6 Généralisation de Houria pour l'intégration d'autres critères de comparaison.

Un des points importants de la théorie des hiérarchies de contraintes est le fait de pouvoir définir différents types de comparateurs. Ces différents types de comparateurs doivent être jugés d'après la réponse à la question suivante : à quel degré représentent-ils l'intention de l'utilisateur dans des problèmes réels ? Le chapitre précédent décrit notre résolveur Houria basé sur le critère global *Nombre-Contraintes-Non-Satisfaites*. Ce résolveur permet de trouver des solutions qui sont plus intuitives que celles obtenues par un résolveur intégrant un comparateur local (Pour un même système surcontraint, les solutions trouvées par le comparateur de ce résolveur satisfont plus de contraintes que celles trouvées par un critère local). Cependant, ce résolveur de même que les autres résolveurs existants, supposent que les contraintes d'une hiérarchie donnée sont uniquement étiquetées et non pondérées par des poids réels.

L'objectif de ce chapitre est de surmonter cette restriction et donc de pouvoir résoudre des hiérarchies où les contraintes sont étiquetées et pondérées par des poids (ou associées à des indices) réels différents. Plusieurs applications illustrent ce besoin, on citera ici deux exemples. Le premier type d'application est celui où l'utilisateur a besoin d'exprimer des contraintes dites de remplacement. Au sein d'une même classe dans la hiérarchie, on peut avoir deux types de contraintes : celles associées à un indice[1] fort dont la satisfaction sera considérée prioritaire, et celles associées à des indices moins forts et qui sont considérées comme des contraintes de remplacement. Dans ce cas, le résolveur doit d'abord considérer la satisfaction des contraintes associées à un indice fort et dans les cas d'échec (si cette satisfaction ne peut pas avoir lieu), considérer la satisfaction des contraintes de remplacement. Le deuxième type d'application est celui où le poids d'une contrainte traduit son degré d'importance dans la classe de la hiérarchie où elle se trouve. Dans ce cas et selon le comparateur utilisé, les contraintes pondérées par des poids forts ne sont pas forcément à satisfaire en priorité (si l'on suppose qu'on utilise le comparateur *Somme-Pondérée* en utilisant l'erreur prédicat, et qu'on a la disjonction exclusive entre la satisfaction d'une contrainte pondérée à 0.9 et de deux contraintes pondérées à 0.5 chacune, alors ces dernières sont prioritaires pour la satisfaction).

1. Ici on préfère utiliser le terme indice au terme poids.

Ce chapitre est composé de cinq parties : la première et la deuxième parties décrivent les définitions d'autres critères de comparaison globaux intégrés dans Houria ainsi que les relations entre les ensembles de solutions produits par ces différents critères et celui produit par le premier critère décrit au chapitre précédent. Chacune de ces deux parties décrit également les techniques utilisées pour la construction des graphes solutions correspondant au critère intégré. La troisième partie décrit les procédures généralisées (paramétrées par un type de comparateur) d'ajout et de retrait d'une contrainte à la hiérarchie. La quatrième partie traite la complexité et l'implémentation de Houria ainsi des mesures effectuées sur des problèmes générés aléatoirement. Enfin, la cinquième partie décrit une comparaison fonctionnelle entre Houria et les autres algorithmes de propagation locale présentés auparavant.

6.1 Le deuxième critère de comparaison intégré dans Houria

6.1.1 Définition formelle

Le deuxième critère de comparaison dans cet algorithme est le comparateur *Cardinal-Pire-Cas*. Ce comparateur utilise le nombre de contraintes satisfaites associées au plus fort indice de chaque classe dans la hiérarchie. L'ensemble des solutions obtenues par l'utilisation de ce comparateur est généralement de taille plus petite que celle de l'ensemble obtenu en utilisant le critère *Localement-Prédicat-Meilleur* ou encore celle de l'ensemble obtenu en utilisant le critère global *Pire-Cas*. Avant de définir ce critère, on présentera d'abord la définition du comparateur *Pire-Cas*.

Définition 6.1:

Une valuation θ est meilleure selon le critère *Pire-Cas* qu'une autre valuation η, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, le plus fort indice parmi ceux des contraintes non satisfaites après application de θ est égal à celui après application de η, et au niveau *k*, cet indice est strictement inférieur.

$$Pire\text{-}Cas(\theta, \eta, H) \Leftrightarrow Globalement\text{-}Meilleure(\eta, \theta, H, g), \text{ avec } g(\phi, H_i) \equiv Max\ \{c\ e(c\ \phi) / (c \in H_i)\}$$

avec e(c φ) est l'erreur prédicat (qui rend 0 si la contrainte est satisfaite et 1 sinon).

Définition 6.2:

Une valuation θ est meilleure selon le critère *Cardinal-Pire-Cas* qu'une autre valuation η, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, le nombre de contraintes non satisfaites associées au plus fort indice après application de θ est égal à celui après application de η, et au niveau *k*, ce nombre est strictement inférieur.

$$Cardinal\text{-}Pire\text{-}Cas(\theta, \eta, H) \Leftrightarrow Globalement\text{-}Meilleure(\eta, \theta, H, g), \text{ avec}$$

$$g(\phi, H_i) \equiv (Max\ \{\ \overline{c}\ e(c\ \phi) / (c \in H_i)\}, Card(Max^1\ \{\ \overline{c}\ e(c\ \phi) / (c \in H_i)\})).$$

Comme il est mentionné au début de ce chapitre, ce comparateur peut être utilisé dans des application où l'utilisateur a besoin d'exprimer des contraintes dites de remplacement. Par exemple, si l'on considère la hiérarchie suivante :

$H = \{ H_1 \}$ où H_1 contient les quatre contraintes suivantes : *{x.A=x.souris}, {y.A = y.souris}, {x.B=x.souris}, {y.B=y.souris}* qui ont respectivement pour indices *0.9, 0.9, 0.8, 0.8*.

Ici, l'utilisation de ce comparateur voudrait dire :

1. Par abus de langage, dans cette partie de la définition on supose que la fonction *Max* rend l'ensemble des maximaux et non un seul

on préfère une valuation qui satisfait les deux contraintes *{x.A=x.souris}* et *{y.A = y.souris}* à une valuation qui satisfait uniquement une de ces deux contraintes (puisqu'on souhaite déplacer par la souris une ligne ayant pour extrémités les points A et B par le point A) ou encore une valuation qui satisfait les contraintes *{x.B=x.souris}, {y.B=y.souris},*

aussi, dans le cas ou il est impossible de satisfaire les contraintes *{x.A=x.souris}* et *{y.A = y.souris}* alors on va essayer de satisfaire les autres contraintes de remplacement.

Le comparateur *Cardinal-Pire-Cas* est un comparateur global qui discrimine mieux l'ensemble des valuations qui satisfont les contraintes requises que le comparateur *Pire-Cas* (ou que le comparateur *Localement-Prédicat-Meilleur*). Par conséquent, la taille de l'ensemble des solutions d'une hiérarchie en utilisant ce critère est généralement plus petite que si l'on utilise le critère *Pire-Cas*. Pour illustrer ceci considérons l'exemple suivant :

Exemple :

Soient $H = \{H_0, H_1, H_2\}$ avec H_0 la classe des contraintes requises. H_1 est la classe des contraintes étiquetées par l'étiquette *forte*. H_2 est la classe des contraintes étiquetées par l'étiquette *moyenne*. H_0 contient deux contraintes c_{01} et c_{02}. H_1 contient quatre contraintes $c_{11}, c_{12}, c_{13}, c_{14}$ et ont respectivement pour indices *0.9, 0.9, 0.9, 0.8*. H_2 contient trois contraintes c_{21}, c_{22}, c_{23} qui ont respectivement pour indices *0.9, 0.9, 0.9*. On rappele que l'erreur prédicat utilisée rend 0 si la contrainte est satisfaite et 1 sinon. On suppose aussi que :

Les valuations θ, φ et τ ne satisfont pas respectivement les ensembles de contraintes : $\{c_{01}, c_{02}, c_{11}, c_{12}, c_{14}, c_{21}, c_{22}\}$, $\{c_{01}, c_{02}, c_{11}, c_{14}, c_{21}, c_{22}\}$ et $\{c_{01}, c_{02}, c_{11}, c_{13}, c_{14}, c_{21}\}$.

Les trois valuations satisfont les deux contraintes requises dans H_0 et donc $S_0 = \{\theta, \phi, \tau\}$. Pour calculer S, les séquences de satisfaction par niveau de chaque valuation en utilisant *Pire-Cas* sont :

$g(\theta, H_1) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9, 1 * 0.8) = 0.9$

$g(\theta, H_2) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9) = 0.9$

$g(\phi, H_1) = Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8) = 0.9$

$g(\phi, H_2) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9) = 0.9$

$g(\tau, H_1) = Max(1 * 0.9, 0 * 0.9, 1 * 0.9, 1 * 0.8) = 0.9$

$g(\tau, H_2) = Max(1 * 0.9, 0 * 0.9, 0 * 0.9) = 0.9$

On observe que : $g(\theta, H_1) = g(\phi, H_1) = g(\tau, H_1)$ et $g(\theta, H_2) = g(\phi, H_2) = g(\tau, H_2)$. Par conséquent $S = \{\theta, \phi, \tau\}$.

Les séquences de satisfaction par niveau de chaque valuation en utilisant *Cardinal-Pire-Cas* sont :

$g(\theta, H_1) = (Max(1*0.9, 1*0.9, 0*0.9, 1*0.8), Card(Max(1*0.9, 1*0.9, 0*0.9, 1*0.8))) = (0.9, 2)$

$g(\theta, H_2) = (Max(1 * 0.9, 1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 1 * 0.9, 0 * 0.9))) = (0.9, 2)$

$g(\phi, H_1) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8))) = (0.9, 1)$

$g(\phi, H_2) = (Max(1 * 0.9, 1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 1 * 0.9, 0 * 0.9))) = (0.9, 2)$

$g(\tau, H_1) = (Max(1*0.9, 0*0.9, 1*0.9, 1*0.8), Card(Max(1*0.9, 0*0.9, 1*0.9, 1*0.8))) = (0.9, 2)$

$g(\tau, H_2) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.9))) = (0.9, 1)$

On observe que : $g(\phi, H_1) <_{lex} g(\theta, H_1)$, $g(\tau, H_1) = g(\theta, H_1)$ et $g(\tau, H_2) <_{lex} g(\theta, H_2)$. Par conséquent $S=\{\phi\}$.

Proposition 6.1:

$\forall\theta, \eta\ \forall H$ *Pire-Cas*$(\theta, \eta, H) \rightarrow$ *Cardinal-Pire-Cas*$(\theta, \eta, H) \vee^1$ *Cardinal-Pire-Cas*(η, θ, H).

Preuve :

Supposons que *Pire-Cas*(θ, η, H) est vrai, et donc qu'il existe un niveau $k>0$ dans H tel qu'après application de chacune des contraintes sur θ jusqu'au niveau $k-1$, le plus fort indice parmi ceux des contraintes non satisfaites est égal à celui obtenu après application sur η. Ceci veut dire que l'on est dans un des deux cas suivants :

1. Après application de θ, le nombre de contraintes non satisfaites ayant le plus fort indice par niveau jusqu'au niveau $k-1$ est égal à celui obtenu après application de η. Dans ce cas au niveau k, après application de θ, il existe au moins une contrainte satisfaite par θ (et non par η) telle qu'elle possède un indice plus fort que celui de toute contrainte satisfaite par η (puisque *Pire-Cas*(θ, η, H)). Par conséquent, on a bien *Cardinal-Pire-Cas*(θ, η, H) et donc l'implication est vraie.
2. Après application de θ, il existe un niveau compris entre 1 et $k-1$ pour lequel le nombre de contraintes non satisfaites ayant le plus fort indice est soit supérieur soit inférieur (mais non égal) à celui obtenu après application de η. Si ce nombre est supérieur, alors *Cardinal-Pire-Cas*(η, θ, H) est vrai. Si ce nombre est inférieur alors *Cardinal-Pire-Cas*(θ, η, H) est vrai. Par conséquent, dans les deux cas l'implication est vraie[2].

Après avoir défini ce critère global de comparaison, la question que l'on peut se poser est : *Peut-on avoir le même ensemble de solutions que celui produit par le comparateur Cardinal-Pire-Cas en subdivisant la hiérarchie et en utilisant le comparateur Nombre-Contraintes-Non-Satisfaites* ? La réponse à cette question est négative comme le montre le paragraphe suivant :

Ici, on donnera un exemple pour lequel l'ensemble de solutions obtenu dans une hiérarchie en utilisant le comparateur *Cardinal-Pire-Cas* est différent de celui obtenu en subdivisant la même hiérarchie et en utilisant le comparateur *Nombre-Contraintes-Non-Satisfaites*.

Soient $H= \{H_0, H_1, H_2, H_3\}$ avec H_0 H_1, H_2, H_3 des classes contenant respectivement des contraintes étiquetées par les étiquettes *requise*, *forte*, *moyenne* et *faible*. H_0 contient deux contraintes c_{01} et c_{02}. H_1 contient deux contraintes c_{11} et c_{12} et sont associées respectivement aux indices *0.9*, *0.9*. H_2 contient trois contraintes c_{21}, c_{22} et c_{23} et sont associées respectivement aux indices *0.9*, *0.9*, *0.8*. H_3 contient deux contraintes c_{31} et c_{32} et sont associées respectivement aux indices *0.9*, *0.8*. On suppose aussi que :

Les valuations θ et τ satisfont respectivement les ensembles de contraintes : $\{c_{01}, c_{02}, c_{11}, c_{21}, c_{31}\}$, $\{c_{01}, c_{02}, c_{12}, c_{22}, c_{23}, c_{32}\}$.

Les deux valuations satisfont les deux contraintes requises dans H_0 et donc $S_0 = \{\theta, \tau\}$. les séquences de satisfaction par niveau de ces deux valuations en utilisant le comparateur *Cardinal-Pire-Cas* sont :

1. Ici il s'agit d'un ou exclusif.
2. C'est ce deuxième cas qui permet de mieux discriminer l'ensemble des solutions dans S_0.

$g(\theta, H_1) = (Max(0 * 0.9, 1 * 0.9), Card(Max(0 * 0.9, 1 * 0.9))) = (0.9, 1)$

$g(\theta, H_2) = (Max(0 * 0.9, 1 * 0.9, 1 * 0.8), Card(Max(0 * 0.9, 1 * 0.9, 1 * 0.8))) = (0.9, 1)$

$g(\theta, H_3) = (Max(0 * 0.9, 1 * 0.8), Card(Max(0 * 0.9, 1 * 0.8))) = (0.8, 1)$

$g(\tau, H_1) = (Max(1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 0 * 0.9))) = (0.9, 1)$

$g(\tau, H_2) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.8), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.8))) = (0.9, 1)$

$g(\tau, H_3) = (Max(1 * 0.9, 0 * 0.8), Card(Max(1 * 0.9, 0 * 0.8))) = (0.9, 1)$

On observe que : $g(\tau, H_1) = g(\theta, H_1)$, $g(\tau, H_2) = g(\theta, H_2)$ et $g(\theta, H_3) <_{lex} g(\tau, H_3)$. Par conséquent la valuation θ est préférée à la valuation τ et donc $S=\{\theta\}$ (Pour cet exemple le critère *Pire-Cas* produit le même ensemble de solution que le critère *Cardinal-Pire-Cas*).

Après subdivision de cette hiérarchie on obtient: $H=\{H_0, H_{1\text{-}0.9}, H_{2\text{-}0.9}, H_{2\text{-}0.8}, H_{3\text{-}0.9}, H_{3\text{-}0.8}\}$. $H_{1\text{-}0.9}$ contient les deux contraintes c_{11} et c_{12}. $H_{2\text{-}0.9}$ contient les deux contraintes c_{21} et c_{22}. $H_{2\text{-}0.8}$ contient la contrainte c_{23}. $H_{3\text{-}0.9}$ contient la contrainte c_{31}. $H_{3\text{-}0.8}$ contient la contrainte c_{32}. Par utilisation du critère *Nombre-Contraintes-Non-Satisfaites*, les séquences d'erreurs obtenues à la suite de l'application respectivement des valuations θ et τ sur cette hiérarchie subdivisée sont : (0, 1, 1, 1, 0, 1) et (0, 1, 1, 0, 1, 0). Le deuxième tuple étant lexicographiquement inférieur au premier, par conséquent on déduit que la valuation τ est préférée à la valuation θ et donc $S=\{\tau\}$.

La section suivante décrit la technique utilisée pour planifier les graphes solutions correspondant à ce deuxième critère intégré.

6.1.2 Graphe solution correspondant au deuxième critère

En considérant ce nouveau critère de comparaison, la définition d'un graphe lexicographiquement meilleur (*GLM*) devient la suivante :

Définition 6.3 :

Soient deux graphes-admissibles s_1 et s_2. s_1 est lexicographiquement meilleur (*GLM*) que s_2 si et seulement si, pour tout niveau jusqu'au niveau k-1, le nombre de contraintes non actives associées au plus fort indice par niveau dans s_1 est égal au nombre de contraintes non actives dans s_2 associées au même indice, et au niveau k ce nombre est strictement inférieur.

Houria utilise les étiquettes et les indices associés aux contraintes pour planifier des graphes solutions. On rappelle ici la définition d'un graphe-solution : un graphe-admissible est un graphe-solution si et seulement s'il ne possède pas de cycle ni de conflit de méthodes, et de plus, il ne doit pas posséder une méthode inactive à un niveau k, telle que si on active cette méthode, elle génère un graphe lexicographiquement meilleur.

Etant donnée une hiérarchie de contraintes, pour obtenir un graphe solution correspondant au critère défini dans cette section, Houria effectue les deux étapes suivantes :

- Au sein d'une même classe, les contraintes attachées à des indices faibles sont laissées non satisfaites (c.à.d. non actives dans le graphe solution) dans le but de satisfaire (c.à.d. activer dans le graphe solution) celles attachées à des indices plus forts de la même classe ou d'autres classes.
- Entre deux classes, les contraintes faiblement étiquetées et rattachées à des indices faibles sont laissées non satisfaites pour satisfaire celles étiquetées fortement.

Par exemple[1], considérons le graphe admissible de la figure 23. Ce graphe n'est pas un graphe solution, car les méthodes des contraintes c_2 et c_4 sont en conflit sur la variable v_2. Puisque l'indice de la contrainte c_2 n'est pas le plus fort indice de la classe des contraintes étiquetées par l'étiquette *forte*, cette contrainte peut être désactivée (ceci correspond à la réalisation de la première étape décrite précédemment). Ceci produit le graphe-admissible de la figure 24 qui n'est pas un graphe solution car la contrainte c_3 peut être activée en désactivant la contrainte c_5, cette dernière n'étant pas rattachée au plus fort indice de la classe contenant les contraintes étiquetées par l'étiquette *moyenne* (ceci correspond à la réalisation de la deuxième étape décrite précédemment). En activant la contrainte c_3 et en désactivant la contrainte c_5, on aboutit au graphe solution de la figure 25. Ce dernier est un graphe solution puisqu'il n'existe aucune contrainte désactivée qui puisse être activée et produire un graphe meilleur que celui de la figure 25.

FIGURE 23 : Graphe-admissible non GLM

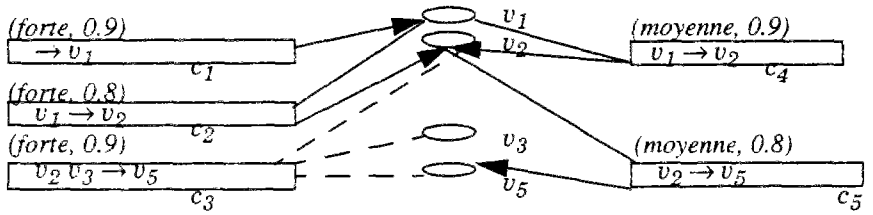


FIGURE 24 : Graphe-admissible non GLM

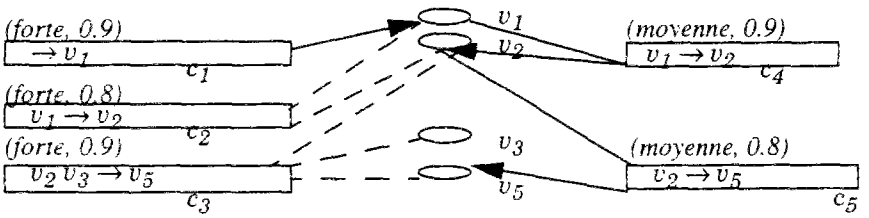
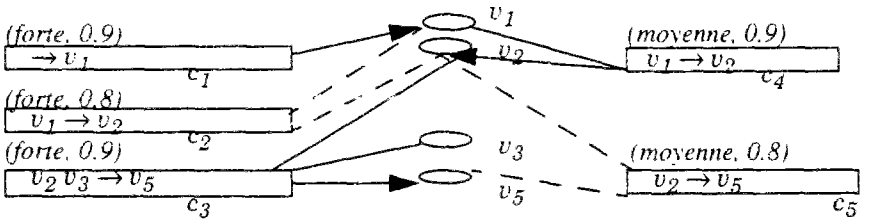


FIGURE 25 : Graphe solution



6.2 Le troisième critère de comparaison intégré dans Houria

6.2.1 Définition formelle

Le troisième critère de comparaison utilisé dans cet algorithme est le comparateur *Somme-Pondérée-Prédicat*. Ce comparateur utilise la somme des poids des contraintes satisfaites par niveau de la hiérarchie. L'ensemble des solutions obtenu par l'utilisation de ce comparateur est généralement

1. Pour simplifier l'exemple, on suppose que chaque contrainte contient une seule méthode.

de taille plus petite que celle de l'ensemble obtenu en utilisant le critère *Localement-Prédicat-Meilleur*. Ce critère est défini comme suit :

Définition 6.4:

Une valuation θ est considérée meilleure qu'une autre valuation η par le comparateur *Somme-Pondérée-Prédicat*, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, la somme des poids des contraintes non satisfaites après application de θ est égale à celui après application de η, et au niveau *k*, cet somme est strictement inférieure.

Somme-Pondérée-Prédicat(θ, η, *H*)⇔ *Globalement-Meilleur*(θ, η, *H*, *g*)

$$avec \quad g(\rho, C_i) \equiv \left(\sum_{c \in H_i} \bar{c} \, e \, (c\rho) \right).$$

On rappelle qu'avec cette définition, *S* ne doit pas contenir une solution moins bonne qu'une autre solution. *S* peut contenir plusieurs solutions (d'où l'existence de plusieurs graphes solutions GLM), aucune de ces solutions n'étant meilleure qu'une autre.

Proposition 6.2:

∀θ, η ∀*H Localement-Prédicat-Meilleur*(θ, η, *H*) → *Somme-Pondérée-Prédicat*(θ, η, *H*).

Preuve :

Supposons que *Localement-Prédicat-Meilleur*(θ, η, *H*) est vrai, et donc qu'il existe un niveau *k>0* dans *H* tel que le vecteur d'erreurs prédicats[1] obtenu après application des contraintes sur θ jusqu'au niveau *k-1* est égal à celui obtenu après application sur η. Ceci implique que : après application sur θ, la somme des éléments de ce vecteur pondérés par les poids correspondant aux contraintes jusqu'au niveau *k-1* est la même que celle obtenue par application de η. De plus au niveau *k*, et par utilisation de *Localement-Prédicat-Meilleur*(θ, η, *H*), toute contrainte satisfaite par η est satisfaite par θ. En plus, θ satisfait au moins une contrainte de plus que η. Donc, le vecteur d'erreurs prédicats obtenu après application sur θ est strictement inférieur lexicographiquement à celui obtenu après application de η. Ceci implique aussi que la somme des éléments de ce vecteur pondérée par les poids correspondant aux contraintes du niveau *k* est strictement inférieure à celle obtenue par application de η. Par conséquent, on a bien *Somme-Pondérée-Prédicat*(θ, η, *H*).

Corollaire 6.2:

Pour une hiérarchie de contraintes donnée, soit S_{LPM} l'ensemble des solutions obtenu par l'utilisation du comparateur *localement-meilleur*, et S_{SPP} celui obtenu par l'utilisation du comparateur *somme-pondérée*, on a : $S_{SPP} \subseteq S_{LPM}$.

Le comparateur *Nombre-Contraintes-Non-Satisfaites* est un cas particulier du comparateur *Somme-Pondérée-Prédicat* utilisant le même poids pour chacune des contrainte de chaque niveau de la hiérarchie. Dans ces conditions $S_{NCNS} \equiv S_{SPP}$.

La section suivante décrit la technique utilisée pour planifier les graphes solutions correspondant à ce troisième critère.

1. Chaque élément dans ce vecteur est égal à 0 si la contrainte est satisfaite et à 1 sinon.

6.2.2 Graphe solution correspondant au troisième critère

En considérant ce critère de comparaison, la définition d'un graphe lexicographiquement meilleur (*GLM*) devient la suivante :

Définition 6.4 :

Soient deux graphes-admissibles s_1 et s_2, s_1 est lexicographiquement meilleur (*GLM*) que s_2 si et seulement si, pour tout niveau jusqu'au niveau *k-1*, la somme des poids des contraintes par niveau non-actives dans s_1 est égale à celle des contraintes non-actives par niveau dans s_2, et au niveau k cette somme est strictement inférieure.

Etant donnée une hiérarchie de contraintes, pour obtenir un graphe solution correspondant au critère défini dans cette section, Houria effectue les deux étapes suivantes :

- Au sein d'une même classe, les contraintes pondérées par des poids faibles sont laissées non satisfaites (c.à.d. non actives dans le graphe solution) dans le but de satisfaire (c.à.d. activer dans le graphe solution) celles pondérées par des poids plus forts.
- Entre deux classes, les contraintes faiblement étiquetées sont laissées non satisfaites pour satisfaire celles étiquetées fortement.

Par exemple, considérons le graphe-admissible de la figure 26. Ce graphe-admissible n'est pas un graphe-solution puisque les méthodes des contraintes c_1 et c_2 sont sur conflit en la variable v_3. Puisque le poids de la contrainte c_2 est plus petit que celui de la contrainte c_1, la contrainte c_2 est désactivée (ceci correspond à la réalisation de la première étape décrite précédemment). Ceci produit le graphe-admissible de la figure 27.

Le graphe-admissible de la figure 27 n'est pas un graphe-solution puisque la contrainte c_3 peut être activée en désactivant la contrainte c_4 du fait que cette dernière est étiquetée par une étiquette plus faible que celle de c_3 (ceci correspond à la réalisation de la deuxième étape décrite précédemment).

En activant les contraintes c_3 et c_5 et en désactivant la contrainte c_4, le graphe-admissible obtenu est celui de la figure 28. Ce dernier est un graphe-solution puisqu'il n'existe aucune contrainte désactivée qui peut être activée et produire un graphe meilleur que celui-là.

FIGURE 26 : Graphe admissible non GLM

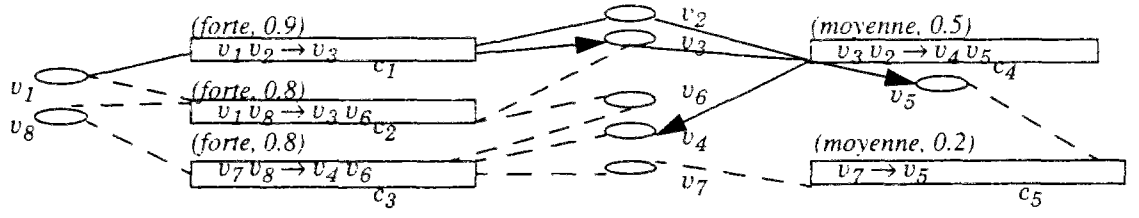


FIGURE 27 : Graphe-admissible non GLM

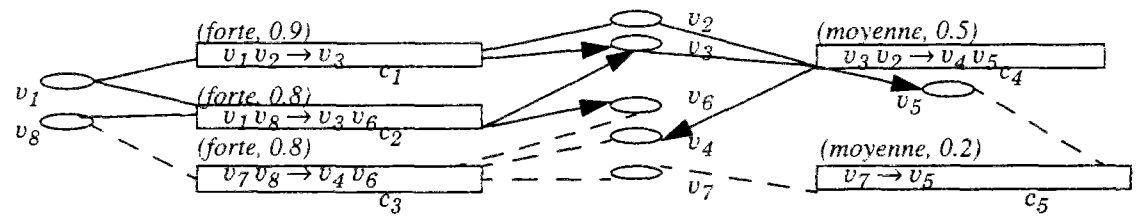
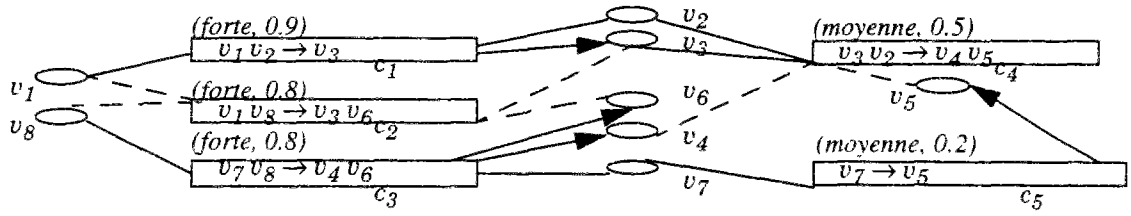


FIGURE 28 : Graphe- solution



6.3 Procédures généralisées d'ajout et du retrait de contrainte

Houria est appelé par deux procédures, *Ajout-Contrainte* et *Retrait-Contrainte*, sur chaque graphe-admissible dans G. C'est ainsi qu'il met à jour ces graphes en tenant compte des étiquettes des contraintes flexibles et de l'étiquette de la contrainte à ajouter ou à retirer pour déterminer les graphes-solutions *GLM*.

Dans cette section, on décrit en détail les procédures généralisées d'ajout et de retrait d'une contrainte c au système de hiérarchie de contraintes. On décrira également quelques propriétés de l'ensemble G (ensemble des graphes admissibles du système). On définira d'abord quelques notations utilisées par ces procédures.

Notations :

On dénote par $\overline{c}$ la paire *(étiquette, poids)* contenant l'étiquette et le poids (ou indice) de la contrainte c, et par M_c l'ensemble des méthodes de la contrainte c. Chaque méthode m dans M_c sera représentée par un couple *(triplet, $\overline{m}$)* où triplet est de la même forme que celui défini dans le chapitre précédent.

On dénote par *gaep* le couple *(ga, ep)* avec *ga* est un *triplet* représentant un graphe-admissible construit par la conjonction (définie au chapitre précédent) des méthodes qui le composent et *ep* une liste de paires *(étiquette, poids)* construite en utilisant l'opérateur $\oplus$[1] et les étiquettes ainsi que les poids des méthodes de ce graphe-solution.

$G = ((g_1,q_1)..... (g_n,q_n))$, l'ensemble G est partitionné en n sous ensembles. Chaque sous ensemble g_i contient les graphes-admissibles ayant la même *liste-réelle-étiquettes* ($g_i = \{gaep_1....gaep_m\}$ telle que $ep_1 = ... = ep_m$).

La *liste-réelle-étiquettes* d'un sous ensemble g_i est noté $\overline{g_i}$ et est égale à la liste des paires *(étiquette, poids)* d'un graphe-admissible dans g_i.

Chaque sous-ensemble g_i est associé à une queue q_i . Si l'on suppose que $q_i = \{c_0, ..., c_n\}$, on dénote par $\overline{q_i}$ la liste des paires *(étiquette, poids)* obtenue par $(\overline{c_0} \oplus ..... \oplus \overline{c_n})$.

La *liste-potentielle-étiquettes* d'un sous ensemble g_i est la liste des paires *(étiquette, poids)* obtenue par $\overline{g_i} \oplus \overline{q_i}$.

1 Cet opérateur sera paramétré par le type de comparateur utilisé par l'utilisateur et aura donc des fonctionnalités différentes

6.3.1 Procédure Ajout-contrainte

Lorsqu'une nouvelle contrainte c est introduite dans le système, la procédure *Ajouter-contrainte* ajoute cette contrainte c dans les queues associées aux sous-ensembles formés dans G, excepté la queue du premier sous ensemble de G (ligne 1; figure 29). Le premier sous-ensemble de G est retiré (ligne 2; figure 29). Ce sous-ensemble retiré est utilisé pour un ajout effectif de la contrainte c sur les différents *gaep* qui le composent (ligne 3; figure 29). La procédure fusionne et ordonne l'union de l'ensemble des couples générés par l'ajout effectif de la contrainte c sur le sous-ensemble g_1 (c.à.d. ce qui résulte de l'appel de la procédure Ajouter et affecté à G'), et les sous-ensembles restant dans G.

Cet ordonnancement est basé en premier lieu sur l'ordre lexicographique de *liste-potentielles-étiquettes* des sous-ensembles, et en second lieu sur liste-réelle-étiquettes de ces sous-ensembles. L'ensemble de ces sous-ensembles ordonné est mis dans G (ligne 4; figure 29).

La procédure examine la queue du sous-ensemble g_1 (g_1 est une variable muette. Elle représente toujours le premier sous-ensemble calculé dans G), dans le cas où cette queue est non vide (ligne 5; figure 29) (c.à.d. on n'est pas certain d'avoir calculé un graphe-solution), la procédure enlève ce premier sous-ensemble de G pour le traiter (ligne 6; figure 29).

La procédure ajoute aux différents *gaep* dans g_1 toutes les contraintes laissées en attente d'ajout dans la queue de ce sous-ensemble. On ajoute d'une manière effective la première contrainte de la queue à l'ensemble des *gaep* de ce sous-ensemble. A l'issue de cet ajout effectif, d'autres *gaep* peuvent être déduits et donc un nouvel ensemble est formé. L'ajout de la deuxième contrainte de la queue se fera sur ce nouvel ensemble. Le processus est le même pour toutes les contraint restantes dans la queue (ligne 7,8; figure 29). L'ensemble des sous-ensembles déduit de cet ajout est fusionné et ensuite ordonné avec l'ensemble des sous-ensembles restant dans G, le résultat est affecté à G (lignes 9; figure 29). La procédure continue de traiter le premier sous-ensemble dans G jusqu'à ce que sa queue devienne vide (ligne 5; figure 29). Cette manœuvre permet de calculer le(s) graphe(s)-solution(s) de la hiérarchie traitée.

La procédure retourne le premier sous-ensemble g_1 de G. Ce premier sous-ensemble contient les meilleurs[1] graphes-admissibles qui sont des graphes-solutions. L'application de l'algorithme de la propagation locale sur ces graphes-solutions produit les valuations qui sont solutions de la hiérarchie.

FIGURE 29 : Procédure d'ajout d'une contrainte au système

Ajouter-contrainte *(c , G, comp)*

1 *Pour chaque* q_i *telle que* $2 \leq i \leq n$ *faire :* $q_i \leftarrow q_i + c$ *Fin-Pour.*
2 $G \leftarrow G \setminus (g_1, q_1)$
3 $G' \leftarrow$ *(Ajouter(c,*g_1*))*
4 $G \leftarrow$ *Trier-Lex (Fusionner (G', G))*
5 *Tantque* $\neg$ *Vide(*q_1*) Faire:*
6 $G \leftarrow G \setminus (g_1, q_1)$
7 $G' \leftarrow g_1$
8 *Pour chaque* $c \in q_1$ *faire :*$G' \leftarrow$ *Ajouter(c, G', comp,G) Fin-Pour.*
9 $G \leftarrow$ *Trier-Lex(Fusionner(G', G))*
Fin-Tantque
10 *Retourner(*g_1*)*

1. Ici meilleur est relatif au comparateur *Comp* (*Comp* $\in \{\tau_{NCNS}, \tau_{CMC}, \tau_{SFP}\}$)

L'ajout effectif d'une contrainte c à un sous-ensemble de couples *gaep* (chaque *gaep* dans g_i est composé d'un graphe-admissible et de la liste construite par les paires *(étiquette, poids)* des méthodes qui composent ce graphe-admissible) est réalisé par la procédure *Ajouter* de la figure 30. Cette procédure tente d'ajouter toutes les méthodes de la contrainte c à tous les *gaep* de g_i (ligne 2,3; figure 30). Chaque ensemble résultant de cet ajout est rangé dans l'ensemble G' qui est de même type que l'ensemble G. Ce rangement consiste à mettre chaque élément d'un ensemble résultant dans le sous-ensemble ayant la même liste-réelle-étiquettes que cet élément. La procédure retourne ces ensembles rangés contenant les nouveaux graphes-admissibles intégrant la contrainte c.

FIGURE 30 : Ajout effectif d'une contrainte sur un ensemble de graphe-admissible

```
Ajouter(c, g_i, comp,G)
1 G'← ∅
2 Pour chaque m ∈ c faire:
3    Ranger( G', Ajouter-méthode(m, g_i, comp,G)))
  Fin-Pour
5 Retourner(G')
```

FIGURE 31 : Ajout effectif d'une méthode sur un ensemble de graphe-admissible

```
Ajouter-méthode(m, g_i, comp, G)
1  G'← ∅
2  G"← ∅
3  Pour chaque gaep ∈ g_i faire:
4        Si Consistant(m, ga) alors Ranger( (ga ∧ m, ep ⊕_comp m̄ ), G')
5              sinon soit M_r ={m' ∈ sg / m̄' =( requise,⊥)}
6                    Si Consistant(m, M_r) alors
7                       G" = {(M, M) ,∅ / (M ⊂ ga) ∧ Consistant(m, M))}
8                       Pour chaque gaep' ∈ G" fair :_
9                         gaep'← (ga' ∧ m, ep' ⊕_comp m)
10                        Si ¬((∃ gaep ∈ g_k /(g_k,∅)∈ G' ∧ gaep' ⊆ gaep)
11                              ∨(∃ gaep ∈ g_j /(g_j,q_j)∈ G ∧ gaep' ⊆ gaep))
12                        alors Ranger((gaep',∅) , G')
                          Fin-Si
                        Fin-Pour
13                    sinon Ranger(gaep, G')
                    Fin-Si
       Fin-Si
    Fin-Pour

14 Retourner(G')
```

La procédure *Ajouter-méthode* effectue un ajout de la méthode m sur les différents graphes-admissibles dans g_i(ligne 3; figure 31). La procédure teste la consistance (définie dans le chapitre précédent) de la méthode avec un graphe-admissible. S'il y a consistance, ce graphe-admissible est modifié en effectuant la conjonction (définie dans le chapitre précédent) de sa représentation et celle de la méthode et en ajoutant la paire *(étiquette, poids)* de la méthode à la liste des paires du graphe-admissible (lignes 4 ; figure 31) (cette ajout est réalisé par l'opérateur $\oplus$. Cet opérateur est paramétré par le comparateur utilisé. On décrira comment cet opérateur procède dans un paragraphe ultérieur).

Dans le cas contraire, la procédure extrait de ce graphe-admissible l'ensemble des méthodes requises et les met dans M_r (qui sont étiquetées par l'étiquette *requise*) (ligne 5; figure 28). Si ce dernier est consistant avec la méthode à ajouter (ligne 6; figure 28) (ceci veut dire qu'il n'y a pas de conflit ni de cycle entre la méthode à ajouter et les méthodes requises du graphe-admissible, et, l'ajout de cette méthode est donc possible), un ensemble de nouveaux graphes-admissibles sera déduit à partir des méthodes consistantes avec la méthode à ajouter (lignes 8; figure 31). L'ensemble $G"$ comporte cet

ensemble déduit de graphes-admissibles. *G"* possède la propriété de ne contenir que des graphes-admissibles maximaux (c.à.d *G"* ne comporte pas deux graphes-admissibles qui se recouvrent).

La méthode *m* est intégrée à chaque graphe-admissible déduit (ligne 9,10; figure 31). Le nouveau graphe-admissible obtenu est rangé dans *G'* s'il est maximal[1] (lignes 10,11,12; figure 31). Si la méthode *m* n'est pas consistante avec une des méthodes requises du graphe-admissible du couple *gaep* alors cette méthode est rejetée et le couple *gaep* est rangé dans *G'* (ligne 13, figure 31). La procédure retourne l'ensemble des sous-ensembles déduits à l'issue de l'ajout de la méthode *m* au sous-ensemble g_i (ligne 14 ; figure 31).

L'opérateur ⊕ est paramétré par le comparateur utilisé pour la résolution de la hiérarchie. Ce comparateur (noté par *Comp* dans les procédures) peut être un des trois comparateurs globaux définis auparavant. Lors de la conjonction d'une méthode avec un ensemble de méthodes, les opérations réalisées par cet opérateur sont comme suit :

> Lorsque le comparateur *Nombre-Contraintes-Non-Satisfaites* est utilisé, l'opérateur ⊕ fusionne deux listes ordonnées d'étiquettes en une seule liste ordonnée.
>
> Lorsque le comparateur *Cardinal-Pire-Cas* est utilisé, l'opérateur ⊕ fusionne deux listes ordonnées de paires *(étiquette, (indice,Card(indice)))* en une seule liste en groupant respectivement les couples ayant les mêmes indices et les mêmes étiquettes.
>
> Si le comparateur *Somme-Pondérée-Prédicat* est utilisé alors l'opérateur ⊕ fusionne deux listes ordonnées de paires *(étiquette, poids)* en une seule liste en ajoutant respectivement les poids dans les couples ayant les mêmes étiquettes.

6.3.2 Procédure Retirer-contrainte

Dans le but de retirer la contrainte *c* du système, la procédure *Retirer-contrainte* partitionne l'ensemble *G* en deux ensembles *G'* et *G"* (ligne 1;figure 32). L'ensemble G" est composé des sous-ensembles qui contiennent la contrainte *c* dans leur queue d'attente de contraintes à ajouter. Ceci implique qu'aucune des méthodes de la contrainte *c* ne figure dans les représentations des graphes-admissibles de ces sous-ensembles. Dans ce cas, la procédure de retrait de contrainte retire tout simplement la contrainte *c* de ces queues (puisqu'il n'y a pas eu un ajout effectif de *c* sur les différents graphes-admissibles de ces sous-ensembles) (ligne 2; figure 32).

L'ensemble G' est composé de sous-ensemble ne possédant pas la contrainte c dans leur queue associée. Certains de ces sous-ensembles contiennent dans leurs différentes représentations une des méthodes de la contrainte *c* (lignes 3,4,5; figure 32) (c.à.d. qu'il y a eu un ajout effectif de la contrainte *c* sur les différents graphes-admissibles des couples de ces sous-ensembles. Parmi ces sous-ensembles, certains peuvent parfaitement avoir des queues non vides). Dans ce cas, la procédure de retrait procède de la manière suivante : pour un graphe-admissible qui inclut dans sa représentation une des méthodes de la contrainte *c* (ligne 5; figure 32), ce graphe-admissible sera enlevé du sous-ensemble qui le contient (ligne 6; figure 32). La méthode de la contrainte *c* sera retirée de ce graphe-admissible en utilisant l'opérateur — et une nouvelle représentation sera déduite (ligne 7; figure 32). Cette dernière est prise en compte s'il n'existe aucune autre représentation dans les sous-ensembles de *G'* ou ceux de *G"* qui la recouvre, ceci pour garder la maximalité des graphes-admissibles (ce

1. Bien que ce graphe-admissible soit maximal par rapport à l'ensemble G", il peut ne pas être maximal par rapport à l'ensemble G' (ou à l'ensemble *G*) comme le montre l'exemple suivant : - hypothèses: seule la méthode m_1 est requise, $g_i=\{(m_1 \wedge m_2,\ \bar{m}_1 \oplus \bar{m}_2),(m_1 \wedge m_3,\ \bar{m}_1 \oplus \bar{m}_3)\}$, ¬consistante($m_3$, m_4) , consistante(m_1 , m_4), consistante(m_2 , m_4) et on veut ajouter la méthode m_4 aux gaep de g_i. Au pas 4 de l'algorithme, l'examen du premier couple de g_i résulte en $G' = \{(m_1 \wedge m_2 \wedge m_4,\ \bar{m}_1 \oplus \bar{m}_2 \oplus \bar{m}_4), \emptyset)\}$. L'examen du deuxième couple au pas 7 résulte en $G''=\{(m_1,\ \bar{m}_1), \emptyset)\}$ au pas 9 : $gaep'=(m_1 \wedge m_4,\ \bar{m}_1 \oplus \bar{m}_4)$. On voit bien que le couple calculé est recouvert par le couple dans *G'* .

nouveau *gaep* déduit ne sera plus un élément de g_i puisqu'il ne possède pas la même *liste-réelle-étiquettes* que celles des éléments restants dans g_i. Un nouveau sous-ensemble contenant ce *gaep* sera construit et la queue q_i lui sera associée. Ce nouveau sous-ensemble sera rangé dans *G")* (ligne 8; figure 32).

Les ensembles *G'* et *G"* sont fusionnés et le résultat est ordonné selon la *liste-potentielle-étiquettes* des sous-ensembles qui la composent (cette opération consiste à fusionner les sous-ensembles ayant la même queue de contraintes à ajouter et la même *liste-réelle-étiquette* en un seul sous-ensemble)(ligne 9; figure 32).

Après cet ordonnancement, il se peut que la queue des contraintes à ajouter du premier sous-ensemble dans *G* soit non vide (ligne 10; figure 32). Dans ce cas, l'opération d'ajout effectif des contraintes de cette queue sur les couples de ce sous-ensemble est déclenchée (lignes 11,12,13,14; figure 32). La procédure s'achève lorsque la queue du premier sous-ensemble dans *G* est vide (c.à.d. *liste-potentielle-étiquettes*(g_1) = *liste-réelle-étiquettes*(g_1)). La procédure retourne le premier sous-ensemble dans *G* (ligne 15; figure 32). Celui-ci contient les graphes-solutions *GLM* du système de contraintes hiérarchiques.

FIGURE 32 : Procédure de retrait d'une contrainte

Retirer-contrainte(c , comp,G)

```
1  Soit G' = {(g_i, q_i) / (c ∉ q_i)} et G'' = {(g_i, q_i) / (c ∈ q_i)}
2  Pour chaque (s_i, q_i) ∈ G'' faire : (q_i ← q_i \ c) Fin-Pour
3  Pour chaque (s_i, q_i) ∈ G' faire :
4    Pour chaque gaep ∈ g_i faire:
5      Si (∃ m ∈ c ∧ m ∈ ga) alors
6        g_i ← g_i \ gaep
7        gaep ← (ga — m, ep- m)
8        Si ¬(∃ gaep' ∈ g_i / ((g_i, q_i) ∈ G' ∨ ∈ G'' ) ∧ gaep ⊆ gaep') alors Ranger( (gaep,q_i), G'')Fin-Pour
       Fin-Si
     Fin-Pour
   Fin-Pour
9  G ← Trier-Lex(Fusionner(G', G''))
10 Tant que ¬ Vide(q_1) Faire:
11     G ← G\(g_1,q_1)
12     G'← g_1
13     Pour chaque c ∈ q_1 faire :G' ← Ajouter(c, G', comp,G) Fin-Pour.
14     G ← Trier-Lex(Fusionner( G', G))
   Fin-Tant que
15 Retourner(g_1)
```

6.3.3 Quelques propriétés et caractéristiques de l'ensemble *G*

Dans cette section, on caractérise d'une façon formelle cet ensemble *G* de graphes-admissibles, dans le but de clarifier la structure de cet ensemble. On imposera des axiomes qui nous serviront à prouver des théorèmes. Ces derniers nous permettront d'obtenir des propriétés sur cet ensemble *G*.

Axiome 6.1 :

$$\forall gaep \in G\ \forall m \in ga \neg(\exists m' \in ga\ /\ m = m')$$

Une méthode d'une contrainte quelconque du système existe au plus une seule fois dans un graphe-admissible de *G*.

Axiome 6.2 :

$Soit(m,m') \in (c,c')(\bar{c} \neq \bar{c}' \Rightarrow m \neq m')$

Deux contraintes différentes ont des méthodes différentes.

Axiome 6.3 :

$\neg(\exists gaep \in G\ \exists gaep' \in G\ /\ (gaep = gaep'))$

L'ensemble G ne contient pas de graphes-admissibles redondants.

Théorème 6.1:

Pour un des trois comparateurs utilisés, si un graphe ga' dans G est plus (ou aussi) important qu'un autre graphe ga dans G alors, ga' n'est pas un sous-graphe de ga.

$(\forall Comp \in \{NCNS, CPC, SPP\})(\forall gaep,gaep' \in G)ep \leq_{Lex} ep' \Rightarrow gaep' \not\subset gaep$

Preuve:

d'après l'axiome 3, on peut déduire que : $ep' \leq_{Lex} ep \Leftrightarrow \exists m \in ga \exists m' \in ga'\ /\ \bar{m}' \neq \bar{m}$ et d'après l'axiome 2, on a $m \neq m'$ et donc : $gaep' \not\subset gaep$.

Proposition 6.3:

Si une méthode est non consistante avec un sous-graphe d'un graphe-admissible, alors elle est non consistante avec ce graphe-admissible.

$(\forall ga)(\forall ga)(\forall m)ga' \subset ga \wedge \neg\text{Consistante}(m, ga') \Rightarrow \neg\text{Consistante}(m, ga)$

Théorème 6.2:

Pour un des trois comparateurs utilisés, si un graphe ga dans G est moins (ou aussi) important qu'un autre graphe ga' dans G alors, ga n'est pas un sous-graphe de ga'.

$(\forall Comp \in \{NCNS, CMC, SPP\})(\forall gaep,gaep' \in G)ep \leq_{Lex} ep' \Rightarrow gaep \not\subset gaep'$

Preuve:

On distingue trois cas :

- Si $|ga'| > |ga| \Rightarrow \exists m \notin ga \wedge m \in ga'$ et donc : $ga' \not\subset ga$

- Si $|ga'| = |ga| \Rightarrow$ par axiome 3 : $\exists m \in ga' \exists m' \in ga$ telle que $\bar{m} \neq \bar{m}'$

Or d'après l'axiome 1 on a : $m \neq m'$ et donc $ga' \not\subset ga$

- Si $|ga'| < |ga|$ on montre que $ga' \not\subset ga$ par l'absurde :

supposons que : $ga' \subset ga$, ceci implique la véracité des propositions suivantes :

- $\forall m' \in ga' \Rightarrow m' \in ga$ (puisqu'on a supposé l'inclusion)

- $\exists m \in ga$ telle que$(m \notin ga') \wedge$ Consistante$(m, ga - m)$ (puisqu'on a l'inclusion stricte)

- $\neg$Consistante(m, ga') (puisque si m était consistante avec ga', on l'aurait déjà ajoutée à ga', car par construction un graphe-admissible est maximal).

Or d'après la proposition on a: $\neg$Consistante(m, ga) et d'après l'hypothèse on a: Consistante$(m, ga - m)$ ce qui est une contradiction.

Propriété 6.1:

$$\forall gaep,gaep' \in G\ (ga - ga' \neq \varnothing \wedge ga' - ga \neq \varnothing)$$

Deux graphes-admissibles quelconques de G sont distincts.

Propriété 6.2:

$$\text{Soit } gaep \in G \text{ et } c \in H_l(\forall m \in c, m \not\subset gaep \Rightarrow \neg \text{Consistant}(c,gaep))$$

Chaque graphe-admissible dans G est maximal et consistant.

6.4 Complexité, implémentation et mesures de performance

Sannella dans [San94] prouve que la classe des problèmes concernant le comparateur *Nombre-Contraintes-Non-Satisfaites* est N-P complet. Etant donné que les deux autres comparateurs utilisés ici (*Somme-Pondérée-Prédicat* et *Cardinal-Pire-Cas*) sont plus complexes que le premier, on a tendance à croire fortement que ces problèmes sont aussi N-P complets. Dans ces conditions, il est clair que dans le pire de cas, Houria est exponentiel. Cependant, les techniques d'optimisation et l'heuristique utilisées permettent de réduire les opérations afin de converger vers une solution en un temps acceptable.

Les algorithmes généralisés et les définitions que nous avons présentés auparavant ont été programmés avec Le-Lisp version 15.25 [Ilg87]. Houria produit des graphes-solutions et leur applique la propagation locale. Dans le but de tester les performances de ce résolveur, nous l'avons expérimenté sur des problèmes sur-contraints générés aléatoirement.

Chaque problème généré est basé sur trois paramètres : le nombre *nv* de variables, le nombre *nc* de contraintes et l'arité de la contrainte *ac*. Chacune des contraintes générée est étiquetée par une étiquette générée aléatoirement parmi l'ensemble des étiquettes *{requise, forte, moyenne, faible}* et pondérée par un poids numérique généré aléatoirement dans l'ensemble fini *{0.1, 0.2, ... 1}*[1]. Parmi les critères intégrés, nous avons exécuté trois types de tests décrits dans les sections suivantes :

6.4.1 Tests avec arité fixe

Le premier type est composé de deux séries de tests :

La première série consiste en des contraintes d'arité égale à 3 (c.à.d. *ac=3*, ici on fixe le nombre de variables conséquentes de chaque méthode de la contrainte à 1. Le nombre de méthodes dans chaque contrainte générée est un nombre tiré aléatoirement entre 1 et 3, ceci est proche des applications réelles puisqu'on suppose qu'on n'a pas toujours des contraintes régulières).

La deuxième série consiste en des contraintes d'arité égale à 4 (c.à.d. *ac=4*, ici les méthodes des contraintes générées sont des méthodes dont le nombres des variables conséquentes est de 1 ou 2. Le nombre de méthodes de chaque contrainte générée dépend du nombre de variable conséquentes de la contrainte. Dans le cas où ce nombre est égal à 1 alors le nombre de méthodes dans la contrainte est un nombre tiré aléatoirement entre 1 et 4, tandis que si ce nombre est égale à 2 alors le nombre de méthodes dans la contrainte est un nombre tiré aléatoirement entre 1 et 6).

1. Excepté pour les tests utilisant le critère *Nombre-Contraintes-Non-Satisfaites* où le poids est 1 pour toutes les contraintes générées.

Pour chacune des séries, nous avons mesuré le temps mis par Houria pour planifier un graphe solution. Nous avons répété les mesures pour 50 problèmes différents sur-contraints générés et nous avons calculé le temps moyen de ces mesures. Les résultats obtenus en utilisant les critères *Nombre-Contraintes-Non-Satisfaites*, *Cardinal-Pire-Cas* sont respectivement dans les figures 33 et 34. L'axe des abscisse représente le nombre de contrainte introduites, et l'axe des ordonnés est en échelle logarithmique et représente le temps mis pour planifier un graphe solution. Les différentes mesures des figures 33 et 34 peuvent être interprétées comme suit :

Pour un nombre donné de variables *nv*, le nombre de contrainte générées est faible par rapport au nombre total qui peut être généré. Par exemple, considérons les contrainte d'arité 4, pour *nv=30* le nombre de méthodes différentes qu'on peut générer est de $(4+6)\times C_{30}^{4} = 274050$. Si l'on suppose qu'en moyenne il y a 5 méthodes par contrainte, on aura 54810 contraintes différentes et donc, 100 contraintes générées représentent une "densité" de 0.002.

Pour un nombre de contraintes donné, plus le nombre de variables est grand moins le temps mis est important. Ceci est parfaitement normal, puisque le tirage des contraintes se fait d'une façon aléatoire et on a plus de chance d'avoir moins de conflits et moins de cycles lorsqu'on considère un ensemble de variables plus important à nombre de contraintes fixé.

Le temps mis en considérant des contraintes d'arité 4 est supérieur à celui mis en considérant des contraintes d'arité 3. Ceci peut s'expliquer par le fait qu'avec les contrainte d'arité 4, on traite plus de conflits et plus de circuits qu'avec les contraintes d'arité 3.

Avec *ca* =3 ou *ca* = 4, le temps mis en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* est strictement inférieur à celui mis par le critère *Cardinal-Pire-Cas*. Ceci peut s'expliquer par le fait suivant :

on effectue plus d'opérations avec ce deuxième critère qu'avec le premier. En effet, puisqu'avec le deuxième critère les contraintes sont étiquetées et possèdent des indices, alors chaque contrainte tirée a plus de chance de déclencher l'examen d'un nombre plus important d'éléments dans *G* que si la contrainte était uniquement étiquetée. Ceci est dû à la définition de ce deuxième critère : on considére qu'une solution est aussi bonne qu'une autre si toutes les deux satisfont le même nombre de contraintes associées au plus fort indice sans se préoccuper des contraintes de remplacement (celles associées à des indices moins forte). Dans ces conditions, deux (ou plusieurs) graphes de méthodes actives feront partie d'un même sous-ensemble (possèdent la même *liste-réelle-étiquettes*). L'ajout d'une nouvelle contrainte se fera d'une façon effective sur tous les graphes de cet ensemble. Par conséquent le temps de traitement devient plus important. Par exemple, considérons les trois graphes de méthodes ga_1, ga_2 et ga_3 telle que :

ga_1 active la contrainte c_1 étiquetée par *forte* et associée à l'indice 0.9,

ga_2 active les deux contraintes c_2 et c_3 étiquetées par *forte* et associées respectivement aux indices 0.9 et 0.8,

ga_3 active les trois contraintes c_4, c_5 et c_6 étiquetées par *forte* et associées respectivement aux indices 0.9 et 0.8 et 0.7,

Ces trois graphes de méthodes ferons partie d'un même sous-ensemble *g* de *G*. Ce sous-ensemble aura la liste *Liste-réelle-étiquettes* égale à *(forte, (0.9,1))*. L'ajout d'une nouvelle contrainte sur ce sous-ensemble se fera sur les trois graphes-solutions dans *g*.

Tandis que si l'on considère le critère *Nombre-Contraintes-Non-Satisfaites* et on suppose également qu'on a les trois graphes de méthodes ga_1, ga_2 et ga_3 et que les contraintes sont uniquement étiquetées alors, on aura trois sous-ensembles g_1, g_2 et g_3 ordonnés dans *G* telle que $g_1=\{ga_3\}$, $g_2=\{ga_2\}$ et $g_3=\{ga_1\}$. L'ajout d'une nouvelle contrainte se fera en

priorité sur g_1 ensuite si on n'est pas certain d'avoir calculé le meilleur graphe, alors on continu sur g_2 etc.

Par le critère *Nombre-Contraintes-Non-Satisfaites* (resp. *Cardinal-Pire-Cas)*, lorsqu'en moyenne le nombre de contraintes est trois fois (resp. 2.2 fois) supérieur au nombre de variables, le temps mis (entre 0 et 100 secondes) est acceptable pour certaines applications graphiques. Mais lorsque le nombre de contraintes est plus que 3 fois (resp. 2.2 fois) le nombre des variables, le temps mis est très long. Par exemple, dans la figure 34.b, en considérant *nv=30* les temps, en secondes, respectifs pour traiter 60, 80 et 100 contraintes sont: 51, 125, 368.

FIGURE 33 : Temps d'exécution avec le critère NCNS lorsque ac=3, ac=4 et nv varie.

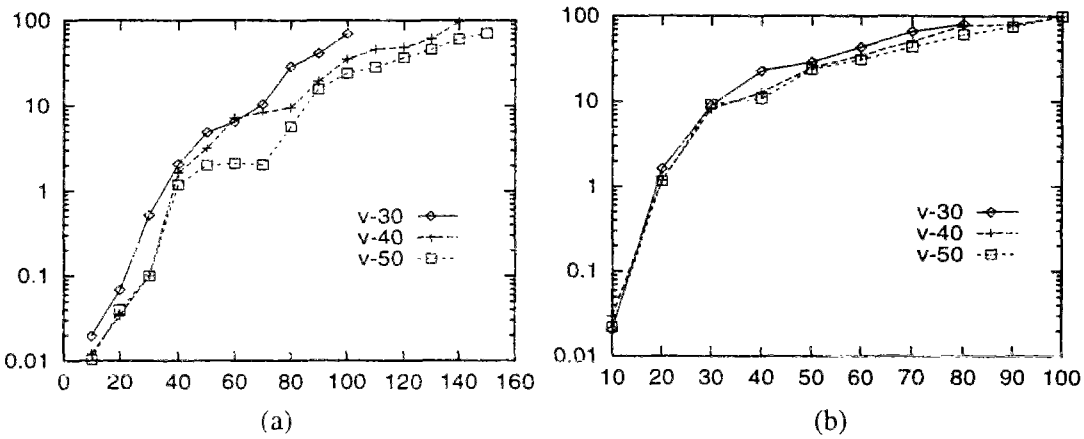
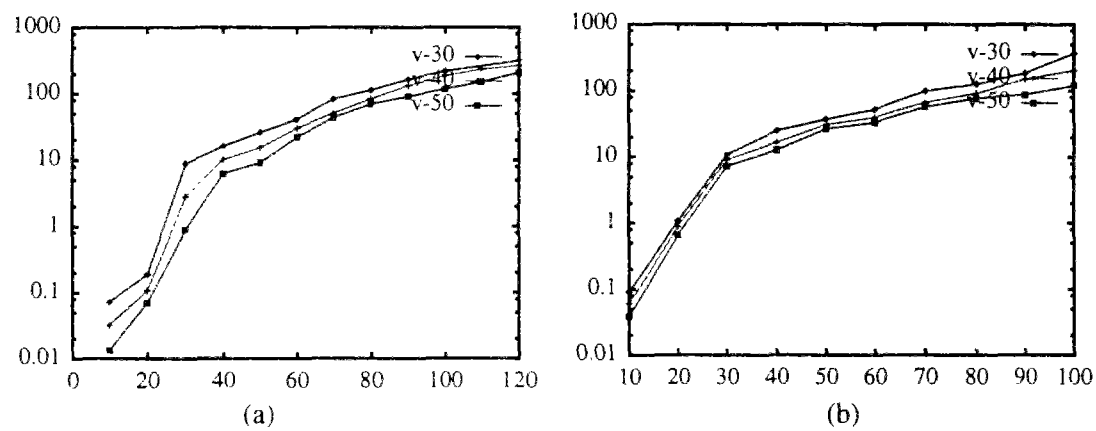


FIGURE 34 : Temps d'exécution avec le critère CPC lorsque ac=3, ac=4 et nv varie



6.4.2 Tests avec arité variable

Dans ce deuxième type de tests, le nombre de contraintes varie de 20 à 200, par pas de 20. Pour ce type de test, les contraintes générées sont d'arités 2, 3 ou 4. 70% des contraintes générées sont d'arité 2, 20% d'arité 3 et 10% d'arité 4. Ici on génére des hiérarchies de contraintes qui sont plus proche de hiérarchies d'applications réelles. On suppose également que :

pour les critères *Nombre-Contraintes-Non-Satisfaites* et *Somme-Pondérée-Prédicat* , le nombre de contraintes est trois fois supérieur au nombre de variables *nv*.

Les résultats obtenus en utilisant ces deux critères apparaissent respectivement dans les figures 35 et 36.

FIGURE 35 : *Temps d'exécution avec le critère NCNS*

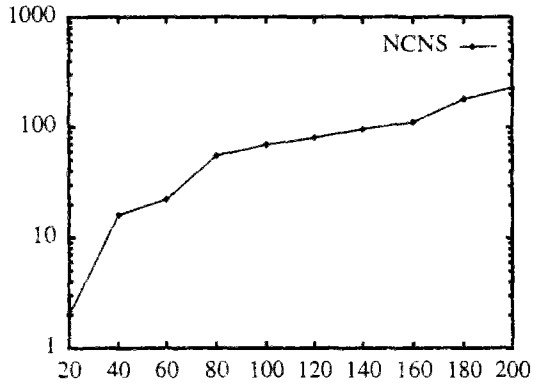
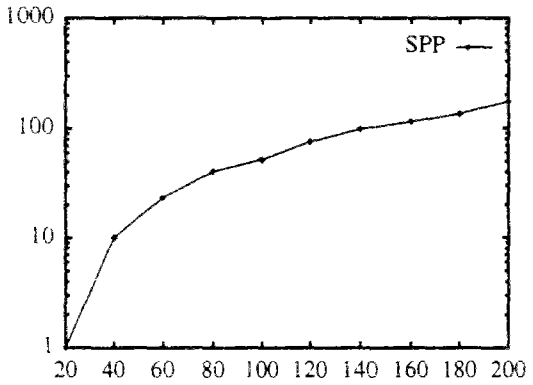


FIGURE 36 : *Temps d'exécution avec le critère SPP*



Ces différentes mesures peuvent être interprétées comme suit :

le temps mis en considérant un ensemble de contraintes contenant 70% de contrainte d'arité 2, 20% de contraintes d'arité 3 et 10% de contraintes d'arité 4 est inférieur celui mis en considérant uniquement des contraintes d'arité 3 ou 4. Ceci peut s'expliquer par le fait que la majorité des contraintes générées sont binaires et donc :

on a plus de chance de traiter un nombre inférieur de conflits et de circuits que celui obtenu si on ne manipulait que des contraintes d'arité 3 ou 4,

aussi, on a plus de chance d'utiliser les théorèmes 5.2 et 5.3 pour prédire l'absence ou l'existance de circuit dans un graphe solution.

le temps mis en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* est légèrement supérieur à celui mis par le critère *Somme-Pondérée-Prédicat.* Ceci peut s'expliquer par les faits suivants :

le critère *Nombre-Contraintes-Non-Satisfaites* a le même comportement, au sens de l'implémentation, que le critère *Somme-Pondérée-Prédicat* puisque si toutes les contraintes sont pondérées par le poids 1, alors ces deux critères sont équivalents,

pour le critère *Somme-Pondérée-Prédicat* les contraintes sont pondérées par un poids tiré aléatoirement de *{0.1,...,0.9}* et donc lors de l'ajout d'une contrainte on a plus de chance de faire un nombre moins important d'opérations pour trouver un graphe solution puisque ici chaque élément de G comporte en moyenne un graphe-admissible (contrairement à lorsqu'on utilisait le critère *Cardinal-Pire-Cas*).

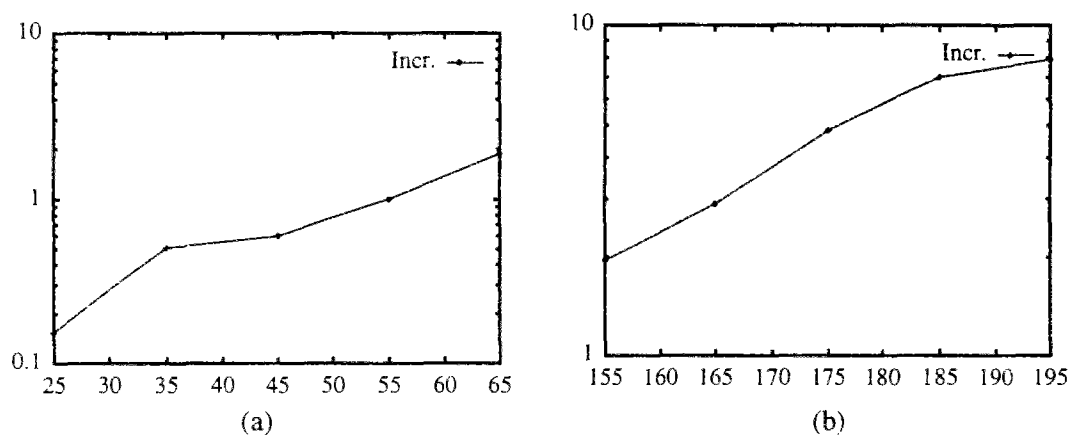
6.4.3 Tests d'incrémentalité

Le troisième type de tests consiste à tester l'aspect incrémental de ce résolveur. Ici, on suppose que le nombre de variable *nv* est le tiers du nombre de contraintes *nc* générées et que les contraintes générées ont les mêmes caractéristiques que celle du deuxiéme type de tests. Le nombre de contraintes varie de 10 à 200, par pas de 10. On calcule le temps nécessaire pour l'ajout de la $25^{ème}$ contrainte (resp. de la $35^{ème}$ contrainte, ..., et de la $195^{ème}$) générée. L'expérience est faite 50 fois et le temps moyen est reporté sur la figure. Les résultats obtenus en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* sont ceux de la figure 37.

Ces différentes mesures peuvent être interprétées comme suit :

plus le nombre de contraintes dans le système est important plus le temps mis pour l'ajout d'une contrainte est important, ceci est dû au nombre d'éléments dans G qui croît vite par rapport au nombre de contraintes introduite au système et par conséquent, d'un ajout à un autre, on est amené à examiner plus d'éléments de G pour trouver une solution.

FIGURE 37 : Temps d'incrémentalité pour le critère NCNS



6.5 Comparaison fonctionnelle avec d'autres résolveurs

Comme il est mentionné au chapitre 4, SkyBlue est le successeur de DeltaBlue. Comme QuickPlan et Deltablue, SkyBlue est un résolveur de contraintes fonctionnelles d'un système hiérarchique. QuickPlan, DeltaBlue et SkyBlue manipulent des contraintes fonctionnelles. Chacune de ces contraintes peut avoir plusieurs méthodes. Chaque méthode peut avoir plusieurs variables en entrée et plusieurs en sortie. QuickPlan, DeltaBlue et SkyBlue utilisent la propagation locale basée sur le comparateur *Localement-Prédicat-Meilleur*, et construisent des graphes localement meilleurs (GLM). SkyBlue supporte les graphes de contraintes cycliques et fait appel à un résolveur qui satis-

fait ces contraintes simultanément. Les principales différences entre Houria, QuickPlan, SkyBlue et DeltaBlue sont les suivantes :

- Skyblue traite les contraintes cycliques (en faisant appel à un résolveur de cycle) et les contraintes possédant plusieurs méthodes, tandis que QuickPlan, DeltaBlue et Houria traitent les contraintes possédant plusieurs méthodes et éliminent les cycles. Houria peut être étendu pour manipuler les cycles, il suffit de prévoir un résolveur de cycle et lors de la détection d'un cycle par la définition *Pas-De-Cycle*, faire appel à ce résolveur de cycle (qui résout les contraintes simultanément).

- Etant donné un système hiérarchique de contraintes, Houria manipule plusieurs graphes solutions (ceci est particulièrement intéressant, si l'utilisateur désire avoir plus d'une solution lors de l'ajout ou du retrait d'une contrainte) tandis que QuickPlan, DeltaBlue et SkyBlue manipulent un seul graphe solution qui est localement meilleur.

- Pour un problème sur-contraint, les critères de comparaison utilisés par Houria sont plus fins[1] et donnent des solutions ayant une qualité meilleure[2] que celles produites par le critère utilisé dans QuickPlan, SkyBlue ou DeltaBlue.

- Si la hiérarchie de contraintes possède une ou plusieurs solutions cycliques et au moins une solution acyclique, alors Houria et QuickPlan garantissent de trouver cette solution acyclique, tandis que SkyBlue est incapable de garantir cette solution.

- Le mode de représentation des contraintes utilisé par Houria est souple : il permet de manipuler les graphes solutions comme des ensembles de variables, ce qui rend les calculs moins coûteux en temps et en espace mémoire. L'intégration dans Houria de n'importe quel critère global de comparaison basé sur l'erreur prédicat est possible. Il suffit d'attribuer des nouvelles fonctionnalités à l'opérateur $\oplus$. Tandis que l'intégration d'un nouveau critère local ou global à l'un des résolveurs suivant QuickPlan, DeltaBlue ou SkyBlue nécessiterait la modification des noyaux de ces derniers.

- Houria manipule des hiérarchies de contraintes où les contraintes d'un niveau donné peuvent être pondérées par des poids numériques (ou indices de satisfaction) tandis que QuickPlan, DeltaBlue et SkyBlue manipulent des hiérarchies de contraintes où les contraintes sont uniquement étiquetées.

L'exemple suivant illustre la comparaison entre un des critères utilisés par Houria (c.à.d *Nombre-Contraintes-Non-Satisfaites*) et celui utilisé par SkyBlue (c.à.d *Localement-Prédicat-Meilleur*).

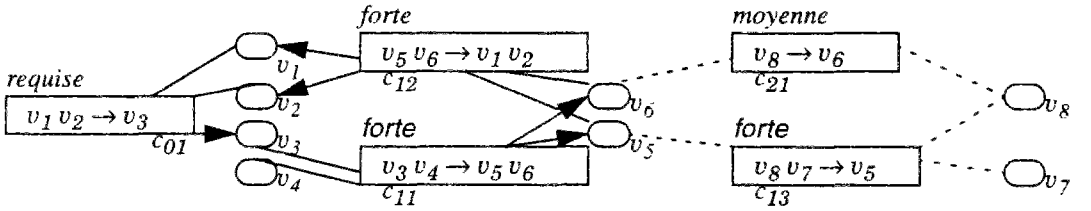
$H = \{ H_0, H_1, H_2 \}, H_0 = \{ c_{01} \}, H_1 = \{c_{11}, c_{12}, c_{13}\}, H_2 = \{c_{21}\}, M_{c_{01}} \equiv \{v_1 v_2 \rightarrow v_3\}$,

$M_{c_{11}} \equiv \{v_3 v_4 \rightarrow v_5 v_6\}$, $M_{c_{12}} \equiv \{v_5 v_6 \rightarrow v_1 v_2\}$, $M_{c_{13}} \equiv \{v_7 v_8 \rightarrow v_5\}$, $M_{c_{21}} \equiv \{v_8 \rightarrow v_6\}$.

Le graphe localement meilleur construit par SkyBlue est celui de la figure 38. *SkyBlue* garde la contrainte c_{13} non active, puisqu'il considère que même si elle devient active, elle ne produirait pas un graphe localement meilleur que celui de la figure 38. L'invariant en SkyBlue est qu'il n'existe pas de contrainte inactive qui puisse être activée en désactivant une ou plusieurs contraintes moins importantes. Aussi lors de l'activation d'une contrainte, il n'y a pas de conflit entre la méthode sélectionnée de cette contrainte et celle d'une autre contrainte plus importante active dans le graphe. SkyBlue garde la contrainte c_{21} inactive, puisqu'elle est en conflit avec la contrainte c_{11}, et elle est moins importante que cette dernière (c.à.d. c_{11} est étiquetée par *forte* tandis que c_{21} est étiquetée par *moyenne*).

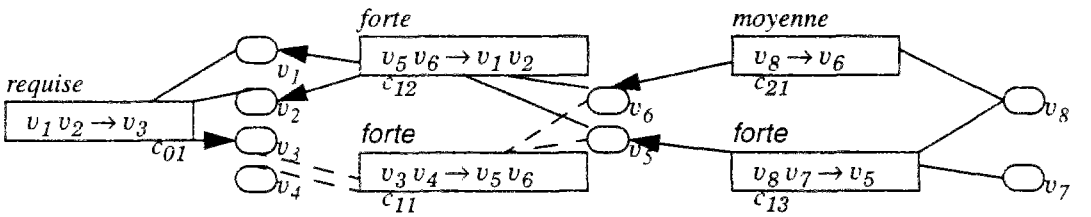
1. Dans le sens où ces critères discriminent mieux l'ensemble des valuations dans S_0 qu'un comparateur local.
2. Pour les critères *Nombre-Contraintes-Non-Satisfaites / Localement-Prédicat-Meilleur*, ces solutions sont meilleures du point de vue du nombre de contraintes satisfaites en respectant la hiérarchie

FIGURE 38 : Graphe localement meilleur de méthodes généré par SkyBlue.



Le graphe lexicographiquement meilleur construit par Houria est celui de la figure 39. La valuation produite par ce graphe doit être préférée à celle produite par le graphe localement meilleur de la figure 38, puisqu'elle est d'une qualité meilleure.

FIGURE 39 : Graphe Lexicographiquement meilleur généré par Houria.



6.6 Perspective

Pour remédier au problème de stockage des états et des tests de maximalité entre ces états, on peut procéder de la manière suivante : la gestion des graphes solutions sera effectuée par un arbre où les noeuds sont des contraintes. Chaque noeud a autant d'arcs que de méthodes plus un arc pour exprimer l'absence de cette contrainte dans la branche. Chaque chemin de la racine à une feuille correspond à un graphe de méthode. Seule la feuille de la branche courante (la branche la plus prometteuse) sera calculée en utilisant l'opérateur de conjonction sur les différentes méthodes qui sont les arcs de ce chemin. L'ensemble G aura la même structure que celle décrite auparavant à savoir un ensemble d'états où chacun de ces états possède une queue. Ici un état est un paquet de noeuds de l'arbre et la queue d'un état renferme les noeuds à ajouter (des contraintes à ajouter). La fonction d'évaluation reste la même.

Lors de l'ajout d'une contrainte, un nouveau noeud va être ajouté à toutes les queues dans G et au chemin courant dans l'arbre. S'il y a consistance entre ce chemin courant et le noeud ajouté, une nouvelle feuille sera calculée et constituera le graphe solution. s'il n'y a pas consistance, il faut remonter de la feuille du chemin courant jusqu'à l'arc le plus haut qui crée cette inconsistance. Ensuite il faut copier cette branche courante en une ou plusieurs sous branches maximales consistantes avec le noeud ajouté. En effet, s'il s'agit d'un conflit de méthodes, seule une seule branche sera ajoutée à l'arbre, mais dans le cas d'un circuit, plusieurs branches peuvent être déduites et ajoutées à l'arbre. La feuille du chemin courant n'est pas stockée. L'ensemble G est ensuite trié selon la fonction d'évaluation f et la branche prometteuse est déterminée. Ce processus continue jusqu'à ce que la queue du premier paquet soit vide.

Le mécanisme pour réaliser cette perspective reste le même que celui décrit auparavant, seulement ici on espère que le fait de ne garder que des références dans G et non des graphes solutions (quitte à les recalculer plusieurs fois lors d'une inconsistance ou lors de changement de branche) nous permet de réduire la complexité en espace mémoire.

Synthèse du chapitre

L'objet de ce sixième chapitre a été l'extension du nouvel algorithme incrémental pour la résolution de hiérarchies de contraintes fonctionnelles conçu au chapitre 5. Cette extension consiste à surmonter la restriction imposée par les résolveurs existants et de prendre en compte des hiérarchies contenant des contraintes étiquetées et pondérées .

D'une façon théorique, si la hiérarchie possède une ou plusieurs solutions acycliques (c.à.d. il existe une ou plusieurs valuations dans S telles que ces valuations satisfassent des ensembles de contraintes ne formant pas de cycle) alors on garantit de trouver ces solutions à partir de ces graphes-solutions en leur appliquant l'algorithme de propagation locale. Maintenant, il se peut que la hiérarchie ne possède pas de solution acyclique et possède uniquement des solutions cycliques, vue la première condition de la définition d'un graphe solution, ces graphes solutions garantissent de trouver les solutions les plus proches de celles de S. Il est à signaler ici qu'on peut aussi trouver ces solutions cycliques, il suffit d'avoir un outil (résolveur de cycle) qui résout un ensemble de contraintes formant un cycle lors de sa détection par le prédicat *Pas-De-Circuit-Bis* (défini au chapitre 5).

Les deux critères de comparaison intégrés à cet algorithme dans ce chapitre sont le comparateur *Cardinal-Pire-Cas* et *Somme-Pondérée-Prédicat*. Le comparateur *Cardinal-Pire-Cas* peut être considéré lorsque l'utilisateur désire exprimer au sein d'une même classe des contraintes de remplacement ou encore des priorités entres les méthodes des contraintes. Par exemple, si l'on considère la contrainte *Somme* $=B+C+D$. Cette contrainte peut avoir 4 méthodes où chacune calcule la valeur d'une des quatre variables à partir des trois autres. Chacune de ces méthodes peut être invoquée pour satisfaire cette contrainte. Si lors d'une modification d'une des valeurs des variable B, C ou D alors il semblerait plus naturel de répercuter cette modification sur la valeur de la variable *Somme* et non sur celle de l'une des variables non modifiées. Pour réaliser ce désir, Il suffit de pondérer la méthode calculant la variable *Somme* par un indice fort, ensuite les méthodes calculant les valeurs des autres variables par un indice moins fort.

Le comparateur *Somme-Pondérée-Prédicat* peut être considéré lorsque le poids d'une contrainte traduit son degré (c.à.d. mesure) d'importance dans la classe de la hiérarchie où elle se trouve si elle est satisfaite.

On montre également que les ensembles de solutions obtenus par ces comparateurs sont différents les uns des autres. Ces ensembles sont aussi différents de celui produit par le comparateur *Nombre-Contraintes-Non-Satisfaites* après subdivision de la hiérarchie.

Le résolveur tel qu'il est conçu au chapitre 5 a permis l'intégration des deux critères globaux décrits dans ce chapitre sans avoir à modifier son noyau. Il a fallu tout simplement attribuer des fonctionnalités différentes à l'opérateur $\oplus$ (à la fonction d'évaluation) selon le comparateur utilisé. Généralement, l'intégration de n'importe quel critère global utilisant l'erreur prédicat (qui retourne 1 si la contrainte n'est pas satisfaite et 0 sinon) dans ce résolveur est possible. Il suffit d'associer sa fonction d'agrégation d'erreur par niveau à l'opérateur $\oplus$.

7 L'utilisation de Houria dans la programmation logique par hiérarchie de contraintes.

Les contraintes fonctionnelles sont utilisées dans plusieurs applications gérées par des langages de programmation logique par contraintes comme par exemple dans les interfaces graphiques, pour des simulations physiques ou encore des contraintes géométriques. Dans plusieurs situations, il est nécessaire de pouvoir exprimer dans ces langages des contraintes dures qui doivent être satisfaites et les contraintes de préférences à satisfaire au mieux selon le comparateur utilisé.

La Programmation Logique par Hiérarchie de Contraintes (*PLHC*) étend la Programmation Logique par Contraintes (*PLC*) en incluant les hiérarchies de contraintes. La *PLC* comme la *PLHC* sont paramétrées par *D* qui est le domaine des contraintes. De plus, la *PLHC* est paramétrée par le comparateur *C* utilisé. Des prototypes de la *PLHC(D,LPM)* sont décrit dans [Wil92, WB89, BMM+89, BDF+87] où *D* est l'ensemble des réels et *LPM* est le comparateur *Localement-Prédicat-Meilleur*. Comme il est mentionné par les auteurs de ces prototypes, l'expérience d'écrire des programmes en *PLHC(R,LPM)* montre que le comparateur *Localement-Prédicat-Meilleur* produit des solutions non intuitives. Ceci résulte du fait que le comparateur *Localement-Prédicat-Meilleur* ne peut comparer que des solutions résultant d'une seule hiérarchie (ceci est appelé comparaison intra-hiérarchie). Cependant, plusieurs applications pratiques en *PLHC* suggèrent le besoin d'une comparaison entre les solutions résultant des différents choix de règles du programme afin d'éliminer celles qui sont non intuitives (c.à.d. jugées moins bonnes) (ceci est appelé comparaison inter-hiérarchies).

Pour réaliser une comparaison inter-hiérarchies, seuls les comparateurs globaux peuvent être utilisés. Le résolveur décrit dans les chapitres 5 et 6 implémente des comparateurs globaux et il peut donc être utilisé à cet effet.

Ce chapitre décrit un algorithme basé sur notre résolveur présenté dans les chapitres 5 et 6 et sur le fondement théorique de comparaison inter-hiérarchies que nous rappellerons dans ce chapitre. Ce fondement est décrit en détail dans [Wil92, WB89], il étend la définition de l'ensemble de solutions afin de donner la possibilité de pouvoir comparer deux solutions résultant de différentes hiérarchies.

Ce chapitre est composé de quatre parties. La première partie est un état de l'art ou nous présenterons l'intégration des hiérarchies de contraintes dans les langages de programmation logique. Il s'agira tout d'abord de décrire globalement la programmation logique par contraintes, ensuite la programmation logique par hiérarchie de contraintes ainsi que le schéma général étendu du comparateur *Globalement-Meilleur* utilisé pour la comparaison inter-hiérarchies. On montre *via* deux exemples le besoin de comparaison inter-hiérarchies. On discutera sur l'aspect des comparateurs, sur les contraintes non primitives et enfin sur les travaux existants dans ce domaine. Ces travaux avaient pour objectif d'obtenir les mêmes résultats que la comparaison inter-hiérarchies. L'exposé de cette partie ne se veut pas exhaustif mais ciblé sur la description des points théoriques auxquels nous ferons référence ultérieurement dans ce chapitre. Dans la deuxième partie, on présente les idées sur lesquelles est basé un nouvel algorithme pour la réalisation de cette comparaison inter-hiérarchies. La troisième partie décrit cet algorithme. Enfin, la quatrième partie décrit un exemple pour bien illustrer cet algorithme.

7.1 Hiérarchie de contraintes et programmation logique

Initialement, les hiérarchies de contraintes ont été une extension du système Thinglab [BDF+87]. Par rapport aux hiérarchies de contraintes et à la programmation logique, nous avons vu que la théorie des hiérarchies de contraintes permet à l'utilisateur de spécifier non seulement les contraintes dures mais aussi les contraintes de préférences réparties en un ou plusieurs niveaux. Ce schéma de hiérarchie de contraintes est paramétré par un comparateur C qui permet la comparaison des différentes solutions possibles d'une seule hiérarchie et le choix de la meilleure solution. Récemment, le paradigme des hiérarchies de contraintes a été intégré avec le schéma de programmation logique par contraintes (*PLC*) afin de produire la programmation logique par hiérarchie de contraintes (*PLHC*). *PLC* et *PLHC* sont paramétrées par D qui est le domaine des contraintes. De plus la *PLHC* est paramétrée par le comparateur C. Cette intégration de hiérarchies de contraintes dans la *PLC* permet à la théorie de la programmation logique d'être complétée par l'expressivité des contraintes de préférence.

Les travaux précédents concernant ce domaine sont scindés essentiellement en deux : la Programmation Logique par Contraintes (*PLC*) et l'utilisation des contraintes dans les applications. Le schéma de la *PLC* est décrit dans [JL87]. Un certain nombre d'interprètes *PLC* ont été implémentés Nons pouvons citer *Prolog III* [Col87], *CLP(R)* [HJM+87, JM87] et *CHIP* [DVS+88]. Il existe aussi des travaux considérables sur l'utilisation des contraintes dans les applications comme par exemple les contraintes géométriques, les simulations physiques, les interfaces utilisateurs, le dessin et le formatage de document, les algorithmes d'animations ou encore le dessin et l'analyse d'instruments mécaniques et de circuits électriques.

Dans cette section, nous illustrons l'extension de la notion de comparateur appliquée à la comparaison inter-hiérarchies. Les définitions précédentes des comparateurs données au chapitre 2 sont des cas particuliers dans le sens où elles utilisent une seule hiérarchie. Etant données les nouvelles définitions de cette section, il serait facile de voir que ces comparateurs inter-hiérarchies exhibent une conduite non monotone. On présentera aussi quelques propriétés intéressantes sur les hiérarchies de contraintes.

En programmation logique par contraintes, les règles sont de la forme : $p(t)$:- $q_1(t)$, ..., $q_m(t)$, $c_1(t)$, ..., $c_n(t)$ avec p, q_1, ..., q_m des symboles de prédicats et c_1, ..., c_n des contraintes. t dénote une liste de terme. En programmation logique par hiérarchie de contraintes les règles sont de la forme : $p(t)$:- $q_1(t)$, ..., $q_m(t)$, $s_1c_1(t)$, ..., $s_nc_n(t)$. s_i indique le niveau de préférence de la contrainte correspondante c_i. Des étiquettes sont associées aux différents niveaux de préférences de la hiérarchie.

L'utilisateur définit un nombre arbitraire d'étiquettes qui correspond au nombre de niveaux de la hiérarchie. Si toute étiquette s_i est "requise", alors il est clair que le programme est équivalent au même programme exprimé en *PLC* sans étiquettes sur les contraintes.

7.1.1 Fondement théorique de la comparaison inter-hiérarchies

Dans [Wil92, WB89, BDF+87], une solution d'un ensemble de hiérarchies de contraintes Δ, consiste en une valuation pour toutes les variables libres dans Δ. L'ensemble Δ consiste en des hiérarchies qui résultent des choix des règles alternatives dans un programme.

La définition de l'ensemble S de solutions au chapitre 2 reposait sur le fait que l'on considérait une seule hiérarchie. Ici, S contient toutes les solutions de Δ. Lorsque Δ contient une seule hiérarchie, la nouvelle définition de S est équivalente à celle du chapitre précédent. On définit en premier l'ensemble S_0 des valuations qui satisfont toutes les contraintes requises (par hiérarchie) des hiérarchies dans Δ. Chaque valuation θ dans S_0 est associée à la hiérarchie qu'elle satisfait. Ensuite l'ensemble S est défini comme avant, à l'exception du comparateur *meilleur* qui est paramétré par un ensemble de hiérarchies de contraintes. Par suite, les valuations potentielles d'une hiérarchie dans Δ qui sont jugées moins bonnes que d'autres valuations potentielles d'une autre hiérarchie dans Δ sont éliminées.

$$S_0 = \{\, \theta_h : \forall h \in \Delta\ \forall c \in h_0\ e\tau_0(c\theta) = 0 \}$$

$$S = \{\theta_h : \theta_h \in S_0 \wedge \forall \eta_{h'} \in S_0 \neg \mathit{meilleur}(\eta_{h'}, \theta_h, \Delta)\}$$

avec $\mathit{meilleur}(\eta_{h'}, \theta_h, \Delta) \Leftrightarrow (G(R(\Delta\eta_{h'})) <_G G(R(\Delta\theta_h)))$.

$S(C)$ est l'ensemble des solutions de l'ensemble des hiérarchies utilisant le comparateur C. Comme il est écrit au deuxième chapitre, les comparateurs sont irréflexifs et transitifs et respectent aussi la sémantique des hiérarchies dans Δ.

Puisque le comparateur *Localement-Meilleur* considère individuellement chaque contrainte de la hiérarchie pour déterminer l'effet d'une valuation, il serait inutile de considérer ce type de comparateur pour la comparaison inter-hiérarchies (c.à.d. pour comparer les différentes solutions des différentes hiérarchies dans Δ). En d'autres termes, *Localement-Meilleur* est défini seulement si Δ consiste en une seule hiérarchie.

Le schéma des comparateurs globaux *Globalement-Meilleur* a été étendu pour satisfaire la comparaison inter-hiérarchies. Nous rappelons que ce schéma est paramétré par une fonction g qui agrège les erreurs de toutes les contraintes d'un niveau de la hiérarchie. La définition de cette extension est :