Introduction

Ce travail s'inscrit dans le cadre de la résolution des problèmes combinatoires notamment les problèmes de satisfaction de contraintes. Il s'agit de développer un noyau sous forme de librairies capables de modéliser et de résoudre des problèmes combinatoire(*Open CSP*).

Sachant que différents outils existent déjà pour effectuer ce genre de tâches, l'ajout principal dans cette réalisation est la possibilité de tracer le processus de résolution afin de déterminer l'existence de contraintes inconsistantes et de tenter d'expliquer certaines situations d'échec lorsque l'algorithme ne réussi pas à fournir une solution.

Il est vrai que la majorité des *solvers* actuels ne donnent aucune ou peu d'informations lorsque le processus de résolution échoue. Connaître les causes d'échec peut être très utiles dans le cas de problèmes réels. Des besoins de débogages par exemple, peuvent tout à fait justifier la nécessité de connaître ce genre d'informations.

Le présent rapport est disposé comme suit :

Un premier chapitre pour présenter *PACTE NOVATION*, l'entreprise chez laquelle s'est effectué ce stage. Une présentation générale, des chiffres et des résultats propres à cette société y sont exposés.

Un deuxième chapitre dans lequel on introduira des notions générales de la programmation par contraintes. On présentera les *CSP* et on parlera de façon globale de l'état de l'art en ce qui concerne la résolution de tels problèmes.

Suivra ensuite, un troisième chapitre pour détailler les algorithmes de backtracking. On parlera surtout des algorithmes de *Back Jump* de *Gaschnig* et du *Conflict Directed Backjumping Learning*.

Le quatrième chapitre est réservé à la consistance d'arcs et à la propagation de contraintes. Un outil indispensable pour la résolution de CSP. On présentera en détail les techniques qui relèvent de ce domaine et que nous avons utilisé dans *Open CSP*.

L'implémentation de notre librairie est le sujet du cinquième chapitre. Le modèle objet sur lequel on s'est basé, le détails de la programmation ainsi que l'environnement globale de cette librairie y seront détaillés.

On présentera ensuite les résultats obtenu et on finira par une conclusion.

Un annexe est prévu à la fin pour servir de manuel d'utilisation de la librairie *Open CSP*.

Chapitre1 :

Présentation de Pacte Novation

I.1 Introduction

Pacte Novation est une société d'ingénierie logicielle pluridisciplinaire, intervenant dans des secteurs d'activités très variés comme le transport, la banque-finance, les télécommunications, l'énergie, l'industrie et le tertiaire.

Son savoir-faire, reconnu auprès de très grands comptes, permet d'intervenir sur des applications à haute valeur ajoutée. Prés de 40% de son activité est réalisé au forfait, ce qui lui a permis de fortement capitaliser autour de 3 axes :

Technologies orientés objets et distribuées :

- Conception et réalisation d'applications à base de technologies orientées objets
- Conseil et pratique des méthodes orientées objets
- Intégration d'applications distribuées dans les environnements Client / Serveur

Interfaces Homme Machine avec une spécialisation en ergonomie du logiciel :

- Prise en compte du facteur humain lors de l'informatisation de processus
- Analyse de la tâche et spécifications des Interfaces Homme/Machine
- Evaluation et recommandation ergonomique sur des IHM existantes
- Réalisation d'IHM graphiques et d'outils utilisant le Langage Naturel

Aide à la décision :

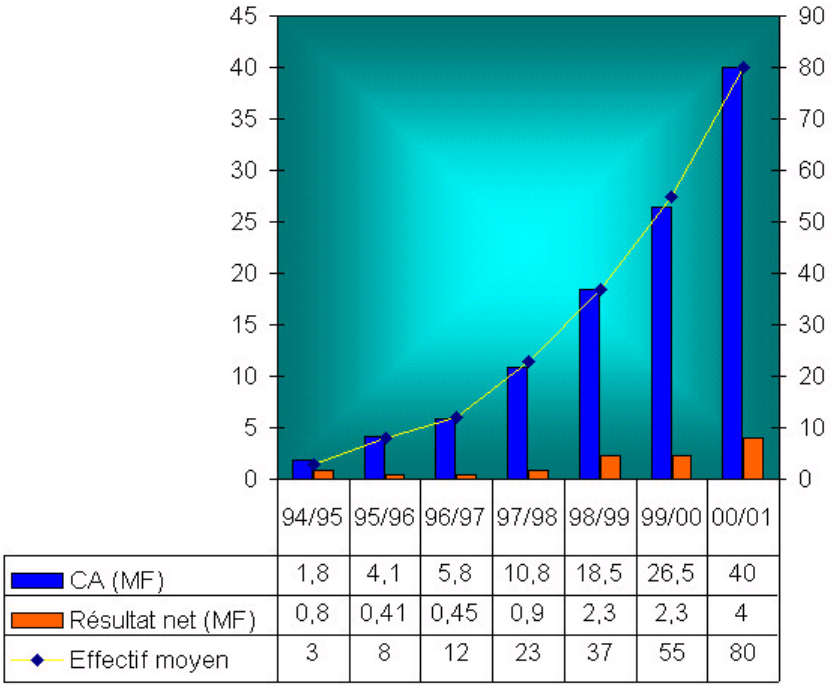
Résolution de problèmes intégrant des raisonnements d'experts, une forte complexité ou bien une grande combinatoire

- *Systèmes à Base de Connaissances* : spécification, conception et développement de systèmes à base de connaissances, modélisation et capitalisation de savoir-faire
- *Systèmes à Base de Contraintes* : allocation de ressources, planification et ordonnancement, optimisation de processus, …

I.2 Profil de l'entreprise

Pacte Novation, créé le 1er avril 1994, est une société anonyme au capital de 512 000 €. Deux associés sont à la tête de Pacte Novation : Christian TORA, Président Directeur Général, ingénieur de formation et Bruno GAUDINAT, également ingénieur. Les actionnaires principaux sont les cadres et dirigeants de l'entreprise.

A ce jour 83 ingénieurs travaillent dans la société et la *figure-1* montre la croissance de Pacte Novation depuis sa création.



	94/95	95/96	96/97	97/98	98/99	99/00	00/01
CA (MF)	1,8	4,1	5,8	10,8	18,5	26,5	40
Résultat net (MF)	0,8	0,41	0,45	0,9	2,3	2,3	4
Effectif moyen	3	8	12	23	37	55	80

Figure 1- courbe de croissance de Pacte Novation

I-3 Ingénierie chez Pacte Novation

I-3.1 Compétences

Pacte Novation est une société à forte compétence et de grande expérience dans tous les domaines avancés de l'aide à la décision. Au dessus des technologies objet, elle a su réunir et mettre en valeur une équipe soudée, complémentaire, curieuse des métiers de ses clients ainsi que des techniques informatiques à forte valeur ajoutée. Ces compétences s'expriment et trouvent leur place à toutes les étapes d'un projet.

Dans les rangs de Pacte Novation se trouvent

- des cogniticiens et ergonomes chargés d'analyser le métier mais aussi le besoin exact du client en vue de rédiger un cahier des charges précis et exhaustif.
- des spécialistes de la résolution de problèmes complexes et combinatoires ou de la mise en place de systèmes à base de connaissances, problèmes pour lesquels une expérience significative est indispensable pour envisager sereinement l'aboutissement des projets qui y ont trait.
- des architectes capables de concevoir et mettre en place des architectures distribuées ainsi que modélisations objets complexes et évolutives.

▪ enfin, une direction technique composée de consultants et de chefs de projets à forte compétence technique se chargeant de garantir la fourniture d'un service de qualité irréprochable.

I-3.2 Clients et projets

Pacte Novation intervient dans de nombreux domaines aussi divers que

▪ l'industrie : SOLLAC, SPSE, Société Le Nickel.

▪ l'énergie : EDF, GDF.

▪ Le transport : Renault Sport, GEC Alsthom, ADP, Eurocontrol, ARAAM, Transnucléaire.

▪ L'industrie de l'informatique et des télécommunications : Alcatel, BULL.

▪ La finance des salles de marché : Crédit Agricole Indosuez, Crédit Lyonnais, Dresdner, Bank, Société Général.

Pacte Novation a participé à des projets ou programmes d'envergure comme :

▪ ALICE pour Renault Sport :

Créé spécifiquement pour Renault Sport dans le contexte de la formule 1, ALICE (Application Logicielle Intelligente pour Course et Essais) analyse les données envoyées par télémesure à chaque tour de piste et déduit les alarmes moteur.

▪ Simulateur de trafic ferroviaire pour ALSTOM :

Pacte Novation a développé pour ALSTOM une série de simulateurs de trafic ferroviaire. Vendus à l'exportation, ces simulateurs sont destinés à valider des commandes centralisées de métro ou de trains comme pour le Caire, Hong Kong, Athènes, Jakarta, Istanbul, Singapour.

▪ AGATE pour SPSE

Planification des arrivées de navires aux môles du Port Autonome de Marseille, placement intelligent des produits pétroliers dans les réservoirs du dépôt, planification des livraisons par pipeline et optimisation des coûts de transport : AGATE a procuré une souplesse incomparable pour tout le personnel de la société du Pipeline Sud Européen.

▪ MATOS

MATOS est un outil logiciel supportant les activités de modélisation des tâches sur lesquelles s'appuie la démarche de formalisation du besoin et de spécifications d'IHM

de Pacte Novation. MATOS implémente en partie les principes de MAD (Scapin, Sébillote, … de l'INRIA).

- Storia sur Internet

  Le projet Storia sur Internet consiste à visualiser, via une applet dans un browser, le trafic aérien civil sur l'Europe. L'applet montre une carte de l'Europe sur laquelle des points symbolisant les avions se déplacent automatiquement sans demande explicite de rafraîchissement. Elle peut utiliser des données historiques (mode « Replay ») ou des données Temps Réel (Mode « live ») en provenance du salon ATC-Maastricht 2001 et c'était, de l'avis de tous les participants, une grande première européenne et l'innovation la plus « décapante ».

I-3.3 Partenariats

Cette stratégie sérieuse fait qu'un climat de confiance tout naturel s'établit entre le client et PACTE NOVATION. Ainsi, après plusieurs années de collaboration, du statut de fournisseur, PACTE NOVATION devient partenaire. Nous intervenons aujourd'hui aux côtés de certains de nos clients pour les aider à répondre à des cahiers des charges, et ce directement en contact avec leur propre client. Nous leur assurons également des formations sur des sujets pointus dans des services de R&D sensibles.

Notre offre, très souvent enrichie par une double compétence fonctionnelle, permet à PACTE NOVATION de nouer des partenariats durables avec :

ILOG Editeur de composants logiciels.

ProACTIVE Projet de recherche mené dans le cadre du programme Esprit sous l'égide de la Communauté Européenne.

EUROTEAM Consortium accrédité par EUROCONTROL, l'organisme européen fédérant les recherches et développements dans le domaine du contrôle aérien.

TEMPOSOFT Editeur de logiciels Intranet pour la gestion et l'optimisation des ressources humaines

SOLLAC Le plus gros projet de l'industrie utilisant des techniques d'Intelligence Artificielle

Chapitre II :

Problème de satisfaction de contraintes :

Généralités

II – 2 <u>Définitions</u> :

Un Problèmes de Satisfaction de Contraintes (*Constraint Satisfaction Problem* ou *CSP*) à valeurs discrètes est composé de :

- un ensemble de *n* variables $X = \{x_1,...,x_n\}$
- un ensemble de *n* domaines $\{D_1,...,D_n\}$ où D_i est le domaine de la variable x_i , $i \in \{1,...,n\}$. Chaque domaine contient un nombre fini de valeurs. Une de ces valeurs doit être assignée à la variable correspondante.
- un ensemble de *contraintes*. Une contrainte R_S sur $S \subseteq X$ est un sous-ensemble du produit cartésien des domaines des variables de S :
  $S = \{x_{i_1},...,x_{i_m}\} \Rightarrow R_S \subseteq D_{i_1} \times ... \times D_{i_m}$, avec
  $j \in \{1,...,m\}, k \in \{1,...,m\}, j \neq k, 0 < m \leq n, i_j \in \{1,...,n\}, i_j \neq i_k$. Ce sous-ensemble représente les valeurs des variables autorisées par la contrainte.

Un CSP est dit *binaire* si toutes les contraintes portent sur au plus 2 variables ($\text{card}(S) \leq 2$).

Une variable est dite *instanciée* si une valeur de son domaine lui a été assignée. On notera l'affectation d'une valeur *a* à une variable x_i par (x_i,a).

Un ensemble de variables instanciées est dit *consistant* avec une contrainte R_S si le n-uple formé par les valeurs des variables de l'ensemble appartenant à S, appartient à R_S (aucune contrainte n'est violée).

Une *solution* au problème est une instanciation de *toutes* les variables du problème de sorte qu'elles soient consistantes avec *toutes* les contraintes du problème.

Un ensemble de variables instanciées ne comportant pas toutes les variables du problème mais étant quand même consistant avec toutes les contraintes est appelé *instanciation partielle*. Elle est notée par $((x_1,a_1), (x_2,a_2), ... , (x_i,a_i))$ ou par $(a_1, a_2, ... , a_i)$ où $\xrightarrow[a_i]{}$ lorsque l'ordre des variables est connue.

L'arbre de construction des solutions du problème est un arbre dont les nœuds sont constitués par des instanciations partielles. Les branches représentent chacune une variable instanciée. La branche située entre deux nœuds *A* et *B* comporte la variable qu'il faut rajouter à l'instanciation *A* pour obtenir l'instanciation *B*. La racine de l'arbre représente l'instanciation partielle nulle. Les feuilles de l'arbre représentent les solutions.

Exemple d'arbre de construction des solutions pour le problème suivant :

$X = \{x_1, x_2, x_3\}, D_1 = \{1,2,3\}, D_2 = \{1,2,3\}, D_3 = \{1,2\}$ avec comme contraintes

$x_1 \neq x_3, x_1 \neq x_2$ et $x_2 \neq x_3$

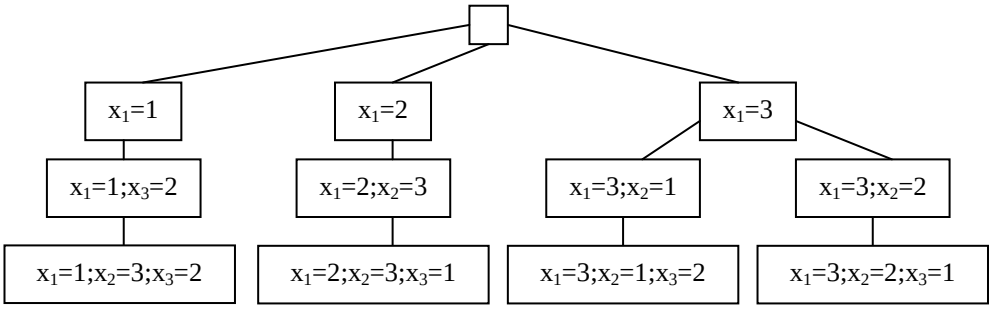


figure 2 - Arbre de construction des solutions-

II – 3 Approche constructive classique :

Algorithme de base

La méthode de base de résolution d'un CSP consiste à construire l'arbre des solutions et à désigner chaque feuille comme étant une réponse au problème. L'arbre se construit branche par branche, de la racine jusqu'aux feuilles. Toutes les combinaisons possibles des valeurs des variables sont essayées. Pour cela, l'algorithme général suivant est utilisé :

1. Créer une instanciation partielle *P* vide, et initialiser une pile vide
2. $D_{1...}D_n$ contiennent les domaines de valeurs pour les variables $x_1...x_n$
3. *Si P* est consistante
4. *Si* il y a encore une variable non-instanciée
5. Choisir un indice *i* correspondant à une variable x_i non-instanciée
6. *Aller* à 14
7. *Sinon*
8. *P* est une solution
9. *Si* la pile est vide, *Arrêter*
10. Dépiler et restaurer les domaines $D_1 \ldots D_n$ précédents
11. *i* reçoit l'indice de la dernière variable affectée dans *P*
12. Enlever de D_i la valeur qui était affectée à x_i dans sa dernière affectation dans *P*
13. Dépiler et restaurer le *P* précédent
14. *Si* le domaine D_i est vide
15. *Aller* à 9
16. Choisir une valeur *v* dans le domaine D_i de la variable x_i
17. Empiler *P* et empiler les $D_1 \ldots D_n$
18. Affecter *v* à x_i (réduire le domaine D_i à *{v}*) et ajouter cette affectation à *P*
19. *Aller* à 3

Figure 3 - Algorithme de base de construction de l'arbre des solutions

Lorsqu'une instanciation générée est inconsistante, ou lorsqu'il n'existe plus de valeur possible pour une variable (situation d'*impasse*), l'algorithme défait les affectations des précédentes variables jusqu'à retrouver une valeur admissible. Cette opération de « retour à l'étape précédente », qui correspond à « remonter » dans l'arbre jusqu'au nœud précédent, est appelée *backtracking* (ou « retour arrière »).

Les choix de la valeur (ligne 16) et de la variable (ligne 5) à traiter sont particulièrement cruciaux. En effet, si l'on choisit d'emblée une *bonne* valeur pour chaque variable, on peut trouver une solution en ne faisant aucun backtrack. De même, si l'on choisit de traiter en priorité une variable dont plus aucune valeur n'est admissible, on évite d'instancier pour rien d'autres variables avant de se rendre compte de l'échec.

On peut par ailleurs s'apercevoir que cet algorithme est en général très inefficace car certaines affectations sont opérées alors même qu'elles n'ont aucune chance de succès. Pour comprendre cela, prenons le problème suivant :

exemple 1 :

$X = \{x_1, x_2, x_3, x_4\}, D_1 = \{1,2,3\}, D_2 = \{1,2,3\}, D_3 = \{1,2,3\}, D_4 = \{1,2\}$ avec comme contraintes $x_1 \neq x_2$, $x_1 \neq x_4$ et $x_2 \neq x_4$.

Considérons l'instanciation partielle consistante suivante *{x_1=1, x_2=2, x_3=1}*. Si les valeurs des variables sont testées dans un ordre croissant, cette situation se produit en tout début de résolution, et les valeurs *2* et *3* de x_3 n'ont pas encore été testées. Lors de l'instanciation de x_4, les valeurs *1* et *2* seront testées sans succès puisque x_4 doit être différente de x_1 et de x_2. Le mécanisme de backtracking va alors remonter à la variable précédente (dans ce cas : x_3) afin de l'instancier avec ses prochaines valeurs possibles (c'est-à-dire *2* puis *3*), ce qui amènera à tester *2*2=4* instanciations inutiles puisque x_3 n'influence pas les contraintes d'exclusion entre x_1, x_2 et x_4. Cet effet néfaste de redécouverte des impasses est appelé *thrashing* [Kum92].

Pour parer à ce problème, on effectue des tests de consistance supplémentaires à certains endroits de l'algorithme. Deux catégories de techniques, sont employées : les méthodes à *test arrière* et les méthodes à *test avant*.

II – 4 Méthodes à test arrière :

Dans ces méthodes (tout comme dans l'algorithme de base *figure-2*), l'affectation des valeurs aux variables se fait sans se préoccuper des autres variables (non-encore instanciées) et pour

lesquelles il faudra trouver une valeur admissible plus tard. Ce type de méthode tombe également souvent en échec en raison du *thrashing*.

Dès lors, l'optimisation de ces méthodes passe par l'optimisation du mécanisme de retour arrière (*backtracking*). Les méthodes optimisées existantes sont des variantes de celui ci et souvent appelées *retour arrière intelligent* (*intelligent backtracking*). Le but que partagent ces systèmes est d'effectuer un backtrack *étendu* (appelé *backjump*) lors de la découverte d'une impasse : plutôt que de remonter au nœud précédent uniquement, ils remontent jusqu'à la première variable susceptible d'influencer la situation d'échec.

Dans notre exemple, ceci se traduit de la manière suivante :

les instanciations $\{x_1=1, x_2=2, x_3=1, x_4=1\}$ et $\{x_1=1, x_2=2, x_3=1, x_4=2\}$ sont connues comme étant inconsistantes car elles violent les contraintes $x_2 \neq x_4$ ou $x_1 \neq x_4$. Or, x_3 ne peut rien changer à cette situation car elle n'intervient dans aucune contrainte liée à x_2, x_1 ou x_4. L'algorithme va donc effectuer un *backjump* directement vers x_2 [FD99].

Plusieurs méthodes de *backjumping* existent. Elles varient suivant la finesse avec laquelle elles arrivent à déterminer la variable incriminée dans l'échec, ou la façon dont elles identifient les liens entre les variables. Les coûts (temps, mémoire) de traitement supplémentaires nécessaires aux différents algorithmes varient également. Les méthodes appartenants à cette famille d'algorithmes utilisées dans Open CSP seront détaillées ultérieurement. On peut néanmoins citer le *conflict-directed backjumping* [Fro97] qui se situe parmi les méthodes les plus performantes. [Kon94] présente en outre des méthodes dérivées encore plus complexes et performantes.

Une alternative au backjumping est constituée par les mécanismes d'*apprentissage* (*Dead-end Driven Learning*) [Fro97]. Ces systèmes génèrent dynamiquement des contraintes sur les variables déjà instanciées à un certain moment de la résolution. Ces contraintes « expliquent » comment éviter les impasses déjà découvertes. Dans notre *exemple-1*, la découverte des instanciations inconsistantes $\{x_1=1, x_2=2, x_3=1, x_4=1\}$ et $\{x_1=1, x_2=2, x_3=1, x_4=2\}$ généreraient la contrainte supplémentaire $\{x_1 \neq 1 \; ou \; x_2 \neq 2\}$, ce qui assurerait deux backtracks successifs (vers x_3 puis vers x_2) puisque l'instanciation partielle $\{x_1=1, x_2=2\}$ serait devenue inconsistante. Là aussi, de nombreux algorithmes existent et peuvent d'ailleurs se greffer sur des systèmes existants de backjumping.

Une caractéristique importante de l'ensemble de ces méthodes à *test arrière* est que le nombre de valeurs possibles lors de l'instanciation d'une variable est important (l'ensemble des valeurs non-testées du domaine de cette variable.) En effet, la réduction de l'espace de recherche se fait, lors de la construction de l'arbre des solutions, par élagage d'un sous-arbre de recherche

potentielle dont la racine se situe à un niveau inférieur au niveau courant (lorsqu'on backjump vers une précédente variable x_{back}, on élague tout le sous-arbre dont la racine est la première instanciation contenant la valeur courante de x_{back}). Il n'y a donc jamais de réduction *a priori* des domaines des variables.

II – 5 Méthodes à test avant (*Look-Ahead*) :

Ici, l'efficacité est trouvée en supprimant, avant instanciation des variables (avant la ligne 5 dans *fig-2*), certaines valeurs des domaines des variables. Cela élague donc les sous-arbres de recherche potentielle dont les racines sont les premières instanciations partielles contenant les valeurs supprimées pour ces variables. Ces valeurs supprimées sont les valeurs inconsistantes avec les affectations déjà effectuées. On voit donc *à l'avance* les échecs potentiels. Ce type d'algorithme provoque donc beaucoup moins de *backtrack* (sans toutefois l'éliminer car il arrive que le domaine d'une variable non-instanciée devienne vide à force d'en supprimer les valeurs : dans ce cas, on ne peut plus trouver de solution avec l'instanciation partielle courante et il faut effectuer un backtrack).

Dans l'*exemple-1*, l'instanciation partielle $\{x_1=1\}$ réduit le domaine D_2 à $\{2,3\}$ et le domaine D_4 à $\{2\}$. L'instanciation partielle suivante $\{x_1=1, x_2=2\}$ réduit alors D_4 à l'ensemble vide. Dès lors, un backtrack a lieu pour modifier x_2.

Il est très important de remarquer qu'une valeur d'une variable non-encore instanciée peut être *indirectement* inconsistante avec l'instanciation partielle courante. Considérons par exemple le problème suivant ***(3)*** :

$X = \{x_1, x_2, x_3, x_4\}, D_1 = \{1,2,3\}, D_2 = \{1,2,3\}, D_3 = \{1,2,3\}, D_4 = \{1,2\}$ avec comme contraintes $x_1 \neq x_4$, $x_2 \neq x_4$ et $x_3 = x_4$

Aucune valeur de x_3 n'est *directement* inconsistante avec l'instanciation partielle $\{x_1=1, x_2=2\}$. Mais, par l'intermédiaire de la variable non-instanciée x_4, toutes les valeurs le deviennent.

Ces indirections peuvent en outre porter sur plusieurs variables à la chaîne. De façon générale, un algorithme de *test avant* élague un espace de recherche d'autant plus grand qu'il peut détecter des indirections importantes dans les inconsistances. Il est toutefois évident que le temps passé à l'élagage augmente avec la taille de l'espace élagué. [Fro97] présente 3 grands types de méthodes de *Look-Ahead* (classés ici dans l'ordre croissant de leur qualité) :

Le *Forward-Checking* consiste à ne tester que la consistance directe avec les variables instanciées (consistance de niveau 1, *cf. ci-dessous*).

Les algorithmes intermédiaires *Full-Looking-Ahead* et *Partial-Looking-Ahead* sont des systèmes heuristiques qui ne fournissent pas exactement la consistance de niveau 2 (*cf. ci-dessous*).

La mise en consistance (*n-consistency*). Ces systèmes testent des inconsistances d'autant plus indirectes que *n* est grand. Ils sont définis comme suit : l'application d'un algorithme de mise en consistance de niveau *n* assure d'avoir au moins une instanciation consistante des *n* prochaines variables, quelles qu'elles soient, et quelle que soit la valeur choisie pour la première de ces variable. Le temps de traitement nécessaire est en $o(e^n)$. Plus *n* est grand et moins il est nécessaire de faire de backtracking lors du parcours de l'arbre. Cependant, [Kum92] montre que pour éliminer totalement le recours au backtracking, il faut, pour un arbre à *p* nœuds, un niveau de mise en consistance *p* (ce qui redonne donc un temps de traitement exponentiel au problème global). Ainsi, il n'est pas forcément intéressant d'appliquer une mise en consistance élevée. Le niveau le plus utilisé est 2 (*arc-consistency*). Les techniques de 2-consistence utilisées dans Open CSP sont détaillées dans le chapitre *Consistance d'arcs et propagation de contraintes*.

L'ensemble de ces méthodes à *test avant* a en outre les caractéristiques suivantes : le temps passé à chaque étape de la construction peut être long car l'élagage est une opération complexe, et le nombre de valeurs possibles lors de l'instanciation est faible car certaines valeurs détectées comme *a priori* inconsistantes ont déjà été élaguées.

Chapitre III :

Algorithmes de Backtrack

III–1 **Introduction** :

Tous les algorithmes de backtracking exploitent la notion d'instanciation partielle. On identifiera deux cas particuliers : l'instanciation nulle et l'instanciation totale. Toutes les procédures de recherche commencent par une instanciations nulle qu'ils feront évoluer successivement en instanciant d'autres variables une à une. Si une situation d'échec (dead-end) est détectée ceci voudra dire que l'instanciation partielle courante ne peut pas être étendu à une instanciation totale, l'algorithme revient en arrière pour affecter une autre valeur à une des variables déjà instanciée.

Les algorithmes de backtrack arrivent à donner une solution (ou plus) lorsque celle ci existe, ou à conclure que le problème est insoluble. La différence entre ces algorithmes réside dans la rapidité avec laquelle les situations d'échec sont détectées et la façon avec laquelle l'algorithme se rétracte.

III-2 **Algorithmes de Backtrack :**

III–2.1 **Backtrack chronologique :**

Dans ce genre d'algorithme, les variables sont instanciées une à une. Dès que toutes les variables d'une même contraintes sont instanciées, on teste la validité de celle ci. Si une instanciation partielle viole au moins une des contraintes, cela signifie que l'affectation courante ne peut pas mener à une solution , un retour arrière (backtracking) est exécuté vers la variable la plus récemment instanciée. Cette technique permet de réduire l'ensemble des cas possibles à chaque fois qu'un backtrack est exécuté.

Procédure BACKTRACKING
Entrées : un graphe de contraintes, les variables {x_1, x_2, …, x_n} et les domaines respéctifs {D_1, D_2, …, D_n}.
Sorties : Une solution si celle ci existe .

Début
I←1 ;
$D'_i \leftarrow D_i$;
Tant que $1 \leq i \leq n$ *faire*
Instancier $x_i \leftarrow$SELECTVAL ;
Si x_i est NULL *alors*
i←i-1 ;
sinon i←i+1 ;
$D'_i \leftarrow D_i$;
FinFaire
Si i==0 *alors*
Return « Problème inconsistent »
Sinon *Return* la solution {x_1, x_2, …, x_n}
Fin ;

Procédure SELECTVAL
Tant que D'_i non vide *faire*
Choisir un élément $a \in D'_i$ puis le supprimer de D'_i
Si (x_i,a) est consistante avec $\xrightarrow[a_{i-1}]{}$ *alors*
Return a ;
Finfaire
Return NULL ;
Fin ;

Figure 4 -Algorithme de backtrack chronologique et la procédure de sélection de valeurs

Depuis la parution de la première version de l'algorithme de backtrack chronologique, de grands efforts ont été fait dans le domaine de la résolution de CSP dans le but d'améliorer cette technique de recherche de solution.

Analyser les raisons des situations d'échecs rencontrées peut aider à « mieux » revenir en arrière et à éviter le thrashing, d'où le *backjumping*.

Une autre approche consiste à se « souvenir » des causes d'échecs (en les formulant sous forme de nouvelle contraintes) pour éviter de rencontrer les mêmes contradictions dans le futur. C'est l'idée de base des techniques de recherche avec apprentissage (*Constraints Recording* ou *learning*).

III – 2.2 <u>Backjumping</u> :

Le backjump procède de la même manière que le backtrack chronologique sauf dans le cas où une contradiction (dead-end) est rencontrée. Dans ce cas le backjump essaye de retrouver la variable précédemment instanciée responsable de cette situation d'échec et effectue son retour arrière vers elle. Cette technique se base essentiellement sur la notion d'ensembles de conflit (conflict sets).

<u>Définitions</u> (ensembles de conflit – Conflict Sets):[FD99]

- Soit $\xrightarrow[a_i]{} = (a_1, a_2, \ldots, a_i)$ une instanciation partielle consistante et soit *x* une variable non-instanciée. Si le domaine de x ne contient aucune valeur consistante avec $\xrightarrow[a_i]{}$, on dira que $\xrightarrow[a_i]{}$ est l'ensemble de conflit de la variable x.

<u>Définitions</u> : [FD99]

- On appellera une instanciation partielle $\xrightarrow[a_i]{}$, contradiction (ou no-good) si celle ci n'apparaît dans aucune solution du problème.
- Soit $\xrightarrow[a_i]{} = (a_1, a_2, \ldots, a_i)$ une instanciation partielle inconsistante avec la variable x_{i+1}. On dira que le saut vers la variable x_j ($j \leq i$) à partir de x_{i+1} , est *sauf* si l'instanciation partielle $\xrightarrow[a_j]{} = (a_1, a_2, \ldots, a_j)$ est un nogood , donc ne peut pas mener à une solution.

En d'autres termes, on sait que si la valeur de x_j est changée, aucune solution ne sera omis.

Le principal enjeu dans le backjump est de s'assurer que le saut arrière qu'effectuera l'algorithme ne va pas exclure une partie de l'espace des solutions qui contiendrait éventuellement une solution.

Trouver la variable responsable d'une situation d'échec $\xrightarrow[a_i]{}$ est simple puisqu'on devra tester la consistance de x_{i+1} avec au plus, *i* tuples $\xrightarrow[a_k]{}$, *k=1,.., i.*

On pourra dire que la variable vers laquelle un backjump est effectué lorsqu'on se trouve face à un nogood $\xrightarrow[a_i]{}$, est la dernière variable dont l'instanciation a rendu inconsistante la dernière valeur du domaine de x_{i+1}.

Une instanciation partielle $\xrightarrow[a_i]{}$ inconsistante avec x_{i+1} peut contenir plusieurs instanciations partielles $\xrightarrow[a_j]{}$ (où $j \leq i$) elles mêmes inconsistantes avec x_{i+1}. Ces « sous-instanciations partielle » ne peuvent jamais paraître dans une des solutions du problème (inconsistance !). Voilà pourquoi le backjump ne revient plus à la variable x_i lorsqu'une situation

d'échec est rencontré à x_{i+1}, mais va plutôt chercher la variable x_b la plus récemment instanciée dont l'instanciation partielle $\xrightarrow[a_b]{}$ ne contient aucun « sous-ensemble » en conflit avec x_{i+1}.

III – 2.3 Algorithme de backjump de Gaschnig (Gaschnig's backjumping) :

Cet algorithme détecte les conflits et stocke certaines informations au moment même où il génère les instanciations partielles pour connaître la variable incriminée dans une situation d'échec lorsque celle ci se produit. Cet algorithme utilise une technique de marquage grâce à laquelle toute variable connaît le plus récent prédécesseur inconsistant avec l'une des valeurs de son domaine.

Lors de la génération de l'instanciation partielle $\xrightarrow[a_i]{}$, l'algorithme dispose d'un pointeur *high[i]* qui pointe sur la dernière variable instanciée, inconsistante avec au moins une valeur du domaine de x_i. C'est vers cette valeur (pointée par *high[i]*) qu'un backjump devrait se produire à partir de la variable x_i.

Procédure BACKJUMPING (*Backjump de Gaschnig*)
Entrées : un graphe de contraintes, les variables {x_1, x_2, …, x_n} et les domaines respectifs {D_1, D_2, …, D_n}.
Sorties : Une solution si celle ci existe .

```
Début
        i←1 ;
D'i←Di ;
Tant que 1≤ i ≤n faire
        Instancier xi←SELECTVAL ;
        Si xi est NULL alors
                i←high[i] ;
        sinon   i←i+1 ;
                D'i←Di ;
                high[i] ←0 ;
FinFaire
Si i==0 alors
        Return « Problème inconsistent »
Sinon   Return la solution {x1, x2, …, xn}
Fin ;
```

```
Procédure SELECTVAL
début
        Tant que D'i non vide faire
                Choisir un élément a∈ D'i puis le supprimer de D'i
                consistant←faux ;
                k←1 ;
                Tant que (k < i ET consistant) faire
                        Si k> high[i] alors
                                high[i] ←k ;
                        si (xi,a) est inconsistant avec ⟶ (a_k) alors
                                consistant←faux
                        sinon   k←k+1 ;
                finfaire
                si consistant alors
                        return a ;
        finfaire
        return NULL ;
fin ;
```

Figure 5 -Algorithme de Backjumping de Gaschnig et la procédure de sélection de valeurs

L'exemple suivant aidera à mieux comprendre le comportement du backjump de Gaschnig.

Soit le CSP dans la figure ci-dessous. Les variables sont les sommets du graphe de contraintes. Leurs domaines respectifs sont mentionnés à l'intérieur des cercles qui représentent les variables. Les arcs du graphe sont les contraintes du problème. On veut colorier chaque nœuds de sorte que toute pair de variables liées par une contrainte aient des couleurs différentes.

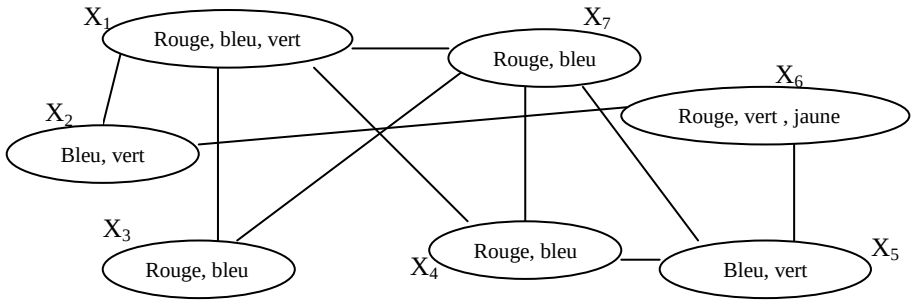


Figure 6 - Problème de coloration de graphe

Une situation d'échec est rencontrée à x_7 pour l'instanciation partielle $\xrightarrow[a_6]{}$ (x_1=rouge, x_2=bleu, x_3=bleu, x_4=bleu, x_5=vert, x_6=rouge). On a alors le pointeur $high_7$ qui vaut 3, car la valeur rouge du domaine de la variable x_7 a été interdite par l'affectation x_1=rouge (sachant qu'on a comme contrainte $x_1 \neq x_7$) de même que la valeur bleu qui a été interdite par l'affectation x_3=bleu (et $x_3 \neq x_7$) ; dès lors plus besoin d'aller tester les variables restantes puisque plus aucune valeurs n'est disponible dans le domaine de x_7 ($D'_7 = \varnothing$).

Un backjump est donc effectué vers la variable d'indice $high_7$, donc x_3. Après le retour à x_3 l'algorithme ne trouve aucune autre valeur pour instancier cette variable ($D'_3=\varnothing$*: la seule autre alternative restante* x_3=*rouge a été interdite par l'affectation* x_1=*rouge*). Dans ce cas, $high_3$ vaudra 2, un saut arrière vers la variable x_2 est donc préconiser.

Le backjump de Gaschnig permet des sauts arrière dans des situation d'échecs lorsque celles-ci se produisent aux niveau des feuilles de l'arbre des solutions. Lorsque les situations d'échecs sont internes (le cas de x_3) , l'algorithme revient vers la variable précédente uniquement.

Les algorithmes détaillés plus bas utilisent la notion *d'ensemble de conflit* associé à une variable (*Conflict Sets*) et arrivent à retrouver la variable incriminée dans une situation d'échec même lorsque le conflit est interne.

III – 2.4 Backjumping à base de graphe de contraintes (Graph-based backjumping) :

Cette variante de l'algorithme du backjumping n'a pas été implémenté dans *Open CSP* à cause de la structure de graphe de contraintes sur lequel elle se base. Néanmoins, nous l'exposerons quand même sans trop de détails pour pouvoir présenter par la suite l'algorithme du *Conflict Directed Backjumping learning*.

Le *graph-based backjumping* (comme son nom l'indique) utilise le graphe de contraintes pour soutirer les informations nécessaires lors de la prise de décision dans une situation d'échec.

Face à un *dead-end* au moment d'instancier la variable *x*, l'algorithme revient vers la variable la plus récemment instanciée *y*, «liée» à *x*. Si le domaine de la variable *y* est vide (aucune autre valeur n'est disponible pour être affecté à *y*) l'algorithme revient vers la variable la plus récemment instanciée *z*, liée à *y* ou à *x*.

Dans cet algorithme , le calcul du pointeur *high[i]* est évité. Mais il est remplacé par une structure de données plus complexe qui a pour tâche de stocker le graphe de contraintes. Toutes les décisions que prendra l'algorithme se feront en se basant sur le graphe initial.

Cependant, cette technique introduit les notions de *parent* et *d'ancêtre* de variables qui nous seront utiles par la suite.

Définitions :[FD99]

Soit un graphe de contraintes. On supposera que les variables sont données dans un ordre fixe. Les ancêtres de la variable *x*, noté *anc(x)* sont le sous-ensemble de variables qui précédent *x* et qui lui sont liées par une contrainte.

Le parent de la variable *x*, noté *p(x)* est la dernière variable contenue dans *anc(x)*.

III- 2.5 Backjumping à base d'apprentissage de conflits (Conflict-Directed Backjumping learning) :

Dans certains cas, la simple découverte d'un nogood ne suffit pas à empêcher l'algorithme de refaire l'instanciation partielle contenue dans ce nogood. En rendant le nogood explicite, on ne risque plus de le parcourir inutilement. Expliciter un nogood sous forme d'une nouvelle contrainte contribue aussi à restreindre l'espace de recherche. Cette technique d'ajout et de sauvegarde de nouvelles contraintes est à la base des algorithmes dits *d'apprentissage* (*learning*).

L'apprentissage dans ce cas signifie se rappeler de certaines informations potentiellement utiles, déduites des données initiales du problème. A chaque backtrack, l'algorithme enregistre le nogood rencontré et l'inclut au problème sous forme d'une nouvelle contrainte.

Dans [FD99], on introduit l'idée d'explication à base d'apprentissage (*explanation based learning*) car sauvegarder explicitement l'instanciation partielle qui conduit à une situation d'échec n'apporte rien à l'algorithme du fait que la stratégie de recherche du backtrack impose de ne pas passer par un même « *état*[*] » plus d'une fois. Le but d'inclure ces nouvelles contraintes induites par les nogood est plutôt de pouvoir retrouver un début de preuve (ou d'explication) aux inconsistances rencontrées.

Dans *Open CSP*, on s'est servi des données fournies par l'algorithme *Conflict Directed backjumping learning* pour pouvoir retracer les échecs rencontrés. Cette approche est détaillée dans le chapitre implémentation.

[*] On entend par état , l'état dans lequel on se trouve dans le parcours de l'arborescence des solutions.

```
Procédure Conflict Directed Backjumping learning
Entrées : un graphe de contraintes, les variables {x1, x2, …, xn} et les
domaines respéctifs {D1, D2, …, Dn}.
Sorties : Une solution si celle ci existe .
Début
        i←1 ;
        D'i←Di ;
        Ji←∅ ;
Tant que 1≤ i ≤n faire
        Instancier xi←SELECTVAL-CBJ ;
        Si xi est NULL alors
           Sauvegarder Ji et les valeurs correspondantes autant qu'un
nogood ;
           i-prec ← i ;
           i ←la plus grande valeur d'index dans Ji ;          // --- Backjump
           Ji ← Ji ∪ Ji-prec – {xi} ;
        sinon  i←i+1 ;
               D'i←Di ;
               Ji ←∅ ;
FinFaire
Si i==0 alors
        Return « Problème inconsistent »
Sinon  Return la solution {x1, x2, …, xn}
Fin ;

Procédure SELECTVAL-CBJ
début
        Tant que D'i non vide faire
               Choisir un élément a∈ D'i puis le supprimer de D'i
               consistant←faux ;
               k←1 ;
               Tant que (k < i ET consistant) faire
                      si (xi,a) est inconsistant avec  ak→  alors
                             ajouter xk à Ji ;
                             consistant←faux
                      sinon  k←k+1 ;
               finfaire
               si consistant alors
                      return a ;
        finfaire
        return NULL ;
fin ;
```

Figure 7 - Algorithme du Conflict Directed Backjumping learning.-

Cet algorithme utilise le même principe que le *Graph-based Backjumping*. Plutôt que de se baser sur les informations déduites du graphe de contraintes comme le fait le *Graph-based Backjumping*, le *Conflict-Directed Backjumpin learning* exploite les informations rassemblées lors de son parcours de l'arborescence des solutions.

Dès qu'une valeur d'une des variables, soit x_j, d'une situation d'échec $\xrightarrow[a_i]{}$ crée un conflit avec l'une des valeurs de la prochaine variable à instancier x_{i+1}, la variable x_j est associée à x_{i+1} comme prochaine « *variable de retour* » potentielle (vers laquelle s'effectuera le Jumpback à partir de x_{i+1}). Ainsi, l'algorithme associe à chaque variable un ensemble de variables de retour potentielles (*Jumpback Set*). On appellera J_i , le *Jumpback Set* associé à la variable x_i.

Les deux notions d'ancêtre et de parents reviennent dans cet algorithme car, le *Jumpback set* J_i contiendra les ancêtres de la variable x_i en conflit avec les valeurs de son domaine D_i et le retour arrière s'effectuera vers le parent de cette variable $p(x_i)$ (aussi contenu dans J_i).

Chapitre IV :

Consistance d'arcs et propagation de contraintes

IV - 1 Introduction

Nous avons introduit plus haut la notion de thrashing dont souffre le backtracking. Cette tendance à redécouvrir inutilement les combinaisons de valeurs inconsistantes peut être réduite si l'on s'assure d'avoir une consistance d'arcs convenable avant d'appliquer l'algorithme du backtracking. Le chapitre suivant expose les différentes techniques utilisées dans *Open CSP* pour assurer la *consistance d'arcs*.

IV – 2 Consistance d'arc et propagation de contraintes :

Définitions :[Bes94]

- Soient i, j deux variables d 'un CSP. Si $a \in D_i$ et $b \in D_j$, b est dite *support* pour la valeur a si et seulement si la contrainte $C_{ij}(a,b)$ est satisfaite.
- Soit $a \in Di$. S 'il existe un support *b* pour la valeur a pour toute contrainte C_{ij} alors *a* est dite *viable*.
- Le graphe de contraintes est dit *consistant*, si et seulement si toutes les valeurs que peuvent prendre les variables de toutes les contraintes sont viables.
- Consistance de nœud : consiste à éliminer les valeurs qui violent les contraintes unaires.
- Consistance d'arc : On dira qu 'un arc (*Vi,Vj*) d 'un graphe de contraintes est consistant si pour toute valeur *x* du domaine D_i, il existe au moins une valeur *y* du domaine D_j, telle que la contrainte entre ces deux variables soit satisfaite.

IV-2.1 *Arc - Consistency – 1 (AC-1)* :

L 'idée de base des premiers algorithme de consistance d 'arcs était de supprimer les valeurs pour lesquelles on ne pouvait pas trouver de support.

La procédure *REVISE(i,j)* élimine les valeurs inconsistantes par rapport à la contrainte qui lie les variables (x_i,x_j).

Figure 8 – Algorithme de consistance d'arcs 1

```
Procédure REVISE( i,j )
début
    supprimer    ← faux;
    Pour tout a ∈ Di faire
        S'il n 'existe pas de support b ∈ Dj pour la valeur a alors
                    supprimer a de Di;
                    supprimer ← vraie;
        finSi
    finFaire
    renvoyer supprimer;
fin ;

Algorithme AC-1
début
    Q ← { tout arc ( i,j ) ∈G / i≠j }
    Répéter
        Changer ← faux;
        Pour tout élément ( i,j ) ∈ Q faire
                Changer ← REVISE ( i,j ) OU Changer;
        finFaire
    jusqu'à (Changer == faux)
fin.
```

Pour s'assurer que tout le graphe est consistant, il ne suffit pas d'exécuter la procédure *REVISE* une seule fois, car cette procédure risque d'éliminer certaines valeurs d 'un domaine D_i et donc, rendre certaines autres valeurs de D_j inconsistantes par rapport à la contrainte (C_{ji}). Un deuxième parcours du domaine D_j s'impose. L'algorithme *AC–1* assure l'obtention d'une consistance de tout le graphe de contraintes[Kum92].

Prenons un exemple pour montrer cela :

Soit le problème de coloration de graphe suivant :

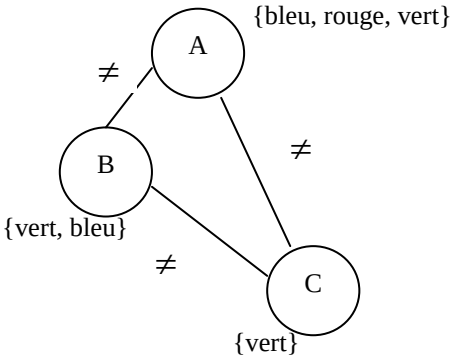


Figure 9 – Graphe de contraintes d'un problème de coloration de graphe –

L'arc (A,B) est initialement consistant, car pour toute valeur de D_A affectée à *A*, il y aura toujours une valeur de D_B à affecter à *B* telle que A≠B. L'arc (*B,C*) est rendu consistant en éliminant la valeur *vert* de D_B. Dès lors, l'arc (*A,B*) devient inconsistant car, pour *A=bleu* , aucune valeur de D_B (réduit à la seule valeur : *bleu*) n'est plus disponible pour être affecter à *B*. Le domaine D_A doit être, à son tour, réduit.

L 'algorithme AC-1 ne différencie pas entre les variables dont les domaines ont subi des modifications et celles dont le domaine reste inchangé. Un parcours inutile de ce genre de domaines est redondant. L 'algorithme AC-3 pallie à ce problème .

IV – 2.2 *Arc - Consistency – 3 (AC-3) :*

L'algorithme *AC-1* perd beaucoup de temps à effectuer des test de consistances inutiles et redondants. Si la procédure *REVISE* réduit un domaine D_k donné, tous les arcs du graphe seront re-tester même ceux qui n'ont subi aucun changement.

La réduction du domaine d'une variable v_k ne devrait affecter que les domaines des variables v_i tels que l'arc (v_i,v_k) existe. Aussi, si la procédure *REVISE* est appliquée à un arc (v_k,v_m) et qu'une réduction du domaine D_k est faite, il n'est pas nécessaire de tester l'arc (v_m,v_k) car aucune des valeurs supprimée de D_k ne risquait d'être un support pour aucune valeur dans D_m.

AC-3 reprend la même procédure REVISE que sa version antérieur AC-1.

```
Algorithme AC-3
début
        Q ← { tout arc ( i,j ) ∈G / i≠j  }
       Tant que Q non vide faire
         Choisir arbitrairement un élément (k,m) dans Q et le supprimer
         Si   REVISE (k,m)  alors
                  Q ← Q∪{(i,k) tel que (i,k) est un arc de G, i≠k, i≠m};
         finSi
       finFaire
fin.
```

Figure 10 - Algorithme de consistance d'arcs 3 -

Durant son parcours, *AC-3* reprend des pairs de valeurs déjà connues pour être consistantes (*ou inconsistantes*). Pour affiner ce parcours et pour qu'il ne concerne que les paires de valeurs dont on ne connaît pas l 'état (*consistantes ou inconsistantes*) , une amélioration a été introduite dans la version quatre de cet algorithme (*AC-4*).

Si on considère *d* comme la taille maximale des domaines de variables et *e* le nombre de contraintes du problème (le nombre d'arcs dans le graphe) [Mac85] montre que la complexité d'un tel algorithme de consistance d'arcs est en $O(ed^3)$.

IV – 2.3 *Arc - Consistency – 4 (AC-4) :*

L'algorithme effectue une première phase d'initialisation afin de connaître les paires de valeurs consistantes/ inconsistantes puis procède au décompte des supports de chaque valeur, à la fin de cette phase, on élimine les valeurs qui n'ont pas de supports pour aboutir à un premier état de semi-consistance. Dès qu'une valeur est éliminée du domaine, on stockera la paire <variable, valeur supprimée> dans une liste pour une vérification de consistance antérieur.

Cette vérification est imposée par le fait que la suppression d'une valeur d'un domaine peut perturber la consistance des variables voisines (lorsque la valeur supprimée constitue le seul support pour d'autres valeurs appartenants aux domaines des variables avoisinantes).

```
Procédure d 'Initialisation:
    début
      Q ← ∅ ;
      S ← ∅ ;
      Pour tout arc (i,j) du graphe de contrainte faire
            Pour chaque élément a de Di faire
                  total ← 0 ;
                  Pour tout élément b de Dj faire
                        Si Cij(a,b) alors
                              total ++;
                              Sj,b ← Sj,b ∪ { <i,a> } ;
                        finSi
                  finFaire
                  counter[(i,j),a] ← total ;
                  Si counter[(i,j),a] ==0 alors
                        supprimer a de Di;
                        Q ← Q ∪ { <i,a> };
                  finSi
            finPour
      finFaire
      renvoyer Q;
    fin Initialize


Algorithme AC-4
    début
      Q ← Initialize();
      Tant que Q non vide faire
            Choisir arbitrairement dans Q une paire <j,b> puis la
    supprimer
            Pour tout élément <i,a> de Sj,b faire
                  counter[(i,j),a] --;
                  Si ( counter[(i,j),a]==0 ) && ( a ∈ Di ) alors
                        Supprimer a de Di;
                        Q ← Q ∪ { <i,a> };
                  finSi
            finFaire
      finFaire
    fin;
```

Figure 11 - Algorithme de Consistance d'arcs 4 -

- Dans l 'algorithme AC-4, on considère que les valeurs des domaines sont ordonnés. On cherche alors, un support pour toute valeur *a* d 'une variable *i* quelle que soit la contrainte *Cij* pour montrer que *(i,a)* est viable.
- L 'algorithme AC-4 exploite la symétrie des contraintes pour minimiser les temps de calcul (*Cij (a,b) = Cji (b,a)*).

Il serait intéressant de se poser la question à savoir si la consistance d'arcs suffit à elle seule à aboutir à une solution ! Cette situation est envisageable dans le cas où tous les domaines de variables se réduisent à un seul élément. On dira alors, que le problème possède « une seule » solution qui s'obtient en affectant à chaque variable, la seule valeur qui existe dans le domaine associé. Cependant, tel n'est pas le cas dans la majorité des problème.

Il existe par contre un degré de consistance pour lequel, la recherche de solution n'aura pas lieu d'être. C'est la *K-consistance*.

Définition : [Kum92]

Un graphe de contraintes est dit *K-Consistant* si pour toutes instanciation partielle de k-1 variables, il existe une affectation possible d'une *k-ème* variable quelconque de sorte que toutes les contraintes soient satisfaites.

Un graphe de contraintes est fortement *K-Consistant* s'il est *J-Consistant* pour tout $J \leq K$.

La consistance d'arcs est équivalente à un état de *forte Consistance de niveau 2*.

La consistance de nœuds quand à elle représente la *forte Consistance de niveau 1*.

Des algorithmes existent pour obtenir la *forte K-Consistance* d'un graphe lorsque K >2 [Coo89]. Un graphe de contraintes à *n* sommets (*n* variables) *fortement n-consistant* n'aura pas besoin de *backtrack* pour être résolu. Cependant, l'inconvénient ici est la complexité exponentielle des algorithmes de *K-Consistance*.

Il existe cependant une autre technique pour réduire les backtrack dans un CSP sans pour autant passer par la K-Consistance où K=n (dans un graphe à n sommets). Une K-consistance où k<n suffirait dans le cas où le graphe de contraintes est *ordonné*.

IV-3 Ordonnancement de valeur et de variable :

Définition :

Un graphe de contrainte ordonnée est un graphe dont les variables sont linéairement ordonnées.

L'ordre des variables dans lequel elles vont être instancié peut influencer le processus de résolution. De même, les valeurs choisies pour être affectées aux variables peuvent être responsable de la vitesse avec laquelle l'algorithme pourra aboutir à une solution (ou conclure qu'il n'y a pas de solutions).

L'ordonnancement de variables dans certains CSP peut éliminer le recours au backtrack. La première branche parcourue de l'arbre des solutions mènera à une solution [Kum92].

Nous allons définir la notion de *Largeur de graphe* pour pouvoir présenter le théorème ci-dessous.

Définitions :[FD99]

Soit un CSP dont les variables ont été ordonné selon un ordre *d*.

La largeur d'un nœud dans un graphe de contraintes ordonné est le nombre d'arcs (de contraintes) qui le lient aux nœuds qui le précédent dans l'ordre *d*.

La largeur d'un graphe ordonné est la largeur maximale de ses nœuds.

La largeur d'un graphe de contrainte quelconque est la largeur minimale qu'on peut obtenir avec tous les ordonnancement de variables possible applicables à ce graphe.

Théorème :[BM96]

Si *G*, un graphe de contraintes est *fortement k-Consistant* et $K>\omega$ où ω est la largeur du graphe, alors il existe un ordre de variables pour lequel il n'y a pas besoin de faire de retour-arrière pour trouver la solution (*backtrack free*).

Preuve :

Si ω est la largeur du graphe, il existe alors un certain ordre de variable *d* pour lequel la largeur des nœuds est au plus égale à ω. Supposons que les variables sont instanciées suivant cet ordre *d* ; à chaque fois qu'une nouvelle variable est instanciée, il est clair qu'on pourra lui trouver une affectation valide car :

Cette valeur devra être consistante avec au plus ω variables connectées à la variable en cours

Le graphe de contraintes est au moins *fortement* $(\omega+1)$*-Consistant* (hypothèse) □

Malgré son intérêt, la K-Consistance lorsque K>2 est peu pratique à cause de l'explosion combinatoire des calculs qu'elle entraîne. Par contre, l'ordonnancement de variables (et de valeurs) peu néanmoins influencer le processus de recherche de solutions.

IV-3.1 Ordonnancement de variables :

Les performances d'un backtracking dépendent essentiellement de la manière dont les variables/valeurs sont choisies pour être instancié.

Il existe deux approches pour ordonner les variables d'un CSP [Bar98]:

Ordre statique : défini à l'initialisation. Utilisé lorsqu'aucune information supplémentaire n'est obtenue durant le processus de résolution.

Ordre dynamique : prend en compte les données courantes de la recherche de solution et doit être mis à jour durant la résolution.

Plusieurs heuristiques d'ordonnancements de variables existent. La plus communément utilisée est celle qui se base sur un principe paradoxal à priori, car elle favorise les variables qui ont le moins de chance de trouver un instanciation valide (*first-fail principle*)!

L'idée qui résume cette technique peut être formuler comme suit :

« Pour réussir, commencez par trouver les sources d'échecs »

Dans cette technique la variable qui a le moins de valeurs dans son domaine est choisie pour être instanciée en premier. Notre but est bien sûr de pouvoir trouver une valeur valide à affecter et, il est clair qu'avec cette méthode, on se donne moins de chance de trouver de telle valeurs. Par contre, les échecs sont vite repérés grâce à cette approche. Plus tôt on découvre les contradictions mieux cela vaudra pour le processus de recherche à long terme.

En appliquant cette heuristique d'ordonnancement, la profondeur moyenne de l'arbre des solutions parcourues est diminuée car les situations d'échecs seront détecté assez tôt.

Une autre approche consiste à favoriser les variables qui participent au plus grand nombre de contraintes. En choisissant d'instancier les variables les plus contraintes d'abord, on force les cas d'inconsistance à paraître. Une fois ces situations d'échecs rencontrées, on peut espérer avoir plus de chance par la suite en allant directement à la solution.

Ces deux heuristiques ont été implémenté dans *Open CSP*. L'ordonnancement de variables est effectué une fois toutes les données du problème réunies (variables, contraintes). Une fois les variables ordonnées, on tente d'appliquer une 2-Consistance au graphe de contraintes avant de lancer l'algorithme de résolution (*cf. Chapitre V*) .

IV-3.2 Ordonnancement de valeurs :

Une fois la variable à instancier choisie, le domaine de celle ci représente souvent un large choix de valeurs. L'ordre dans lequel ces valeurs doivent être considérées peut aussi avoir une impact sur l'évolution du processus de résolution.

Il est clair que si le CSP n'a pas de solutions ou si l'on cherche à déterminer toutes les solutions possibles d'un problème, le choix des valeurs est alors indifférent.

Si par contre le processus de résolution s'arrête à la première solution trouvée, on cherchera à optimiser le temps dans lequel l'algorithme aboutisse, en préférant certaines valeurs à d'autres.

Le choix de la valeur à affecter suit une logique de propagation, dans le sens où, chaque valeur candidate devra estimer les chances des variables non encore instanciées à trouver une valeur (à partir de leurs domaines respectifs) qui ne crée pas de conflit avec elle.

Cette techniques évite les cas de conflits et n'autorise une instanciation que lorsque celle ci ne pénalise aucune variable non encore instanciée.

Les coûts des calculs en terme de temps de ce genre de propagation étant assez conséquent, nous avons opté pour un choix de valeur aléatoire au moment de l'instanciation (*cf. Chapitre V*).

Chapitre V :

Implémentation

V - 1 Introduction

Le noyau final se présentera sous forme de librairies de classes capables de modéliser et de résoudre des CSP discrets à contraintes binaires. L'utilisation de cette librairie devra suivre les étapes suivantes :

Création d'un objet « Manager du Problème »

Déclaration des variables et de leurs domaines respectifs

Déclaration des contraintes

Résolution du problème

Le « Manager » encapsule toutes les données qui se rattachent au problème, il est donc possible de manipuler plusieurs problèmes, en parallèle d'une façon totalement indépendante.

Un exemple d'une exécution pourra être le suivant :

```
#include <PPc_Lib.h>
        int main(int argc, char ** argv) {
          PPc_Manager  m;
          PPc_IntVar  x(m,‘ var1 ’,8,25);
          PPc_IntVar  y(m,‘ var2 ’,2,14);
          PPc_IntVar  z(m,‘ var3 ’,1,20);

          PPc_BinIntCons C1=x<y ;
          PPc_BinIntCons C2=x<=z ;
          PPc_BinIntCons C3=y+2 !=z ;

          m.add(C1, C2, C3);
          if (m.Solve() )
                    m.Solution();
          return 0 ;
        }
```

V-2 <u>Architecture du noyau :</u>

V-2.1 <u>Le modèle Objet</u> :

Le noyau Open CSP se base sur une structure modulaire qui permet une grande liberté dans la gestion des données et dans leurs réutilisabilités .

On s’est basé sur le schéma conceptuel de données UML ci-dessous pour implémenter notre noyau :

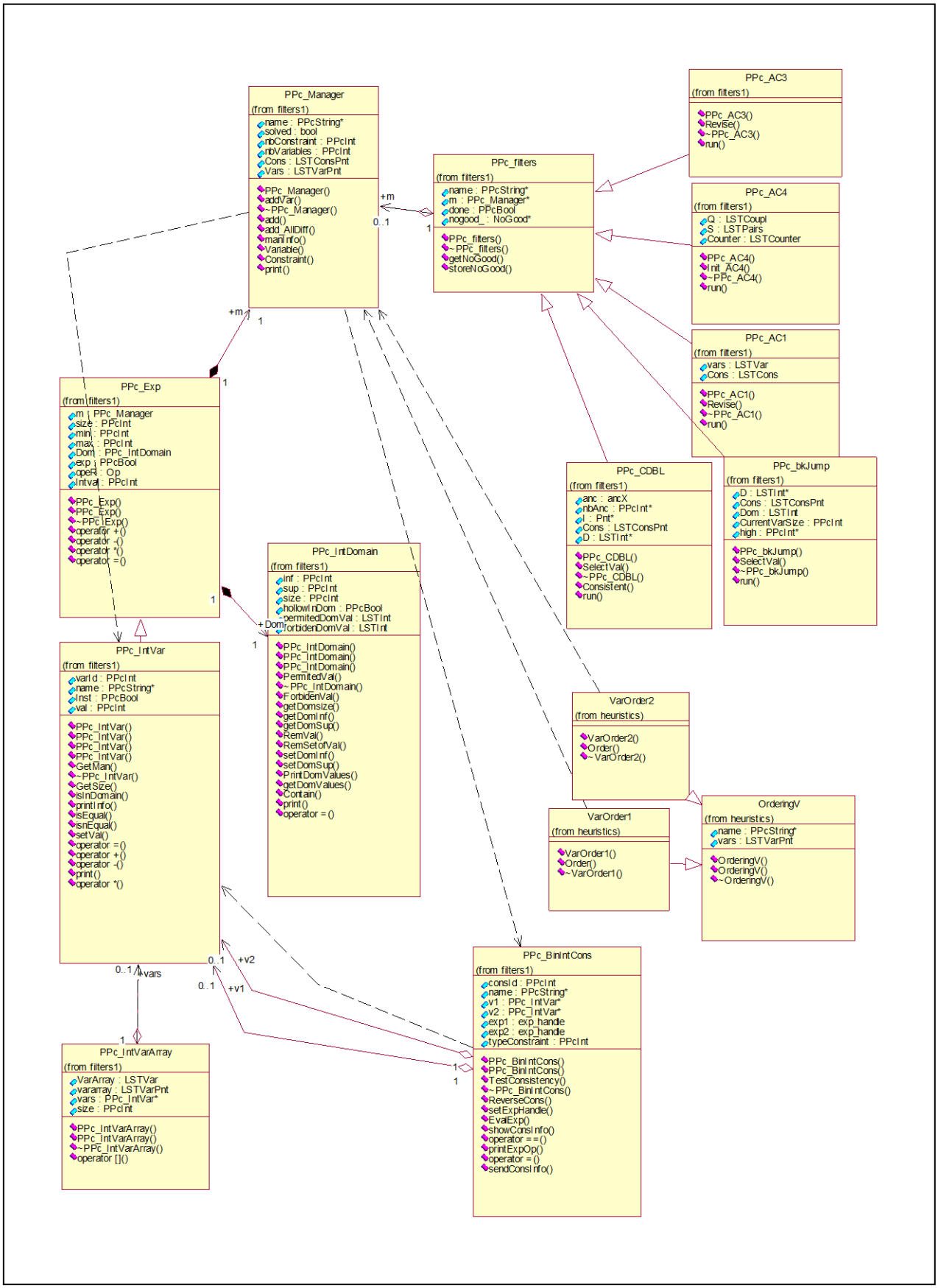


Figure 12 - Modèle objet du noyau Open CSP -

V-2.2 **Descriptif des classes :**

Open CSP est présenté sous forme de librairie capable de modéliser et de résoudre des CSP binaires discrets, avec possibilité de retracer la résolution pour expliquer certaines décisions de l'algorithme.

Le noyau *d'Open CSP* se devait d'être réutilisable et extensible. Il est fait de sorte qu'il soit modulaire, où toute les parties du noyau sont représentées par des classes qui définissent entièrement les objets à manipuler, tout en restant générique et ouvert pour toute extension éventuelle (faire en sorte que le noyau prennent en compte d'autres types de données, d'autres types de contraintes …etc).

Ce qui suit est le détail des principales classes qui représentent le noyau :

Classe PPc_Manager :

Utilisée pour modéliser les CSP. Toutes les données d'un problème sont encapsulées dans une instance de cette classe.

Un Manager contiendra des pointeurs vers les variables et les contraintes qui constituent le problème auquel il est associé :

Variable : est un vecteur de pointeur sur les différentes variable du problème en cours.

Constraint : est le vecteur des Contraintes.

Des flags de contrôle serviront d'indicateurs si le problème est résolu ou pas.

La méthode *add(PPc_BinIntCons)* est appelée pour inclure une nouvelle contrainte dans le problème.

La méthode *Solve()* est la principales méthodes de résolution. Elle est appelée à partir de la classe Manager.

Classe PPc_Var :

Classe abstraite qui servira de base de dérivation pour tous les types de variables à manipuler.

Classe PPc_Exp :

Classe dérivée de *PPc_Var* . *PPc_Exp* sert à gérer les expressions arithmétiques. Les opérateurs qui serviront à définir les expressions sont surchargés au niveau de cette classe.

Sont implémentés donc, les opérateurs : +, - , * , / .

Classe PPc_IntVar :

Classe dérivée de *PPc_Exp*. Les instances de cette classe sont les seules à pouvoir être manipulées par l'utilisateur.

Open CSP se limite dans un premier temps aux variables entières et les méthodes de la classe *PPc_IntVar* sont donc prévues pour des valeurs de ce type (entières). Pour des extensions éventuelles, de nouvelles dérivations à partir de *PPc_Var* sont à prévoir, ainsi que la redéfinition des méthodes de cette classe.

Chaque *PPc_IntVar* est rattachée à un domaine et toute modification de celui ci entraîne un processus de propagation de contraintes.

Classe PPc_IntVarArray :

Reprend les attributs de la classe *PPc_IntVar* en prévoyant un « vecteur de variables ». L'utilisation des instances de cette classe est très utile lorsque les variables du problème sont quasiment identiques (toutes les variables possèdent le même domaine). L'utilisation de *PPc_IntVarArray* permet une définition aisée des variables dans ces cas là.

Classe PPc_Domain :

Classe associée à chaque instance de *PPc_IntVar*. Une instance de *PPc_Domain* contient toutes les informations nécessaires pour retrouver les valeurs du domaine d'une variable, sans que ces valeurs ne soient entièrement stockées en mémoire.

Le domaine est repéré par ses bornes *sup* et *inf* dans le cas où il est contiguë. Si le domaine contient des vides , une variable booléenne *HollowInDom* , initialement à faux prendra la valeur vrai, et un des deux vecteurs *forbidenDomVal* ou *permitedDomVal* se chargera de repérer ou bien les valeurs à supprimer , ou bien alors les valeurs à retenir, selon le nombre de ces valeurs, on choisira de charger l'un ou l'autre de ces deux vecteurs en optant pour celui qui contiendra le moins de valeurs.

Différentes méthodes pour la propagation de contraintes sont incluses dans cette classe.

Classe PPc_BinIntCons :

Classe qui servira à modéliser les contraintes *'binaires-entières'*, les contraintes peuvent s'écrire de façon intuitive grâce à la surcharge des opérateurs binaires nécessaires.

La principale méthode de cette classe est *TestConsistency(),* qui vérifie la consistance de l'instance de *PPc_binIntCons* appelante.

Classe PPc_filters :

La classe *PPc_filters* sert de base de dérivation pour toutes les méthodes utilisées pour résoudre les CSP. On y trouve les principales procédure de propagation (*Propagate(), Consistent()*), ainsi que les appels aux sous-routines de gestion de données telle que (*getNoGood(), storeNoGood()*) qui sont essentiellement utilisées pour retracer le parcours du processus de résolution pour le débogage en cas d'échec.

Classes PPc_AC1, PPc_AC3 et PPc_AC4 :

Dérivées de *PPc_filters*, dans ces classes sont implémentés les procédures de consistance d'arcs d'après les différentes versions d'algorithmes *AC1, AC3* et *AC4*. L'implémentation de ces méthodes a volontairement été séparé pour marquer leur indépendance, en prévoyant une classe mère commune (la classe *PPc_filters*) pour toute dérivation ultérieure.

Classes PPc_bkJump, PPc_CDBL :

Dérivées de *PPc_filters*, dans ces classes sont implémentés les deux algorithmes de résolution utilisé dans Open CSP, à savoir , le *Backjump* et le *Conflict Directed Backjumping Learning*.

Open CSP offre la possibilité d'être surchargé par d'autres méthodes de résolution. Celles ci doivent être dérivé de *PPc_filters* et obéir aux mêmes règles que *PPc_bkJump* et *PPc_CDBL*. Un pointeur sur le Manager en cours est fourni en entrée. A partir de ce dernier on peut avoir accès à toutes les données du problème, qu'on utilise comme entrées pour l'algorithme de résolution. En sortie, l'algorithme doit retourner la solution obtenue ou alors, une information pour dire que le problème est sans solution. La gestion des *Nogoods* et des *Conflict Set* doit être faite de la même manière que dans *PPc_bkJump* et *PPc_CDBL*.

Classe PPc_OrderingV :

Cette classe sert de base de dérivation pour les principales heuristiques d'ordonnancement de variables utilisées. De cette classe découle les deux classes *PPc_VarOrder1* et *PPc_VarOrder2*. Dans la première classe dérivée on a implémenté l'heuristique d'ordonnancement de variables par degré de contraignace. Dans la seconde classe dérivée se trouve l'heuristique d'ordonnancement de variables par nombre de choix de valeurs (cardinalité des domaines).

Classe PPc_exemple :

Classe abstraite à partir de laquelle on a dérivé un certain nombre d'exemples prêt à être utilisés et avec lesquels on a validé le module de résolution d'Open CSP.

Parmi les exemples on retrouve le problèmes des N-reines (classe *PPc_nqueen*), des problèmes de coloration de graphe (classes *PPc_color* et *PPc_colouring*) et des problèmes sans solutions (classe *PPc_unsolvable*) pour tester le module de traçabilité.

Les classes dérivées de *PPc_exemple* peuvent servir de modèle pour implémenter tout type de problèmes qu'Open CSP peut supporter.

V-3 Résolution des CSP :

Pour résoudre un CSP en utilisant la librairie *Open CSP* on doit respecter les étapes suivantes :

a/ définir le problème en utilisant les structures de données qu'offre la librairie, cela revient à définir les variables du problème ainsi que les contraintes qui les lient et faire en sorte que toutes les données soient encapsulées dans un même objet *Manager* pour désigner un seul et unique problème. Il est possible de manipuler plusieurs problèmes à la fois en déclarant plusieurs *Manager*.

b/ Passer par une phase de propagation de contraintes où un algorithme de consistance d'arcs est exécuté sur le problème en cours. Ceci aura pour effet de réduire la taille des domaines des variables en détectant et en supprimant les valeurs inconsistantes. *Open CSP* offre différents algorithmes de consistance d'arcs. Il est aussi possible d'implémenter d'autres algorithmes de consistance en faisant des dérivations à partir de la classe *PPc_filters*.

c/ Procéder à la résolution du problème. Les deux algorithmes de résolutions prévus dans Open CSP sont des backtrack intelligents. Dépendant de la nature du problème, on peut aussi implémenter d'autres algorithmes de résolution comme par exemple des algorithmes à test-avant (forward checking).

V-4 Gestion de la mémoire :

La représentation des données dans *Open CSP* est dynamique. Les différentes structure de données prévues pour les CSP sont allouées à partir des classes qui les contiennent. Les désallocation se fait au niveau des destructeurs à la sortie du programme pour l'ensemble des objets manipulés.

Il appartient à l'utilisateur de libérer correctement tout espace supplémentaire qu'il aura à ajouter lors de son utilisation.

Le compromis entre l'espace mémoire et le temps d'exécution a bien évidemment été posé. Notre but étant de pouvoir tracer les situations d'échecs, on se devait de stocker un grand nombre d'informations supplémentaire en mémoire. Nous avons fait le choix d'alléger la mémoire en reportant les informations nécessaires à l'explication des échecs dans les fichiers 'rapport' en temps réel. Les temps d'exécution reste cependant assez moyen, ceci est la conséquence directe d'une lourde série de test et de vérification supplémentaires imposée par le module de traçabilité.

V-5 Environnement de développement :

Open CSP est une librairie réalisée en C++. Un compilateur C++ est donc nécessaire pour toute utilisation. Nous avons essayé de tirer profit des performances de ce langage pour manipuler certaines structures de données complexes (listes de classes, … etc).

Les librairies fournies avec ce compilateur contiennent un grand nombre d'outils qui ont aidé à implémenter certaines tâches assez compliquées. On s'est aussi basé sur la modélisation Objet , qui a structuré notre démarche lors de la programmation. Le but étant d'aboutir à un librairie ouverte à toute extension.

Chapitre VI :

Résultats

VI-1 **Traçabilité et explications** :

Dès la fin d'un processus de résolution, 3 fichiers rapports sont générés automatiquement : solution.ppc, out.ppc et xplain.ppc.

Solution.ppc :

Dans ce fichier on peut retrouver l'état des variables telles qu'elles étaient à la fin du processus de résolution. Un rappel des informations de chaque variable y est reporté ainsi que l'instanciation finale de l'ensemble des variables du problème (dans le cas où le problème a au moins une solution !).

Out.ppc :

Reprend l'état du Manager à la fin de la résolution. Toutes les données initiales du problème sont disponibles dans ce fichier : contraintes, variables, domaines …

Xplain.ppc :

Un nogood peut être considéré comme une instanciation partielle qui ne mène jamais à une solution. Pour comprendre ce qui empêche une telle instanciation d'aboutir à une solution, on peut l'associer à certaines contraintes du problème. Ces contraintes sont celles qui lient la variable responsable d'un retour arrière aux autres variables du problème, déjà instanciées. Connaître ces contraintes nous permettra de comprendre l'impossibilité d'instancier la variable en cours et donc saisir les causes du backtrack.

Définition :

On appellera *Contraintes de choix* (*choice constraints*) les contraintes du type affectation : $v_i=a_i$ que génère l'algorithme à chaque instanciation.

Dans certains solvers et plus précisément, les solvers à base d'explications (*explanation based constraints programing*) on inclut la négation de ce type de contraintes ($v_i \neq a_i$) dans le problème au courant de la résolution pour empêcher l'algorithme d'affecter les même valeurs à ces variables impliquées dans un nogood. Ces contraintes sont ensuite propagées pour conditionner le comportement futur de la résolution.

Un nogood peut être écrit sous la forme :

$$C \vdash\!\!\!- \neg(C' \wedge a_1 = v_1 \wedge a_2 = v_2 \wedge ... \wedge a_k = v_k)$$

Où C' $\subset$ C est un sous ensemble des contraintes initiales et les $v_i=a_i$ $i=1,...k$ sont des contraintes de choix incluses dans le système de contraintes initial lors du processus de recherche de solutions.

Si la réécriture d'un nogood (contradiction) ne contient aucune contrainte de choix, on peut en déduire une contradiction, le problème n'a donc pas de solution. ($C \vdash\!\!\!- (\neg C')$, l'ensemble des contraintes *C* ne peut pas être satisfait par la négation d'un sous-ensemble de lui même !)

Lors de la résolution, *Open CSP* reprend les nogoods les plus pertinents et les stock en mémoire dans une liste de paires du type <IdentifiantVariable, valeur>.

Ces nogoods par lesquels l'algorithme serait passé pour aboutir à une situation d'échec (dead-end) seront reportés dans le fichier *xplain.ppc* sous la forme suscitée.

Garder en mémoire tous les nogoods peut être prohibitif en terme d'espace mémoire. On se débarrassera donc au fur et à mesure des nogoods qui ne sont plus pertinents. Un nogood est dit pertinent, si toutes les instanciations qu'il contient sont encore valides.

Nous avons résolus un certain nombre de CSP[*] avec la librairie *Open CSP* d'une part, pour valider le module de résolution qui s'est montré capable de fournir la solution d'un problème (lorsque celle ci existe) et d'autre part, pour montrer sa capacité à fournir une explication aux situation d'échecs rencontrées. Il est clair que les problèmes les plus « intéressants » pour notre étude sont ceux pour lesquels *Open CSP* est destiné , c'est à dire les problèmes insolubles. La majorité des solvers actuels arrivent à savoir si le CSP à résoudre n'a pas de solution, par contre très peu fournissent les raisons d'un tel échec qui peut très bien être provoquer par un bug dans le solver lui même, une mauvaise modélisation du problème ou tout simplement, parce que le problème est insoluble [Narendra2001].

En gardant la trace des nogoods rencontrés dans des structures de données adéquates, on peut comprendre la comportement de l'algorithme et interpréter certaines décisions prises et ainsi, retracer l'évolution du processus de recherche.

Ci-dessous, des exemples pour mieux comprendre les résultats d'*Open CSP* :

[*] Ces CSP sont implémentés dans le dossier Exemples de la librairie.

VI-2 Exemples de CSP résolu avec Open CSP :

VI-2.1 Open CSP et le problème des N-reines :

Un CSP connu est celui des N-reines. Soit un échiquier de taille n, on dispose de n reines qu'on veut placer dans cet échiquier sans qu'aucune d'entre elles ne puisse attaquer l'autre.

Ce problème sera modéliser comme suit :

Les variables représentent les n-reines $\{x_1, \ldots, x_n\}$, chaque reine dispose d'un domaine constitué des valeurs qu'elle peut prendre dans l'échiquier $D_i = (1, \ldots, n)$.

Les contraintes seront de la forme :

$$\left.\begin{array}{l} x_i \neq x_j \\ x_i - i \neq x_j - j \\ x_i + i \neq x_j + j \end{array}\right\} \forall i, j \in (1,\ldots,n), i \neq j$$

Pour n=6, *Open CSP* fourni le résultat suivant :

Fichier Solution.ppc :

```
MANAGER INFO :
Manager : N-Queen Manager
Number of Variables ................. 6
Number of Constraints ............... 45
Solved in 0.852 seconds.............
Variable Info :
      name: vararray1   referenced in Manager as var1
      DomMin: 1   DomMax: 6
Instanciated to ..................... 5
Variable Info :
      name: vararray2   referenced in Manager as var2
      DomMin: 1   DomMax: 6
Instanciated to ..................... 3
Variable Info :
      name: vararray3   referenced in Manager as var3
      DomMin: 1   DomMax: 6
Instanciated to ..................... 1
Variable Info :
      name: vararray4   referenced in Manager as var4
      DomMin: 1   DomMax: 6
Instanciated to ..................... 6
Variable Info :
      name: vararray5   referenced in Manager as var5
      DomMin: 1   DomMax: 6
Instanciated to ..................... 4
Variable Info :
      name: vararray6   referenced in Manager as var6
      DomMin: 1   DomMax: 6
Instanciated to ..................... 2
```

Fichier xplain.ppc :

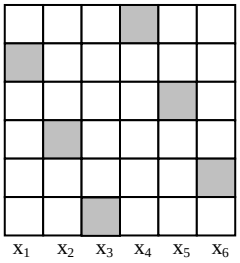
```
Variables Ordering....
variable vararray1 (id:var1) , variable vararray2 (id:var2) ,
variable vararray3 (id:var3) , variable vararray4 (id:var4) ,
variable vararray5 (id:var5) , variable vararray6 (id:var6)

Resulotion performs a back Jump 2 times......
Number of cheked nodes : 9 nodes......

BackJump occured from variable vararray6 (Id:var6) to variable
vararray5 (Id:var5).........
      Conflict when :
            variable vararray1(Id:var1)=5 ,
            variable vararray2(Id:var2)=3 ,
            variable vararray3(Id:var3)=6 ,
            variable vararray4(Id:var4)=4 ,
            variable vararray5(Id:var5)=2 ,
            AND ...
            Constraint 5:var 1 != var 6
            Constraint 12:var 3 != var 6
            Constraint 14:var 4 != var 6
            Constraint 9:var 2 != var 6
            Constraint 29:var 4+4 != var 6+6
            Constraint 30:var 5+5 != var 6+6

BackJump occured from variable vararray5 (Id:var5) to variable
vararray3 (Id:var3).........
      Conflict when :
            variable vararray1(Id:var1)=5 ,
            variable vararray2(Id:var2)=3 ,
            variable vararray3(Id:var3)=6 ,
            AND ...
            Constraint 4:var 1 != var 5
            Constraint 8:var 2 != var 5
            Constraint 19:var 1+1 != var 5+5
            Constraint 38:var 2-2 != var 5-5
            Constraint 26:var 3+3 != var 5+5
```

Le fichier Solution affiche la solution obtenue. La disposition des reines dans ce cas est illustré dans l'échiquier ci-dessous :



- Positionnement des 6-reines -

L'algorithme fournit une solution dans ce cas là. Le fichier xplain.ppc expose quand même les parcours de l'algorithme et nous informe sur les backtrack effectués.

Par exemple, on lit que l'algorithme à effectué deux retour arrières et qu'il a parcourus 9 sommets de l'arbre des solutions. L'ordre des variables dans l'algorithme est détaillé en premier dans le fichier *xplain.ppc*. Dans cet exemple, toutes les variables sont équivalentes, l'ordre reste alors, inchangé.

Le premier backjump enregistré s'est produit lorsqu'un conflit est survenu en instanciant la variable x_6. Pour l'instanciation partielle $\xrightarrow[a_6]{}$= (x_1=5, x_2=3, x_3=6, x_4=4, x_5=2) (*fig-13.a*)aucune valeur consistante de D_6 n'a été trouvé. $\xrightarrow[a_6]{}$ constitue alors un ensemble en conflit avec x_6. Le noggod peut donc être pris en considération. Xplain.ppc nous informe aussi des contraintes qui pénalisent cette variable.

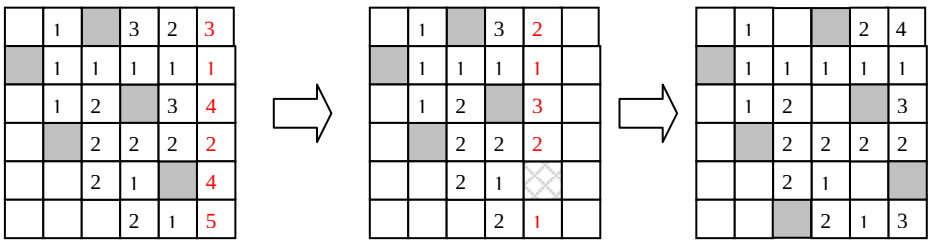


Figure 13 – Les nogood rencontrés dans le problème des 6-reines -

On remarque alors dans le premier nogood que pour l'instanciation $\xrightarrow[a_6]{}$ on a :

La contrainte (`var 1 != var 6)` la valeur 5 de D_6 n'est plus valide,

La contrainte (`var 2 != var 6)` la valeur 3 de D_6 n'est plus valide,

La contrainte (`var 3 != var 6)` la valeur 6 de D_6 n'est plus valide,

La contrainte (`var 4 != var 6)` la valeur 4 de D_6 n'est plus valide,

La contrainte (`var4+4 != var6+6)` la valeur 2 de D_6 n'est plus valide,

La contrainte (`var5+5 != var6+6)` la valeur 1 de D_6 n'est plus valide,

Le backtrack est effectué alors vers le plus proche ancêtre récemment instanciée, qui est donc x_5. Il se trouve que cette variable n'a elle aussi plus aucune valeur consistante avec l'instanciation partielle $\xrightarrow[a_5]{}$. De même, le backtrack est effectué vers l'ancêtre le plus récemment instancié. D'après le schéma, on constate que c'est la variable x_3. Les raisons sont aussi explicitées dans xplain.cpp en formulant le nogood comme ça a été décrit plus haut.

Dès lors, le parcours arborescent qu'a effectué l'algorithme peut être retracé et on retrouve les chemin suivant :

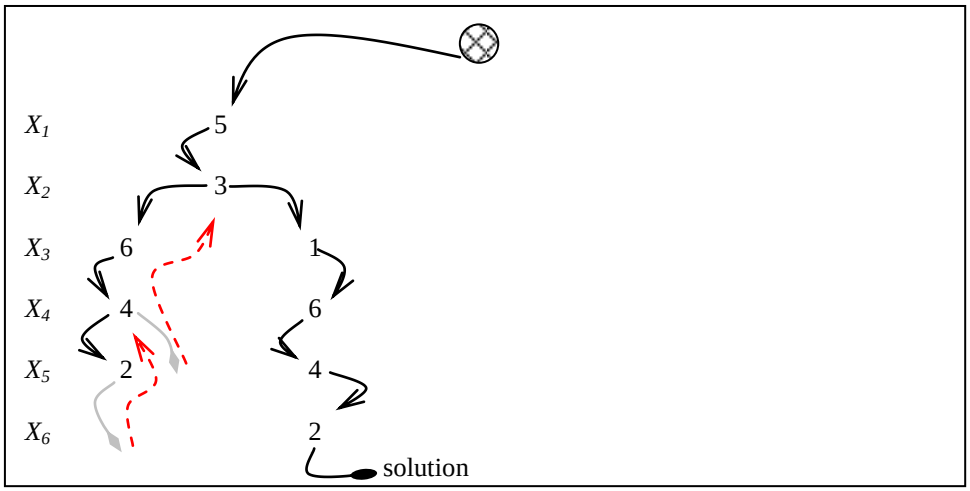


Figure 14 - Évolution de l'algorithme dans l'arbre des solutions. -
Les arcs décrivent les instanciations successives. Les feuilles en gris sont les situations d'échecs. La feuille en noir est la solution. Les arcs en pointillés désignes les backtrack.

VI-2.2 Open CSP : un problème sans solution :

Un deuxième exemple est un problème sans solution, il est représenté par son graphe de contraintes :

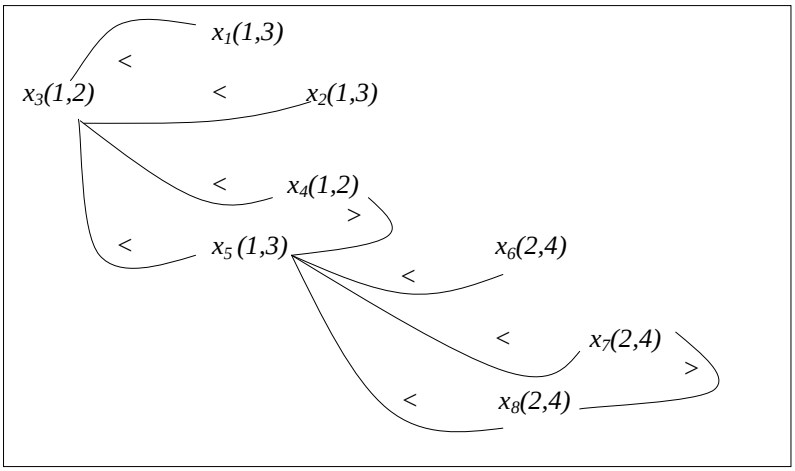


Figure 15 - Graphe de contraintes d'un problème sans solution -

Détail du fichier ***xplain.ppc*** après la résolution :

```
Variables Ordering....
 variable x5 (id:var4) , variable x3 (id:var1) , variable x4
 (id:var5) , variable x7 (id:var7) , variable x8 (id:var8) ,
 variable x6 (id:var6) , variable x1 (id:var2) , variable x2
 (id:var3)

Resulotion performs a back Jump 5 times......
Number of cheked nodes : 8 nodes......

 BackJump occured from variable x4 (Id:var5) to variable x5
 (Id:var4).........
 Conflict when :
   variable x3(Id:var1)=1 ,
   variable x5(Id:var4)=2 ,
   AND ...
   Constraint 5:var 5 < var 4
   Constraint 4:var 1 < var 5

 BackJump occured from variable x4 (Id:var5) to variable x3
 (Id:var1).........
 Conflict when :
   variable x3(Id:var1)=2 ,
   AND ...
   Constraint 4:var 1 < var 5


 BackJump occured from variable x8 (Id:var8) to variable x7
 (Id:var7).........
 Conflict when :
   variable x5(Id:var4)=3 ,
   variable x7(Id:var7)=4 ,
   AND ...
   Constraint 8:var 4 < var 8
   Constraint 9:var 7 < var 8

 BackJump occured from variable x7 (Id:var7) to variable x5
 (Id:var4).........
 Conflict when :
   variable x5(Id:var4)=3 ,
   AND ...
   Constraint 7:var 4 < var 7

 BackJump occured from variable x3 (Id:var1) to variable x5
 (Id:var4).........
 Conflict when :
   variable x5(Id:var4)=1 ,
   AND ...
   Constraint 3:var 1 < var 4
```

Lorsque l'algorithme n'aboutit pas à une solution, un fichier supplémentaire NoGood.ppc est généré. Dans ce fichier on retrouve l'historique des instanciations effectuées au moment des backtracks consécutifs. Le contenu de ce fichier est listé ci-dessous :

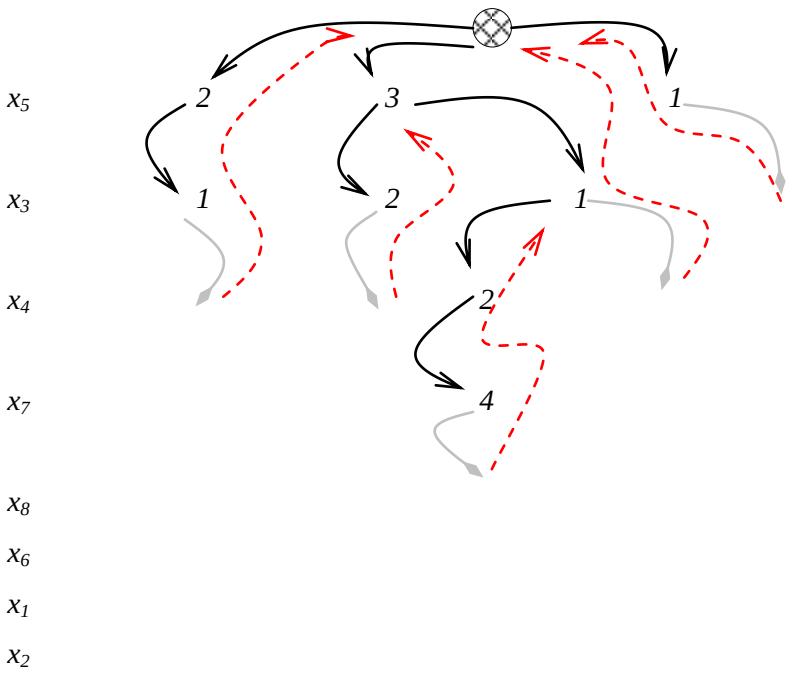
```
 MANAGER INFO :
 Manager name : Example Manager
 Number of Variables ................. 8
```

```
Number of Constraints ..............  9

Resulotion perform a back Jump 5 times......

Nogoods nb 1
 Stored when a jumpback occured from var5  named x4
  var 4 valuated to 2
  var 1 valuated to 1
Nogoods nb 2
 Stored when a jumpback occured from var5  named x4
  var 4 valuated to 3
  var 1 valuated to 2
Nogoods nb 3
 Stored when a jumpback occured from var8  named x8
  var 4 valuated to 3
  var 1 valuated to 1
  var 5 valuated to 2
Nogoods nb 4
 Stored when a jumpback occured from var7  named x7
  var 4 valuated to 3
  var 1 valuated to 1
  var 5 valuated to 2
Nogoods nb 5
 Stored when a jumpback occured from var1  named x3
```

En se basant sur les informations fournies par ces deux fichiers, on peut aisément dans un premier temps, retracer le parcours de l'algorithme dans l'arbre des solutions ;



Dans un second temps, on pourra interpréter ces résultats pour comprendre l'échec de la résolution. On voit dans ce parcours que toutes les valeurs du domaine D_5 mènent à un échec. La variable x_5 est fortement contrainte (elle est liée à 5 contraintes parmi les 9 que compte le problème). À partir du graphe de contraintes initial et les informations données par les fichiers rapport, on remarque certaines inconsistances qui feront que le problème n'aura jamais de solution.

Pour pouvoir interpréter ces résultats on s'intéressera d'abords au problème initial. D'après le graphe de contraintes, le système de contraintes se résume à l'inégalité suivante :

$$x_3<x_4<x_5<x_7<x_8\,. \qquad \textbf{(1)}$$

En d'autres termes, pour un domaine de valeur contiguë, x3 devra avoir la plus petite valeur et x_8 la plus grande. Or, les domaines dans ce problème sont de '1' à '4'. Il est donc impossible de trouver cinq valeurs ordonnées à affecter à l'ensemble des variables.

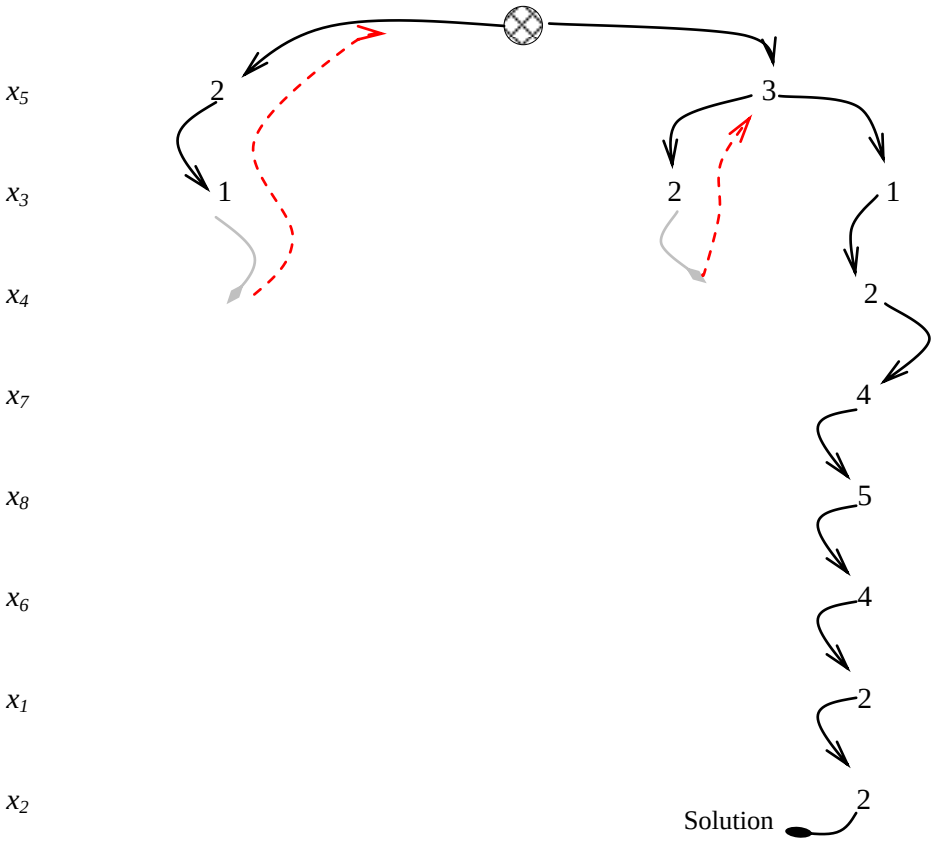
Les nogoods dans le fichier *xplain.ppc* s'explique à partir de là :

Le premier nogood :

De x_4 vers x_5 : x_5 vaut 2 et x_3 vaut 1. On sait maintenant que la valeur de x_4 doit être comprise entre celles de x_3 et de x_5. Or dans ce cas, les valeurs de ces deux variables se succèdent, de plus x_5=2 élimine la valeur 2 du domaine D_4 et x_3=1 fait de même pour la valeur 1. D_4 étant vide, il y a donc impossibilité d'instancier x_4 , un backtrack est effectué vers l'ancêtre le plus proche (dans ce cas x_5).

Le deuxième nogood peut s'expliquer de la même manière que le premier sauf que dans ce cas, la variable x_5 ne fait plus partie de l'ensemble en conflit avec la variable x_4 (car x_5 vaut 3 et cette valeur n'influence pas le domaine de x_4 !) et le retour arrière se fait vers x_3.

Le troisième nogood est plus intéressant à voir car dans ce cas, l'inégalité (***1***) résumant les contraintes du problème est en partie respectée sauf pour la variable x_8 où, il est impossible de lui trouver une valeur à partir de son domaine. L'affectation x_5=3 élimine à elle seule les deux valeurs '2' et '3' du domaine D8. Lorsque , en plus x_7=4, le domaine de x_8 se vide et le retour arrière devient inévitable. On pourrait prévoir par exemple de relâcher la contrainte entre x_7 et x_8 ($x_7 \neq x_8$) ou bien alors, d'étendre le domaine D_8 (D_8=(2,5)). Ces deux possibilités constitueraient des alternatives si l'on voulait obtenir une solution. Dans la deuxième possibilité, l'algorithme aboutira à une solution après le parcours suivant :



- *Figure 16 - Arborescence des solutions après extension de D_8 à (2,5)* –

VI-2.3 Conclusion :

Il est clair que les explications fournies par *Open CSP* restent indicatives. Néanmoins, on réussi quand même à interpréter la source d'échec d'une résolution à partir des informations fournies par les fichiers *xplain.ppc* et *NoGood.ppc*.

Un raisonnement semblable à celui utilisé dans les solvers à base d'explications serait parfaitement envisageable dans *Open CSP*. Une amélioration dans laquelle la résolution prendrait en compte les contraintes induites par les nogood rencontrés (contraintes de choix) serait possible. Le choix même de l'algorithme de résolution, le *Conflict Directed Backjumping Learning* est motivé par le fait que cette technique prévoit de suivre la trace des conflits tout au long de la résolution, d'effectuer un apprentissage sur la base de ce qui c'est déjà fait (instanciations

précedentes, nogoods, …) et de prendre les décisions adéquates en fonctions de tout ces paramètres. Notre but étant de prévoir des explications aux situations d'échec, cette méthode s'est donc avéré, très appropriée.

Chapitre VII :

Conclusion

Divers problèmes relevant de domaines aussi variés que l'industrie ou les télécommunications sont modélisés sous forme de CSP.

Les enjeux qui leurs sont liés deviennent de plus en plus importants. Il est donc devenu nécessaire de savoir les résoudre de la meilleure façon qu'il soit.

Plusieurs solvers existent pour effectuer ce genre de tâche. Cependant, lorsque ces outils 'échouent', on se retrouve sans aucune indication pouvant nous informer sur les causes de l'échec.

Open CSP se place donc dans cette optique, comme un outils conçu pour modéliser, résoudre et déboguer les problèmes discrets de satisfaction de contraintes binaires.

Notre but étant d'éviter d'avoir une n-ième librairie capable de résoudre les CSP, nous nous sommes plus penchés sur l'explication d'échec dans les résolutions de tels problèmes.

On s'est inspiré du travail fait dans l'outil de résolution d'Ilog Ilog Solver ainsi que du travail fait dans CHOCO, une librairie développer par F. Laburthe pour Bouygues Télécoms.

Notre module de résolution utilise deux algorithmes connus qui sont le backjump de Gaschnig et le Conflict Directed Backjumping Learning (CDBL). Ce deuxième algorithme répondait mieux à nos attentes car, notre objectif était de pouvoir retracer le processus de résolution en cas d'échec et d'essayer de comprendre les causes de ce dernier. Dans le CDBL est prévu une gestion particulière des nogood rencontrés, ce qui nous a permis de nous rapprocher de notre objectif.

Open CSP reste cependant un outil ouvert et extensible grâce à sa structure modulaire. Voilà pourquoi, la faire évoluer encore plus est une des perspectives les plus intéressantes. Inclure de nouveaux types de données (réels, énumérés, …) et de nouveaux types de contraintes (contraintes

booléennes, possibilité de définir ses propres contraintes….) est plus que souhaitable car ceci permettra de voir le comportement du module de traçabilité face à de nouveaux types de données.

Références bibliographiques :

[Bak95] Baker A., *Intelligent backtracking on constraints satisfaction problems : experimental and theorical results*, Department of Computer and Informations Science University of Oregon, 1995.

[Bar98] Bartak R. , Guide to Constraint Programming, http://kti.ms.mff.cuni.cz/~bartak/contraints, 1998.

[Bes94] Bessière C., A fast Algorithm to Establish Arc-consistency in Constraint Networks, LIRMM, University of Montpellier II, 1994.

[BM96] Bayardo R., Miranker D.P., *A Complexity Analysis of Space-bounded Learning Algorithms for the Constraint Satisfaction Problem*, AAAI 96 :298-304, 1996.

[Coo89] Cooper M. C., *An Optimal k-Consistency Algorithm*. Artificial Intelligence 41 :89-95, 1989.

[DV90] Deville Y., Van Hentenryck P., *Efficient Arc Consistency Algorithm for a Class of CSP Problems*. Department of Computer Science Brown University. Technical Report No. CS-90-36, 1990.

[FD99] Dechter R., Frost D. *Backtracking algorithms for constraint satisfaction problems*, Dept. of Information and Computer Science, University of California, Irvine, 1999 .

[Fro97] Frost D.H. *Algorithms and Heuristics for Constraint Satisfaction Problems*, University of California, Irvine, 1997.

Ilog optimization suite White Paper www.ilog.com

[Kam97] Kambhampati S., *On the Relation between Intelligent Backtracking and Failure-driven Explanation Based Learning in Constraint Satisfaction planning*, Department of Computer Science and Engineering Arizona State University, technical repport 97-018, 1997.

[Kon94] Kondrak G. *A.Theoretical Evaluation of Selected Backtracking Algorithms*, Dpt. of Computing Science, Thesis, University of Alberta, 1994.

[KV97] Kondrak G., Van Beek P, *A theorical Evaluation of Selected Backtracking Algorithms*, AI Magazine, 89 :365-387, 1997.

[Kum92] Kumar V. , *Algorithms for Constraint Satisfaction Problems : A Survey*, AI Magazine 13(1) : 32-44, 1992.

[Lab00] Laburthe F., OCRE Team, Bouygues e-Lab, 2000.

[MacW85] Mackworth A. K. and Freuder E. , The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. Artificial Intelligence 25 :65-74, 1985.

[Nar01] Narendra J., The PaLM system : explaination-based constraint programming, Écoles des Mines de Nantes, JNPC 2001, 2001.

ANNEXE

***OPEN CSP* manuel d'utilisation :**

1- Détail des classes et modèle objet :

cf. Chapitre V : implémentation.

2 – Problème de satisfaction de contraintes avec Open CSP :

Open CSP est une librairie conçue pour résoudre les CSP binaires entiers avec possibilité de tracer la résolution en cas d'échec pour des besoins de débogage.

Une fois le CSP modélisé, on procède à sa déclaration :

Le Manager :

Une instance de la classe PPc_Manager doit être déclarée au tout début de chaque programme. Cette objet devra encapsuler toutes les données d'un même problème. La déclaration se fait comme suit :

PPc_Manager nom_du_manager ;

Un manager déclaré tout seul ne sert à rien. Les variables et les contraintes du problème doivent être déclaré puis liés à ce manager.

Le manager peut ensuite lancer la résolution en appelant la méthode *solve()* à partir de laquelle se feront quasiment tout les appels des méthodes de résolution.

Les variables :

Les variables dans Open CSP sont des instances de la classe PPc_IntVar. Cette classe prend en charge les variables entières uniquement.

La déclaration peut se faire suivant les deux constructeurs prévus pour cette classe :

PPc_IntVar variableX (nom_manager, nom_variable,borne_min_du_domaine,
borne_max_du_domaine) ;

La déclaration prend dans ce cas quatre paramètres :

Le nom du manager du problème auquel la variable est associée.

Le nom de la variable qui servira de premier identifiant

Les deux bornes du domaine de la variable.

Il est aussi possible de déclarer les variables en ne précisant que le nom du manager et un nom à la variable :

PPc_IntVar variableX (nom_manager, nom_variable) ;

Dans ce cas, Open CSP affecte un domaine par défaut à la variable qui est (-100,+100) .

Il est également possible de déclarer des variables temporaires où, il n'est plus nécessaire de préciser ni le manager, ni le nom ni les bornes du domaines. Ce sont des variables qui prendront par la suite les valeurs d'autres variables déjà existantes.

Tous les objets *PPc_IntVal* possèdent une donnée membre *val* qui se verra affecter la valeur entière en cas d'instanciation. Le champ booléen *Inst*, initialement à *false* prendra la valeur *true* dès que son instance se voit affecter une valeur de son domaine.

Les Domaines :

Les domaines des variables sont à priori inaccessibles à l'utilisateur. Leurs définitions se fait de façon induite lorsque les variables sont définies. Cependant, ceci n'est le cas que lorsque ces domaines sont contiguës. Dans le cas contraire, il est important de préciser au manager les valeurs à considérer et les valeurs à ignorer.

Les domaines sont gérés par la classe *PPc_Domaine*. Dès qu'une variable est déclarée, une instance de cette classe est crée et associé à cette variable. La classe *PPc_Domaine* possède deux champs *inf* et *sup* qui se verront affecté les valeurs passées en définition de variable *borne_inf_du_domaine* et *borne_sup_du_domaine* respectivement.

Dans l'exemple suivant, la variable *x* a le domaine suivant $d_x = (1, 2, 3, 4, 5, 8,10)$.

Sa déclaration se fera comme suit :

```
PPc_IntVar x(m,'varx', 1, 10) ;
PPcInt * tmpTab ;
tmpTab=new PPpcInt (4) ;
tmpTab[0]=3 ; tmpTab[1]=6 ; tmpTab[2]=7 ; tmpTab[3]=9 ;
x.Dom.forbidenVal(tmpTab) ;
```

Après avoir déclaré la variable et les bornes de son domaine, on procède à l'élimination des valeurs qui ne sont pas comprises dans ce dernier. Le vecteur *tmpTab* indiquera dans son élément d'indice *0* le nombre d'élément à supprimer (*tmpTab[0]=3*). On affecte ensuite ce vecteur comme paramètre à la méthode *forbidenVal()* qui a pour tâche de supprimer du domaine en cours , les valeurs contenues dans ce vecteur.

Supposons que le domaine de la variable *x* soit le suivant :

$$d_x=(1, 2, 10) ;$$

Dans ce cas, on aura à supprimer les six valeurs *(3, 4, 5, 6, 7, 8, 9)* du domaine de *x*. Il est plus judicieux d'indiquer au manager les valeurs à retenir uniquement, vu que leur nombre est moins important ! On utilisera alors la méthode *permitedVal()* au lieu de *forbidenVal()* :

```
PPc_IntVar x(m,'varx', 1, 10) ;
PPcInt * tmpTab ;
tmpTab=new PPpcInt (4) ;
tmpTab[0]=3 ; tmpTab[1]=1 ; tmpTab[2]=2 ; tmpTab[3]=10 ;
x.Dom.permitedVal(tmpTab) ;
```

L'appel de la méthode *forbidenVal()* sera suivi par la méthode *RemSetofVal(PPcInt *)* qui a pour tâche de supprimer du domaine en cours les valeurs contenues dans un vecteur d'entier passé en paramètre.

Toute instance de la classe *PPc_Domaine* possède une variable membre booléenne *hollowInDom* qui est initialisée à *false*. Dès qu'une des deux méthodes *forbidenVal(),* ou *permitedVal()* est appelée , cette variable passera à *true* pour indiquer que les valeurs du domaine ne sont pas contiguë. Les valeurs retenues (ou supprimées respectivement) seront chargées dans une liste *PermitedDomVal* (ou *forbidenDomVal*).

Les membres d'une instance de classe *PPc_Domaine* ne sont pas accessibles à l'utilisateur. Mis à part le fait d'indiquer les '*creux*' dans les domaines toute autre donnée membre devra être accédé par une méthode approprié :

GetDomSize() pour la taille du domaine,

GetDomInf() et getDomSup() pour les bornes du domaine,

GetDomValues() : retourne une liste d'entier dans laquelle seront contenues les valeurs du domaine.

Les vecteurs de variables :

Dans certains problèmes CSP, les variables du problème sont toutes identiques. Elles possèdent le même domaine et doivent être différenciées par un identifiant. Lorsqu'en plus le nombre de ces variables est important, il est conseillé de procéder à leur déclaration en passant par un vecteur de variables. La déclaration se fait comme suit :

PPc_IntVarArray vars(nom_manager, nombre_de_variables, borne_inf, borne_sup);

La création d'une instance de *PPc_IntVarArray* aura pour effet de créer *nombre_de_variables* variables, toutes associées au manager *nom_manager* et ayant le domaine (*borne_inf, …, borne_sup*). Les noms des variables leurs seront affectés automatiquement (*var1, var2, …, varN*).

Ceci évitera de retaper le même code pour déclarer des variables identiques.

Les contraintes :

L'écriture des contraintes dans *Open CSP* se fait de façon intuitive. Les contraintes sont des instances de la classe *PPc_BinIntCons*.

Open CSP étant prévu pour supporter les CSP binaires entiers, une redéfinition des opérateurs binaires était nécessaire. Le fichier *BinOp.h* contient cette surcharge effectué sur les opérateurs suivants :

<, >, >=, <=, *!*=, = =

Ces opérateurs serviront à définir la majorité des contraintes binaires. Une déclaration peut être la suivante :

PPc_Manager m ;
PPc_IntVar x (m, 'varX', 1, 15) ;
PPc_IntVar y (m, 'varY', 1, 12) ;
PPc_BinIntCons C = x >=y ;

Une contrainte *C* ainsi crée doit être ensuite rattaché au manager du problème. La classe *PPc_Manager* dispose d'une méthode *add()* destinée à cette tâche là. On écrira alors :

m.add (C) ;

L'appelle de cette méthode aura pour effet d'attribuer un identifiant à la contrainte (le champs *consId* de la classe *PPc_BinIntCons*) et donc de lier les variables *x, y* et la contraintes *C* au même manager. La contrainte *C* possède aussi deux pointeurs sur des objets *PPc_IntVar* grâce auxquels elle pourra identifier les deux variables qui la constituent.

La méthode add( ) aura pour effet aussi d'identifier les variables qui constituent les contraintes comme, appartenants au manager en cours. Les champs varId de la classe PPc_IntVar est prévu pour ça. A chaque fois qu'une nouvelle variable apparaît dans la définition d'une contrainte, celle ci reçoit un identifiant qui la différenciera des autres variables déjà définies. La manager l'inclut ensuite dans sa liste de variables *m.Vars*. Le même traitement est réservé aux contraintes qui seront listées dans *m.Cons*.

La classe *PPc_BinIntCons* possède également deux membres du types *exp_handle*. Dans le cas où une des variables de la contrainte est une expression (ex : *C= x+2= = y*) le champ *exp* de type *exp_handle* se verra attribuer la valeur de l'expression qui suit la variable *x*.

Exp_handle est une structure dont les champs sont les suivants :

Struct exp_handle { Booléen exp ;
Entier IntVal ;
Entier opeR ; }

Le champ *exp* indique que la donnée *exp_handle* est une expression ou pas selon qu'il est à *false* ou à *true*. Le champ *IntVal* porte la valeur entière qui constitue l'expression (dans le même exemple plus haut , *IntVal=2*). Quand au champ *opeR* il sera égale à l'identifiant qui représente l'opérateur (1 pour l'opérateur '+', 2 pour l'opérateur '-', 3 pour l'opérateur '*').

Les contraintes quand à elles sont identifiées par leurs types dans le champ '*typeConstraint*' (1 pour '<=', 2 pour '>=', 3 pour '<', 4 pour '>', 5 pour ' !=', 6 pour '= =').

La classe *PPc_BinIntCons* a une méthode qui lui sert à tester la consistance de ces instances pour deux valeurs données, passé en paramètres. L'appelle de cette méthode se fait comme suit :

PPc_BinIntCons C = x+2 != y ;

C.TestConsistency (PPcInt valX , PPcInt valY) ;

Cette méthode renvoie une valeur booléenne pour indiquer la consistance ou pas, de cette instance pour les deux valeurs (*valX, valY*) passées en paramètres.

D'autre types de contraintes ont été implémenter explicitement dans Open CSP. Ceci concerne le cas où les variables sont déclarer sous forme de *vecteur de variables*. Lorsqu'on dispose d'un vecteur de variables et que l'on veut appliquer les mêmes contraintes à l'ensemble de ces variables, il serait plus intéressant de les inclure toutes sous une même procédure qui aura pour tâche de créer les contraintes (du même type) et de les *poster* au manager. Un exemple plus concret serait le problème des N-Reines. Les contraintes dans ce problème sont du type (*!=*). On a alors défini dans le manager la méthode *add_AllDiff(PPc_IntVarArray &)*. Elle prend comme paramètre un tableau de variables, elle exécute ensuite la procédure *m.add(C)* en boucle pour des contraintes de type (*!=*) sur autant de variables que contient le vecteur de variables passé en paramètre.

Le résolution de CSP :

La résolution de CSP passe par les filtres prévus dans la classe PPc_filters. Cette classe rassemble les différents algorithmes qui vont contribuer à la résolution. Ces méthodes sont dérivées d'une même classe mère PPc_filters et sont au nombre de cinq :

PPc_AC1, PPc_AC3, PPc_AC4, PPc_bkJump et PPc_CDBL.

Toutes les classes dérivées de PPc_filters étant prévues pour un parcours de l'arbre des solutions, cette classe mère dispose des champs :

PPcBool done : à true lorsque l'algorithme abouti. A false sinon.

PPcInt nbBckJump : le nombre de backjump effectués (ne concerne que les deux classes PPc_CDBL et PPc_bkJump).

PPcInt nodeChkd : renvoie le nombre de sommets parcourus (node checked). Information comparative pour connaître la vitesse avec laquelle l'algorithme converge vers la solution (ne concerne que les deux classes PPc_CDBL et PPc_bkJump).

Les algorithmes de consistance d'arcs sont implémentés dans leur classes respectives PPc_AC1, PPc_AC3, PPc_AC4. Ces classes ont toutes en commun la méthode

run(PPc_Manager & m) ;

qui prend comme paramètre une référence du manager sur lequel on veut exécuter l'algorithme.

Les algorithmes de résolutions sont implémentés dans les classes PPc_bkJump (pour le back Jump de Gaschnig) et PPc_CDBL (pour le Conflict Directed Backjumping Learning).

L'appelle d'une résolution est simple. Ces deux classes ont en commun la méthode run() (redéfinie pour chaque algorithme ! !) qu'il faut appeler en désignant le manager.

Si l'algorithme réussit à trouver une solution, la donnée membre done est à true , elle est à false sinon. Un exemple où on appelle une de ces méthodes serait le suivant :

```
PPc_cdbl resolution ;
resolution.run(m) ; // où m est une référence sur un objet PPc_Manager
if (resolution.done)
  //Récupérer les résultats
else
  // pas de solution trouvée
```

Un simple parcours des membres *val* des objets *PPc_IntVar* nous informera sur les différentes instanciations faites.

Il est à noter que par un souci de complétude du noyau, l'ordonnancement des variables ainsi que la réduction de leur domaines est déjà prévu dans l'implémentation des méthodes de résolution. Ceci revient à dire que dès que la procédure de résolution est appelée, celle ci commence par une propagation de contrainte (création d'un objet *AC* de type *PPc_AC4*; puis appel de la procédure *AC.run(m)*) puis un ordonnancement de ses variables (création d'un objet *order* de type *VarOrder1* ou *VarOrder2*, puis appel de la procédure *order.Order(m.Vars,m)*). Il n'est donc pas nécessaire de passer par ces phases à l'extérieur de la méthode de résolution.

Il est aussi possible d'implémenter d'autres algorithmes de résolution sous condition de respecter le même schéma sur lequel ont été implémenter les deux algorithmes *CDBL* et *bkJump*. Les classes qui prendront en charge de telles tâches doivent impérativement être des descendantes de la classe *PPc_filters*.

Pour résumer cette annexe, ci-dessous est le code source pour implémenter un exemple dans lequel on a essayé de réunir le plus grand nombre de fonctionnalités d'*Open CSP*.

C'est le code source qui sert à modéliser le problème de coloration de graphe (*cf. page 19*). Les commentaires pour chaque ligne de code accompagne l'implémentation.

Fichier colouring.h

```
// --- PPc_colouring.h -----------------------------------
// Colouring exemple :
// theorical exemple : from " Backtracking algorithms for
//constraint satisfaction
```

```
// problems" (Rina Dechter, Daniel Frost), sept, 1999.
//
// Colouring problem :
//      lets  seven  variables  with  each  domain  variable
//        written between brackets
//      a(red,blue,green)
//      b(blue,green)
//      c(red,blue)
//      d(red,blue)
//      e(blue,green)
//      f(red,green,teal)
//      g(red,blue)
//      for an easiest modelisation let red=1, blue=2,
//       green=3,teal=4
//
// ------------------------------------------------------------

#include "PPc_Exemple.h"  // fichier à inclure car c à
                          //l'intérieur qu'est défini la
                          //classe mère PPc_exemple

class PPc_colouring : public PPc_exemple {

  public:
        PPc_colouring();      //Constructeur par défaut
        PPc_colouring(PPcString);  //Constructeur avec
                                   //passage de nom au
                                   //problème.

        ~PPc_colouring();          //destructeur

        void run(PPc_Manager );    //procédure qui exécute
                                   //l'exemple.
};
```

fichier colouring.cpp :

```
// --- PPc_colouring.cpp -----------------------------------

  #include "PPc_colouring.h"

  PPc_colouring::PPc_colouring(){} // const par défaut

  PPc_colouring::PPc_colouring(PPcString s){      this->name=s; }
  // constructeur

  PPc_colouring::~PPc_colouring(){}
  // destructeur

  void PPc_colouring::run(PPc_Manager m)
  {
       clock_t start, finish; // calcul du temps d'exécution
       double duration;

       PPc_IntVar a(m,"a",1,3);        //
       PPc_IntVar b(m,"b",2,3);        //
       PPc_IntVar c(m,"c",1,2);        //
       PPc_IntVar d(m,"d",1,2);        // definig variables
       PPc_IntVar e(m,"e",2,3);        //
       PPc_IntVar f(m,"f",1,4);        //
       PPc_IntVar g(m,"g",1,2);        //

       PPcInt * tab;         //
       tab=new PPcInt[2];    //
       tab[0]=1;             //
       tab[1]=2;             //   suppression de la valeur '2' du
                             //   domaine de f
       f.Dom.ForbidenVal(tab);//
       delete[] tab;         //
       tab=NULL;             //

  // --- constraints definition ---------------------------------
       PPc_BinIntCons C1;
       C1=a!=b;
       m.add(C1);

       PPc_BinIntCons C2;
       C2=a!=c;
       m.add(C2);

       PPc_BinIntCons C3;
       C3=a!=d;
       m.add(C3);

       PPc_BinIntCons C4;
       C4=a!=g;
       m.add(C4);
```

```
      PPc_BinIntCons C5;
      C5=c!=g;
      m.add(C5);

      PPc_BinIntCons C6;
      C6=d!=g;
      m.add(C6);


      PPc_BinIntCons C7;
      C7=d!=e;
      m.add(C7);

      PPc_BinIntCons C8;
      C8=e!=f;
      m.add(C8);

      PPc_BinIntCons C9;
      C9=e!=g;
      m.add(C9);

      PPc_BinIntCons C10;
      C10=b!=f;
      m.add(C10);
// --- end constraints definition -----------------------------



      cout<<endl<<endl<<"Colouring Exemple ............"<<endl;
      cout <<".....................Vars added ......."<<endl;
      cout<<m.nbConstraint<<"  -  "<<m.nbVariables<<endl;
      cout <<".....Constraints added ..........."<<endl;
      cout<<m.nbConstraint<<"  -  "<<m.nbConstraint<<endl;




      PPc_CDBL cdbl;  // déclaration de l'objet cdbl pour la
                      //  résolution

start=clock();        // -------------- TIMER -----------------
      cdbl.run(m);
finish=clock();       // -------------- END TIMER -------------

      duration = (double)(finish - start) / CLOCKS_PER_SEC;
      m.duration=duration;
      if ( cdbl.done)
      {
      cout<<"Resolution DONE in "<<duration<<"  seconds"<<endl;
      cout << "Check the solution.txt file ....."<<endl;
           ofstream fileOut("Clrout.ppc"); // création du
                                           // fichier Out.ppc
```

```
            m.print(fileOut);
            fileOut.close();

            ofstream solution("Clrsolution.ppc"); // création du
                                                  //fichier
                                                  //solution.ppc
            m.solution(solution);
            solution.close();
      }
      else
      {     cout << "Resoluition failed ............."<<endl;
            cout << "Check the nogood.txt file ......"<<endl;
      }


}// --- end run()
```

Fichier main.cpp :

```
#include «PPc_Lib.h » // entête de tous les fichiers utilisés

int main (int argc, char ** argv)
{
      PPc_Manager m;
      m.name="Example Manager";

      PPc_colouring color;
      color.run(m);

      return 0 ;
}
```

Autres exemples implémenté dans *OpenCSP* :

Exemple 2 Coloration de graphe :

Fichier PPc_colour.h :

```
// --- PPc_colour.h ------------------------------------------------
// Colouring a six  countries map exemple :
// theorical exemple : from Ilog Solver lib examples.
//
// Colouring problem :
//having a map of six european countries . We want to colour the map
// with at most 4 colours.
//for an easiest modelisation let blue=1 white=2 red=3 and green=4.
//
// -----------------------------------------------------------------
```

```
#include "PPc_Exemple.h"

class PPc_colour : public PPc_exemple {
      public:
            PPc_colour();
            PPc_colour(PPcString);
            ~PPc_colour();
            void run(PPc_Manager &);
};
```

fichier PPc_coulour.cpp :

```
#include "PPc_colour.h"


PPc_colour::PPc_colour(){}

PPc_colour::PPc_colour(PPcString s)
{
      this->name=s;
}

PPc_colour::~PPc_colour(){}

void PPc_colour::run(PPc_Manager &m)
{



      clock_t start, finish;
      double duration;

      cout<<endl<<"Colouring a six countries map ................."<<endl;
      cout<<endl;

      PPc_IntVar Belgium(m,"belgium", 1, 4), Denmark(m,"Denmark", 1, 4),
      France(m,"France", 1, 4), Germany(m,"Germany", 1, 4),
      Netherlands(m,"Netherlands", 1, 4), Luxembourg(m,"Luxembourg", 1, 4);

      m.add(France != Belgium);
      m.add(France != Luxembourg);
      m.add(France != Germany);
      m.add(Luxembourg != Germany);
      m.add(Luxembourg != Belgium);
      m.add(Belgium != Netherlands);
      m.add(Germany != Netherlands);
      m.add(Germany != Denmark);

      cout <<".........Vars added ................................."<<endl;
      cout<<m.nbConstraint<<"  -  "<<m.nbVariables<<endl;

      PPc_CDBL cdbl;

start=clock();            // -------------- TIMER -----------------

try{
      cdbl.run(m);

      } catch(exception &e)
      {
            cerr<<e.what();     }
```

```
	catch(...)
	{
		cerr<<"error   on   resolution   process.............!!!"<<endl;
}

finish=clock();		// -------------- END TIMER -------------
	duration = (double)(finish - start) / CLOCKS_PER_SEC;
	m.duration=duration;
	if ( cdbl.done)

	{
		cout<<"Resolution  DONE  in "<<duration<<"  seconds ...."<<endl;
		cout << "Check the solution.txt file …................."<<endl;
		ofstream fileOut("IlgClrout.ppc");
		m.print(fileOut);
		fileOut.close();

		ofstream solution("IlgClrsol.ppc");
		m.solution(solution);
		solution.close();
	}
	else
	{	cout << "Resoluition failed ............................"<<endl;
		cout << "Check the nogood.txt file ..................."<<endl;
	}
}// --- end run()
```

fichier main.cpp :

```
#include «PPc_Lib.h » // entête de tous les fichiers utilisés

int main (int argc, char ** argv)
{
	PPc_Manager m;
	m.name="Example Manager";

	PPc_colouring color;
	color.run(m);

	return 0 ;
}
```

détail du fichier solution.ppc après la résolution :

```
MANAGER INFO :
Manager : Example Manager
Number of Variables ................. 6
Number of Constraints ............... 8
Solved in 0.06 seconds.............
Variable Info :
   name: Germanyreferenced in Manager as var4
   DomMin: 1    DomMax: 4
Instanciated to ..................... 2
Variable Info :
   name: belgiumreferenced in Manager as var2
   DomMin: 1    DomMax: 4
Instanciated to ..................... 2
Variable Info :
   name: Luxembourg    referenced in Manager as var3
   DomMin: 1    DomMax: 4
```

```
Instanciated to ..................... 3
Variable Info :
   name: France referenced in Manager as var1
   DomMin: 1    DomMax: 4
Instanciated to ..................... 4
Variable Info :
   name: Netherlands   referenced in Manager as var5
   DomMin: 1    DomMax: 4
Instanciated to ..................... 3
Variable Info :
   name: Denmarkreferenced in Manager as var6
   DomMin: 1    DomMax: 4
Instanciated to ..................... 3
```

Détails du fichier xplain.ppc après la résolution :

```
MANAGER INFO :
Manager name : Example Manager
Number of Variables ................. 6
Number of Constraints ............... 8

Resulotion perform a back Jump 0 times......
Number of cheked nodes : 6 nodes......
Variables Ordering....
  variable Germany (id:var4) , variable belgium (id:var2) ,
variable Luxembourg (id:var3) , variable France (id:var1) ,
variable Netherlands (id:var5) , variable Denmark (id:var6)
```