minimax

Lors d'un jeu on s'interesse particulièrement à établir une stratégie gagnante, c'est-à-dire une stratégie qui garantit que le joueur A gagne indépendamment de la façon dont joue le joueur B. Par conséquent une stratégie gagnante est un sous-arbre G^+ dont toutes les feuilles sont étiquetées G.

Considérons maintenant deux stratégies une pour le joueur A : $G^+ = (X^+, \Gamma)$ et une pour le joueur B : $G^- = (X^-, \Gamma)$. Dans ce cas les deux ensembles de sommets X^+ et X^- des sous-arbres ont un sommet en commun qui est d'ailleurs un sommet terminal : $\mid X^+ \cap X^- \mid = 1$ et si $x \in X^+ \cap X^-$, alors $\Gamma(x) = \emptyset$. Ce sommet commun sera noté (X^+, X^-).

Nous savons que le joueur A cherchera à maximiser ses gains et, symétriquement, le joueur B à minimiser ses pertes. Comme le jeu est à information complète, toutes les informations concernant les différentes situations du jeu, les gains et les pertes sont parfaitement connues à chaque moment. Dans ce cas, le joueur A pour remplir son objectif, chaque fois qu'il aura le trait, il examinera les situations qu'il soit possible d'atteindre à partir de la situation actuelle, il retiendra celles qui minimisent les pertes de son adversaire[7] et, parmi celles-ci, il choisira la ou les situations qui maximisent son propre gain. Donc ce faisant, le joueur A retient tous les successeurs d'un sommet MIN et au moins un successeur d'un sommet MAX. Par conséquent le joueur A, lorsqu'il jouera, cherchera ses situations dans le sous-arbre G^+. Pour des raisons symétriques le joueur B cherchera ses situations dans le sous-arbre G^-. Si les deux joueurs jouent de cette façon, ils vont aboutir au sommet terminal $x^* \in X^+ \cap X^-$ et par conséquent l'étiquette de la racine sera l'étiquette de x^* : $etat(\alpha) = etat(x^*)$ et $f(\alpha) = f(x^*)$.

En formalisant nous avons :

– Pour le joueur A : $f(\alpha) = \max_{x \in X^+} \min_{x \in X^-} (X^+, X^-)$, c'est-à-dire le joueur

[7] Le joueur A est obligé de retenir toutes ces situations car il doit faire l'hypothèse que son adversaire jouera de façon rationnelle et cherchera à minimiser ses pertes, donc il choisira, quand il aura le trait, la situation qui convient pour que cette minimisation ait lieu.

A, chaque fois qu'il aura le trait, examinera les situations qu'il peut atteindre à partir de la situation actuelle, il retiendra celles qui minimisent les gains de son adversaire et parmi celles-ci il choisira les stratégie(s) qui maximise(nt) son propre gain.

– Pour le joueur B : $f(\alpha) = \min_{x \in X^-} \max_{x \in X^+} (X^+, X^-)$

Cette stratégie est appelée *stratégie minimax* et elle est mise au point initialement par J. von Neumann. La valeur $f(\alpha)$ est appelée *valeur minimax*.

Il est évident que pour pouvoir étiqueter la racine il faut développer tout l'arbre. Pour la plupart de jeux une telle démarche est impossible à cause de l'espace mémoire et du temps nécessaire pour l'effectuer. Nous pouvons donc de décider de limiter la recherche jusqu'à une profondeur donnée, e.g. $L \geq 1$. Ainsi la méthode consiste à

– développer l'arbre jusqu'à une profondeur L ;
– déterminer les meilleurs mouvements en utilisant l'arbre incomplet développé auparavant.

Pour écrire l'algorithme qui établit la stratégie minimax et calcule la valeur minimax d'un jeu, nous avons besoin de deux fonctions auxiliaires :

– `eval(POSITION, JOUEUR)` qui retourne la valeur du sommet `POSITION` pour le joueur `JOUEUR`. On suppose que quand on appelle `EVAL` c'est pour calculer la valeur maximale. Quand donc, on revient en arrière pour calculer la valeur du sommet qui se trouve au niveau précédent, il faut multiplier cette valeur par -1 afin de tenir compte du point de vue de l'autre joueur.
– `testChemin(POSITION,JOUEUR)` qui retourne `VRAIE` s'il faut arrêter l'exploration à la profondeur actuelle et `FAUSSE` s'il faut continuer.

La procédure `minimax` que nous allons présenter par la suite est une procédure recursive et elle retourne le chemin calculé et sa valeur actualisée pour le joueur qu'il l'a appelée.

Procédure `minimax(POSITION, CHEMIN, VALEUR, JOUEUR)`

1 Si `testChemin(POSITION, JOUEUR)` alors retourne

`VALEUR` ← `eval(POSITION, JOUEUR)` ;

`CHEMIN` ← $\emptyset$ Dans ce cas il n'y a pas de suite du chemin depuis `POSITION` et sa valeur est `VALEUR`.

Sinon

2 Développement du sommet `POSITION` :

2.1 SUCCESSEURS $\leftarrow \Gamma$(POSITION)

2.2 MeilleureValeur $\leftarrow -\infty$

3 Tant que SUCCESSEURS $\neq \emptyset$ faire

Début

3.1 Mettre dans Succ le premier élément de la liste SUCCESSEURS

Succ $\leftarrow$ car(SUCCESSEURS)

3.2 Supprimer de la liste SUCCESSEURS son premier élément

SUCCESSEURS $\leftarrow$ SUCCESSEURS - {Succ}

3.3 SuccSuivant $\leftarrow$ minimax(Succ, CHEMIN, VALEUR, $\overline{JOUEUR}$)

avec $\overline{JOUEUR}$ l'adversaire de JOUEUR.

3.4 NouvValeur $\leftarrow$ -VALEUR

3.5 Si NouvValeur $>$ MeilleureValeur alors

3.5.1 MeilleureValeur $\leftarrow$ NouvValeur

3.5.2 Mettre au début de la liste CHEMIN le sommet Succ

MeilleurChemin $\leftarrow$ {Succ} $\cup$ CHEMIN

Fin

4 Mise à jour de VALEUR et CHEMIN

VALEUR $\leftarrow$ MeilleureValeur

CHEMIN $\leftarrow$ MeilleurChemin

3.10 L'algorithme $\alpha - \beta$

L'algorithme de minimax nécessite le développement de tous les sommets de l'arbre. On pourrait envisager une démarche qui arrête la progression du développement d'une branche de l'arbre du jeu si on savait que les valeurs de ses sommets ne sont pas interessantes pour la solution. Par exemple supposons que le joueur A a parcouru une branche et il a évalué sur cette branche une valeur maximale pour le jeu. S'il est en train de parcourir une autre branche qui a un sommet commun avec la branche déjà parcourue et si ce sommet est un sommet Max, alors si une feuillle de cette branche a une valeur inférieure à la valeur maximale déjà obtenue, il n'est pas nécessaire de continuer le développement de cette branche car le joueur MAx ne l'utilisera pas étant donné que sur cette branche la valeur du jeu est inférieure à la valeur obtenue sur la barnche précédente.

Pour appliquer cette reamarque on associera à cahque sommet, en plus de sa valeur, deux autres quantités, à savoir :

– la *valeur alpha* qui est une approximation par défaut de la vraie valeur du sommet. Elle est initialisée à $-\infty$. Son calcul se fait comme suit

$$\alpha(x) = \begin{cases} f(x), & \text{si } \Gamma(x) = \emptyset \\ \max\limits_{y \in E(x)} f(y), & \text{si } \Gamma(x) \neq \emptyset \text{ et } x \text{ sommet MAX} \\ \alpha(z) \text{ ; avec } z \in \Gamma^{-1}(x), & \text{si } \Gamma(x) \neq \emptyset \text{ et } x \text{ sommet MIN} \end{cases}$$

– la *valeur beta* qui est une approximation par excès de la vraie valeur du sommet. Elle est initialisée à $+\infty$. Son calcul se fait comme suit

$$\beta(x) = \begin{cases} f(x), & \text{si } \Gamma(x) = \emptyset \\ \min\limits_{y \in E(x)} f(y), & \text{si } \Gamma(x) \neq \emptyset \text{ et } x \text{ sommet MIN} \\ \alpha(z), \text{ avec } z \in \Gamma^{-1}(x), & \text{si } \Gamma(x) \neq \emptyset \text{ et } x \text{ sommet MAX} \end{cases}$$

où on a noté par $E(x) \subseteq \Gamma(x)$ l'ensemble de successeurs du sommet x développés à une étape de l'algorithme.

Les valeurs alpha et beta d'un sommet x encadrent sa valeur $f(x)$, i.e. $\alpha(x) \leq f(x) \leq \beta(x)$. Du fait que sur le parcours d'une branche il y a alternance des sommets MAX et MIN, il s'ensuit que $\alpha(x) = \max\limits_{y \in \Gamma(x)} \beta(y)$ si x est un sommet MAX et $\beta(x) = \min\limits_{y \in \Gamma(x)} \alpha(x)$ si x est un sommet MIN. Par conséquent nous avons pour un sommet x de type MAX $\alpha(y) \leq \beta(x)$ et $\beta(y) \leq \alpha(x)$ où $y \in \Gamma(x)$.

Lors de l'exécution d'un algorithme qui utilise ces valeurs on peut « couper » le développement d'une branche si

– sur un sommet $y \in \Gamma(x)$ de type MAX, on a $\alpha(y) \geq \beta(x)$ car sur le sommet x, qu'il est du type MIN, le joueur B ne choisira pas cette branche dans la mesure où ce choix ne diminue pas ses pertes ;
– sur un sommet $y \in \Gamma(x)$ de type MIN, on a $\beta(y) \leq \alpha(x)$ car sur le sommet x, qu'il est du type MAX, le joueur A ne choisira pas cette branche qui n'augmente pas ses gains.

Ainsi l'algorithme alpha-beta ne développe pas les successeurs d'un sommet dès qu'on constate que ce sommet ne pourra pas faire partie de l'arbre du jeu.

Pour l'algorithme alpha-beta on construit une fonction qui retourne la valeur de la racine.

Données : $\alpha(x) = -\infty$, $\beta(x) = +\infty$ pour tout sommet x du graphe.

$f(x)$ pour tout sommet terminal

`fonction alpha_beta(x,` $\alpha(x)$, $\beta(x)$, $f(x)$, `JOUEUR)`

1 Si $\Gamma(x) = \emptyset$, alors retourner $f(x)$.

Sinon

2 Si `JOUEUR` = MAX faire

Début

2.1 $\alpha(x) \leftarrow -\infty$

2.2 Si $\Gamma(x) \neq \emptyset$ faire

Début

2.2.1 Extraire un successeur y de x : $\Gamma(x) \leftarrow \Gamma(x)$ -$\{y\}$

2.2.2 $\alpha(y) \leftarrow \alpha(x)$

2.2.3 `Valeur` $\leftarrow$ `alpha_beta(y,` $\alpha(y)$, $\beta(y)$, $f(x)$, `MIN)`

2.2.4 $\alpha(y) \leftarrow \max\{$`Valeur`, $\alpha(y)\}$

2.2.5 Si $\alpha(y) \geq \beta(x)$, alors retourner la valeur $\alpha(x)$.

Sinon

Fin (du parcours sur $\Gamma(x)$)

3 Retourner la valeur $\alpha(y)$.

Fin (`JOUEUR` = MAX)

4 Sinon (i.e. `JOUEUR` = MIN) faire

Début

4.1 $\beta(x) \leftarrow +\infty$

4.2 Si $\Gamma(x) \neq \emptyset$ faire

Début

4.2.1 Extraire un successeur y de x : $\Gamma(x) \leftarrow \Gamma(x)$ -$\{y\}$

4.2.2 $\beta(y) \leftarrow \beta(x)$

4.2.3 `Valeur` $\leftarrow$ `alpha_beta(y,` $\alpha(y)$, $\beta(y)$, $f(x)$, `MAX)`

4.2.4 $\beta(y) \leftarrow \min\{$`Valeur`, $\beta(y)\}$

4.2.5 Si $\alpha(x) \geq \beta(y)$, alors retourner la valeur $\beta(x)$.

Sinon

Fin (du parcours sur $\Gamma(x)$)

5 Retourner la valeur $\beta(y)$

Fin (JOUEUR = MIN)

Fin de la fonction

En moyenne l'algorithme alpha-beta réalise une économie de 30% sur le nombre de sommets développés. Cette économie peut être atteinte, voire dépassée, si on ne parcourt pas l'arbre au hasard. Pour élaborer une tactique pour ce parcours, trouvons d'abord, une condition nécessaire et suffisante pour qu'un sommet soit développé par l'algorithme alpha-beta.

Soit $x \in X$ un sommet du graphe du jeu et supposons qu'il existe un chemin de la racine à x. On définit les deux quantités suivantes :

- $A(x)$ = la plus grande valeur minimax calculée sur les successeurs qui sont à gauche de x de tous les prédecesseurs MAX de x ;
- $B(x)$ = la plus petite valeur minimax calculée sur les successeurs qui sont à gauche de x de tous les prédecesseurs MIN de x.

Le sommet x sera développé par l'algorithme alpha-beta si et seulement si

$$A(x) < B(x)$$

A contrario un critère pour que le sommet x ne soit pas développé par l'algorithme alpha-beta est d'avoir $A(x) \geq B(x)$. Selon ce critère, si on veut minimiser le nombre de sommets à développer par l'algorithme alpha-beta, il faut commencer par les chemins qui assurent des grandes valeurs pour minimax.

Il existe d'autres algorithmes, en particulier les algorithmes SSS*, SCOUT et PVS, qui minimisent encore plus le nombre de sommets à développer. L'élève interessé pourra se reporter à la bibliographie.

3.11 Exercices

EXERCICE 3.1 Considérons le jeu de fan-tan à deux joueurs suivant :

- Il y a deux rangées d'allumettes, une contenant trois allumettes et l'autre une.
- Chaque joueur à son tour peut prendre d'une rangée autant d'allumettes qu'il souhaite et au minimum une.
- Le joueur qui prend en dernier gagne la partie.

(1) Faire le graphe du jeu et calculer une fonction de Grundy pour ce jeu.

(2) À l'aide de la fonction de Grundy, élaborer une stratégie qui permet au joueur MAX de gagner à coup sûr.

(3) Établir l'arbre de solution pour ce jeu en utilisant l'algorithme du minimax.

(4) Même question mais en utilisant l'algorithme alpha-beta.

(5) Est-il possible d'améliorer l'application de cet algorithme ?

EXERCICE 3.2 Considérons un tas de m allumettes et deux joueurs. Chaque joueur à son tour peut prendre $k = 3$ allumettes à la fois. Le joueur qui prend en dernier gagne la partie.

(1) Établir le graphe du jeu.

(2) Donner les conditions qui permettent à ce graphe d'avoir une fonction de Grundy. Calculer cette fonction.

(3) Évaluer la fonction de Grundy à l'aide du numéro du sommet et de la valeur de k. N.B. Un sommet aura comme numéro n, si ol contient n allumettes.

(4) Donner le noyau du graphe et trouver une position initiale gagnante pour le joueur A lors du premier coup.

(5) Généralisation. On considère qu'il y ait p tas contenant $m_1, m_2, \ldots, m_p$ allumettes et on peut prendre chaque fois $k_1, k_2, \ldots, k_p$ allumettes.

Calculer la fonction de Grundy du graphe et trouver une position gagnante pour le joueur A lors du premier coup.

Application : $p = 3$, $m_1 = 4$, $m_2 = 7$, $m_3 = 5$, $k_1 = 5$, $k_2 = 3$, $k_3 = 2$.

3.12 Bibliographie

Pour la redaction de ce chapitre nous avons utiliser essentiellement les livres suivants :

[1] R. B. BANERJII : *Articial Intelligence*, North-Holland, 1980

[2] C. BERGE : *Théorie générale des jeux à n personnes*, Gauthier-Villars, 1957

[3] —— : *Théorie des graphes et ses applications*, Dunod, 1958

[4] N. J. NILSSON : *Principes d'intelligence artificielle*, Cepadues, 1988

[5] J. PEARL : *Heuristics*, Addison-Wesley, 1984

[6] J. VON NEUMANN & O. MORGENSTERN : *Theory of games and economic behavior*, Wiley, 1964

4

ARBRES DE DÉCISION

Les concepts que nous avons examiné au chapitre précédent étaient des concepts relatifs à des objets qui appartenaient à la même classe, c'est-à-dire les objets qui composaient le concept étaient homogènes. Par exemple un livre dans une bibliothèque. Il est possible d'avoir des situations où les objets – tout en ayant une certaine homogénéité – appartiennent à plusieurs classes. Par exemple un article qui est publié soit dans un livre collectif (première classe), soit dans une revue scientifique éditée en Angletterre (deuxième classe). Dans ce cas le concept apparaît comme une disjonction des conjonctions : *(article dans un livre)* $\vee$ *(article dans une revue* $\wedge$ *pays d'édition : Angleterre)*.

La mise en évidence et l'apprentissage des concepts de cette nature ne peut pas se faire en utilisant l'algorithme d'élimination des candidats. Il nous faut des algorithmes capables de détecter et de déterminer les différentes classes auxquelles se décompose le concept et, aussi, capables d'attribuer une classe à un objet. En d'autres termes, il faut pouvoir obtenir des généralisations du concept qui doivent couvrir toutes les classes de ses objets. Il s'agit, en partant des propriétés élémentaires des objets – qui sont considérées comme constituant des classes élémentaires – d'agréger, à l'aide d'une procédure ascendante, ces propriétés en construisant des classes des pro-

priétés de plus en plus complexes jusqu'à obtenir une seule classe qui sera celle qui représente le concept. La procédure que nous venons de décrire est en réalité ce qu'en Analyse de Données on appelle classification ascendante hiérarchique et pour laquelle il y a beaucoup d'algorithmes qui utilisent des données numériques.

Quand nous avons des données symboliques ou un melange des données symboliques et numériques, la démarche est toute autre. Sur la base des exemples avec des données symboliques, nous ne pouvons pas établir un espace des attributs muni d'une métrique qui nous permetrait de trouver les deux attributs les plus proches pour former la première classe et poursuivre ensuite de la même façon. Nous allons donc procéder autrement. On commence par la racine, qu'on associe à un attribut, et on développe pour chaque valeur de cet attribut un arc qui part de la racine et qui aboutit à un autre attribut. Si les exemples sont correctement classés par les feuilles de l'arbre, on arrête l'algorithme. Sinon, on continue. Donc les algorithmes que nous verrons construisent progressivement un arbre de décision de haut en bas. Chaque sommet de l'arbre repésente un attribut et les arcs qui partent d'un sommet représentent les différentes valeurs de l'attribut du sommet. Le chemin qui mène de la racine à une feuille s'établit en choisissant un sommet-attribut, soit au hasard, soit selon un critère.([1]).

4.1 L'algorithme de classification

Il s'agit d'un algorithme non incremental qui produit une hypothèse – que doit refléter le concept – sous la forme d'un arbre de décision. À chaque étape de l'algorithme on rajoute à l'arbre un sommet qui représente un attribut choisi de façon aléatoire. L'ensemble des exemples est testé par rapport à ce sommet. Si les valeurs du concept pour tous les exemples et pour ce sommet sont identiques, le sommet est étiqueté avec la valeur du concept et devient une feuille. Sinon, on repète l'algorithme récursivement jusqu'au moment où tous les sommets engendrés sont des feuilles.

Formellement l'algorithme se présente comme suit :

Algorithme de classification : fonction recursive `classification(`X`,` A_X`,` V`)`. Le retour de cette fonction est l'arbre de décision T.

[1] Remarquons qu'en 1973 J.-P.Benzécri a établi un algorithme de construction d'un arbre des feuilles vers la racine en utilisant des mesures de la théorie de l'information. (cf J.-P.Benzécri : L'analyse des données, vol. 1, Dunod 1973, pp.207-236).

0 Données : Un ensemble d'exemples X. Un ensemble d'attributs A_X pour les exemples. Pour chaque attribut $a \in A_X$ l'ensemble de ses valeurs $V(a;X)$.

1 Si pour tout $\mathbf{x} \in X$ nous avons la même valeur pour $v(c;\mathbf{x})$, alors créer la racine avec comme étiquette cette valeur et arrêt de l'algorithme.

Sinon

2 Choisir au hasard un attribut $a \in A_X$ et soit $m \leftarrow \mid V(a;X) \mid$.

3 $A_X \leftarrow A_X - \{a\}$.

4 On crée le sommet relatif à cet attribut a.

5 On partage l'ensemble X des exemples en sous-ensembles $X_1, X_2, \ldots, X_m$ selon les différentes valeurs $v(a;\mathbf{x}) \in V(a;X)$ pour l'attribut a, i.e. $X_i = \left\{\mathbf{x} \in X \;/\; v(a;\mathbf{x}) = v_{a,i}\right\}$.

6 Pour chacun des sous-ensembles X_i construits, on crée un sommet successeur du sommet de l'attribut a et un arc de a vers X_i, valué à la valeur $v(a;\mathbf{x})\,;\mathbf{x} \in X_i$.

7 Appel récursif de la fonction `classification`

$T_i \leftarrow$ `classification(`X_i`,` A_{X_i}`,` V`)` pour $i = 1, 2, \ldots, m$.

8 $T \leftarrow$ union des arbres $T_1, T_2, \ldots, T_m$.

9 Retourner T.

En choisissant les attributs, lors de l'étape 2, dans un ordre différent, on construit des arbres différents avec, éventuellement, des profondeurs différentes. Or ce qui est intéressant pour la détermination du concept, ce sont des arbres de profondeur minimale car dans ce cas le concept s'exprime plus simplement. Il devient donc évident qu'il faut effectuer le choix de l'attribut selon un critère susceptible de minimiser la profondeur de l'arbre de décision. Dans la bibliographie on trouve plusieurs critères de cette nature. Nous commençons par présenter un critère fondé sur la théorie de l'information. Il a été mis au point par Quinlan et il l'a utilisé dans son algorithme ID3.

4.2 l'information apportée par les attributs

L'idée de base pour la construction d'un arbre de décision est la suivante : Pour classifier correctement un exemple, c'est-à-dire pour identifier correctement son concept-classe parmi un ensemble des concepts-classes, il

ne faut pas examiner tous les attributs mais seulement ceux qui sont les plus pertinents pour la classe à laquelle appartient cet exemple. Il nous faut donc, chaque fois que nous allons choisir un attribut pour un exemple, une mesure de pertinence de cet attribut. L'analyse de données offre une pléthore de critères qui s'appuient sur des informations numériques. Comme en IA nous avons aussi des informations symboliques, nous allons élaborer un critère qui refletera la capacité de l'attribut de classer les exemples d'apprentissage aux différentes classes que se décompose le concept à apprendre.

Considérons un ensemble X d'exemples d'un concept-cible qui peut prendre q valeurs différentes, avec $q \geq 2$. Soit X^{c_k} l'ensemble des exemples dont le concept-cible c prend la k-ième valeur $v_{c_k} : X^{c_k} = \{\mathbf{x} \in X \ / \ v(c;\mathbf{x}) = v_{c_k}\}$. L'information véhiculée par X est donnée par la formule :

$$I(X) = -\sum_{k=1}^{q} \frac{|\, X^{c_k} \,|}{|\, X \,|} \log_2 \frac{|\, X^{c_k} \,|}{|\, X \,|}$$

Pour mesurer l'apport d'un attribut au classement des exemples, on utilise l'information fournie par le partage des exemples selon cet attribut et selon le concept-cible. Ainsi, soit $a \in A_X$ un attribut et soit $V(a;X) = \{v_{a,1}, v_{a,2}, \ldots, v_{a,n_a}\}$ l'ensemble de ses valeurs. Dans ce cas nous avons $q \times n_a$ classes qui se forment par l'association de q classes des exemples selon le concept-cible avec les n_a classes de ces mêmes exemples selon les valeurs de l'attribut a. Bien évidemment certaines de ces classes peuvent être vides. On note $X_{a,i}^{c_k}$ l'ensemble des exemples de X qui ont pour le concept-cible la valeur v_{c_k} et la valeur $v_{a,i} \in V(a;X)$ pour l'attribut $a \in A_X : X_{a,i}^{c_k} = \{\mathbf{x} \in X \ / \ v(c;\mathbf{x}) = v_{c_k}$ et $v(a;\mathbf{x}) = v_{a,i}\}$ et par $X_{a,i}$ qui ont la valeur $v_{a,i} \in V(a;X)$ pour l'attribut $a \in A_X : X_{a,i} = \{\mathbf{x} \in X \ / \ v(a;\mathbf{x}) = v_{a,i}\}$. L'information sur X induite par l'attribut a et le concept-cible c est donnée par

$$I(X;a) = -\sum_{v_{a,i} \in V(a;X)} \frac{|\, X_{a,i} \,|}{|\, X \,|} \sum_{k=1}^{q} \frac{|\, X_{a,i}^{c_k} \,|}{|\, X_{a,i} \,|} \log_2 \frac{|\, X_{a,i}^{c_k} \,|}{|\, X_{a,i} \,|}$$

qui peut, encore, s'écrire :

$$I(X;a) = -\frac{1}{|\, X \,|} \sum_{v_{a,i} \in V(a;X)} \sum_{k=1}^{q} |\, X_{a,i}^{c_k} \,| \log_2 \frac{|\, X_{a,i}^{c_k} \,|}{|\, X_{a,i} \,|}$$

Si on note

$$I(X_{a,i};a) = -\sum_{k=1}^{q} \frac{|\, X_{a,i}^{c_k} \,|}{|\, X_{a,i} \,|} \log_2 \frac{|\, X_{a,i}^{c_k} \,|}{|\, X_{a,i} \,|}$$

l'information apportée par le i-ième valeur de l'attribut a[2], la formule précédente s'écrit

$$I(X;a) = \sum_{v_{a,i} \in V(a;X)} \frac{|\, X_{a,i} \,|}{|\, X \,|} I(X_{a,i};a)$$

Le *gain d'information* $K(X,a)$ de l'attribut $a \in A_X$ est :

$$K(X,a) = I(X) - I(X;a);\ a \in A_X$$

Le gain d'information est ce qui est retirée de l'information sur X lorsqu'on connaît la valeur de l'attribut a pour chaque exemple et il est la mesure utilisée par ID3 afin de choisir le meilleur attribut à chaque étape de la construction de l'arbre.

4.3 L'algorithme ID3

L'algorithme *Inductive Decision Tree* – ID3, proposé par Quinlan en 1983, est en réalité l'algorithme de classification avec l'étape 2 modifiée comme suit :

2' Calculer pour chaque attribut $a \in A_X$ le gain de l'information $K(X,a)$.

Choisir l'attribut $a^* \in A_X$ tel que $K(X,a^*) = \max_{a \in A_X} K(X,a)$.

Posons $n \leftarrow |\, V(a^*;X) \,|$.

La mesure du gain de l'information pour le choix de l'attribut à examiner ne garantit pas que l'arbre de décision sera le moins profond. Elle constitue tout simplement une heuristique pour aboutir à des arbres de décision qui ont des profondeurs petites.

L'algorithme ID3 peut s'arrêter par manque d'exemples. Dans ce cas, pour continuer, il faut trouver d'autres exemples. Il est possible aussi que lors de son déroulement, l'algorithme examine un exemple dont les valeurs

[2] qui est la formule de Shannon et qui exprime la quantité moyenne d'information apportée à l'ensemble X par la i-ième valeur de l'attribut a (cf. poly du cours de la théorie d'information).

des attributs contredisent l'arbre de décision construit sur les exemples précédents. La solution consiste à ignorer pour cet exemple les valeurs d'un ou plusieurs attributs afin de rétablir la compatibilité avec l'arbre de décision. Ce faisant, on considère implicitement que les réponses pour les attributs non pris en compte pour cet exemple étaient eronnées. Nous voyons donc que cet algorithme peut fonctionner même avec des données qui contiennent des erreurs ou des imperfections contrairement à l'algorithme d'élimination des candidats.

La fin normale de l'algorithme arrive quand dans l'arbre de décision aucun sommet terminal ne peut être développé davantage. Il est possible que dans ce cas il reste des exemples que l'algorithme n'a pas examiné. Et il possible que certains parmi ces exemples ne soient pas couverts par l'arbre de décision. Bien sûr nous pouvons agir comme précédement, à savoir relaxer certains attributs pour que l'arbre de décision couvre aussi ces exemples. Remarquons simplement que nous n'avons aucune indication quant au bien fondé de cette démarche. En effet il est possible que les données en question soient erronées. Mais il est possible aussi qu'en suivant l'algorithme nous avons trop avancé en decoupage des classes. Nous avons ainsi des classes petites dont la structure ne reflète pas un concept et ses ramifications générales, mais plutôt des exemples particuliers de ce concept. Nous sommes donc devant un problème de *sur-apprentissage*, problème qui est sous-jacent à tout algorithme d'apprentissage symbolique et sous-symbolique. Il n'y a pas de solution générale à ce problème, seulement des recettes globales et/ou locales que nous pouvons essayer d'appliquer.

Une recette globale est l'utilisation concomitante d'une population d'exemples de test avec la population d'exemples d'apprentisssage. Nous exécutons l'algorithme en utilisant les exemples d'apprentissage et chaque fois qu'on établit une branche de l'arbre de décision, on l'applique sur les exemples de test. La structure de l'algorithme est telle que chaque nouvelle branche de l'arbre apporte plus de précision sur les exemples d'apprentissage. Normalement la même chose doit se passer aussi avec les exemples de test. Mais à partir d'un certain moment la précision sur les exemples de test n'augmente pas et elle a même tendance à diminuer. Dans ce cas, nous sommes en train de rentrer dans le domaine du sur-apprentissage. La poursuite de l'algorithme fera que l'apprentissage du concept évoluera vers l'assimilation des cas particuliers – qui sont présents dans la population d'apprentissage – au détriment d'autre cas particuliers – qui sont présents dans la population de test – par une décomposition excessive en classes des attributs

qui ne sont pas significatifs pour le concept.

Une recette locale consiste en l'utilisation du test du χ^2 pour les tableaux de contingence. L'analyse précédente montre qu'il faut décider de ne pas poursuivre la décomposition en classes d'un attribut si cet attribut n'est pas significatif pour le concept. Nous pouvons admettre que pour un attribut non significatif la distribution des valeurs de cet attribut sera aléatoire. On construit donc, pour un attribut $a \in A_X$ qui a n_a valeurs différentes, un tableau de contingence $X(a;c)$ avec n_a lignes et q colonnes, chaque colonne correspondant à une valeur du concept-cible c. Chaque case (i,k) de ce tableau contient la valeur $X_{a,i}^{c_k}$ qui est le nombre d'exemples qui ont la valeur $v_{a,i}$ pour l'attribut a et la valeur v_{c_k} pour le concept-cible c. L'hypothèse nulle pour ce tableau est l'hypothèse selon laquelle la distribution de valeurs pour l'attribut est indépendante de celle des valeurs pour le concept-cible. La valeur de χ^2 pour ce tableau est donnée par

$$\chi^2 = \sum_{v_{a,i} \in V(a;X)} \sum_{k=1}^{q} \frac{\left(\mid X_{a,i}^{c_k} \mid - \frac{|X_a^{c_k}| \cdot |X_{a,i}|}{|X|} \right)^2}{\frac{|X_a^{c_k}| \cdot |X_{a,i}|}{|X|}} = \mid X \mid \left(\sum_{v_{a,i} \in V(a;X)} \sum_{k=1}^{q} \frac{\mid X_{a,i}^{c_k} \mid^2}{\mid X_a^{c_k} \mid \cdot \mid X_{a,i} \mid} - 1 \right)$$

Si χ^2 est inférieure à la valeur de $\chi^2_{\alpha;(q-1),(n_a-1)}$ qui pourra être obtenue par les tables, alors l'hypothèse nulle est acceptée avec un risque de $\alpha\%$([3]) et on doit donc arrêter la poursuite de l'algorithme sur cette branche.

Cette discussion nous amène à nous poser une autre question concernant cette fois-ci le nombre n d'exemples qu'il soit nécessaire de disposer pour pouvoir appliquer convenablement l'algorithme ID3. Il n'y a pas une réponse simple à cette question. Dans [2, pp. 553-556] il y a un calcul approximatif dans un cas particulier que nous reprenons ici.

Soit X l'ensemble des exemples à utiliser pour établir un concept c. On suppose que cet ensemble d'exemples est un échantillon issu d'une population $D(c)$ d'exemples relatifs au concept c. Soit $H(X)$ l'espace des hypothèses induit par X. On suppose que la vraie hypothèse h^* qui permet d'apprendre complètement et correctement le concept c se trouve dans $H(X)$. Pour une hypothèse quelconque $h \in H(X)$, l'erreur par rapport à la vraie

[3] cf. E.L.Crow & al. : *Statistics manual*, Dover, 1960, pp. 97-100.

hypothèse serait

$$e(h) = \Pr\left\{h(\mathbf{x}) \neq f(\mathbf{x}) \ / \ \mathbf{x} \in \mathbf{D}(c)\right\}$$

Bien sûr il s'agit d'une probabilité théorique mais qui peut être estimée en utilisant divers ensembles d'exemples X.

Une hypothèse h est approximativement correcte si $e(h) \leq \varepsilon$ avec ε une petite constante positive. Si maintenant on a une hypothèse $h^{\wr}$ mauvaise, alors son erreur est $e\left(h^{\wr}\right) > \varepsilon$. Par conséquent la probabilité pour qu'elle satisfasse à un exemple $\mathbf{x} \in X$ est $\leq (1-\varepsilon)$. Nous avons donc que la probabilité pour qu'elle satissfasse les n exemples de X est

$$\Pr\left(h^{\wr} \text{ satisfait tous les exemples de } X\right) \leq (1-\varepsilon)^n$$

Ainsi la probabilité pour que $H(X)$ contient une mauvaise hypothèse qui est cohérente avec l'échantillon d'exemples X est

$$\Pr(H(X) \text{ contient une hypothèse mauvaise et cohérente}) \leq \mid H(X) \mid (1-\varepsilon)^n$$

Nous pouvons fixer comme objectif que cette probabilité soit plus petite qu'un nombre positif petit δ, à savoir que

$$\mid H(X) \mid (1-\varepsilon)^n \leq \delta$$

d'où on obtient finalement une borne inférieure pour le nombre $n(\varepsilon, \delta)$

$$n(\varepsilon, \delta) \geq \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + \ln \mid H(X) \mid\right)$$

qui s'appelle la *complexité de l'échantillon*. Si donc $\mid X \mid \geq n(\varepsilon, \delta)$, alors l'hypothèse élaborée par l'algorithme ID3 a, avec une probabilitè au moins égale à $1-\delta$, une erreur au plus ε, donc elle est approximativement correcte.

La valeur de $\mid H(X) \mid$ est, même pour des cas avec un nombre d'attributs modeste, très grande. Par exemple si on considère que les m attributs sont tous binaires et le concept-cible est aussi binaire, alors nous avons $\mid H(X) \mid = 2^{2^m}$. Faisons encore une hypothèse simplificatrice. Nous avons, au début de ce chapitre, vu que le concept que nous voulons élaborer pour décrire une situation présentée par des exemples, peut se mettre sous la forme logique des disjonctions des conjonctions, e.g. $(p_{11} \wedge \cdots \wedge p_{1m_1}) \vee \cdots \vee (p_{n1} \wedge \cdots \wedge p_{nm_n})$ où p_{ij} est un littéral. Supposons que chaque groupe de conjonctions $(p_{i1} \wedge \cdots \wedge p_{im_i})$ contient K littéraux. Alors dans ce cas nous

avons pour la complexité de l'échantillon :

$$n(\varepsilon,\delta) \geq \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + O\left(m^K \log_2 m^K\right)\right)$$

qui est inférieur à la précédente valeur pour un K petit.

4.4 Algorithme C4.5

Il s'agit d'une amélioration de l'algorithme ID3, proposé aussi par Quinlan en 1993.

L'amélioration concerne le calcul du meilleur attribut a^*. ID3 établit a^* sur la base du plus grand gain d'information. Mais si, pour un attribut, ses valeurs sont équiréparties dans l'ensemble des exemples, alors cet attribut réalise le maximum du gain d'information tout en ayant le plus petit pouvoir discriminatoire. Afin d'éviter des telles situations, on calcule pour chaque attribut $a \in A_X$, qui peut prendre n_a valeurs différentes, sa fonction d'information par rapport à X :

$$I(a;X) = -\sum_{v_{a,i} \in V(a;X)} \frac{|X_{a,i}|}{|X|} \log_2 \frac{|X_{a,i}|}{|X|}$$

Le choix du meilleur attribut se fera sur la base du *gain modifié d'information*

$$G(X,a) = \frac{K(X,a)}{I(a;X)};\ a \in A_X$$

L'algorithme C4.5 peut fonctionner avec des exemples qui ont des données manquantes, en calculant le gain modifié d'information uniquement pour les exemples pour lesquels les valeurs des attributs sont définis. Il peut aussi fonctionner quand les valeurs des attributs se trouvent dans un intervalle continu des valeurs.

4.5 Exercices

EXERCICE 4.1 *Pour le diagnostic entre trois infections A, B et C on utilise une analyse morphologique des cellules bactériennes prélevées dans le sang des patients.*

L'ensemble d'apprentissage est donnée par le tableau suivant :

No	Nb noyaux	Nb flagelles	Coloration	Paroi	Infection
1	1	1	Pâle	Fine	A
2	2	1	Pâle	Fine	A
3	1	1	Pâle	Épaisse	A
4	1	1	Foncée	Fine	A
5	1	1	Foncée	Épaisse	A
6	2	2	Pâle	Fine	B
7	2	2	Foncée	Fine	B
8	2	2	Foncée	Épaisse	B
9	2	1	Foncée	Fine	C
10	2	1	Foncée	Épaisse	C
11	1	2	Pâle	Fine	C
12	1	2	Pâle	Épaisse	C

(1) Calculer les deux premiers sommets de l'arbre de décision.

(2) Appliquer le programme ID3 sur les données de la table. Commentaires.

EXERCICE 4.2 *Soit X l'ensemble des exemples suivants :*

N^o	P_1	P_2	P_3	Cl
1	0	V	N	A
2	1	V	I	A
3	0	F	O	B
4	1	V	N	A
5	1	V	O	A
6	1	F	N	A
7	0	F	O	B
8	0	V	I	A
9	0	F	N	B
10	1	V	I	B
11	1	F	O	A
12	1	F	I	A
13	0	V	O	B

(1) Considérons l'ensemble d'apprentissage composé des exemples 1 à 9. Construire l'arbre de décision T_1 en choisissant les attributs dans l'ordre P_1, P_2, P_3.

(2) Même question avec T_2 en choisissant les attributs en ordre inverse P_1, P_2, P_3.

(3) Évaluer les conséquences si on rajoute à l'ensemble d'apprentissage l'exemple 10.

(4) Considérons l'ensemble test composé des exemples 11 à 13. Évaluer l'erreur de classification sur l'ensemble d'apprentissage, sur l'ensemble test et sur l'échantillon complet pour les deux arbres T_1 et T_2. Commentaires.

(5) Utiliser le programme `id3.pl` pour calculer l'arbre de décision selon le critère du gain d'information en utilisant l'ensemble d'apprentissage et l'échantillon complet. Commentaires.

EXERCICE 4.3 *Soit le tableau des données suivant*

Client	Montant	Âge	Résidence	Études	Cl
1	moyen	moyen	village	oui	oui
2	élevé	moyen	bourg	non	non
3	faible	âgé	bourg	non	non
4	faible	moyen	bourg	oui	oui
5	moyen	jeune	ville	oui	oui
6	élevé	âgé	ville	oui	non
7	moyen	âgé	ville	oui	non
8	faible	moyen	village	non	non

qui concerne les clients d'une banque. La classe oui correspond à un client qui a effectue une consultation de ses comptes bancaires en utilisant Internet. Nous avons aussi comme ensemble de test le tableau suivant

Client	Montant	Âge	Résidence	Études	Cl
9	moyen	âgé	village	oui	oui
10	élevé	jeune	ville	non	oui
11	faible	âgé	village	non	non
12	moyen	moyen	bourg	oui	non

(1) Construire un arbre de décision en prenant les attributs dans l'ordre. Évaluer l'erreur sur l'ensemble de test.

(2) Utiliser le programme `id3.pl` pour calculer l'arbre de décision selon le critère du gain d'information. Comparer avec l'arbre de décision obtenu précédement.

4.6 Bibliographie

Pour la redaction de ce chapitre nous avons utiliser essentiellement les livres suivants :

[1] T. M. MITCHELL : *Machine learning*, McGraw-Hill, 1997
[2] S. RUSSELL, P. NORVIG : *Artificial intelligence*, Prentice Hall, 1995
[3] C. J. THORNTON : *Techniques in computational learning*, Chapman and Hall, 1992

5

PROGRAMMATION LOGIQUE INDUCTIVE

L'apprentissage du concept soit par la formation des ensembles S et G, soit par des arbres de décision, consiste en l'élaboration d'une hypothèse h qui est représenatative du concept. L'hypothèse h est formulée dans le même langage que les exemples et, par conséquent, elle ne peut pas être générale. Pour le dire autrement, elle est exprimée dans un langage de logique des propositions, tandis que, pour qu'elle soit générale, il aurait fallu qu'elle soit exprimée dans un langage des prédicats avec, notament, l'utilisation des variables.

La *programmation logique inductive* (PLI) permet la construction, à partir d'un ensemble d'exemples, d'une hypothèse avec des variables. Une telle hypothèse est un ensemble de clauses de Horn, i.e. un programme défini, qui satisfait à tous les exemples positifs et ne satisfait pas aux exemples négatifs. On dit dans ce cas que l'hypothèse *couvre* les exemples.

5.1 La subsomption des clauses

L'application des algorithmes d'élimination des candidats et ID3 est fondée sur la relation d'ordre partiel $\succeq$ qui existe entre les hypothèses. L'utilisation des clauses pour formuler des hypothèses, nous oblige à établir une relation analogue entre les clauses. La première relation qui vient à l'esprit et qui pourrait jouer ce rôle est bien évidemment l'implication syntaxique ou sémantique. En effet, nous pouvons envisager que si, par exemple, pour deux clauses C_1 et C_2 nous avons $C_1 \vdash C_2$, alors la clause C_1 est plus générale que la clause C_2. Mais nous savons depuis 1932 avec A. Church qu'une telle implication n'est pas décidable([1]). A. Robinson dans son article fondateur (A Machine-Oriented Logic Based on the Resolution Principle, Journ. ACM, v.12, pp.23-41, 1965), où il a présenté l'algorithme de résolution SLD (cf. poly de logique computationnelle), propose une relation d'implication plus stricte, qu'il appelle θ-*subsomption* dont la définition est la suivante :

DÉFINITION 5.1.1 *La clause C_1 θ-subsume la clause C_2, que l'on note $C_1 \vdash_\theta C_2$, si et seulement si*

- *le nombre de prédicats de C_1 n'est pas supérieur au nombre de prédicats de C_2, et*
- *il existe une substitution θ telle que $C_1 \circ \theta \vdash C_2$.*

La θ-subsomption – qu'on appellera dorénavant *subsomption* – est en réalité une approximation de l'implication logique qui a l'avantage d'être décidable. Son inconvénient est qu'il n'est pas complète, c'est-à-dire qu'on peut avoir, pour deux clauses C_1 et C_2, que C_1 ne subsume pas C_2 : $C_1 \nvdash_\theta C_2$ et aussi que C_1 implique C_2 : $C_1 \models C_2$. En effet, nous avons cette situation si C_1 est recursive (c'est-à-dire qu'elle peut être résolue par elle-même) et C_2 est une tautologie. Dans tous les autres cas nous avons $C_1 \vdash_\theta C_2 \Longleftrightarrow C_1 \models C_2$.

Nous pouvons donc utiliser la subsomption pour établir un ordre partiel entre les clauses.

5.2 Algorithmes de couverture séquentielle

Nous allons d'abord présenter une famille d'algorithmes pour l'élaboration d'un concept fondés sur la création séquentielle à l'aide de la logique des propositions. De façon repetitive, on introduit une règle, on supprime tous les exemples positifs qu'elle couvre et on poursuit avec une autre règle. De cette façon à chaque itération de l'algorithme de couverture séquentielle

[1] c'est-à-dire que nous ne pouvons pas montrer en un temps fini, que $C_1 \models C_2$ est logiquement valide ou non en logique des prédicats

on « apprend » une seule règle. Nous présentons d'abord l'algorithme d'apprentissage d'une seule règle et ensuite l'algorithme de couverture séquentielle qui fait appel à l'apprentissage d'une règle.

Formelement on considère un ensemble X d'exemples avec ensemble d'attributs A_X et pour chaque attribut $a \in A_X$ l'ensemble de ses valeurs $V(a;X) = \{v_{a,1}, v_{a,2}, \ldots, v_{a,n_a}\}$ dans X. On considère aussi le concept-cible c qui, en théorie, peut prendre q valeurs différentes, mais dans les algorithmes que nous présentons ici il ne prend que deux valeurs, 1 et 0, partageant ainsi X en deux sous-ensembles X^+ et X^- appelés *ensemble des exemples positifs* (i.e. les exemples qui ont pour le concept-cible la valeur 1) et *ensemble des exemples négatifs* respectivement.

Pour une hypothèse $h \in H$, nous définissons l'ensemble $X(h)$ des exemples qui sont filtrés par l'hypothèse h : $X(h) = \left\{\mathbf{x} \in X \;/\; h \succeq \mathbf{x}\right\}$ où la relation $h \succeq \mathbf{x}$ signifie que h subsume $\mathbf{x}$.

ALGORITHME D'APPRENTISSAGE D'UNE RÈGLE : Son objectif est de calculer la meilleure règle compte tenu de l'ensemble des exemples X.

Fonction `apprentissageUneRegle(`c, A_x, K`)`

Début

0 Données : Le concept-cible c, l'ensemble des attributs A_X et un naturel $K > 0$.

1 On place dans h^* l'hypopthèse la plus générale que nous pouvons faire eu egard à l'ensemble des attributs A_X.

2 $H \leftarrow \{h^*\}$

3 Tant que $H \neq \emptyset$ faire

Début

3.1 *Création de l'ensemble des hypothèses plus spécifiques qui sont cadidates au remplacement de* h^*.

3.1.1 $H' \leftarrow \emptyset$

3.1.2 Pour chaque $h \in H$ faire

Début

3.1.2.1 $h' \leftarrow$ spécialisation de h en y ajoutant la contrainte $a = v_{a,i}$ pour $a \in A_X$ et $v_{a,i} \in V(a;X)$.

3.1.2.2 $H' \leftarrow H' \cup \left\{h'\right\}$

Fin

3.2 *Mise à jour de* h^*.

3.2.1 Pour tout $h' \in H'$ faire
Début

3.2.1.1 Si $\texttt{performance}\left(h', X, c\right) > \texttt{performance}(h^*, X, c)$
, alors
Début
$h^* \leftarrow h'$.
Fin

3.3 *Mise à jour de* H.

3.3.1 Ordonner H' selon les valeurs croissantes de la fonction performances appliquée sur $h' \in H', X$ et c. Soit $H' = \left\{h'_1, h'_2, \ldots, h'_p\right\}$ le résultat obtenu.

3.3.2 Si $K < p$, alors
Début

3.3.2.1 $H \leftarrow \{h'_1, h'_2, \ldots, h'_K\}$

sinon

3.3.2.2 $H \leftarrow H'$.
Fin

Fin

4 $v_{c,k^*} \leftarrow arg \max_{v_{c,k} \in V(c;X)} \mid X\left(h^*; v_{c,k}\right) \mid$ où

$X\left(h^*; v_{c,k}\right) = \left\{\mathbf{x} \in X \;/\; h^* \succeq \mathbf{x} \text{ et } v(c;\mathbf{x}) = v_{c,k}\right\}$; $k = 1, \ldots, q$.

5 Retourne la meilleure règle apprise qui est de la forme

$$r^* : \text{SI } h^* \text{ ALORS } v_{c,k^*}$$

Fin

La fonction performance calcule l'adéquation d'une hypothèse h en fonction des exemples qui sont couverts par cette hypothèse. Son algorithme est le suivant :

ALGORITHME DE CALCUL DE L'ADÉQUATION D'UNE HYPOTHÈSE :
Il retourne la performance d'une règle $f(h, X, c)$.

Fonction $\texttt{performance}(h, X, c)$
Début

1 Retourner la valeur de $f(h,X,c)$ qui est calculée

– soit en utilisant la fréquence relative

$$f(h,X,c) = \frac{|\,X(h;c)\,|}{|\,X(h)\,|}$$

où $X(h;c) = \{\mathbf{x} \in X(h) \;/\; v(c;h) = v(c;\mathbf{x})\}$

– soit en utilisant l'information de l'ensemble $X(h)$:

$$f(h,X,c) = -\sum_{k=1}^{q} \frac{|\,X(h;c_k)\,|}{|\,X(h)\,|} \log_2 \frac{|\,X(h;c_k)\,|}{|\,X(h)\,|}$$

Fin

Notons que dans la bibliographie on trouve des formules supplémentaires pour le calcul de l'adéquation d'un hypothèse.

L'algorithme de la couverture séquentielle, qui utilise l'algorithme précédent, est le suivant :

ALGORITHME DE COUVERTURE SÉQUENTIELLE : Il retourne l'ensemble des meilleures règles R.

Fonction `CSeq(`c`,` X`,` A_x`,` α`)`

Début

0 Données : Le concept-cible c, l'ensemble des exemples X, l'ensemble des attributs A_X et un réel $\alpha > 0$.

1 $R \leftarrow \emptyset$

2 $r \leftarrow$ `apprentissageRegle(`c`,` A_x`,` K`)`

3 Tant que `perfoRegle(`r`,` X`)` $>$ α faire

Début

3.1 $R \leftarrow R \cup \{r\}$

3.2 $X \leftarrow X - \{\mathbf{x} \in X \;/\; \mathbf{x}$ est correctement classé par $r\}$

3.3 $r \leftarrow$ `apprentissageRegle(`c`,` A_x`,` K`)`

Fin

4 Ordonner R selon les valeurs décroissantes des performances des règles.

5 Retourner R.

Fin

`perfoRegle(`r`,` X`)` est une fonction analogue à `performance(`h, X, c`)` à ceci près que le calcul de la fonction f se fait en utilisant la conclusion de la règle à la place du concept-cible c.

5.3 Algorithme FOIL

Quinlan en 1990 a introduit FOIL, qui est une variante de l'algorithme de couverture séquentielle dans le cas de la logique des prédicats. L'algorithme utilise des clauses de Horn dans lesquelles les prédicats ne peuvent pas avoir des foncteurs parmi leurs arguments. Ceci afin de réduire la complexité de l'espace de recherche. Nous présentons ci-après cet algorithme.

ALGORITHME FOIL : Il retourne l'ensemble des meilleures règles R.

Fonction `foil(`c`,` X`,` A_X`)`

Début

0 Données : Le concept-cible c, l'ensemble des exemples X, l'ensemble des prédicats P_X

1 $R \leftarrow \emptyset$

2 Tant que $X^+(c) \neq \emptyset$ faire

Début

2.1 Apprendre une nouvelle règle

2.1.1 $r \leftarrow (c \leftarrow)$, où $(c \leftarrow)$ est la clause de Horn qui induit le concept-cible sans conditions, c'est-à-dire le concept-cible est une donnée.

2.1.2 $X^-(c;r) \leftarrow X^-(c)$

2.1.3 Tant que $X^-(c;r) \neq \emptyset$ faire

Début

2.1.3.1 Engendrer l'ensemble des littéraux : $L \leftarrow$ `creerLitteraux(`r`,` P_X`)`

2.1.3.2 $p^* \leftarrow arg\max_{p \in L}$ `gain(`p`,` r`,` X`)`

2.1.3.3 $r \leftarrow (c \leftarrow p_1, \ldots, p_l, p^*)$, où on suppose que la règle r était $c \leftarrow p_1, \ldots, p_l$.

2.1.3.4 $X^-(c;r) \leftarrow X^-(c;r) - \{\mathbf{x} \in X^-(c;r) \ / \ \mathbf{x}$ satisfait à $p_1, \ldots, p_l, p^*\}$

Fin

2.1.4 $R \leftarrow R \cup \{r\}$

2.1.5 $X^+(c) \leftarrow X^+(c) - \{\mathbf{x} \in X^+(c) \ / \ \mathbf{x}$ est couvert par $r\}$

Fin

3 Retourner R.

Fin

La fonction `creerLitteraux` ajoute des nouveaux prédicats et leur négations, qu'on appelle ici lttéraux, à la règle qui est en train de se former et ceci selon deux techniques :

- soit on ajoute un prédicat – ou sa négation –, sauf le prédicat égal, et dans ce cas il doit avoir parmi ses arguments au moins une variable qui se trouve déjà dans la règle ;
- soit l'on ajoute le prédicat égal – ou sa négation – avec comme arguments des variables qui font partie des variables de la règle.

Nous avons donc l'algorithme suivant :

ALGORITHME CREERLITTERAUX : Il retourne une liste L des prédicats et de leurs négations.

fonction `creerLitteraux(`r`,` P_X`)`

Début

0 Données : La règle en formation r : $(q \leftarrow p_1, \ldots, p_N)$ et les prédicats des exemples P_X.

On forme l'ensemble

$M(r) = \{Z \text{ variable } / \ Z \text{ argument dans au moins un de prédicats } q, p_1, \ldots, p_N \text{ de la règle } r\}$.

1 $L \leftarrow \left\{ \begin{array}{cc} p(Z_1,\ldots,Z_l), \neg p(Z_1,\ldots,Z_l) \ ; & \text{avec } p \in P_X \text{ et } \{Z_1,\ldots,Z_l\} \cap M(r) \neq \emptyset \\ egal(Z_1,\ldots,Z_l), \neg egal(Z_1,\ldots,Z_l) \ ; & \text{avec } \{Z_1,\ldots,Z_l\} \subseteq M(r) \end{array} \right\}$

Fin

La fonction `gain(`p`,` r`,` X`)` calcule l'adéquation d'un prédicat p avec la règle r par rapport aux exemples X. Le calcul se fait selon la formule :

$$\begin{aligned} \texttt{gain}(p,r,X) \ = \ & \mid X^+(r) \cap X^+(r \cup p) \mid \cdot \Big(\log_2 \frac{|X^+(r \cup p)|}{|X^+(r \cup p)| + |X^-(r \cup p)|} \\ & - \log_2 \frac{|X^+(r)|}{|X^+(r)| + |X^-(r)|} \end{aligned}$$

où on a noté par $X^+(r)$ l'ensemble d'exemples de X qui sont classés en positif avec r et $X^+(r \cup p)$ l'ensemble d'exemples de X qui sont classés en positif avec la règle r à laquelle on a ajouté le prédicat p. La valeur de $\texttt{gain}(p, r, X)$ représente la diminution de l'information de la règle r quand on rajoute à celle-ci le prédicat p.

5.4 L'induction comme une déduction inverse

Une autre approche de la programmation logique inductive est fondée sur l'observation que l'induction est juste l'inverse de la déduction ! En réalité il s'agit d'inverser la SLD-résolution telle qu'elle a été présentée en cours de logique computationnelle pour les programmes définis.

Considérons un ensemble d'exemples X pour lesquels nous connaissons aussi la valeur du concept-cible $v(c;\mathbf{x})$, $\mathbf{x} \in X$. Supposons aussi que nous disposons de connaissances partielles B sur l'environnement des exemples. Le problème d'apprentissage est l'élaboration d'une hypothèse h telle que la valeur du concept-cible $v(c;\mathbf{x})$ pour tout $\mathbf{x} \in X$ se déduit chaque fois de l'hypothèse h, des valeurs des attributs pour l'exemple $\mathbf{x}$ et de la connaissance supplémentaire B :

$$(\forall \mathbf{x} \in X,\ v(c;\mathbf{x})) \ :\ \ (B \wedge h \wedge \mathbf{x}) \vdash\ v(c;\mathbf{x})$$

Cette relation fournie la contrainte qui doit être satisfaite par l'hypothèse h, à savoir que pour chaque exemple $\mathbf{x}$ la valeur du concept-cible doit être déduite de B, h et $\mathbf{x}$.

La SLD-résolution est une règle pour l'inférence déductive en logique du premier ordre. On peut montrer que nous pouvons inverser la règle de résolution afin d'avoir in opérateur d'induction comme celui de la relation précédente.

Nous examinons d'abord le cas du calcul propositionnel. Étant données deux clauses C_1 et C_2, l'opérateur de résolution calcule, s'il existe, leur resolvante C. Pour ce faire il faut que C_1 ait un littéral p et C_2 ait la négation de ce littéral $\neg p$. Dans ce cas la résolvante est

$$C = (C_1 - \{p\}) \cup (C_2 - \{\neg p\})$$

C constitue une déduction de C_1 et C_2. En appliquant la formule précédente, nous pouvons obtenir C_2 comme induction de C et de C_1. En effet on a :

$$C_2 = (C - (C_1 - \{p\})) \cup \{\neg p\}$$

Nous allons ensuite faire une extension de cette technique dans le cas de la logique des prédicats. Considérons toujours les deux clauses C_1 et C_2 et soient un littéral p_1 de C_1 et aussi un littéral p_2 de C_2 tels qu'il existe un unificateur σ qui unifie p_1 à $\neg p_2$, i.e. $p_1\sigma = \neg p_2\sigma$. Dans ce cas la résolvante est donnée par

$$C = (C_1 - \{p_1\})\,\sigma \cup (C_2 - \{\neg p_2\})\,\sigma$$

Factorisons maintenant l'unificateur σ en deux unificateurs : $\sigma = \sigma_1\sigma_2$, avec

σ_1 unificateur qui contient toutes les substitutions concernant uniquement de variables de la clause C_1 et σ_2 unificateur qui contient toutes les substitutions qui mettent en jeu uniquement de variables de la clause C_2. La relation précédente s'écrit :

$$C = (C_1 - \{p_1\})\sigma_1 \cup (C_2 - \{\neg p_2\})\sigma_2$$

Supposons que la clause C_2 ne contient pas des littéraux en commun avec C_1. Dans ce cas on peut écrire la relation précédente comme suit :

$$C - (C_1 - \{p_1\})\sigma_1 = (C_2 - \{\neg p_2\})\sigma_2$$

En utilisant le fait que $p_2 = \neg p_1\sigma_1\sigma_2^{-1}$ en résolvant par rapport à C_2, on obtient

$$\text{Résolution inverse :} \quad C_2 = (C - (C_1 - \{p_1\})\sigma_1)\sigma_2^{-1} \cup \left\{\neg p_1\sigma_1\sigma_2^{-1}\right\}$$

qui est la résolution inverse pour la logique du 1er ordre.

Remarquons que cet opérateur n'est pas déterministe car le choix de l'unificateur pour les deux clauses C_1 et C_2 n'est pas unique.

Certains auteurs appellent cet opérateur, *opérateur d'absorption*. Il est utilisé dans des systèmes de PLI comme Marvin, Cigol ou Progol.

5.5 Exercices

EXERCICE 5.1 *En appliquant la résolution inverse donner au moins deux solutions possibles pour C_2 si on a*

$$C_1 = a \vee b \vee c \;\; et \;\; C = a \vee b$$

EXERCICE 5.2 *Soient les clauses*

$$C_1 = s(b,Y) \vee r(Z,X) \;\; et \;\; C = r(b,X) \vee p(X,a)$$

Donner quatre résultats possibles pour C_2.

EXERCICE 5.3 *On cherche à apprebdre les règles pour le prédicat* `suc2(Y,X)` *qui dans un graphe exprime le fait que le sommet* `Y` *est "petit-flis" du sommet* `X`.

On a comm exemple X {`suc2(c,a)`} *et comme connaissances complémentaires B* = {`pred(a,b), pred(b,c)`}.

En utilisant la résolution inverse, donner la clause pour `suc2(Y,X)` .

EXERCICE 5.4 *Ordonner les clauses suivantes selon la* θ*-subsomption.*

(1) `f(a, b) :- m(a), m(b), p(a,b).`

(2) `f(X, Y) :- m(X), p(X, Y).`

(3) `f(X, X) :- m(X), p(X, Y), p(X, Y).`

(4) `f(X,X) :- m(X), p(X, X), m(X).`

(5) `f(X,X) :- m(X), p(X, X).`

(6) `f(X,X) :- m(X), p(X, Y).`

EXERCICE 5.5 *Calculer la plus petite généralisation pour les formules suivantes :*

(1) `m(a, c(a, nil))` *et* `m(b, c(d, nil)).`

(2) `add(s(s(0)), s(s(0)), s(s(s(s(0))))` *et* `add(s(0), s(0), s(s(0)))`

EXERCICE 5.6 *Utiliser le programme FOIL avec les exemples donnés. Commenter les résultats obtenus.*

EXERCICE 5.7 *Utiliser le programme FOIL avec les données du chapitre précédent. Comparer les résultats obtenus avec ceux du programme ID3.*

5.6 Bibliographie

Pour la redaction de cd chapitre nous avons utiliser essentiellement les livres suivants :

[1] A. CORNUÉJOLS, L. MICLET : *Apprentissage artificiel*, Eyrolles, 2002

[2] N.LAVRAČ, S.DŽEROSKI : *Inductive logic programming*, E.Horwood, 1994

[3] T. M. MITCHELL : *Machine learning*, McGraw-Hill, 1997

6

FOUILLE DE DONNÉES : APPROCHE DE RÈGLES D'ASSOCAITION

Nous présentons dans ce chapitre une approche assez récente de fouille de données qui est fondée sur la découverte de *règles d'association* à partir d'un ensemble de données qu'on appellera *transactions* (Agrawal et al. 1993). Ce thème est considéré aujourd'hui comme faisant parti des approches d'apprentissage symbolique non supervisé, utilisé dans le domaine de fouille de données (data mining) et d'extraction de connaissances. Un exemple d'application assez courant est l'analyse des logs web sur un serveur web afin de découvrir de comportements utilisateur (web usage mining) dans le but d'adapter ou de personnaliser le site ou de découvrir des comportements types sur certains sites (e-commerce par exemple).

Nous avons vu dans les chapitre précédents trois approches d'apprentissage de concept (espace de versions, arbre de décision et FOIL) dans un

contexte d'apprentissage supervisé ; autrement dit, la classe de sortie était présentée est évaluée dans l'ensemble d'apprentissage.

Les règles d'association, étant une méthode d'apprentissage non supervisé, permettent de découvrir à partir d'un ensemble de transactions, un ensemble de règles qui exprime une possibilité d'association entre différents *items*. Une transaction est une succession d'items exprimée selon un ordre donné ; de même, l'ensemble de transactions contient des transactions de longueurs différents.

Un exemple classique de l'utilité de cette approche est *le panier du ménagère* qui décrit un ensemble d'achats effectué au supermarché ; les règles d'association permet de découvrir de régularités dans l'ensemble de transactions comme par exemple : *Si fromage alors vin rouge*, etc. Ces règles permettent par exemple au gérant de proposer des bons de réductions significatifs sur les achats futurs des clients ! !

6.1 Description du domaine

Un domaine d'application donné doit être décrit par une liste limitée d'atomes qu'on appellera *items*. Par exemple, pour l'application du *panier de ménagère* la liste des items correspond à l'ensemble d'articles disponibles dans le supermarché $[vin, fromage, chocolat, etc.]$. *Un ensemble d'items* est une succession d'items exprimée dans un ordre donné et pré-défini. *Une transaction* est un ensemble d'items. Un ensemble D de transactions correspond à un ensemble d'apprentissage qu'on va utiliser dans la suite pour déterminer les règles d'associations.

Par exemple, deux transactions possibles qui décrivent les achats dans un supermarché sont : $T_1 = \{Vin\ Fromage\ Viande\ \}$ et $T_2 = \{Vin\ Fromage\ Chocolat\ \}$. Remarquer bien qu'un ordre doit être défini sur l'ensemble d'items, autrement dit, dans toutes les transactions qui contiennent *Vin* et *Fromage*, *Vin* doit figurer avant *Fromage*.

6.1.1 Définition

Un ensemble d'items est dit *fréquent* ssi il correspond à un motif fréquent dans la base de transactions. Nous définissons le *support d'un motif* comme étant le nombre de transactions dans D contenant ce motif divisé par card(D). Un seuil minimal de support *minSupp* est fixé à partir duquel un

ensemble d'items est dit fréquent.

Le but de ce chapitre est de trouver tous les ensembles d'items fréquents qui auront certainement des longueurs différentes, et d'en construire des règles d'association. Par exemple, supposons que ABCD soit un ensemble d'items fréquent de longueur 4, nous construisons la règle $AB \Rightarrow CD$ ssi $support(ABCD)/support(AB) \geq minConf$ où nous avons noté par $minConf$ la valeur de la confiance minimale associée à cette règle. nous appelons cette mesure *la confiance associée à une règle*. Si cette confiance dépasse un certain seuil, alors la règle est induite.

Nous présentons dans la suite deux algorithmes qui gênèrent tous les ensembles d'items fréquents suivi de l'algorithme qui construit les règles d'association à partir de ces ensembles.

6.2 Deux algorithmes pour la générations des sous-ensembles fréquents

Nous présentons deux algorithmes pour la génération des sous-ensemble d'items fréquents, le premier étant un algorithme très coûteux en terme d'accès à la base de transactions D, le deuxième optimise le coût d'accès.

6.2.1 Terminologie

Nous adoptons dans la suite la terminologie suivante :

L_k est l'ensemble constitué des sous-ensembles d'items fréquents de longueur k.

C_k est un ensemble constitué des sous-ensembles d'items candidats de longueur k, notons bien que $L_k \subset C_k$

6.2.2 Propriété des sous-ensembles fréquents

Soit X_k un sous-ensemble d'items fréquent, tous les sous-ensembles d'items contenus dans X_k et qui soient de longueurs inférieurs à k sont fréquents.

Par exemple si $ABCD$ est un sous-ensemble d'items fréquents, alors, les sous ensembles :ABC, ABD, BCD, AB, AC, BC, BD, CD, A, B, C, D les sont aussi.

6.2.3 L'algorithme `Apriori`

L'algorithme `Apriori` est le suivant :

Calculer L_1

$k \leftarrow 2$

TANTQUE $L_{k-1} <> \phi$ **FAIRE**

  $C_k \leftarrow apriori - gen(L_{k-1})$

  TANTQUE $t \in D$ **FAIRE**

    $C_t = sousensemble(C_k, t)$

    TANTQUE $c \in C_t$ **FAIRE**

      c.count++

    FIN TANTQUE

  FIN TANTQUE

  $L_k \leftarrow \{c \in C_k c.count \geq minSup\}$

  $k \leftarrow k+1$

FIN TANTQUE

RETOURNER $\bigcup_k L_k$

Algorithme 4: L'algorithme `Apriori`

L'algorithme `Apriori` permet de découvrir les sous-ensembles d'items fréquents en partant de ceux dont la longueur est 1 et en augmentant la longueur au fur et à mesure. Cet algorithme est fondé sur la propriété des sous ensembles d'items fréquents. Il fait appel à deux algorithmes :

apriori-gen L'algorithme `apriori-gen` est constitué de deux phases la première phase nommé `Joindre` trouve tous les candidats possibles de longueur k à partir de l'ensemble L_{k-1}, La deuxième phase `Effacer` efface de C_k les éléments qui ne vérifient pas la propriété des sous-ensemble fréquents. Nous donnons le code de `Joindre` en SQL :

Soient $p, q \in L_{k-1}$

(1) insert into C_k

(2) select $p[1], p[2], ..p[k-1], q[k-1]$

(3) from p,q

(4) Where $p[1] = q[1]..p[k-2] = q[k-2], p[k-1] < q[k-1]$

sous-ensemble L'algorithme `sous-ensemble` calcule le sous ensemble $C_t \subseteq C_k$ qui correspondent à des sous-ensembles présents dans les transactions contenues dans D.

Par exemple si $L_3 = \{\{123\},\{124\},\{134\},\{135\},\{234\}\}$, la phase `Joindre` donne comme résultat $C_4 = \{\{1234\},\{1345\}\}$ ensuite la phase `Effacer` donne le résultat : $C_4 = \{\{1234\}$ car l'élément $\{145\}$ n'est pas dans L_3 et donc $\{1345\}$ est effacé.

L'inconvénient majeur de cet algorithme est le nombre considérable de l'accès à la base de données D. Une amélioration qui consiste à intégrer les identificateurs des transactions est proposée dans la sous-sections suivante.

6.2.4 L'algorithme `AprioriTid`

L'algorithme `AprioriTid` est donné par le tableau de la page suivante.

L'amélioration que apporte cet algorithme par rapport au précédent est le fait de stocker à chaque itération les identificateurs des transactions contenant les sous-ensembles fréquents dans l'ensemble $\hat{C}_k$. La propriété des sous-ensembles fréquents nous permet de voir clairement que les transactions sollicitées dans l'itérations $k+1$ seront parmi celles identifiées dans l'itération k. Par conséquence, l'accès à D est effectuée seulement pour l'itération 1.

6.3 Génération des règles

Nous adoptons une approche descendante de recherche de règles d'association à partir des ensembles d'items fréquents trouvés par l'un des algorithmes précédents. Cette approche est justifiée par deux propriétés de redondance sur l'ensemble de règles d'association qu'on définira dans la suite.

La première propriété est celle de la *redondance simple* : Soient les deux règles :

R1 $A \Rightarrow B,C$

R2 $A,B \Rightarrow C$

Nous remarquons que $confiance(R1) = support(ABC)/support(A)$, $confiance(R2) = support(ABC)/support(AB)$ et par conséquent, $confiance(R2) > confinace(R1)$. Autrement dit, il est intéressant pour un ensemble fréquent

Calculer L_1

$\hat{C_1} \leftarrow D$

$k \leftarrow 2$

TANTQUE $L_{k-1} <> \phi$ **FAIRE**

$C_k \leftarrow apriori - gen(L_{k-1})$

$\hat{C_k} \leftarrow \phi$

TANTQUE $t \in \hat{C_{k-1}}$ **FAIRE**

$C_t = \{c \in C_k$ `(c[1].c[2]..c[k-1])` $\in t.ensemble \wedge (c[1].c[2]..c[k-2].c[k]) \in t.ensemble\}$

TANTQUE $c \in C_t$ **FAIRE**

c.count++

SI $C_t <> \phi$ **ALORS**

$\hat{C_k} \leftarrow + < t.TID, C_t >$

FIN SI

FIN TANTQUE

FIN TANTQUE

$L_k \leftarrow \{c \in C_k$ `c.count \geq minSup\}$\STATE $k` $\leftarrow k+1$

FIN TANTQUE

RETOURNER $\bigcup_k L_k$

Algorithme 5: L'algorithme `AprioriTid`

donné, de commencer par chercher les règles ayant le nombre de conditions minimal, si une telle règle est acceptée les autres règles dérivées du même sous-ensemble fréquent ayant plus de conditions incluant la condition minimale, auront des taux de confiance plus élevés. Autrement dit si $A \Rightarrow B,C$ alors $A,B \Rightarrow C$ et $A,C \Rightarrow B$.

La deuxième propriété est celle de la *redondance stricte* ; Soient les deux règles :

R1 $A \Rightarrow B,C,D$

R2 $A \Rightarrow B,C$

Nous remarquons que $confiance(R1) = support(ABCD)/support(A)$,

$confiance(R2) = support(ABC)/support(A)$ et par conséquent, $confiance(R2) > confinace(R1)$. Autrement dit, il est judicieux de commencer la recherche de règles d'association dans les ensembles d'items fréquents les plus grands, si une telle règle est acceptée, toutes les règles inclues seront encore considérées.

La stratégie de recherche de règles que nous adoptons est la suivante :

(1) Trouver les plus grands ensembles d'items fréquents.

(2) En extraire les règles de confiance supérieure au seuil, ayant des conditions minimales.

(3) Pour les configurations n'ayant pas engendré des règles d'association, explorer d'une façon analogue, les sous-ensembles immédiats.

6.4 Exercice

Les règles d'association

Génération des règles d'association :

EXERCICE 6.1 *Soit D la base de transactions suivante :*

100 *1 3 4*

200 *2 3 5*

300 *1 2 3 5*

400 *2 5*

(1) Appliquer l'algorithme Apriori pour extraire les sous-ensembles d'items fréquents avec $minSupp = 2$.

(2) Appliquer l'algorithme AprioriTid pour extraire les sous-ensembles d'items fréquents avec $minSupp = 2$.

(3) Générer les règles d'association avec $minConf = 1$

EXERCICE 6.2 *Amélioration de la prédiction*

Soit D la base de transactions contenant un ensemble de transaction décrivant des achats de produits dans l'ensemble $\{A, B, C, D, E\}$

1 *A D*

2 *A B C*

3 *A E*

4 *A D E*

5 *B D*

(1) Considérons seulement l'ensemble $\{A,B,C\}$ *; Trouver l'ensemble de règles qui permettent de prédire l'achat de deux produits tout en améliorant la prédiction par rapport à la mesure statistique induite par la base de données.*

(2) Répéter la même question pour trouver l'ensemble de règles qui permettent de prédire l'achat d'un produit.

6.5 Références

R. AGRAWAL, T. IMIELINSKI, A. SWAMI : Mining association rules between sets of items in large databases, *ACM SIGMOD Conference on Management of Data*, 1993

7

PROBLÈMES DE SATISFACTION DE CONTRAINTES

De nombreux problèmes peuvent être exprimés en terme de contraintes comme les problèmes d'emploi de temps, de gestion d'agenda, de gestion de trafic ainsi que certains problèmes de planification et d'optimisation (comme le problème de routage de réseaux de telecommunication). Ce chapitre contient trois parties principales : la première partie permet d'une part, de fournir les éléments de base pour formaliser un problème de satisfaction de contraintes et, d'autre part, de montrer comment un tel problème peut être ramené à une recherche d'une solution dans un espace d'états. La deuxième partie propose trois algorithmes de recherche de solutions qui mettent à jour les domaines de définitions de variables. La troisième partie permet de comprendre la notion de la *consistance* dans un système particulier contenant des contraintes unaires ou binaires et montre leur utilisation lors de la recherche d'une solution.

7.1 Définitions et notations

Un *problème de satisfaction de contraintes* (PSC) est modélisé par un triplet (X,D,C), où $X = \{X,..X_n\}$ est un ensemble fini de variables à résoudre, D est la fonction qui définit le domaine de chaque variable ($D(X_i)$) contenant l'ensemble de valeurs que peut prendre la variable. Nous nous limitons ici aux domaines finis. $C = \{C_1,..C_m\}$ est un ensemble fini de contraintes. Une contrainte est une relation entre un sous-ensemble de variables noté W et un sous ensemble de valeurs noté T : $C = (W,T)$ avec $W \subseteq X$ et $T \subseteq D^W$. W est l'arité de la contrainte C et est notée $var(c)$. T est la solution de C et est notée $sol(C)$.

Une *affectation* est un ensemble de couples (variables-valeurs) : $A = \{(X_j \leftarrow v_j)\}$ avec $X_j \in X$ et $v_j \in D(X_j)$. Une affectation est *partielle* si elle ne concerne qu'une partie de variables et *totale* sinon. Une affectation A est *valide par rapport à C* si la relation définie dans chaque contrainte C_i est vérifiée pour les valeurs des variables affectées dans A.

7.2 Méthodes de résolution

Une solution à un problème de resolution de contrainte est une affectation totale valide A.

Un problème de résolution de contrainte peut être formalisé comme problème de planification en considérant une affectation valide comme étant un état et une affectation d'une variable donnée non affectée encore, comme étant une action. Le problème peut se formaliser ainsi :

S est l'ensemble des états, un état est une affectation partielle (ou totale) valide ;

E0 l'état initial est l'ensemble vide ;

F l'ensemble des états finals est l'ensemble des affectations totales et valides.

T est donnée par : $T(A) = \{affect(X_r,vr),A \cup \{X_r \leftarrow vr\}\}$

où A est un état, $affect(X_r,vr)$ est une action, et $A \cup \{X_r \leftarrow vr\}$ est l'état suivant.
En fait, dans cette modélisation, le nombre d'états est fini car l'ensemble X

est fini, De plus on ne peut pas par définition (et par convention) re-visiter un état.

7.2.1 Le backtrack

L'algorithme "*simple backtrack*" correspond à une recherche aveugle de la solution en essayant successivement les ensembles des affectations de variables jusqu'à trouver une solution. Cet algorithme est fondé sur les points de retour. Chaque fois qu'un choix est effectué, un point de retour est mentionné dans l'algorithme. L'ordre des variables à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations. Le chemin en cours de test est éliminé de l'ensemble de solutions si un domaine donné devient vide. La procédure choisir revient à l'itération précédente et essaie une nouvelle valeur pour la variable en question, si aucune valeur n'est possible on revient au point précédent. Si par contre, nous sommes au premier point alors on sort de l'algorithme en échec (autrement dit avec impossibilité de trouver une solution).

FONCTION simple-backtrack(X,D,C) : affectation
VAR A :affectation
  i : ENTIER
  $i \leftarrow 1$
  TANTQUE $i \leq n$ **FAIRE**
    (*) choisir $v_i \in D(X_i)$
    SI $A \cup \{X_i \leftarrow v_i\}$ est valide **ALORS**
      $A \leftarrow A \cup \{X_i \leftarrow v_i\}$
    SINON
      retour au dernier point de choix (*)
    FIN SI
    $i \leftarrow i+1$
  FIN TANTQUE
RETOURNER A

Algorithme 6: L'algorithme simple backtrack

7.2.2 L'anticipation

Cet algorithme est une simple amélioration de l'algorithme précédent. En fait, à chaque choix d'une affectation d'une variable, les domaines des variables non affectés encore, sont réduits aux valeurs qui sont compatibles avec l'affection en cours et ceci en utilisant le système des contraintes. Cet algorithme est également fondé sur l'utilisation des points de retour ainsi que sur les domaines vides. De même, l'ordre des variables à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations.

FONCTION anticipation(X,D,C) : affectation

VAR A :affectation

i,j : ENTIER

$i \leftarrow 1$

TANTQUE $i \leq n$ **FAIRE**

(*) choisir $v_i \in D(X_i)$

$A \leftarrow A \cup \{X_i \leftarrow v_i\}$

$j \leftarrow i+1$

TANTQUE $j \leq n$ **FAIRE**

$D(X_j) = \{v_j \in D(X_j) tq A \cup \{X_j \leftarrow v_j\} estvalide\}$

SI $D(X_j) = \{\}$ **ALORS**

retour au dernier point de choix (*)

FIN SI

$j \leftarrow j+1$

FIN TANTQUE

$i \leftarrow i+1$

FIN TANTQUE

RETOURNER A

Algorithme 7: L'algorithme anticipation

7.2.3 Le choix du plus petit domaine

L'algorithme du choix du plus petit domaine est fondé sur le principe suivant : à chaque itération la variable non affectée ayant le plus petit domaine est choisie. Ensuite l'anticipation sur les autres domaines est réalisée. Par contre, l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations, reste aléatoire.

FONCTION min-domaine(X,D,C) : affectation

VAR A :affectation

i,j : ENTIER

$i \leftarrow 1$

TANTQUE $i \leq n$ **FAIRE**

choisir la variable X_k non affectée ayant le plus petit domaine

(*) choisir $v_k \in D(X_k)$

$A \leftarrow A \cup \{X_k \leftarrow v_k\}$

TANTQUE il existe des variables X_j non affectée **FAIRE**

$D(X_j) = \{v_j \in D(X_j) tq A \cup \{X_j \leftarrow v_j\} est valide\}$

SI $D(X_j) = \{\}$ **ALORS**

retour au dernier point de choix (*)

FIN SI

FIN TANTQUE

$i \leftarrow i+1$

FIN TANTQUE

RETOURNER A

Algorithme 8: L'algorithme choix du plus petit domaine

7.3 La notion de consistance binaire

Nous nous intéressons dans cette section à l'étude de la notion de

consistance dans un système ayant des contraintes d'arité 2 au plus[1].

Un tel système peut être représenté par un graphe. Chaque nœud représente une variable X ainsi que le domaine actuel de cette variable. Un arc entre deux nœuds représente une contrainte donnée.

Une *consistance* est une propriété qui doit être assurée à chaque étape de la recherche de la solution dans l'objectif de couper l'arbre de recherche. Cette contrainte ne doit pas être coûteuse à calculer. Elle retire des domaines des variables les valeurs qui ne participent pas à aucune solution.

La plupart des consistences et des algorithmes concernent un graphe de contraintes binaires. L'adaptation aux contraintes n-aires ne fait pas l'objet de ce chapitre. Les consistances sont des propriétés globales du graphe et le fait de retirer une valeur d'un domaine peut relancer la mise à jour des autres domaines dans le graphe et ceci jusqu'à la stabilité.

Nous définissons ici deux types de consistance :

La nœud-consistance consiste à éliminer des domaines de variables toutes les valeurs qui n'appartiennent pas aux solutions des contrainte unaires.

L'arc consistance qui est défini ainsi : Soit s_X le domaine courant de la variable X, soit c une contrainte sur X et Y (Y étant la deuxième variable de la contrainte binaire c). La valeur $v_x \in s_x$ est *arc-consistante* pour c si : $\exists v_y \in s_y$ tel que $(v_x, v_y) \in sol(c)$? De même on dit que la contrainte c est *arc-consistante* si : $\forall v_x \in s_x$, v_x est arc-consistante et vice-versa en échangeant le rôle de X et de Y. Finalement, un système de résolution de contraintes est *arc-consistant* ssi chaque contrainte est arc-consistante.

Nous présentons dans la suite trois algorithmes Ac1, Ac3, Ac4 qui effectuent des mises à jour des domaines des variables pour rendre un système de contrainte arc-consistant.

Ces algorithmes sont basés sur la fonction *réviser* qui prend en paramètre une contrainte binaire $c(X,Y)$ et qui met à jour les domaines des deux variables pour que cette contrainte deviennent arc-consistante.

L'algorithme Ac1 propage la mise à jour des domaines dans le graphe de façon aléatoire jusqu'à la stabilisation des domaines sur tous les nœuds.

L'algorithme Ac3 utilise une file d'attente. Dès qu'un domaine est modifié, on ne place dans la file que les arcs pouvant être affectés par la modi-

[1] c'est-à-dire chaque contrainte exprime une relation entre deux variables au plus.

fication. Le but étant d'améliorer la vitesse de convergence vers la stabilité en éliminant les vérifications inutiles.

L'algorithme Ac4 affine mieux la performance en ajoutant une stratégie permettant de choisir d'une façon optimale les valeurs à verifier dans un domaine donné.

FONCTION reviser(c : contrainte ; X,dx : variable) : Bool

VAR supprime :Bool

v,w : Valeurs

$supprime \leftarrow faux$

TANTQUE $v \in dx$ **FAIRE**

SI il n'y a pas de $w \in d_y$ tq $(v,w) \in sol(c)$ **ALORS**

supprimer v de dx

$supprime \leftarrow vrai$

FIN SI

FIN TANTQUE

RETOURNER supprime

Algorithme 9: L'algorithme Réviser : c est une contrainte binaire, dx est le domaine de X

FONCTION AC1 (C :PSC)

VAR change :bool

c : Contrainte, X : Variable

$change \leftarrow vrai$

TANTQUE *change* **FAIRE**

$change \leftarrow faux$

TANTQUE il y a des couples (c,X) **FAIRE**

$change \leftarrow change \vee reviser(c,X)$

FIN TANTQUE

FIN TANTQUE

Algorithme 10: L'algorithme Ac1

FONCTION AC3 (C :PSC)

VAR Q :File

$change \leftarrow vrai$

TANTQUE Q est non vide **FAIRE**

récupérer la tête (c,X)

SI reviser(c,X) **ALORS**

$Q \leftarrow (Q \cup \{(c',Z) \mid (var(c') = \{X,Z\}) \wedge Z \neq Y\})$

FIN SI

FIN TANTQUE

Algorithme 11: L'algorithme Ac3

7.4 Applications en Prolog

Nous présentons dans cette section deux applications de résolution de contraintes décrites en Prolog. Les programmes présentés ici correspondent à la façon la plus coûteuse pour trouver une solution en recherchant dans

toutes les possibilités. Nous allons travailler sur ces programmes pendant les TPs dans le but d'y intégrer la vérification de l'arc-consistance ce qui permet de les rendre plus performants.

7.4.1 Les huit reines

Il s'agit de placer huit reines de façon à ce que aucune paire de reines soit en situation de prise. L'échiquier est modélisé par une liste de taille 8, le i-ème élément correspond à la position d'une reine sur la colonne i. Nous définissons les prédicats suivants :

form est un prédicat qui génère une liste qui prend ses valeurs dans un intervalle donné ;

queen est le programme principle qui trouve les bonnes configurations de l'échiquier ;

safe est un prédicat qui teste si une configuration donnée sur une partie de l'échiquier soit bonne ou non ;

noattack est un prédicat qui teste si une reine récemment positionnée sur un colonne attaque l'ensemble de reines déjà bien positionnées sur l'échiquier ;

diff exprime les contraintes que doivent verifier deux reines sur un échiquier en connaissant le nombre de colonne qui les sépare.

PROGRAMME 7.4.1

```
form(N,L,N) :- member(N,L).
form(I,L,N) :- I<N, member(I,L),I1 is I+1, form(I1,L,N).

queen(L) :- length(L,8), form(1,L,8), safe(L).

safe([]).
safe([X|L]):-noattack(L,X,1), safe(L).

noattack([],_,_).
noattack([Y|L],X,I) :- diff(X,Y,I),I1 is I+1, noattack(L,X,I1).

diff(X,Y,I) :- X=\=Y, X=\=Y+I, X+I=\=Y.
```

7.4.2 La série magique

La série magique de taille N est une série qui prend ses valeurs dans l'intervalle $[0,N-1]$ et qui vérifie la propriété suivante : le $i$ème élément correspond au nombre d'occurrences de l'élément i dans la série ! Le programme Prolog est composé des prédicats suivants :

geninter est un prédicat qui génère un intervalle entre deux bornes fixées.

gensuite est un programme qui génère une suite de taille N qui prend ses valeurs dans $[0, N-1]$

magic est le programme qui génère une suite magique de taille N.

contraintes est un programme qui vérifie si chaque élément de la suite correspond bien au nombre d'occurrences de sa position.

PROGRAMME 7.4.2

```
geninter(N,N,[N]).
geninter(I,N,[I|R]) :- I<N, I1 is I+1, geninter(I1,N,R).

suite(0,_,[]) :- !.
suite(N,I,[X|R]) :- member(X,I), N1 is N-1, suite(N1,I,R).

gensuite(N,L) :- N1 is N-1, geninter(0,N1,I), suite(N,I,L).

magic(N,L) :- gensuite(N,L), contraintes(L,L,0).

contraintes([],_,_).
contraintes([X|Xs],L,I) :- somme(L,I,X), I1 is I+1, contraintes(Xs,L,I1).

somme([],_,0).
somme([X|Xs],I,S) :- somme(Xs,I,S1), X==I, S is 1+S1.
somme([X|Xs],I,S) :- somme(Xs,I,S1), X=\=I, S is S1.
```

7.5 Exercices

EXERCICE 7.1 (AVEC QUI PARLE PIERRE ? AVEC QUI PARLE ELISA ?) *Dans une soirée, trois couples se sont réunies. Chaque personne parle avec une autre personne qui est du sexe opposé et qui n'est pas son conjoint.*

(1) Marie parle avec Pascal.

(2) George Parle avec la femme de Pascal.

(3) Denise parle avec le mari de Marie.

(1) Modéliser ce problème en tant qu'un problème de salification de contraintes.

(2) Appliquer l'algorithme simple-backtrack pour trouver la solution.

(3) Modéliser ce problème en tant q'un problème de planification.

(4) Appliquer l'exploration en largeur pour trouver la solution.

EXERCICE 7.2 (PROBLÈME DE HUIT REINES) *Le but est de placer sur un échiquier 8 reines, aucune paire de reines ne doit être en position de prise. Placez vous dans le contexte de 4 reines sur un échiquier de 4 lignes et 4 colonnes.* [2]

(1) Nous imposons la modélisation $X = \{X1, X2, X3, X4\}$*,* X_i *étant la position de la reine sur la colonne i. Supposons que les domaines des quatres variables sont initialisés à [1,2,3,4], Appliquer l'algorithme anticipation pour trouver la solution.*

(2) Revenons à la situation initiale, est ce que la propriété de arc-consistance est vérifiée ?

(3) On initialise le domaine de la variable représentant la première colonne à $D(X_1) = [1]$ *Appliquer l'algorithme AC1.*

(4) On initialise maintenant le domaine de la variable représentant la première colonne à $D(X_1) = [2]$ *Appliquer l'algorithme AC3.*

[2] Histoire de vous montrer que je suis sympa ! !

Table des matières