4 Algorithmes de propagation locale

En préambule de ce chapitre, nous dirons que la recherche d'un algorithme efficace qui pourrait satisfaire tout type de contraintes en utilisant n'importe quel domaine et n'importe quel comparateur serait un essai futile. Ce chapitre donne un aperçu des différents algorithmes existants pour le traitement des hiérarchies de contraintes. Ce chapitre décrit la réponse à la question essentielle posée avant de concevoir tel ou tel algorithme, et qui est : "que fait le système lorsque l'ensemble des contraintes est un ensemble sur-contraint (pas de solution qui satisfasse cet ensemble) ou lorsque l'ensemble des contraintes est un ensemble sous-contraint (il y a plusieurs solutions qui satisfont cet ensemble)". Si le résolveur gère les contraintes d'une application d'interface utilisateur, alors il n'est pas acceptable de signaler une erreur de manipulation. La théorie des hiérarchies de contraintes décrite dans les chapitres précédents indique une voie pour spécifier déclarativement comment le résolveur doit réagir face à cette situation.

Une hiérarchie de contraintes est un ensemble de contraintes où chaque contrainte possède une étiquette qui exprime l'importance de cette contrainte dans le système (ou la préférence de cette contrainte par rapport aux autres contraintes). Etant donnée une hiérarchie de contraintes sur-contrainte, le résolveur peut laisser certaines contraintes parmi les moins préférées non satisfaites pour satisfaire celles qui sont les plus préférées. Si la hiérarchie est sous-contrainte, le résolveur peut choisir n'importe quelle solution. L'utilisateur peut contrôler quelle solution choisir en ajoutant des contraintes étiquetées par des étiquettes faibles. Ces contraintes pourront porter sur les variables pour exprimer le maintien de leurs valeurs (i.e. l'utilisateur exprime via ces contraintes son désir de ne pas changer les valeurs de certaines variables). Ce chapitre discutera aussi un peu plus en détail des algorithmes basés sur la propagation locale. Nous nous attarderons sur quelques aspects intéressants d'ingénierie de certains algorithmes, sur les critères de comparaison utilisés et enfin sur la complexité de ces algorithmes.

Ce chapitre est composé de quatre parties. La première décrit le principe de la propagation locale. La deuxième partie décrit quelques algorithmes existants basés sur la propagation locale pour la résolution de hiérarchie de contraintes fonctionnelles. La troisième partie donne un aperçu sur des algorithmes existants pour la résolution de hiérarchies ayant des contraintes d'égalité et d'inégalité. Et enfin la quatrième partie donne un aperçu sur d'autres algorithmes existants.

4.1 Propagation locale

Parmi les techniques ordinaires pour satisfaire les contraintes, il y a la technique nommée propagation locale. Dans cette technique, la contrainte peut être utilisée pour déterminer la valeur d'une de ses variables au cas où les valeurs de ses $n - 1$ autres variables sont connues. Dans ces conditions, on parle de contrainte à une seule variable de sortie (uni-sortie). La contrainte peut être utilisée pour déterminer les valeurs de m de ses variables si les valeurs de ses $n - m$ autres variables sont connues et si l'on dispose d'une méthode qui calcule ces m valeurs à partir des $n - m$ variables. Dans ces conditions, on parle de contrainte à plusieurs variables de sortie (multi-sorties). Il est bien évident que ces $n - m$ variables peuvent aussi être calculées par d'autres contraintes et ainsi de suite.

La propagation locale est similaire à la propagation des valeurs dans le réseau de contraintes. Cependant, un problème se pose qui est l'existence de contraintes multi-directionnelles dans le réseau et donc il existe plusieurs chemins potentiels de propagation. Le rôle des résolveurs de contraintes en général est de choisir quel chemin prendre. Dans le cas de hiérarchie de contraintes, le chemin à choisir est celui qui doit fournir "*la meilleure*[1] " solution (ou une des meilleures).

Chaque contrainte du réseau possède une ou plusieurs méthodes : il s'agit de petites procédures dont l'exécution d'une seule produit la satisfaction de la contrainte. Chaque méthode détermine la valeur d'une ou plusieurs variables de sortie à partir de celles d'entrée. Par exemple, la contrainte d'addition suivante : $A + B = C$ a trois méthodes possibles : $A \leftarrow C - B$, $B \leftarrow C - A$ et $C \leftarrow A + B$.

L'algorithme de propagation locale produit un chemin de propagation en sélectionnant une méthode de chaque contrainte de la hiérarchie (si la contrainte ne peut pas être satisfaite, cela voudrait dire que l'algorithme n'a pas pu sélectionner de méthode pour cette contrainte). Par conséquent, un chemin produit par l'algorithme de propagation locale utilise au plus une méthode pour déterminer la valeur d'une variable de sortie d'une méthode d'une contrainte. Dans ce cas on considère que toute contrainte possédant une méthode dans ce chemin est satisfaite.

Les résolveurs basés sur la propagation locale (et qui donc ne manipulent que des hiérarchies n'ayant que des contraintes fonctionnelles) sont contraints d'utiliser la fonction d'erreur prédicat qui retourne *0* lorsque la contrainte est satisfaite et *1* sinon (contrairement à la fonction d'erreur métrique). Cette restriction vient du fait que ces résolveurs utilisent les méthodes des contraintes pour les satisfaire.

La propagation locale est incapable de résoudre un ensemble de méthodes de contraintes qui forment un circuit, puisque le chemin produit par ces méthodes sera cyclique, et dans la phase d'exécution de ces méthode on risque de les exécuter un nombre infini de fois.

4.2 Algorithmes pour la résolution d'une hiérarchie de contraintes

Dans les applications interactives, les contraintes d'une hiérarchie viennent souvent s'ajouter graduellement, ceci constitue une raison pour les résoudre par un algorithme incrémental. Un algorithme incrémental maintient une solution courante résolvant les contraintes. Puisque les contraintes sont ajoutées ou retranchées au système, l'algorithme modifie cette solution courante dans le but de trouver une solution qui satisfasse la nouvelle hiérarchie de contraintes.

1. Le mot meilleur ici est relatif au comparateur intégré par le résolveur.

La plupart des algorithmes (exemple : *Blue, DeltaBlue, SkyBlue, QuickPlan* ...) traitant les hiérarchies de contraintes sont basés sur la propagation locale et sur le critère de comparaison *Localement-Prédicat-Meilleur* pour un domaine de contraintes arbitraire où le graphe de méthodes de contraintes ne contient pas de circuit.

Une version incrémentale de l'algorithme *Blue* est connue par le nom *DeltaBlue*. *Blue* et *DeltaBlue* manipulent des hiérarchies de contraintes multi-directionnelles. Les méthodes des contraintes manipulées par ces deux algorithmes ne forment pas de circuits entres elles. Pour l'algorithme *Blue*, les contraintes ne possèdent que des méthodes ayant une seule variable de sortie. Dans le but de surmonter ces restrictions, un algorithme incrémental nommé *SkyBlue* a été conçu. Cet algorithme manipule des contraintes ayant des méthodes possédant plusieurs variables de sortie. Cet algorithme fait aussi appel à un sous-résolveur qui résout les cycles de contraintes lorsqu'ils sont détectés. Dans la suite, on présentera l'architecture de ces algorithmes ainsi que leurs fonctionnalités. On présentera également l'algorithme *QuickPlan* qui manipule les mêmes types de hiérarchies que *SkyBlue*. *QuickPlan* permet de trouver une solution acyclique si elle existe. Ceci nous permettra de situer nos travaux décrits dans les autres chapitres de ce mémoire.

Pour plus de détail sur les algorithmes incrémentaux acycliques, le lecteur est invité à voir [FM89, FMB89, FMB90, Mal91]. [FMB90] et [Mal91] incluent les résultats de complexité et la preuve de complétude. Pour plus de détail sur d'autres algorithmes utilisant la propagation locale, le lecteur est invité à consulter [FW90, FWB92, Wib92, Tro95].

4.2.1 L'algorithme *DeltaBlue*

DeltaBlue est une version incrémentale de l'algorithme *Blue*. La complexité en temps de cet algorithme est de $O(N)$, N étant le nombre de contraintes du système [FW90], tandis que celle de *Blue* est de $O(N^2)$ sur le nombre de contraintes du système. *DeltaBlue* utilise trois catégories de données: les contraintes de la hiérarchie H, les variables contraintes V et la solution courante P. P est aussi appelé "plan", il s'agit d'un graphe orienté acyclique composé de contraintes et de variables. Chaque variable connaît la contrainte qui détermine sa valeur et chaque contrainte sait si elle est actuellement satisfaite et connaît la méthode utilisée (la contrainte utilise une méthode pour sa satisfaction).

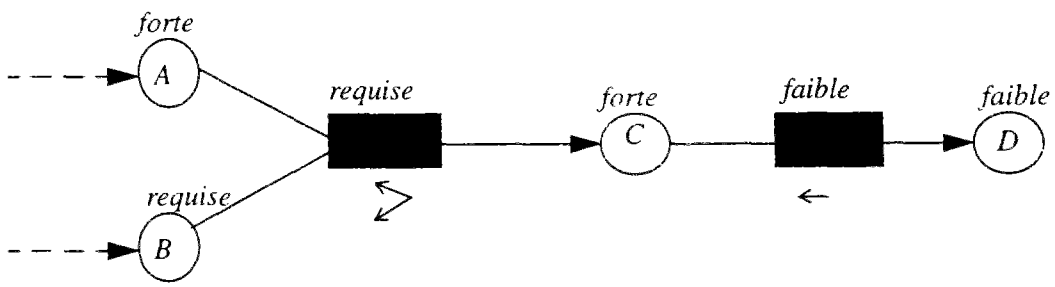
DeltaBlue comporte un programme client qui joue le rôle d'une interface à quatre points d'entrée : *ajout-contrainte*, *ajout-variable*, *enlever-contrainte* et *enlever-variable*. Les variables doivent être ajoutées au graphe avant que la contrainte qui les utilise ne soit introduite. Le retrait d'une variable entraîne le retrait de toute contrainte attachée à cette variable. D'une façon incrémentale, la solution courante P est mise à jour après exécution d'une des deux opérations *ajout-contrainte* et *enlever-contrainte*. Cette algorithme ne résout pas les hiérarchies sous-contraintes. Pour ce faire, une contrainte invisible étiquetée par *très-faible* est attachée à chaque variable quand elle est ajoutée au système. Cette contrainte invisible contraint la valeur de la variable à rester inchangée. Ceci n'a aucun effet secondaire sur la hiérarchie, car cette contrainte invisible est étiquetée par la plus faible étiquette qui puisse exister et donc n'importe quelle autre contrainte fournie par le programme client sera étiquetée par une étiquette plus forte. Cette contrainte invisible sera utilisée seulement si sa variable n'est pas contrainte par la suite (c.à.d. que la variable est sous-contrainte).

L'idée clé de cet algorithme consiste à associer une information suffisante à chaque variable le long de l'exécution de l'algorithme afin de prédire les effets d'un ajout d'une contrainte donnée. Cette prédiction se fera seulement par examen des opérandes (étiquette, variables) de la contrainte ajoutée. Cette information est appelée *étiquette-voyageuse*[1] de la variable et définie par :

La variable v est déterminée par la méthode m de la contrainte c (dans ce cas la variable v est une variable de sortie de la contrainte c). L'étiquette *étiquette-voyageuse* de v est le minimum entre l'étiquette de la contrainte c et les étiquettes associées aux variables d'entrée de la méthode m.

L'étiquette *étiquette-voyageuse* d'une variable peut être vue comme l'étiquette de la plus faible contrainte en amont, c.a.d. la contrainte précédente étiquetée faiblement et qui peut être atteinte via la séquence réversible de contraintes. Ainsi l'étiquette *étiquette-voyageuse* représente l'étiquette de la contrainte faible qui peut être enlevée du système (devient non satisfaite) pour permettre à une autre contrainte d'être satisfaite. Par exemple, la figure 7 montre un graphe de contraintes comprenant quatre variables et deux contraintes. Les variables sont représentées par des cercles et les contraintes par des rectangles. Les variables de sortie d'une méthode sont indiquées par des flèches. Une flèche en pointillé sur une variable indique que la méthode n'est pas sélectionnée pour déterminer cette variable. Un petit diagramme au-dessous de chaque contrainte indique les méthodes non sélectionnées dans la contrainte (méthodes inactives). L'étiquette *étiquette-voyageuse* de la variable D est égale à *faible*. La méthode qui détermine cette variable doit être remplacée par une autre méthode si nécessaire (c.à.d. si par la suite une contrainte portant sur la variable D d'étiquette plus forte que *faible* est ajoutée au système). L'étiquette *étiquette-voyageuse* de la variable C est *forte*, ceci vient du fait que l'*étiquette-voyageuse* de la variable A est *forte* et la contrainte portant sur les variables A et C est étiquetée par *requise* (*requise* est considérée comme supérieure à *forte*).

FIGURE 7 : étiquette-voyageuse



Ajout d'une contrainte par *DeltaBlue*

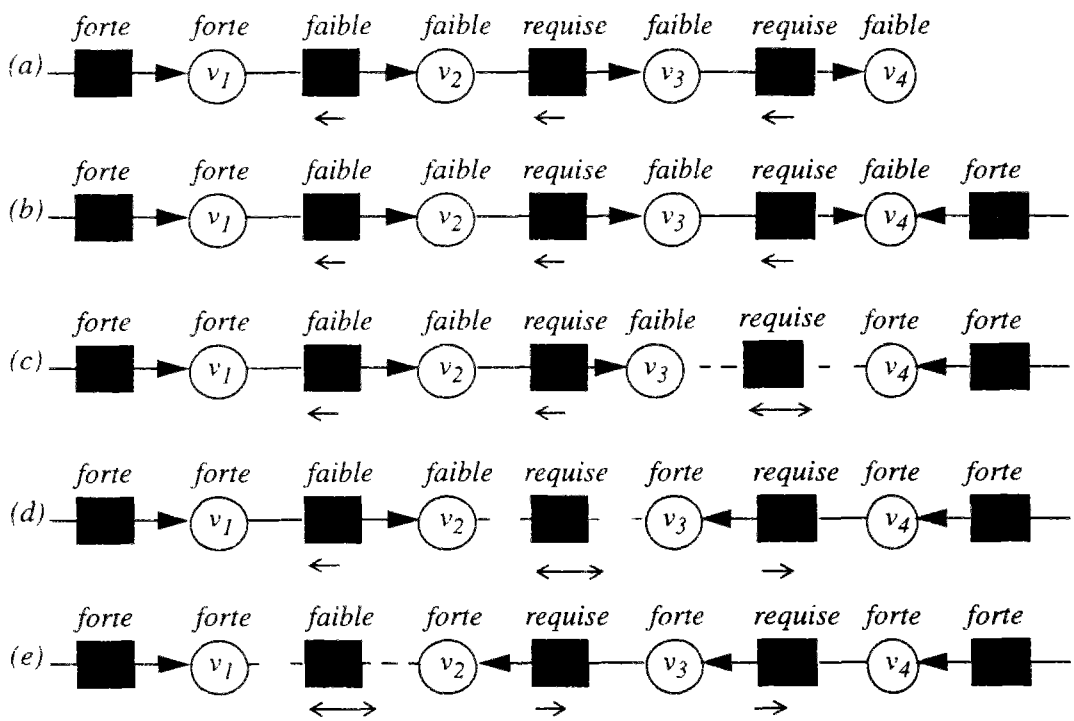
Pour satisfaire une contrainte c ajoutée au système, *DeltaBlue* doit trouver une des méthodes de cette contrainte telle que chacune de ses variables de sortie dans le réseau ait son *étiquette-voyageuse* inférieure à l'étiquette de la contrainte c. Si une telle méthode n'est pas trouvée, alors la contrainte c ne peut pas être satisfaite sans le retrait d'une autre contrainte étiquetée par une étiquette de même importance ou d'importance supérieure à celle de c. L'enlèvement d'une contrainte possédant la même étiquette (même importance) que celle de la contrainte c produira une solution différente que la solution courante mais non meilleure au sens de *Localement-Prédicat-Meilleur*. Dans ce cas, la contrainte c est laissée non satisfaite. L'enlèvement d'une contrainte possédant une étiquette plus forte que celle de la contrainte c ne peut créer qu'une solution plus mauvaise que la solution courante (mauvaise dans le sens où la solution produite satisfait la contrainte c qui est moins importante que la contrainte enlevée du système, ce qui crée une violation de la sémantique du comparateur utilisé). Dans ce cas, la contrainte c est laissée non satisfaite.

1. En anglais, cette appellation est connue par : walkabout strength. Ce nom vient d'une coutume australienne qui dit : "Pour voir plus clair il faut faire une longue marche".

La figure 8 illustre le processus d'ajout d'une contrainte. La figure 8.a montre la situation initiale avant que la contrainte ne soit ajoutée. On observe que l'*étiquette-voyageuse* de la variable v_4 est à faible, ceci est dû à l'étiquette de la contrainte portant sur les variables v_1 et v_2. Dans la figure 8.b, le programme client ajoute une contrainte étiquetée par *forte* et portant sur la variable v_4. *DeltaBlue* considère que cette contrainte doit être ajoutée puisque son étiquette est supérieure à *faible* et par conséquent elle aura comme effet le retrait de la contrainte qui a eu pour effet de mettre la valeur *faible* à l'étiquette *étiquette-voyageuse* de la variable v_4. *DeltaBlue* satisfait la nouvelle contrainte ajoutée au système et retire la contrainte qui déterminait précédemment la variable v_4 (la contrainte portant sur v_3 et v_4) en créant l'état décrit dans la figure 8.c.

Maintenant, du fait que les *étiquette-voyageuse* de v_3 et *étiquette-voyageuse* de v_4 sont moins fortes que celle de la contrainte qui relie ces deux variables, *DeltaBlue* sait qu'il doit resatisfaire cette contrainte. L'algorithme choisit toujours de modifier la variable étiquetée par l'*étiquette-voyageuse* la moins forte. Dans ce cas, c'est la variable v_3 qui est concernée, ainsi l'algorithme resatisfait la contrainte dans la nouvelle direction comme il est montré par la Figure 8.d. Cette étape a pour effet de retirer la contrainte qui détermine la variable v_3. Cette contrainte est alors considérée candidate pour être satisfaite dans une nouvelle direction. Le processus de propagation continue pour atteindre la contrainte responsable de la transmission du contenu de l'étiquette *étiquette-voyageuse* (=*faible*) de la variable initiale. L'algorithme se termine puisque cette dernière contrainte n'est pas assez importante pour qu'elle soit resatisfaite comme il est montré par la Figure 8.e. Si la contrainte initiale à ajouter était étiquetée par *très-faible* alors elle ne serait pas assez importante pour pouvoir déterminer la valeur de la variable v_4 (du fait que la valeur de la variable v_4 est au moins influencée par une contrainte étiquetée par *faible* qui est en tout cas supérieur à l'étiquette *très-faible*).

FIGURE 8 : Ajout d'une contrainte par DeltaBlue

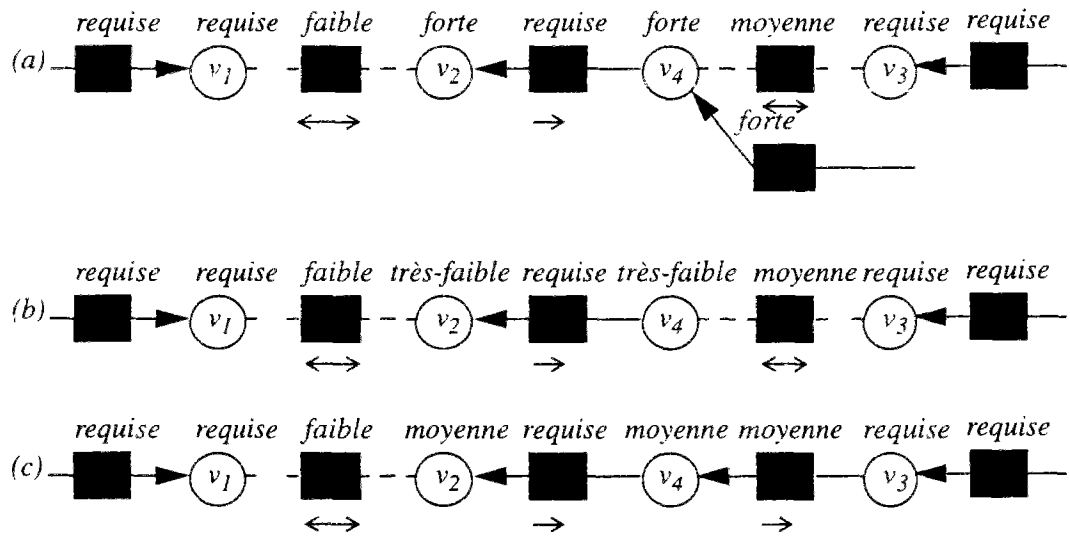


Enlèvement d'une contrainte par *DeltaBlue*

Si la contrainte à enlever est non satisfaite par la solution courante P alors le fait de l'enlever ne modifie en rien la solution courante P. Dans le cas où cette contrainte est satisfaite, son enlèvement du système peut changer le contenu de l'étiquette-voyageuse des variables en aval. Ceci permettra peut être à une ou plusieurs contraintes précédentes non satisfaites de devenir satisfaites. Considérant l'exemple de la figure 9. La figure 9.a montre la situation initiale, le programme client veut retirer la contrainte du côté droit de la variable v_4 étiquetée par *forte*. Il est à noter que la contrainte étiquetée par *faible* et portant sur les variables v_1 et v_2 ainsi que la contrainte étiquetée par *moyenne* portant sur les variables v_3 et v_4 sont toutes les deux non satisfaites puisque leurs étiquettes sont plus faibles que les étiquettes: *étiquette-voyageuse* de v_1, de v_2, de v_3 et celle de v_4. Dans la figure 9.b, la contrainte vient d'être enlevée et l'étiquette *étiquette-voyageuse* en aval vient d'être recalculée puisqu'il n'y a pas de contrainte déterminant la valeur de la variable v_4. L'étiquette *étiquette-voyageuse* de v_4 est maintenant identique à celle de la contrainte invisible introduite avec chaque variable (c.a.d. *très-faible*). Cette étiquette *très-faible* est propagée à travers la contrainte portant sur v_2 et v_3 et affecte l'*étiquette-voyageuse* de la variable v_2.

Maintenant, les deux étiquettes des contraintes non satisfaites sont plus fortes que celles des variables sur lesquelles elles portent et donc ces deux contraintes sont éligibles pour être satisfaites. *DeltaBlue* considère toujours en priorité la contrainte possédant l'étiquette la plus forte. Dans ce cas, la contrainte étiquetée par moyenne est ajoutée et les *étiquettes-voyageuses* de la variable v_2 et celle de la variable v_4 sont recalculées, ceci est illustré dans la figure 9.c. Finalement, la contrainte étiquetée par *faible* portant sur les variables v_1 et v_2 est considérée pour une éventuelle satisfaction. Ceci n'est pas accompli puisque l'étiquette de cette contrainte n'est pas plus forte que les *étiquettes-voyageuses* de v_1 et de v_2. L'algorithme se termine puisqu'il n 'y a plus de contraintes à considérer.

FIGURE 9 : Enlèvement d'une contrainte



Les exemples des figures précédentes contiennent uniquement des contraintes à une seule variable en sortie (c.à.d. que leurs méthodes correspondantes contiennent une seule variable de sortie). Cependant, *DeltaBlue* est capable de manipuler des contraintes ayant plusieurs variables de sortie dans la limite où ces contraintes ne possèdent chacune qu'une seule méthode (on parle de contrainte uni-directionnelle). Il est bien évident que si les contraintes à une seule variable de sortie ne sont pas multi-directionnelles ou si les contraintes à plusieurs variables de sortie ne sont pas uni-directionnelle l'utilisation de l'*étiquette-voyageuse* ne serait pas d'un grand intérêt. Un exemple de contrainte uni-directionnelle, est celui de la contrainte qui capte la position de la souris et détermine la position du curseur sur l'écran. La souris peut altérer la position du curseur tandis que le curseur ne peut pas effectuer un déplacement physique de la souris.

Le principe de *DeltaBlue* est de résoudre seulement les contraintes qui déterminent d'une façon unique les valeurs de leurs variables. Un exemple de contrainte ne déterminant pas d'une façon unique la valeur de sa variable est la contrainte $x > 5$. Il s'agit d'une contrainte de restriction qui ne fixe pas la valeur de sa variable.

Chaque solution générée par *DeltaBlue* est une solution *Localement-Prédicat-Meilleur* à la hiérarchie de contrainte courante. S'il existe un cycle ou un conflit entre des contraintes étiquetées par l'étiquette *requise* (c.à.d. deux méthodes sélectionnées des deux contraintes forment un cycle ou ayant toutes les deux une même variable en sortie) alors l'algorithme s'arrête en générant un message d'erreur.

Pour prouver que les solutions de *DeltaBlue* à des hiérarchies ayant des contraintes qui ne forment pas de cycles sont *Localement-Prédicat-Meilleur*, la notion de contrainte bloquée est définie. Cette notion consiste à dire qu'une contrainte est bloquée si elle est étiquetée par une étiquette considérée supérieure à une des étiquettes de ses variables potentielles de sortie. Ensuite, un lemme est calqué sur cette notion exprimant que s'il n'y a pas de contrainte bloquée dans une solution alors cette solution est *Localement-Prédicat-Meilleur*. La complétude est montrée par récurrence sur la solution courante. Si la solution courante ne possède pas de contrainte bloquée alors l'exécution d'un des quatre points d'entrée du programme ne produit pas de contrainte bloquée. Par exemple, soit une solution P_1 produite par *DeltaBlue*. Supposons qu'il existe une solution P_2 meilleure que P_1 et non produite par *DeltaBlue*. Alors par la définition du comparateur *Localement-Prédicat-Meilleur*, il existe un niveau k de la hiérarchie telle que P_2 satisfait toutes les contraintes que P_1 satisfait et au moins une contrainte c de plus. Soit v_c la variable de c possédant la plus faible *étiquette-voyageuse*, et soit $W_{P1}(v_c)$ l'*étiquette-voyageuse* de la variable v_c dans P_1. Maintenant, puisque c est non satisfaite dans P_1 et P_1 n'a pas de contrainte bloquée, c n'est pas étiquetée par une étiquette plus forte que $W_{P1}(v_c)$. Cependant, puisque c est satisfaite dans P_2, l'étiquette de c doit être plus forte que $W_{P1}(v_c)$. Ce qui constitue une contradiction et donc il ne doit pas y avoir de solution meilleure que P_1.

Contrairement à l'algorithme *Blue* qui réexamine chaque contrainte à la suite d'un ajout ou d'un enlèvement de contrainte au système, *DeltaBlue* utilise la solution courante comme un guide pour trouver la prochaine solution en examinant seulement la contrainte affectée par un changement récent. Dans le cas où le nombre de contraintes affectées par un changement est petit par rapport au nombre de contraintes dans la hiérarchie alors *DeltaBlue* est beaucoup plus rapide que *Blue*. Dans le cas où le nombre de contraintes affectées par ce changement est proche de celui de la hiérarchie alors *DeltaBlue* est moins rapide que *Blue* puisque *DeltaBlue* doit exécuter plus d'opérations pour maintenir sa structure de données.

DeltaBlue est étendu pour obtenir l'algorithme *UltraViolet*. Cette extension consiste à appeler un résolveur de cycle lors de la détection d'un cycle afin de déterminer les valeurs des variables des contraintes ayant les méthodes qui forment ce cycle et ensuite les propager en aval dans le graphe.

4.2.2 L'algorithme *SkyBlue*

Comme nous l'avons vu dans la section précédente décrivant l'algorithme *DeltaBlue*, cet algorithme possède deux restrictions majeures. La première est qu'il est incapable de gérer les contraintes cycliques (si toutefois une hiérarchie contient des contraintes cycliques, *DeltaBlue* s'arrête en générant un message d'erreur). La deuxième restriction est qu'il ne peut pas gérer des contraintes multi-directionnelles ayant plusieurs variables en sortie (c.à.d chaque contrainte peut avoir plusieurs méthodes et chaque méthode peut calculer plusieurs variables).

SkyBlue a été conçu pour remédier à ces deux restrictions. C'est ainsi qu'il permet la construction de cycles et traite les contraintes possédant des méthodes à multi-variables de sortie. *SkyBlue* ne satisfait pas les contraintes qui forment un cycle. Il fait appel à un résolveur plus puissant pour résoudre cette tâche. Cependant, il maintient correctement les contraintes non cycliques du graphe. *SkyBlue* traite les contraintes contenant des méthodes à multi-variables de sortie. Ce genre de contraintes est utilisé dans plusieurs situations. Par exemple, supposons que les variables x et y représentent les coordonnées cartésiennes d'un point et φ et θ représentent les coordonnées polaires du même point. Pour garder ces deux représentations consistantes en parallèle, on aimerait définir une contrainte avec la méthode à 2-variables de sortie x et y : $(x,y) \leftarrow (\varphi \cos \theta, \varphi \sin \theta)$ et une autre méthode à 2-variables de sortie φ et θ : $(\varphi,\theta) \leftarrow (\sqrt{(x^2+y^2)}, Arctang(x,y))$. Les méthodes à multi-variables de sortie sont aussi utilisées pour accéder aux éléments d'une structure de donnée composée. Par exemple, lorsqu'on veut décomposer un objet composé d'un point cartésien à deux variables utilisant une contrainte avec les méthodes : $(x,y) \leftarrow (Point.x, Point.y)$ et $Point \leftarrow Crée\text{-}Point(x,y)$.

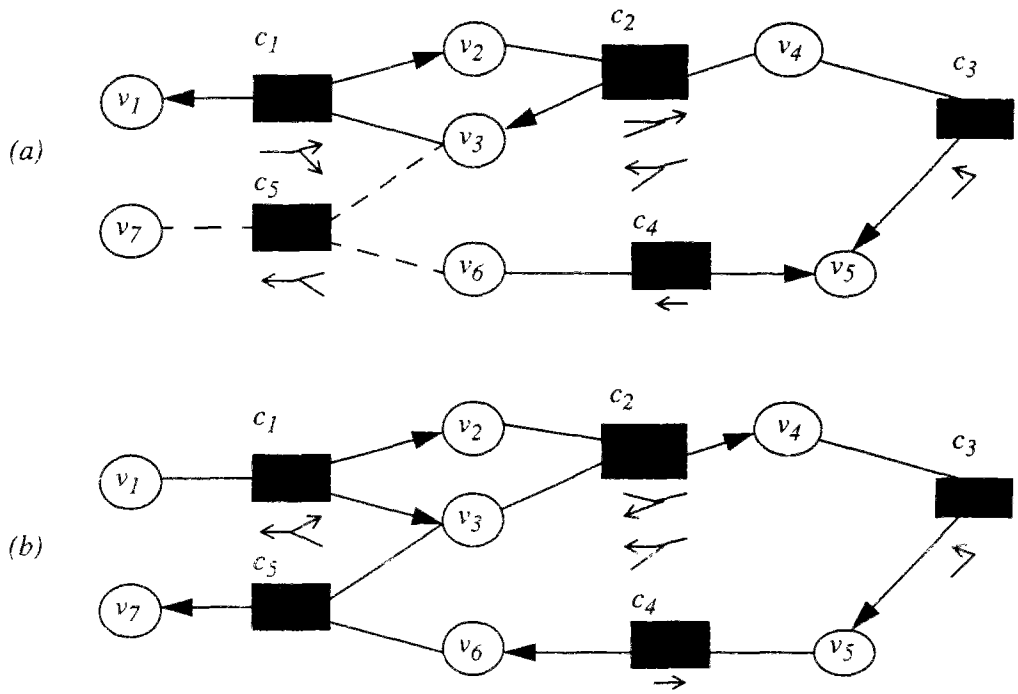
Maloney dans [Mal91] a établi que générer un graphe solution à partir d'un problème de contraintes à sorties multiples était NP-complet. Les auteurs de la ligne *Blue* affirment pourtant que dans la plupart des cas, le complexité en temps n'est pas un problème puisque ces algorithmes changent un sous-graphe de petite taille lors de l'ajout ou du retrait d'une contrainte et donc la complexité en temps est souvent linéaire en nombre de contraintes. Si l'on considère des contraintes uni-variable de sortie non cycliques, *DeltaBlue* est 2 fois plus rapide que *SkyBlue*. Des travaux futurs seront effectués sur *SkyBlue* pour qu'il puisse prédire si la hiérarchie ne contient pas de contraintes cycliques et si elles sont uni-variable de sortie. Dans ce cas, il pourra atteindre les mêmes performances que *DeltaBlue*.

Pour satisfaire un ensemble de contraintes, *SkyBlue* choisit une méthode à exécuter pour chaque contrainte dans cet ensemble. Chaque méthode sélectionnée est considérée comme méthode active dans le graphe. Si un graphe de méthodes contient deux ou plusieurs méthodes actives possédant une même variable de sortie alors il s'agit d'un conflit de méthodes. Dans la figure 10.a, les deux méthodes actives des deux contraintes c_3 et c_4 sont en conflit. *SkyBlue* prohibe le conflit de méthodes en affectant la $n^{ième}$ variable puisqu'il empêche la satisfaction de leurs contraintes correspondantes simultanément. Dans la figure 10.a, si la contrainte c_3 est satisfaite en exécutant sa méthode active (la valeur de la variable v_5 sera déterminée) ensuite la contrainte c_4 est satisfaite en exécutant sa méthode active (la valeur de la variable v_5 sera redéterminée de nouveau). Ceci implique que la contrainte c_3 devienne certainement non satisfaite puisque la valeur de v_5 vient d'être modifiée.

Un graphe de méthodes ne possédant pas de conflit de méthodes ni de cycle peut être utilisé pour la satisfaction des contraintes actives[1] en exécutant ces méthodes dans l'ordre topologique[2]. Par exemple la figure 10.b montre un graphe de méthodes des mêmes contraintes que celles de la figure (a) où toutes les contraintes peuvent être satisfaites en exécutant dans l'ordre les méthodes correspondantes au contraintes c_1, c_2, c_3, c_4 et c_5.

1 On désigne par contrainte active une contrainte possédant une de ces méthodes active dans le graphe.
2 Dans les figures, chaque variable désignée par une flèche est déterminée avant d'être lue.

FIGURE 10 : (a) graphe de méthodes contenant la contrainte inactive c_5, un conflit de méthodes en v_5 et un cycle direct (entre c_1 et c_2). (b) graphe où chaque contrainte est satisfaite.



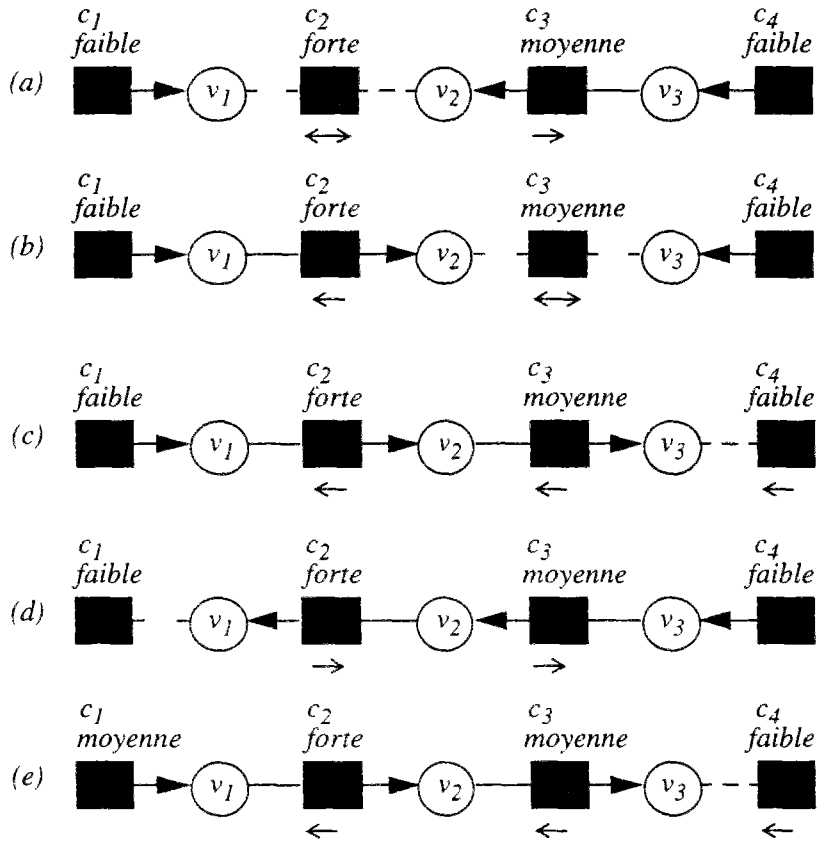
Si un graphe de méthodes contient un cycle direct (comme par exemple entre les deux méthodes des deux contraintes c_1 et c_2 de la figure 10.a) alors il devient impossible de trouver un ordre de tri topologique pour ses méthodes actives. Dans ce cas, *SkyBlue* trie et exécute seulement les méthodes actives en amont du cycle. Toutes les méthodes qui forment un cycle (ou l'aval du cycle) ne sont pas exécutées et leurs variables sont marquées pour spécifier que leurs valeurs ne satisfont pas nécessairement les contraintes actives. Si le cycle est cassé après, les méthodes de ce cycle sont triées et celles en aval sont exécutées correctement.

Traitement d'une hiérarchie de contraintes par *SkyBlue*

SkyBlue utilise les étiquettes des contraintes pour construire un *Graphe-Localement-Meilleur* (*GLM*) de méthodes. Un graphe de méthodes est un *GLM* s'il ne contient pas de conflit entres ses méthodes et s'il ne contient pas de méthode inactive qui puisse être activée en désactivant une ou plusieurs méthodes d'importances inférieures (étiquetées par des étiquettes moins fortes).

Par exemple, considérons le graphe de méthodes de la figure 11.a. Ce graphe n'est pas un *GLM* puisque la contrainte c_2 (étiquetée par *forte*) peut être activée en activant sa méthode qui a pour variable de sortie v_2 et en désactivant la contrainte c_3 (étiquetée par *moyenne*) ce qui produit la figure 11.b. Ce graphe de méthode n'est toujours pas un *GLM* puisque c_3 peut être activée en désactivant la contrainte c_4 en produisant la figure 11.c. Ce dernier est un *GLM* puisque la seule contrainte désactive (c_4) ne peut pas être active en désactivant une contrainte d'importance inférieure (étiquetée par une étiquette inférieure).

FIGURE 11 : Construction de GLM de méthodes



Il peut exister plusieurs *GLM* de méthodes pour un graphe donné de contraintes. Ceci est compatible avec la définition théorique attribuée à l'ensemble S (qui est l'ensemble des valuations qui sont solutions à la hiérarchie). S peut contenir plusieurs valuations puisque les comparateurs utilisés n'établissent pas un ordre total sur les valuations dans l'ensemble S_0.

La figure 11.d montre un autre *GLM* de méthodes qui n'est ni meilleur ni pire que celui de la figure 11.c. *SkyBlue* se contente de construire un des *GLM* possibles d'une façon arbitraire. Si l'on change les étiquettes des contraintes, ceci imposera le choix d'une alternative par rapport à une autre. Par exemple si l'étiquette de la contrainte c_1 est remplacée par *moyenne* alors le seul *GLM* de méthodes doit être celui de la Figure 11.e.

Comme dans *DeltaBlue*, pour contrôler la construction des graphes de méthodes, l'utilisateur ajoute des contraintes invisibles (aussi appelé contraintes de maintien sur les variables). Chacune de ces contraintes possède une seule méthode à une variable de sortie et aucune variable d'entrée. Ces contraintes invisibles (de maintien) expriment la non modification des valeurs de leurs variables. Un type de contraintes similaire est celui des contraintes d'initialisation. Ces contraintes initialisent leurs variables de sortie à des valeurs constantes. Les contraintes d'initialisation peuvent être utilisées pour injecter des nouvelles valeurs à des variables dans un graphe de contraintes. Dans la figure 11, les contraintes c_1 et c_4 sont des contraintes de maintien ou d'initialisation.

Le concept de *lecture-seulement* défini dans le chapitre 3 étend la théorie décrite au chapitre 2 aux contraintes qui ne possèdent pas de méthodes pour déterminer certaines de leurs variables. Comme dans *Blue*, *DeltaBlue* ou *SkyBlue*, les contraintes peuvent ne pas avoir toutes les méthodes de toutes les directions possibles.

Pour certains graphes de contraintes, les *GLM* de méthodes de ce graphe déterminent les solutions du comparateur *Localement-Prédicat-Meilleur*.

SkyBlue maintient les contraintes en construisant un *GLM* de méthodes et en exécutant ces méthodes pour satisfaire les contraintes actives. Initialement, le graphe de contraintes et son *GLM* de méthodes sont vides. *SkyBlue* est invoqué par l'appel d'une des deux procédures d'ajout et de retrait d'une contrainte. Puisque les contraintes sont ajoutées et retranchées au système, *SkyBlue* met a jour incrémentalement le *GLM* de méthodes et exécute ces méthodes pour resatisfaire les contraintes actives.

L'ajout d'une contrainte par *SkyBlue*

Lorsqu'une nouvelle contrainte est ajoutée au graphe de contraintes, il est possible de modifier le graphe de méthodes en activant cette nouvelle contrainte. L'activation de cette nouvelle contrainte se fait par la sélection d'une de ses méthodes. Cette sélection peut créer un changement de méthodes actives des contraintes libellées par des étiquettes plus fortes ou aussi fortes que celle de la nouvelle contrainte. Cette sélection peut aussi désactiver une ou plusieurs contraintes étiquetées par des étiquettes moins fortes que celle de la nouvelle contrainte. Ce processus est connu comme la construction d'une vigne de méthodes. L'ajout d'une contrainte c par *SKyBlue* est effectué par l'exécution des pas suivants :

1. Ajouter la contrainte c inactive au graphe de contraintes et essayer de l'activer en construisant une vigne. S'il n'est pas possible de construire une telle vigne alors laisser la contrainte c inactive (Dans ce cas le graphe est inchangé et donc il est toujours un *GLM*).

2. D'une manière répétitive, essayer d'activer toutes les contraintes non actives dans le graphe en construisant des vignes de méthodes jusqu'à ce qu'aucune des contraintes restantes ne puisse être activée. Il est à signaler que chaque fois qu'une contrainte inactive devient active, une ou plusieurs contraintes faiblement étiquetées sont enlevées. Ces contraintes enlevées sont remises dans l'ensemble des contraintes inactives pour être réessayées après (lors de l'ajout d'une autre contrainte).

3. Exécuter les méthodes sélectionnées (actives) dans le graphe de méthodes pour satisfaire les contraintes actives comme il est décrit précédemment.

Le deuxième pas doit terminer puisqu'il y a un nombre fini de contraintes. Chaque fois qu'une contrainte est enlevée, elle est ajoutée à l'ensemble des contraintes inactives. Ces contraintes ajoutées peuvent être activables après, lors de l'ajout d'autres contraintes étiquetées plus faiblement que ces dernières. Ce processus n'est pas infini et peut s'arrêter éventuellement avec un ensemble de contraintes non activables.

Lorsque le pas 2 se termine, le graphe de méthodes actives résultant est un *GLM* puisqu'aucune vigne supplémentaire ne peut être construite. Par exemple, supposons que l'on vient d'ajouter la contrainte c_2 au graphe de contrainte de la figure 11.a. Une des voies possibles de construction de la vigne est d'enlever la contrainte c_3 et d'activer la contrainte c_2 en choisissant sa méthode qui a pour variable de sortie v_2 (figure 11.b).

Etant donné ce graphe de méthodes, le deuxième pas va essayer de construire une vigne pour activer la contrainte c_3. Ceci est possible en enlevant la contrainte c_4 (figure 11.c). Arrivé à ce point il n'est plus possible de construire une vigne pour activer la contrainte c_4 et par conséquent le pas 2 se termine. Le graphe de méthodes résultant est un *GLM*. Alternativement, si la première vigne avait été construite en enlevant la contrainte c_1 alors le *GLM* de méthodes de la figure 11.d aurait été produit immédiatement et le deuxième pas n'aurait pas été capable d'activer la contrainte c_1.

Le retrait d'une contrainte par *SkyBlue*

La procédure de retrait d'une contrainte est similaire à celle d'ajout d'une contrainte au système. Lorsqu'une contrainte active est retirée du graphe, l'activation d'une ou plusieurs contraintes inactives devient possible. Cela aboutit au même processus de construction de vignes. Le retrait d'une contrainte c par *SKyBlue* est effectué par l'exécution des pas suivants :

1. Si c est inactive, alors son retrait ne permet en aucun cas l'activation d'autres contraintes inactives et donc le graphe de méthode reste un *GLM*.
2. D'une manière répétitive, essayer d'activer toutes les contraintes non actives dans le graphe en construisant des vignes de méthodes (ajouter l'ensemble des contraintes désactivées à l'ensemble des contraintes inactives) jusqu'à ce qu'aucune des contraintes restantes ne puisse être activée. Ce pas se termine en laissant un *GLM* de méthodes.
3. Exécuter les méthodes sélectionnées dans le graphe de méthodes pour satisfaire les contraintes actives.

La construction d'une vigne de méthodes par *SkyBlue*

SkyBlue tente d'activer les méthodes inactives en changeant les méthodes actives des contraintes dans le graphe. Seules les méthodes de contraintes étiquetées par des étiquettes de même préférences ou de préférences supérieures sont concernées par ce changement. Les méthodes des contraintes étiquetées par des étiquettes de préférences strictement inférieures peuvent être inactivées à la suite de cette activation. La technique utilisée par *SkyBlue* est la construction de *vigne* de méthodes (graphe de méthodes) utilisant la recherche en profondeur d'abord en opérant un retour en arrière.

Une vigne est construite en sélectionnant une des méthodes de la contrainte qu'on essaye d'activer (la contrainte racine). Si cette méthode est en conflit avec une méthode d'une autre contrainte active, alors on sélectionne une autre méthode de cette autre contrainte active. Cette dernière peut être aussi en conflit avec d'autres méthodes, etc. Ce processus continue dans le graphe de méthodes et résulte en une vigne de nouvelles méthodes débutant par une méthode de la contrainte racine. Ce processus de construction de vigne se manifeste dans l'un des sens suivants :

1. Si les variables de sortie de la nouvelle méthode sélectionnée dans la vigne ne sont pas déterminées par une autre méthode d'une autre contrainte, alors la branche courante de la vigne s'arrête sur ce point.
2. Si la nouvelle méthode sélectionnée dans la vigne est en conflit avec une méthode d'une contrainte étiquetée faiblement par rapport à la contrainte racine alors cette contrainte est désactivée. Par conséquent, toutes les méthodes d'une vigne appartiennent à des contraintes étiquetées par des étiquettes supérieures ou égales à celle de l'étiquette de la contrainte racine.

3. Si une méthode alternative choisie est en conflit avec une méthode de la vigne alors cette méthode n'est pas ajoutée à la vigne et l'essai d'autres alternatives est effectué. Si toutes les alternatives sont en conflit avec les méthodes de la vigne alors le processus de construction de cette vigne opère un retour arrière . La méthode sélectionnée précédemment est enlevée de la vigne. Cette vigne est ensuite étendue en choisissant d'autres méthodes de ces contraintes. Si aucune méthode de la contrainte racine ne permet la construction d'une vigne sans conflit, alors cette contrainte n'est pas activée.

La figure 12 présente un exemple qui montre le processus de construction d'une vigne. Une vigne complète est un sous-graphe connecté. Une vigne n'est pas nécessairement un arbre puisque les branches séparées peuvent se rencontrer et former un cycle. Si toute méthode de la vigne possède une seule variable de sortie alors la vigne doit avoir la structure d'une tige commençant par la contrainte racine et finissant par des séries d'autres contraintes avec des changements de méthodes sélectionnées. Si des méthodes de la vigne possèdent plusieurs variables de sortie alors la vigne doit être divisée en plusieurs branches avec une branche pour chaque variable de sortie. Les différentes branches ne peuvent pas être étendues indépendamment puisque les méthodes dans les différentes branches ne peuvent pas déterminer les mêmes variables. La recherche par backtrack doit prendre cela en compte en essayant toute combinaison possible de méthodes sélectionnées pour les contraintes dans différentes branches.

Les heuristiques intégrées à SkyBlue

Les performances de *SkyBlue* chutent lorsque le graphe de contraintes devient très large puisque le graphe peut contenir un nombre très important de contraintes inactives que *SkyBlue* va tenter d'activer en essayant de construire les vignes correspondantes. Dans ces conditions, *SkyBlue* exhibe une complexité en temps de $O(M^N)$ avec N est le nombre de contraintes et M est le nombre maximal de méthodes par contrainte.

Dans cette section, on décrira des heuristiques utilisées par *SkyBlue*. Ces heuristiques améliorent les performances de *SykBlue* sur un graphe de contraintes très grand.

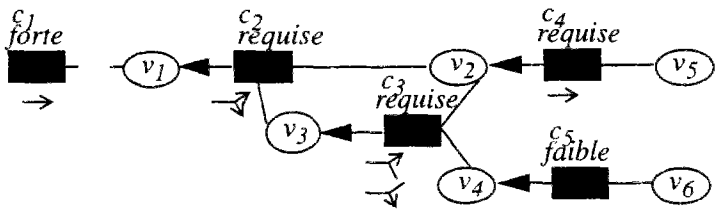
Technique d'assemblage d'étiquettes

Le résultat de chaque appel à l'une des procédures d'ajout ou de retrait de contrainte est un *GLM* de méthodes. Cependant, le graphe de méthode courant doit être un *GLM* si jamais l'une des deux procédures est appelée. Ce fait peut être utilisé pour éviter d'essayer l'activation de certaines contraintes inactives.

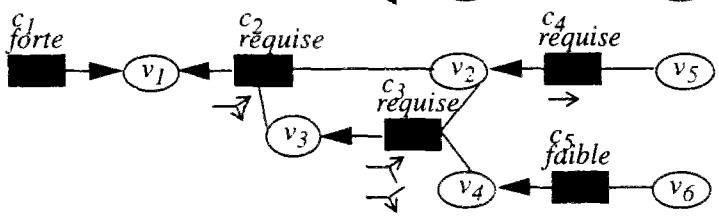
Si à l'issue de l'appel de la procédure d'ajout d'une contrainte c au graphe, cette contrainte est ajoutée effectivement alors il est impossible d'activer une autre contrainte inactive étiquetée par une étiquette de même importance ou d'importance supérieure à celle de c. Puisque, s'il était possible d'activer une telle contrainte après l'ajout de c, il aurait été possible aussi de l'activer avant l'ajout de c et le graphe de méthodes n'aurait pas été un *GLM*.

Après le retrait d'une contrainte c par la procédure de retrait, il est impossible d'activer une contrainte étiquetée par une étiquette plus forte que celle de c. Puisque s'il était possible d'activer une telle contrainte après ce retrait, il aurait été possible de l'activer avant ce retrait et le graphe de méthode n'aurait pas été un *GLM*. Il faut rappeler que la procédure peut permettre à des contraintes étiquetées par la même étiquette ou par des étiquettes moins fortes que celle de c d'être activées.

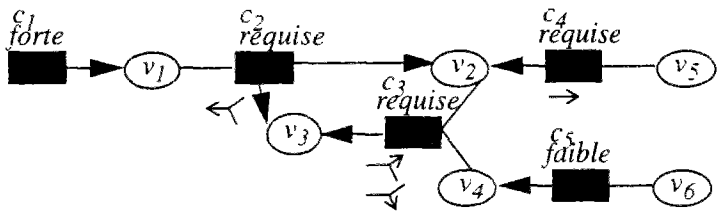
FIGURE 12 : Construction d'une vigne de méthodes



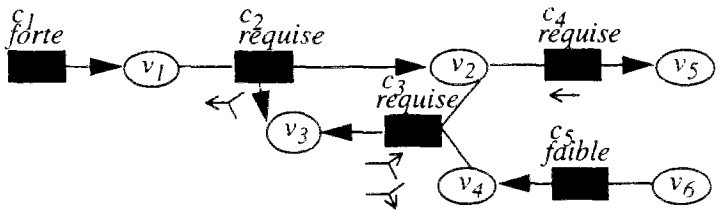
Supposons que l'on commence par ce graphe de méthodes et qu'on veuille activer la contrainte c_1 étiquetée par l'étiquette forte.



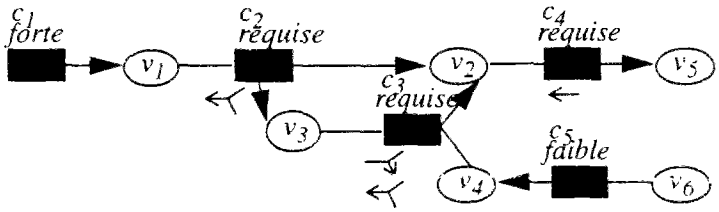
c_1 contient une seule méthode et donc cette méthode est sélectionnée pour être activée. Ceci crée un conflit de méthodes entre c_1 et c_2 et donc une autre méthode de c_2 doit être choisie.



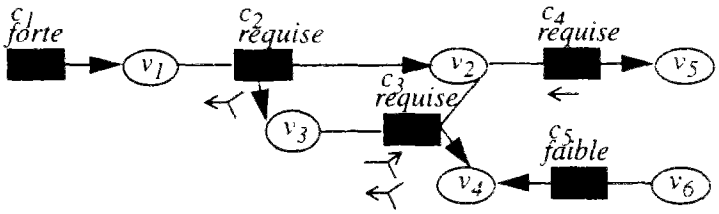
La méthode alternative dans c_2 crée un conflit avec c_3 et un conflit avec c_4.



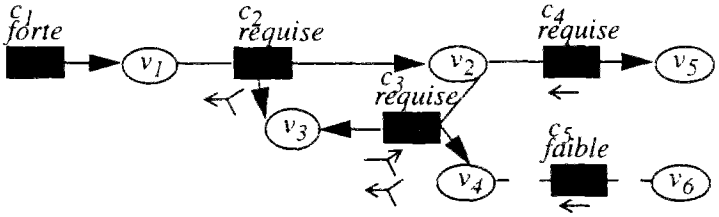
On suppose qu'on procède avec c_4 en premier : on intervertit seulement les méthodes de c_4 et donc la méthode déterminant v_5 est activée. v_5 n'est déterminée par aucune autre méthode et donc le développement de cette branche de la vigne s'arrête en ce point.



On doit considérer une des autres méthodes de c_3. Supposons qu'on essaye celle qui détermine v_2. Cette dernière n'est pas permise puisqu'elle crée un conflit avec c_2 qui se trouve déjà dans la vigne.



Cependant, on opère un retour arrière et on essayer une autre méthode de c_3 Supposons maintenant qu'on essaye la méthode qui détermine la variable v4 (ceci produit un conflit de méthode avec c_5)



Maintenant on considère la contrainte c_5. Puisque cette contraint est étiquetée plus faiblement que la contrainte c_1, on n'a pas à trouver une méthode alternative. On peut tout simplement la supprimer et arrêter la construction de la vigne en ce point.

Technique locale d'assemblage

Si le graphe de méthodes est un *GLM* à la suite d'un ajout ou d'un retrait d'une contrainte *c*, il est clair qu'aucune contrainte inactive non connectée directement à *c* ne sera activée. Il est possible d'être plus sélectif sur les contraintes à activer :

> Lors de l'appel de la procédure d'ajout d'une contrainte *c*, si cet appel termine par la construction d'une vigne qui active la contrainte *c*, il est suffisant de collecter les contraintes inactives qui contraignent les variables en aval dans le graphe. Ces variables sont extraites des variables redéterminées par d'autres contraintes différentes des contraintes initiales.
>
> Après le retrait d'une contrainte *c*, il est suffisant de collecter les contraintes inactives qui contraignent les variables en aval parmi celles déterminées précédemment par *c*.
>
> Si SkyBlue construit avec succès une vigne, les contraintes inactives résultant de cette construction sont ajoutées à l'ensemble des contraintes inactives en examinant les variables en aval redéterminées une nouvelle fois. Comme chacune de ces contraintes est traitée (elle est activée ou il est déterminé qu'elle ne sera pas activée) elle peut être enlevée de l'ensemble. Lorsque cet ensemble devient vide, il n'y a plus de contrainte inactive qui puisse être activée.

Une technique identique peut être utilisée pour réduire l'ensemble des méthodes à exécuter. Au lieu d'exécuter toutes les méthodes sélectionnées des contraintes actives dans le graphe des contraintes, il est seulement nécessaire de collecter et d'exécuter les méthodes sélectionnées des nouvelles contraintes activées et les méthodes en aval des variables redéterminées.

L'étiquette-voyageuse

La vigne est construite par le processus répétitif qui est la sélection d'une nouvelle méthode d'une contrainte et l'essai d'étendre la vigne en partant des variables de sortie de cette nouvelle méthode. Cette extension n'est réalisée que s'il n'y a pas de conflit entre les différentes branches et que la vigne rencontre des variables non déterminées par des méthodes de contraintes étiquetées par des étiquettes identiques ou supérieures à celle de la contrainte racine. Si *SkyBlue* peut prédire qu'une de ces conditions est fausse, alors la méthode sélectionnée doit être rejetée immédiatement sans essayer l'extension de la vigne.

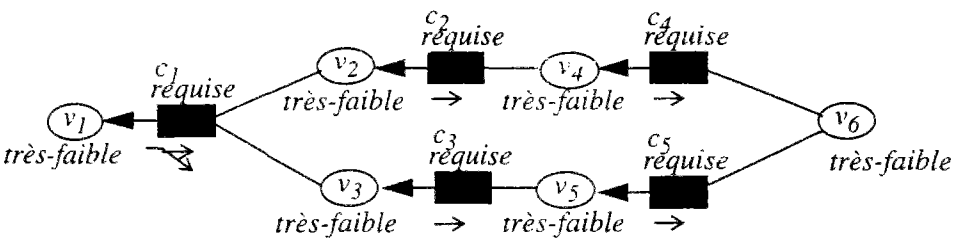
Nous avons vu que le résolveur *DeltaBlue* prédit l'activation d'une nouvelle contrainte introduite au système en utilisant le concept de l'*étiquette-voyageuse*. L'*étiquette-voyageuse* d'une variable indique l'étiquette de la contrainte qui doit être retirée (ne servira pas à déterminer cette variable) pour que cette variable puisse être déterminée par une autre contrainte. L'*étiquette-voyageuse* d'une variable peut être l'étiquette de la contrainte courante qui détermine cette variable, ou bien elle peut être la plus faible étiquette d'une contrainte dans la branche partant de cette variable. Cette contrainte peut être amenée à être retirée lors d'une resélection de méthodes. Si la variable n'est déterminée par aucune des contraintes du système, son *étiquette-voyageuse* est assignée à *très-faible*[1].

1 Cette étiquette est considérée comme la plus faible étiquette de toutes les étiquettes du système.

Une variable peut aussi porter cette étiquette si elle est laissée indéterminée lorsque le résolveur resélectionne les méthodes sans enlever aucune contrainte (une autre interprétation attribuée à l'étiquette *très-faible* est que chaque variable a un état implicite préféré par *très-faible* qui spécifie que la valeur de la variable peut être changée si une contrainte étiquetée par une étiquette plus forte la détermine).

Comme nous l'avons mentionné auparavant, une propriété importante de l'algorithme *DeltaBlue* concernant le calcul de l'*étiquette-voyageuse* est qu'elle peut être déterminée en utilisant les informations locales. L'*étiquette-voyageuse* d'une variable peut être calculée à partir de l'étiquette de la contrainte qui la détermine et des *étiquettes-voyageuses* des autres variables de cette contrainte. Si le graphe de méthodes ne possède pas de cycle[1], toutes les *étiquettes-voyageuses* peuvent être mises à jour en associant les *étiquettes-voyageuses* des variables non déterminées à *très-faible* et en traitant chaque contrainte active dans un ordre topologique (afin d'initialiser les *étiquettes-voyageuses* des variables déterminées).

FIGURE 13 : étiquettes-voyageuses en SkyBlue



Cette technique de calcul n'est plus valable dans *SkyBlue* puisqu'on considère des contraintes possédant des méthodes ayant plusieurs variables de sortie. Considérons le graphe de méthodes de la figure 13. *DeltaBlue* doit correctement calculer les *étiquettes-voyageuses* des variables v_2 ... v_6 (dans ce cas, elles sont égales à *très-faible*).

Mais qu'en est-il de l'*étiquette-voyageuse* de la variable v_1? L'*étiquette-voyageuse* de v_2 et celle de v_3 impliquent que celle de v_1 doit être à *très-faible* puisque la méthode alternative de la contrainte c_1 peut être choisie pour déterminer les variables v_2 et v_3. Cependant, il n'est pas possible pour cette méthode de déterminer ces deux variables simultanément sans révoquer une contrainte requise du graphe. Une simple resélection de méthode résultera en un conflit en v_6 entre la contrainte c_4 et la contrainte c_5. Cette information ne peut pas être connue sans explorer le graphe. Ceci conduit à perdre le gain apporté par cette technique d'*étiquette-voyageuse* (c.à.d. qu'on pouvait les calculer en utilisant uniquement les informations locales).

Dans *SkyBlue*, la définition de l'*étiquette-voyageuse* est modifiée. L'*étiquette-voyageuse* d'une variable est définie par la borne inférieure sur l'étiquette de la contrainte la plus faible qui devrait être enlevée afin de permettre à cette variable d'être déterminée par une nouvelle contrainte. *SkyBlue* utilise cette nouvelle définition pour rejeter des méthodes lors de la construction d'une vigne : si une des variables de sortie d'une méthode a son *étiquette-voyageuse* égale ou supérieure à l'étiquette de la contrainte racine. Dans ce cas, il n'est pas possible de compléter la vigne en utilisant cette méthode. L'utilisation de cette technique ne peut pas éliminer tous les retours-arrière durant la construction d'une vigne mais elle les réduit considérablement.

1. Cette condition est requise pour *DeltaBlue*.

Lors de la construction d'une vigne avec succès, le graphe de méthodes est donc modifié et les *étiquettes-voyageuses* doivent être mises à jour pour correspondre aux nouvelles méthodes dans le graphe. Ceci est réalisé en considérant toutes les méthodes actives dans le graphe dans un ordre topologique et en recalculant ces *étiquettes-voyageuses* des variables déterminées par ces méthodes. Il est possible d'appliquer *la technique locale d'assemblage* décrite auparavant à cette situation en traitant seulement les contraintes actives en aval des variables redéterminées.

SkyBlue utilise cette nouvelle définition de l'*étiquette-voyageuse* pour simplifier le traitement des cycles. Si le graphe de méthodes contient des cycles, il n'est donc pas possible de trouver un ordre de tri topologique pour les contraintes. Les *étiquettes-voyageuses* des variables dans le cycle doivent être calculées en examinant toutes les contraintes dans le cycle ce qui nécessite un calcul non local. Au lieu de procéder de cette façon, *SkyBlue* choisit une méthode dans le cycle et calcule les *étiquettes-voyageuses* de ces variables de sortie comme si toutes les *étiquettes-voyageuses* de ces variables d'entrées étaient à *très-faible*. Ceci garantit une borne inférieure et simplifie la mise à jour des *étiquettes-voyageuses* par rapport au coût croissant de la recherche pendant la construction d'une vigne puisque les *étiquettes-voyageuses* en aval dans le cycle peuvent être étiquetées plus faiblement que nécessaire.

4.2.3 L'algorithme *QuickPlan*

Comme nous l'avons vu, les sections précédentes ont décrit les algorithmes *DeltaBlue et SkyBlue*. Ces algorithmes possèdent deux défauts majeurs. Le premier est qu'ils ne garantissent pas une solution acyclique si elle existe. C'est-à-dire, supposons que la hiérarchie manipulée contient des solution cycliques et au moins une solution acyclique. *DeltaBlue* comme *SkyBlue* ne garantissent pas de la trouver. Il suffit que *DeltaBlue* (resp. *SkyBlue*) parte sur la construction d'une chaîne de méthodes formant un cycle et lors de la construction de ce cycle le message d'erreur "*Contraintes Formant un Cycle*" apparaît et il s'arrête (resp. appelle un sous-résolveur de cycle et il continue). *DeltaBlue* comme *SkyBlue* ne cherchent pas à déterminer s'il y a existence d'une autre solution acyclique. Le deuxième défaut est que dans le pire des cas ces algorithmes acquièrent une complexité en temps exponentielle pour trouver une solution à une hiérarchie de contraintes à plusieurs variables de sorties. L'algorithme *QuickPlan* a été conçu pour remédier à ces deux défauts. *QuickPlan* garantit de trouver une solution acyclique si elle existe en $O(N^2)$ (N est le nombre de contraintes).

Dans la suite de cette section, nous présentons des vues générales sur les algorithmes de *QuickPlan*. Seules les idées clés de ces algorithmes suivies d'exemples seront présentées. Les versions complètes de ces algorithmes ainsi qu'une version incrémentale sont dans [Van95]. On présentera en premier une vue sur l'algorithme manipulant des contraintes à une seule variable de sortie. Ensuite, on présentera l'extension de cet algorithme afin de manipuler des contraintes ayant plusieurs variables en sortie. Et enfin, on présentera la résolution d'une hiérarchie de contraintes par utilisation du comparateur *Localement-Prédicat-Meilleur*.

Propagation de degré de liberté et résolution d'un ensemble de contraintes à une seule variable de sortie.

QuickPlan est basé sur la propagation de degré de liberté [Sut63a, Bor81, Van88]. Cette technique manipule un ensemble de contraintes non satisfaites et consiste à :

1- chercher une variable dans le graphe telle que :

cette variable est une variable de sortie d'une méthode d'une contrainte et en plus,

cette variable doit être attachée à une seule contrainte.

2- pour cette variable,

sélectionner la méthode (pour satisfaire la contrainte) qui calcule cette variable,

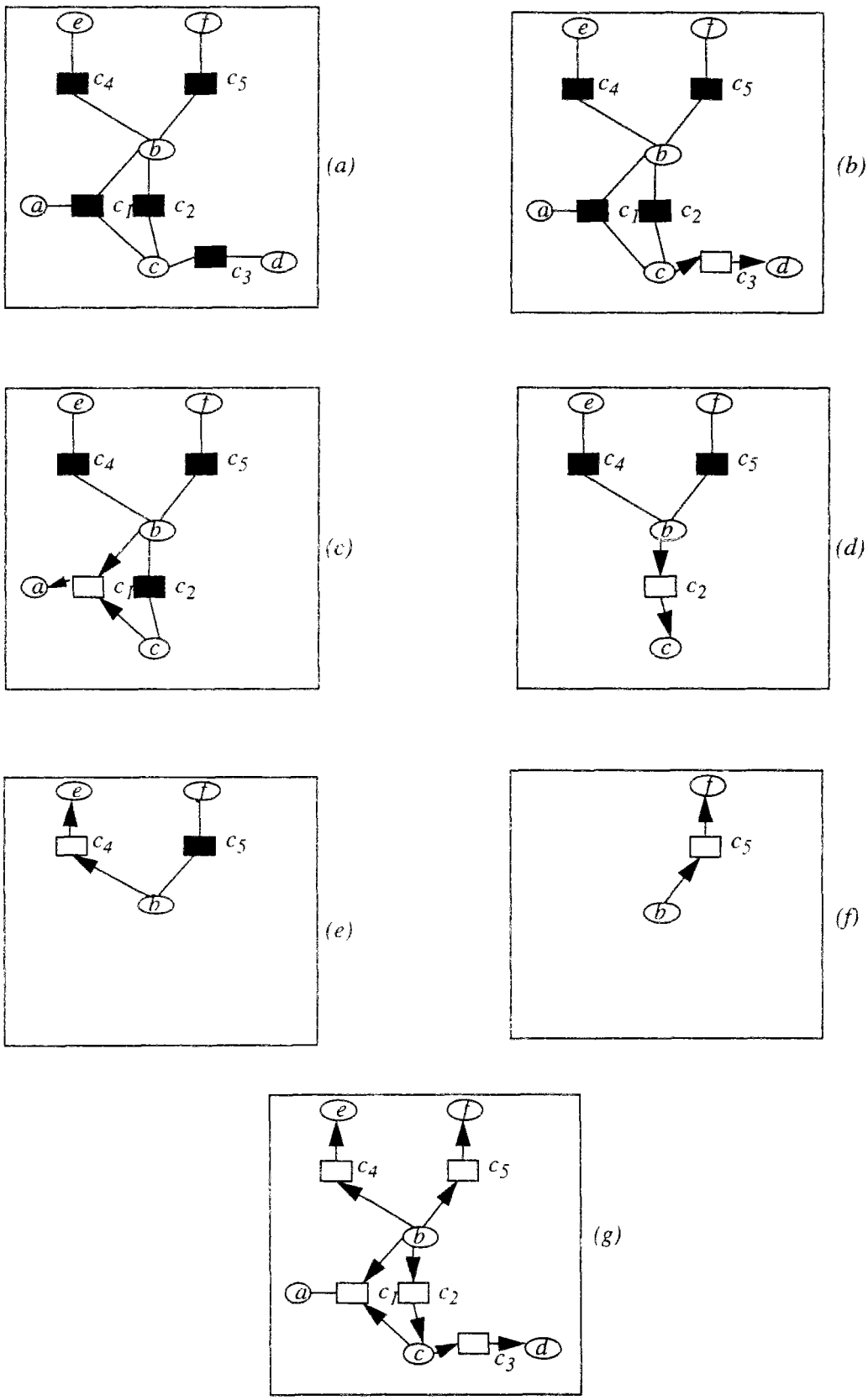
enlever la contrainte possédant cette méthode de l'ensemble des contraintes non satisfaites.

3- retourner au pas 1.

Cet algorithme se termine lorsqu'il ne reste plus de contraintes dans l'ensemble des contraintes non satisfaites ou lorsqu'une variable est attachée à plus d'une contrainte. Dans le cas favorable, le graphe de méthode constitue une solution acyclique. Les contraintes sont satisfaites en exécutant tout simplement cet ensemble de méthodes dans un ordre topologique. Dans le cas défavorable, le sous-graphe restant est considéré cyclique puisqu'il n'y a pas de possibilité de diriger le flot sans aboutir à un cycle entres les méthodes de ce sous-graphe.

Par exemple, considérons la figure 14 qui montre le fonctionnement de cet algorithme. Dans la figure 14.a, la variable *d* est attachée à une seule contrainte (c_3). Dans un premier temps, l'algorithme sélectionne la méthode qui calcule la variable *d* de cette contrainte (figure 14.b). Ensuite, il élimine la contrainte c_3 ainsi que ses arrêtes du graphe (figure 14.c). Ces deux pas sont répétés jusqu'à ce que toutes les contraintes soient éliminées du graphe (figure 14.c-14.f). Le résultat est le graphe acyclique de la figure 14.g.

FIGURE 14 : Résolution d'un graphe par propagation de degrés de liberté.



Résolution d'un ensemble de contraintes ayant plusieurs variables de sortie

L'extension de l'algorithme précédent pour la résolution des contraintes ayant plusieurs variables de sortie est très simple à réaliser. Cette extension est décrite par la procédure ci-dessous :

1- chercher un ensemble de variables dans le graphe tel que :

ce sous ensemble est attaché à une seule contrainte et,

ce sous ensemble constitue les variables de sortie d'une méthode de cette contrainte

2- identifier la méthode sélectionnée qui calcule cet ensemble de variables dans la contrainte,

enlever la contrainte possédant cette méthode de l'ensemble des contraintes non satisfaites.

3- retourner au pas 1.

Le pas 1 peut être réalisé en procédant comme suit :

1'- chercher une variable dans le graphe telle que :

cette variable est une variable de sortie d'une méthode[1] d'une contrainte et en plus,

cette variable doit être attachée à une seule contrainte.

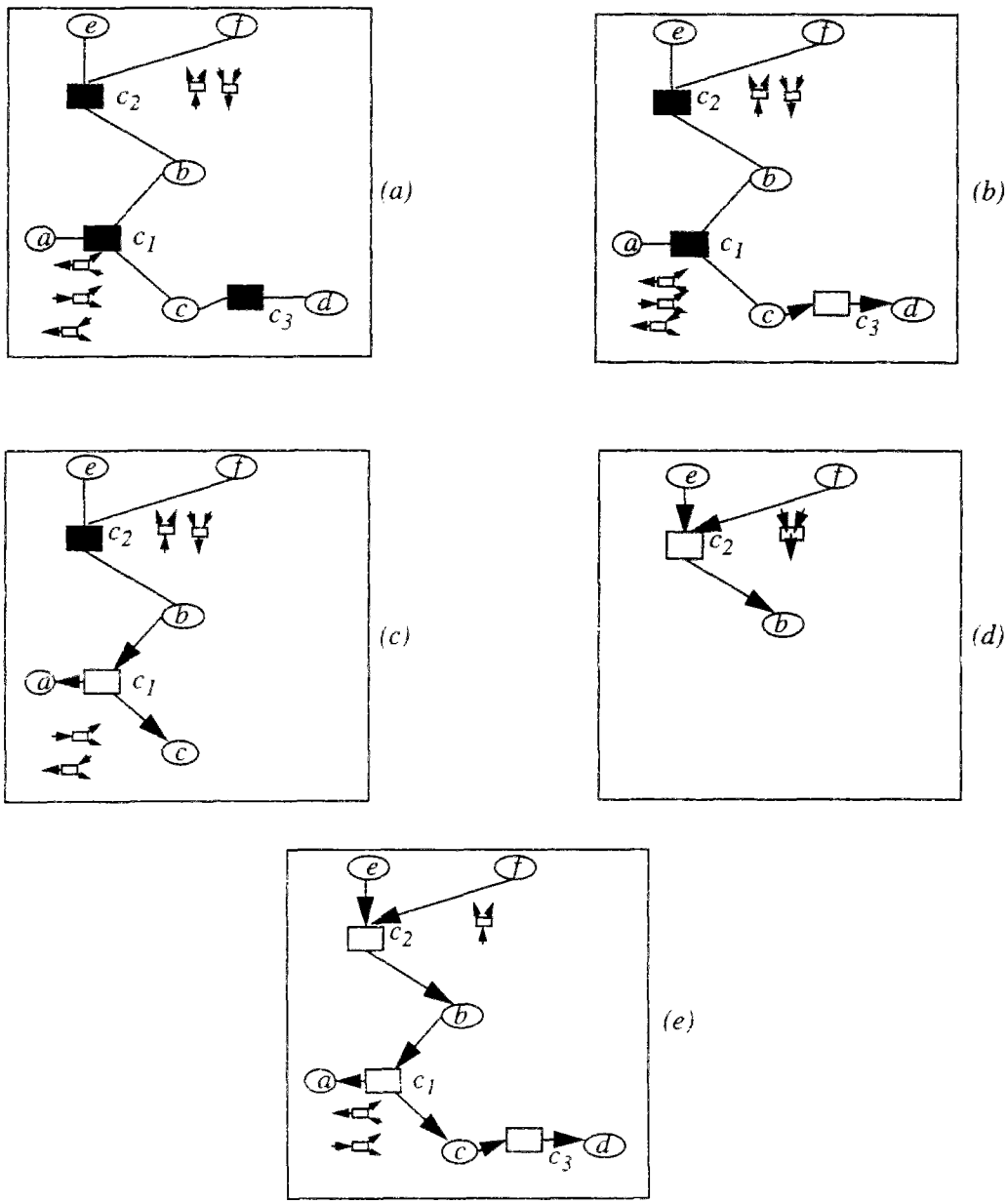
2'- s'il existe une variable de l'ensemble des autres variables de sortie de cette méthode qui n'est pas attachée à une seule contrainte alors retourner au pas 1'.

Ce nouvel algorithme obtenu se termine s'il ne reste aucune contrainte dans le graphe (dans ce cas la solution obtenue est une solution acyclique) ou s'il reste un ensemble de contraintes formant un cycle.

Par exemple, considérons la figure 15 qui montre le déroulement de ce nouvel algorithme obtenu à partir du premier. Dans la figure 15, les méthodes iconifiées représentent les méthodes non sélectionnées pour la satisfaction des contraintes. Dans la figure 15.a, la variable d est attachée à une seule contrainte (c_3). L'algorithme sélectionne la méthode qui calcule la variable d de cette contrainte (figure 15.b). Ensuite, il élimine la contrainte c_3 ainsi que ses arêtes du graphe (figure 15.c). Arrivé à ce stade, l'algorithme sélectionne la variable a. Il existe deux méthodes dans la contrainte c_1 pour lesquelles la variable a est une variable de sortie. La première méthode a pour variables de sortie les variables a et c et la deuxième méthode a pour variables de sortie les variables a et b. L'algorithme choisit la première puisque, pour la deuxième, la variable b est attachée à deux contraintes (condition du pas 2'). La première méthode est identifiée et la contrainte c_1 ainsi que ses arrêtes sont éliminées du graphe (figure 15.d). L'algorithme choisit ensuite la variable b et identifie la méthode qui détermine cette variable (figure 15.d). Après ce pas, il ne reste plus de contrainte dans le graphe et, par conséquent, l'algorithme se termine. Le graphe de méthodes dans la figure 15.e constitue une solution acyclique du graphe de contraintes initial.

1. S'il y a plusieurs choix de méthodes, l'algorithme sélectionne la méthode qui possède le plus petit nombre de variable en sortie. Ceci afin de maximuser le nombre de contraintes satisfaites.

FIGURE 15 : Résolution d'un graphe de contraintes ayant plusieurs variables en sortie



Résolution d'une hiérarchie de contraintes.

Pour la résolution d'une hiérarchie de contraintes où les contraintes sont étiquetées, *QuickPlan* modifie légèrement la technique décrite dans le paragraphe précédent. Si l'algorithme rencontre un sous-graphe dans lequel chaque variable est attachée à plusieurs contraintes, au lieu de terminer, il supprime la contrainte ayant l'étiquette la plus faible pourvu qu'elle ne soit pas requise. L'algorithme continue à satisfaire le nouveau sous-graphe obtenu. L'algorithme alterne les deux pas : suppression et résolution jusqu'à ce qu'il ne reste plus de contrainte dans le graphe ou jusqu'à ce que chaque variable du sous-graphe restant ne soit attachée qu'à des contraintes requises.

La preuve de la terminaison de l'algorithme est construite avec les même arguments que celles présentées auparavant. Si l'algorithme résout successivement les contraintes et en supprime quelques unes alors la solutions générée n'est pas forcément une solution meilleure au sens *Localement-Prédicat-Meilleur.*

Pour obtenir une solution qui est *Localement-Prédicat-Meilleur, QuickPlan* réessaie les contraintes éliminées par ordre décroissant sur leurs étiquettes. C'est ainsi qu'il peut satisfaire certaines de ces contraintes en exécutant l'algorithme de propagation de degré de liberté sur l'ensemble obtenu par l'union des contraintes satisfaites précédemment et la contrainte qu'il réessaie de satisfaire.

L'essai effectué par l'algorithme de propagation de degrés de liberté, qui consiste à satisfaire une contrainte éliminée, peut créer l'élimination d'une ou plusieurs contraintes ayant des étiquettes inférieures à celles de la contrainte essayée. Cependant, l'algorithme ne doit pas éliminer une contrainte étiquetée par une étiquette égale ou supérieure à celle de la contrainte essayée, puisque ceci conduirait à une solution moins bonne ou, au mieux, aussi bonne que la précédente. Par conséquent, lorsque l'algorithme atteint le point où il doit éliminer une contrainte étiquetée par une étiquette plus forte ou égale à celle de la contrainte réessayée pour avancer, il doit terminer et communiquer à *QuickPlan* que la contrainte qu'il a essayée ne peut pas être satisfaite. Tandis que si l'algorithme réussit à satisfaire la contrainte qu'il a essayée, alors toute contrainte éliminée à cause de cette satisfaction est remise dans la liste des contraintes qui doivent être réessayées.

Puisque *QuickPlan* tente de satisfaire les contraintes éliminées selon l'ordre décroissant de leurs étiquettes, alors chaque contrainte éliminée est essayée une seule fois. La hiérarchie contient un nombre fini de contraintes et par conséquent *QuickPlan* se termine. La solution générée est une solution *Localement-Prédicat-Meilleur.*

4.3 Algorithmes de résolution de contraintes d'égalité et d'inégalité

Un désavantage des algorithmes existants utilisant le principe de la propagation locale est l'incapacité de résoudre des contraintes cycliques. Dans certains cas, ces algorithmes peuvent trouver une solution acyclique si elle existe mais ceci n'est pas garanti. Ces algorithmes s'arrêtent souvent en générant le message d'erreur "*Graphe de contraintes cyclique*". De plus, si les contraintes sont simultanément vraies (c.à.d. deux contraintes sont en conflit sur une variable, et il existe une valeur pour cette variable qui satisfait ces deux contraintes), ces algorithmes ne peuvent pas les résoudre.

Un autre ensemble d'algorithmes existe. Ces algorithmes peuvent résoudre les hiérarchies contenant une collection arbitraire de contraintes linéaires d'égalité et d'inégalité. Ces algorithmes sont basés sur les comparateurs suivants : *Localement-Métrique-Meilleur* (τ_{LMM}), *Somme-Pondérée-Métrique*(τ_{SPM}), *Cas-Pire-Métrique*(τ_{CPM}), *Moindre-Carrés-Métrique*(τ_{MCM}). Ces comparateurs utilisent le domaine métrique pour compter les erreurs produites par les valuations sur les contraintes. Ces algorithmes sont des instances de l'algorithme *DeltaStar* [FW90, FWB92] et sont collectivement référencés comme les algorithmes *Orange.*

DeltaStar est un algorithme incrémental qui résout une hiérarchie de contraintes. Il est basé sur une alternative équivalente à la description de la théorie des hiérarchies de contraintes. La preuve de cette équivalence est décrite dans [Wil92, FW90, Fre91]. La principe de la théorie des hiérarchies de contraintes est décrit dans les chapitres précédents et attaché à la dichotomie entre les niveaux des contraintes dure et de préférence, alors que celui de la théorie alternative est attaché au raffinement hiérarchique de l'ensemble de solutions.

Orange est une série de trois algorithmes utilisant le principe de *DeltaStar* en transformant une hiérarchie de contraintes en une série de problèmes de programmation linéaire. L'ensemble de contraintes d'un niveau donné de la hiérarchie est transformé en un programme linéaire qui est résolu par l'algorithme du simplexe. Ces algorithmes sont spécialisés pour la recherche d'une ou plusieurs solutions d'une hiérarchie contenant des contraintes d'égalité ou d'inégalité. Aucun de ces trois algorithme ne traite les hiérarchies à ordre partiel sur les niveaux ni les annotations de *lecture-seulement* ou *écriture-seulement* sur les variables.

4.4 Autres algorithmes

Bien qu'elles ne soient pas destinées à résoudre des hiérarchies de contraintes, plusieurs autres techniques de résolution sont disponibles comportant la relaxation, la recherche de solution dans un domaine fini et la réécriture d'équation [KJ84, Bor81, Lel87]. Cette dernière est empruntée aux langages de programmation fonctionnelle avec ajout de support pour les contraintes multi-directionnelles et pour les objets.

Mackworth [Mac77], Van Hentenryck [Van89] et plusieurs autres auteurs décrivent des algorithmes efficaces pour la résolution d'un ensemble de contraintes où les variables ont un domaine de valeurs fini. Ces algorithmes considèrent la relaxation comme une technique numérique itérative dans laquelle chaque valeur de variable est ajustée pour minimiser l'erreur dans le processus de satisfaction des contraintes qui lui sont attachées.

Red est un algorithme utilisant un système de généralisation de réécriture de graphe basé sur *Bertrand* [Lel86]. Cet algorithme est capable de résoudre des hiérarchies de contraintes cycliques ainsi que des équations simultanées.

Yellow est un algorithme utilisant la relaxation classique qui produit une solution de moindre carré [Sut63b]. La complexité en temps de cet algorithme est supérieure à celle de *Orange*. Cependant, il peut traiter des hiérarchies contenant des équations non-linéaires.

Green est un algorithme de résolution de contraintes portant sur des variables ayant des domaines finis. Cet algorithme combine la propagation locale et la technique de génération et test de l'arbre de recherche.

Synthèse du chapitre

Ce chapitre présente une vue générale sur la plupart des résolveurs existants pour la résolution ou le maintien de la cohérence entre les contraintes d'une hiérarchie. Ces algorithmes permettent d'obtenir une ou plusieurs valuations de l'ensemble S des solutions tel qu'il a été construit d'une façon théorique dans le chapitre 2.

Nous avons essentiellement exposé dans ce chapitre plusieurs idées qui nous ont semblé intéressantes et qui sont utilisées dans les algorithmes (*DeltaBlue*, *SkyBlue* et *QuickPlan*) manipulant des contraintes fonctionnelles. Ces algorithmes sont basés sur la propagation locale et implémentent tous le comparateur *Localement-Prédicat-Meilleur*.

Le tableau de la figure 16 est une synthèse sur les fonctionnalités des quatre algorithmes étudiés de près dans ce chapitre.

FIGURE 16 : Comparaison entre résolveurs basés sur la propagation locale[1].

utilise / résolveurs	contraintes ayant : plusieurs variables en sortie	contraintes sont : multi-directionnelles	contraintes sont dans : une hiérarchie	compatible avec résolveur de cycle	garantit solution acyclique	complexité dans le pires de cas
Blue	–	+	–	–	–	O(N)
DeltaBlue	+ –	– +	+	–	–	O(N)
SkyBlue	+	+	+	+	–	$O(M^N)$
QuickPlan	+	+	+	+	+	$O(N^2)$

1. Concernant *DeltaBlue*, le carré en pointillé dans cette figure indique que si les contraintes possèdent plusieurs variables en sortie alors elles ne doivent pas être multi-directionnelles.

5 Un nouveau résolveur : Houria

Après avoir décrit dans le détail la théorie des hiérarchies de contraintes ainsi que certaines extensions qui lui sont apportées, nous avons examiné au chapitre 4 certaines fonctionnalités d'une série de résolveurs existants pour la résolution de ces hiérarchies de contraintes. Chacun de ces résolveurs est basé sur un critère de comparaison de valuations. La plupart de ces derniers sont des algorithmes de propagation locale manipulant des hiérarchies de contraintes fonctionnelles et utilisant des erreurs binaires et travaillant en deux passes. La première consiste à planifier (en tenant compte d'un critère de comparaison) un graphe orienté de méthodes sans conflits et souvent sans circuits. Et la deuxième passe consiste à exécuter les méthodes de ce graphe planifié. Ainsi une valuation qualifiée de solution respectant le critère utilisé est obtenue.

La plupart des algorithmes de propagation locale manipulant des hiérarchies de contraintes fonctionnelles utilisent un mode de comparaison local assez simple entre les différentes valuations : une valuation sera meilleure qu'une autre s'il existe un niveau pour lequel elle satisfait un sur-ensemble des contraintes de la seconde, et si pour tous les niveaux plus importants, elles satisfont toutes deux les mêmes ensembles de contraintes. D'après cet ordre local, deux valuations qui satisfont deux ensembles de contraintes non inclus l'un dans l'autre au sein d'un même niveau de la hiérarchie sont incomparables. Dans la plupart des cas, cette incapacité à comparer ces valuations est une situation indésirable puisque d'après la sémantique de la hiérarchie, les contraintes d'un niveau donné sont aussi importantes les unes que les autres et on souhaiterait privilégier la satisfaction d'un ensemble de contraintes dont le nombre est supérieur à celui d'un autre ensemble de contraintes du même niveau.

Dans ce chapitre, on reviendra en détail sur ces arguments qui justifient en partie notre démarche qui était de concevoir un nouveau résolveur de propagation locale basé sur un ordre plus fin que celui utilisé par les algorithmes de propagation locale existants. Ce mode de comparaison prend en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons.