PACTE NOVATION
2, rue du Dr Lombard
92441 Issy-les-Moulineaux

Projet de Fin d'Etudes

RESOLUTION DE GRANDS PROBLEMES DE SATISFACTION DE CONTRAINTES. APPLICATION AU PROJET MAXIME

V 1.5

Nicolas Strauel, 5IF

Direction : Philippe Morignot (Pacte Novation)
Guy Caplat, Louis Frécon (INSA Lyon)

Institut National des Sciences Appliquées de Lyon
Département Informatique

PACTE NOVATION
2, rue du Dr Lombard
92441 Issy-les-Moulineaux

Projet de Fin d'Etudes

RESOLUTION DE GRANDS PROBLEMES DE SATISFACTION DE CONTRAINTES. APPLICATION AU PROJET MAXIME

V 1.5

Nicolas Strauel, 5IF

Direction : Philippe Morignot (Pacte Novation)
Guy Caplat, Louis Frécon (INSA Lyon)

Institut National des Sciences Appliquées de Lyon
Département Informatique

MOTS CLEFS

Programmation par contraintes, Emploi du temps, Planification,
Heuristique, Ilog Solver.

RESUME

Dans le cadre de la résolution de grands problèmes de satisfaction de contraintes (CSP) à valeurs discrètes, nous présentons plusieurs méthodes récentes visant à réduire les temps de calcul.

Nous effectuons ensuite l'adaptation et l'implémentation de certaines de ces méthodes en vue de traiter le problème de *génération d'emploi du temps* posé par l'application MAXIME. Ces implémentations sont faites au-dessus du moteur de résolution Ilog Solver 4.3.

Enfin, nous comparons les performances obtenues avec les différentes méthodes et leurs variantes.

KEYWORDS

CSP, Constraints, Scheduling, Timetabling, Heuristic, Ilog Solver.

ABSTRACT

We present in this paper several recent methods aiming at saving computation time when solving large discrete Constraint Satisfaction Problems.

We then adapt and implement them in order to deal with the scheduling problem stated by our application : MAXIME. These algorithms are developed on top of Ilog Solver 4.3 solving engine.

Finally, we compare the results given by these different methods and their variants.

TABLE DES MATIERES

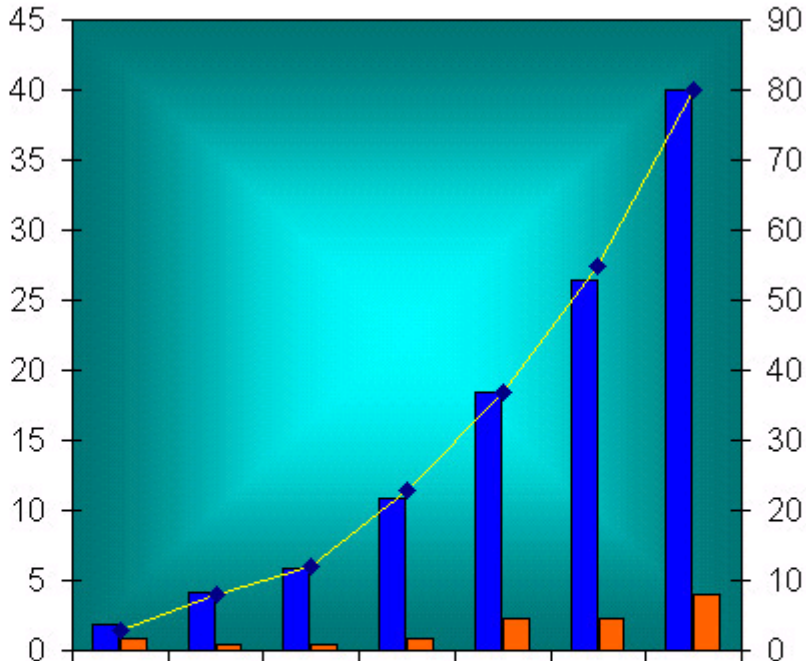
1 INTRODUCTION

1.1 PACTE NOVATION

PACTE NOVATION, entreprise créée en avril 1994, est une Société de Conseil et d'Ingénierie en Informatique spécialisée dans l'utilisation et l'intégration des techniques d'Informatique Avancée. Elle est actuellement composée d'une quarantaine d'ingénieurs. Son chiffre d'affaire est d'environ 25 millions de francs pour un résultat net de plus de deux millions.

Deux associés sont à la tête de PACTE NOVATION : Christian Tora, Président Directeur Général, et Bruno Gaudinat, Directeur Technique.

1.1.1 Chiffres d'affaire et résultats nets



	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

1.1.2 Domaines de compétences

Les compétences de PACTE NOVATION sont réparties sur trois axes :

- **La communication homme/machine (incluant l'ergonomie et la réalisation d'IHM) :**
  Prise en compte du facteur humain lors de l'informatisation de processus
  Analyse de la tâche et spécification des Interfaces Homme/Machine
  Evaluations et recommandations ergonomiques sur des IHM existantes
  Réalisation d'IHM graphiques et d'outils utilisant le Langage Naturel

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

- **L'aide à la décision :** résolution de problèmes intégrant des raisonnements d'experts, ayant une forte complexité ou 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, ...

Signalons enfin que PACTE NOVATION est membre du Comité Richelieu, dont la mission est de promouvoir les petites et moyennes entreprises innovantes, dans le milieu des hautes technologies, au sein de la D.G.A. et d'autres organismes publics.

1.1.3 Partenaires et clients

PACTE NOVATION intervient dans de nombreux domaines aussi divers que l'industrie métallurgique (SOLLAC, SPSE, Société Le Nickel), l'énergie (EDF, GDF), le transport (Renault Sport, GEC Alstom, ADP, Eurocontrol, ERAAM, Transnucléaire), l'industrie de l'informatique et des télécommunications (Alcatel, Bull) ou encore la finance des salles de marché (Crédit Agricole Indosuez, Crédit Lyonnais, Dresdner Bank, Société Générale).

La société s'est associée à un partenaire de choix : ILOG. ILOG, société française cotée au NASDAQ, est le premier éditeur français dans le domaine des composants logiciels orientés objets. Le partenariat est fort puisque PACTE NOVATION se voit confier l'évaluation de certains de leurs produits et la sous-traitance d'un certain nombre de leurs formations. Ce partenariat est d'ailleurs mentionné sur le site D'ILOG, à l'adresse : http://www.ilog.fr/html/partners/pacte.htm

Un autre partenariat, dans le domaine de l'ergonomie du logiciel, s'est organisé avec ERGO SOFT France, qui leur permet d'exploiter leurs laboratoires d'utilisabilité.

Enfin, PACTE NOVATION a également réussi à créer des liens scientifiques solides notamment dans le domaine des Sciences cognitives et du langage naturel, avec le C.A.M.S. de Paris.

1.1.4 Types d'activités

PACTE NOVATION exerce trois types d'activités :

- Réalisation de projets au forfait, branche importante de la société puisqu'elle représente 50% de ses activités.
- Conseil, notamment en Ergonomie du Logiciel et en Architectures Orientées Objet.
- Assistance technique, délégation de personnel, dans ses domaines de compétence.

1.1.5 Quelques références

En un peu plus de cinq années d'existence, PACTE NOVATION a participé à des projets ou programmes d'envergure comme :

- Projet ALICE-CARROLL pour Renault Sport F1, un système expert pour la supervision en temps réel des moteurs de Formule 1.
- Un Simulateur de Trafic Ferroviaire pour GEC-Alsthom, vendu dans des affaires à l'export (Hongkong, Athènes, Indonésie...).
- Projet SACHEM pour la SOLLAC, dans le domaine de la supervision des hauts-fourneaux.
- Projet AGATE pour SPSE (Société Pétrolière du Sud Européen), dans le domaine de la planification et l'optimisation d'un pipeline pétrolier.
- Projet MAXIME pour ADP (Aéroports De Paris), qui concerne la planification du personnel de l'escale des aéroports parisiens.

1.2 Sujet de l'étude

Cette étude a pour but de dégager de nouvelles stratégies en vue de l'amélioration des performances de résolution de Problèmes de Satisfaction de Contraintes (*Constraint Solving Problems* ou *CSP*) à valeurs discrètes.

Ces nouvelles méthodes seront orientées vers l'augmentation de la taille des problèmes pouvant être traités par l'application MAXIME. Cette application a pour but de générer des emplois du temps pour le personnel au sol des Aéroports De Paris (ADP). Elle utilise pour cela, dans un premier temps, une stratégie de résolution de CSP *constructive* dont le déroulement est géré par l'outil *Ilog Solver 4.3*. Dans un second temps, plusieurs autres méthodes sont mises en place (notamment de la recherche *locale*) afin d'améliorer la solution initiale.

Avec un algorithme standard, le temps de génération de cette solution initiale croît en général de façon exponentielle avec le nombre de tâches à incorporer dans l'emploi du temps (le problème est dit *NP-complet*). Il s'agit ici de juguler au mieux cette explosion combinatoire.

Ainsi, cette étude ne traitera pas de recherche locale (déjà réalisée en phase deux de l'application), et se concentrera uniquement sur les processus de résolution de CSP pouvant réutiliser le modèle préexistant sous Ilog Solver. Ceci écarte donc également, entre autres, les systèmes utilisant exclusivement les algorithmes génétiques ou les réseaux de neurones.

2 PROPAGATION DE CONTRAINTES ET METHODES CONNEXES

2.1 Principes de base

2.1.1 Définitions

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.
- en outre, le problème peut être doté d'une *fonction objectif*. C'est une fonction à valeur réelle de l'ensemble des variables X, et qui doit être minimisée (ou maximisée – dans ce texte nous considérerons que cette fonction doit être minimisée, ce qui est équivalent à en maximiser l'inverse) par l'algorithme de résolution. Cette fonction permet de pré-ordonner les différentes solutions du problème (et donc de définir des solutions *optimales.*)

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. 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.

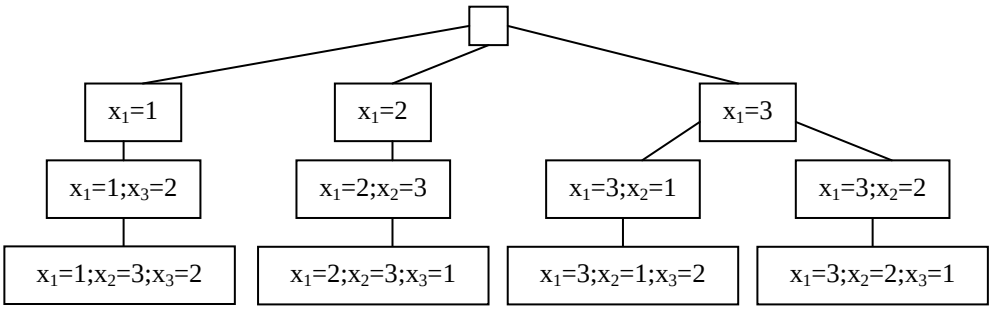
Une *solution* au problème est un ensemble de variables instanciées comportant *toutes* les variables du problème et étant consistant avec *toutes* les contraintes du problème.

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* ou *instanciation partielle consistante*. Un ensemble de variables instanciées ne comportant pas toutes les variables du problème et n'étant pas consistant avec toutes les contraintes est appelé *instanciation partielle inconsistante.*

Le but de la résolution d'un CSP peut être de trouver *toutes* les solutions du problème, ou alors *la* meilleure. **Dans cette étude, par contre, nous chercherons à obtenir le plus rapidement possible *une* solution** (la première trouvée), tout en nous laissant la possibilité d'en chercher d'autres, meilleures, par la suite.

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 $x_2 \neq x_3$



Notons que, en suivant deux branches parallèles, les variables ne sont pas forcément instanciées dans le même ordre.

2.1.2 Approche constructive classique

2.1.2.1 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. $D_1 \ldots D_n$ contiennent les domaines de valeurs pour les variables $x_1 \ldots x_n$
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

***(1)** : 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 « reprise 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.

C'est pourquoi ces deux points font l'objet d'heuristiques discutées dans le chapitre *2.2 : Amélioration de la stratégie constructive*

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 ***(2)*** :

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

Considérons l'instanciation partielle consistante *{x_1=1, x_2=2, x_3=1}*. Si les valeurs des variables sont essayées dans l'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* [Kuma92].

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

2.1.2.2 Méthodes à test arrière

Dans ces méthodes (tout comme dans l'algorithme de base ***(1)***), 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. Du coup, ce type de méthode peut également souvent tomber en échec en raison du thrashing.

Dès lors, l'optimisation de ces méthodes passe par l'optimisation du mécanisme de backtracking. Les méthodes optimisées existantes sont des variantes du *backjumping*. 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 jusqu'au nœud précédent uniquement, ils remontent jusqu'à la première variable susceptible d'influencer la situation d'échec.

Dans notre exemple ***(2)***, 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 découvertes 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 [FroDec99-2].

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. Sans détailler ces systèmes, on peut citer le *conflict-directed backjumping* [Fros97] qui se situe parmi les méthodes les plus performantes. [Kond94] 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*) [Fros97]. 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 ***(2)***, 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 immédiatement 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-essayées 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.

On peut toutefois agrémenter ces mécanismes d'un minimum de « test avant », comme le montre le système *BJ-DVO-LVO* [Fros97]. Ce point est discuté dans le chapitre *2.2 : Amélioration de la stratégie constructive*.

2.1.2.3 Méthodes à test avant (*Look-Ahead*)

Ici, l'efficacité est trouvée en supprimant, avant instanciation des variables (avant la ligne 5 de ***(1)***), 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 *backtracking* (sans toutefois l'éliminer car il arrive que le domaine d'une variable non-instanciée devienne vide à force d'en supprimer des 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 ***(2)***, 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. [Fros97] présente 3 grands types de méthodes de *Look-Ahead* (classés ici dans l'ordre croissant de leur qualité) :

1. Le *Forward-Checking* consiste à ne tester que la consistance directe avec les variables instanciées (consistance de niveau 1, *cf. ci-dessous*).
2. 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*).
3. 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, [Kuma92] 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*), appliqué à l'aide de l'algorithme AC-3 [Fros97]. AC-3 étant défini pour des CSP binaires uniquement, ce sont bien souvent des variantes qui sont utilisées : elles portent le nom générique d' « algorithmes à 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.

2.1.3 Performances et modélisation du problème

Certaines caractéristiques du problème indépendantes de l'algorithme utilisé peuvent influer directement sur les performances de résolution :

2.1.3.1 Contraintes redondantes

L'introduction de contraintes redondantes (*User's Manual* Ilog Solver 4.3, p. 61 et [TsaKwa93]) peut améliorer les capacités de réduction de la taille des domaines des variables (ou la qualité du backjumping), et donc minimiser la taille du problème. Cette « redondance » a pour but de rendre explicite certaines contraintes contenues implicitement dans la combinaison d'autres contraintes exprimées.

Comme le CSP ne peut pas être rendu totalement consistant à chaque instanciation d'une nouvelle variable (*cf. chapitre 2.1.2.3 : Méthodes à test avant (Look-Ahead)*), les contraintes (ou plus souvent les groupes de contraintes) ayant des implications trop indirectes sur les domaines de certaines variables ne sont pas propagées. Par contre, une contrainte unique « résumant » ces implications et ayant un effet direct sur les domaines des variables sera elle propagée et les domaines de ces variables seront effectivement réduits.

2.1.3.2 Valeurs fictives et relaxation de contraintes

Plusieurs méthodes (citées dans les prochains chapitres) fonctionnement mieux lorsqu'elles peuvent savoir si une instanciation partielle inconsistante *A* est (ou n'est pas) très éloignée de l'instanciation partielle consistante la plus proche. Ce critère permet alors de connaître la « non-qualité » *q* de *A*.

On peut, par exemple, définir ceci comme étant le nombre de variables à retirer à *A* pour obtenir une instanciation consistante. Une approximation de cette quantité est facile à déterminer en introduisant une valeur « fictive » dans les domaines des variables (procédé utilisé actuellement dans MAXIME). Cette valeur n'interviendrait dans aucune contrainte et serait choisie « par défaut » par l'algorithme, après échec de toutes les autres valeurs possibles. Pour toute instanciation inconsistante *A*, il suffit alors de compter le nombre de valeurs fictives dans *A* pour obtenir *q*. En général, on ajoute également à la fonction objectif une pénalité proportionnelle au nombre de valeurs fictives utilisées dans la solution, afin de dévaloriser cette solution par rapport à une autre utilisant de « vraies » valeurs pour ces variables.

Il peut aussi s'agir du nombre de contraintes à retirer au problème pour obtenir une instanciation consistante. Là encore, on peut obtenir une approximation de cette quantité en « relaxant » certaines contraintes ([Cost94] et [Cost95]). Relaxer une contrainte consiste à supprimer cette contrainte du problème mais à la remplacer par un terme supplémentaire dans la fonction objectif. Ce terme est conçu de façon à pénaliser les instanciations partielles inconsistantes qui ne respectent pas la contrainte d'origine. Ainsi, la valeur de la fonction objectif peut donner une mesure de *q*.

De façon générale, un problème *relaxé* est plus rapidement résolu que son original. De plus, aucune solution du problème original ne peut être meilleure que la meilleure des solutions d'une version *relaxée* de ce problème.

2.2 Amélioration de la stratégie constructive

Ce chapitre expose des méthodes susceptibles d'améliorer les performances des stratégies constructives classiques exposées dans le chapitre précédent.

2.2.1 Choix des valeurs à base de test avant

2.2.1.1 Description

Il s'agit ici de proposer une solution heuristique efficace au problème de choix de valeur posé en ligne 16 de l'algorithme de base ***(1)***. Le but est de choisir une valeur pour la prochaine variable à instancier.

Le principe de cette heuristique [FroDec95] est le suivant : on teste toutes les valeurs possibles pour la variable (en l'instanciant « à l'essai » avec chacune des valeurs de son domaine) et on note, à chacun de ces essais, l'impact de ce choix sur les variables non-instanciées restantes. Pour cela, on effectue un élagage de type « test avant » après chaque essai. L' « impact » est mesuré par la taille des domaines restants après cette étape. Pour l'instanciation définitive, on essaie alors en priorité les valeurs qui laissent les domaines les plus grands aux variables non-encore instanciées. Intuitivement, on peut en effet penser que ces valeurs ont moins de chances de provoquer de futures inconsistances, et ont donc plus de chances d'aboutir à une solution.

Après ajout de cette fonctionnalité l'algorithme ***(1)*** devient donc :

(on considère dans cet algorithme que la gestion des domaines (ensembles de valeurs non-ordonnées) des variables est indépendante de la gestion des listes ordonnées des valeurs possibles pour ces variables. En théorie, la gestion des domaines n'est plus nécessaire puisque les valeurs possibles sont contenues dans les listes ordonnées. Toutefois, si cette heuristique se greffe sur un algorithme de génération de l'arbre préexistant, la gestion des domaines ne pourra pas être altérée et on se trouve dans le cas exposé ici où les deux entités sont maintenues.)

```
1.  Créer une instanciation partielle P vide, et initialiser une pile vide
2.  D1…Dn contiennent les domaines de valeurs pour les variables x1...xn
3.  Si P est consistante
4.      Si il y a encore une variable non-instanciée
5.          Choisir un indice i correspondant à une variable xi non-instanciée
6.          Créer une nouvelle liste vide Li associée à xi, un entier t et une instanciation partielle Q
7.          Pour chaque valeur e de Di
8.              Construire Q composée de P et de xi=e
9.              Si Q est consistante
10.                 t reçoit une mesure de la taille des domaines des variables restantes après un « test
                    avant » fait sur Q
11.                 Ajouter à Li un élément composé de t (champ « taille ») et de e (champ « valeur »)
12.             Sinon
13.                 Enlever e de Di
14.         Trier Li dans l'ordre décroissant des champs « taille »
15.         Aller à 24
16.     Sinon
17.         P est une solution
18. Si la pile est vide, Arrêter
19. Dépiler et restaurer les domaines D1…Dn précédents
20. i reçoit l'indice de la dernière variable affectée dans P
21. Enlever de Di la valeur qui était affectée à xi dans sa dernière affectation dans P
22. Enlever le premier élément de Li
23. Dépiler et restaurer le P précédent
24. Si le domaine Di est vide
25.     Aller à 18
26. v reçoit le champ « valeur » du premier élément de Li
27. Empiler P et empiler les D1…Dn
28. Affecter v à xi (réduire le domaine Di à {v}) et ajouter cette affectation à P
29. Aller à 3
```

***(4)** : Algorithme avec heuristique de test avant*
*Les modifications par rapport à **(1)** sont indiquées par les bandes noires*

Une petite amélioration peut être définie concernant cet algorithme : durant la phase de test avant (ligne 10), il peut arriver qu'une des variables non-instanciées voit son domaine se vider. Dans ce cas, il est inutile de chercher une solution avec la valeur *e* en cours, et on peut directement passer à la ligne 13.

La fonction qui donne une « mesure de la taille des domaines » a fait l'objet de tests, notamment dans [FroDec95]. Les solutions suivantes ont été essayées :

1. Additionner les tailles des domaines des variables restantes
2. Prendre la taille du domaine restant le plus petit
3. Prendre la taille du domaine restant le plus petit, et départager les *ex æquo* avec le nombre de variables de cette taille
4. Affecter des « points » à certaines tailles de domaines (plus la taille est grande, plus on donne de points) et totaliser ensuite les points correspondants aux tailles des domaines des variables restantes.

De ces 4 fonctions, la première s'est révélée être la meilleure dans tous les tests de [FroDec95].

Indiquons enfin que cette heuristique est *générique* (c'est-à-dire qu'elle est efficace sur une grande variété de problèmes.) Elle peut naturellement être surpassée par des heuristiques particulières à un problème donné.

2.2.1.2 Discussion sur les performances

L'efficacité de l'heuristique précédente est largement conditionnée par les points suivants :

- Comme toute heuristique de choix de valeur, elle est d'autant plus efficace qu'elle réduit de beaucoup la probabilité de choisir une valeur n'aboutissant pas à une solution. Or, cette probabilité est déjà largement plus faible avec un système de génération de l'arbre à base de test avant (car les domaines

sont réduits à l'avance) qu'avec un système à base de test arrière. Cette heuristique va donc mieux fonctionner sur des systèmes à test arrière.

- Le temps passé à faire les tests avants de la ligne 10 de l'algorithme ***(4)*** est crucial pour l'efficacité de la méthode. En effet, un test est effectué à chaque instanciation de variable, et pour toutes les valeurs présentes dans le domaine. Le test gagnera donc à être sommaire mais rapide. Dans [FroDec95], ce test est un *Forward-Checking* (détection des conflits directs uniquement, *cf. chapitre 2.1.2.3 : Méthodes à test avant (Look-Ahead)*).
- Lors du tri de la liste des valeurs, il est possible de départager les *ex æquo* par une autre heuristique, éventuellement particulière au problème.

2.2.2 Choix des valeurs et évaluations mathématiques

2.2.2.1 Introduction

Plutôt que de classer les valeurs possibles en fonction d'une quantité dépendant directement de la taille des domaines restants à explorer, il existe une théorie permettant d'évaluer la probabilité de trouver une solution à un problème donné, moyennant la connaissance de certains paramètres [Hogg9x]. On peut alors appliquer ce principe au sous-problème restant à résoudre après le choix d'une certaine valeur, et donc ordonner les valeurs en fonction de cette probabilité. Cette théorie s'étend également à l'estimation du temps de résolution d'un problème.

Comme la plupart des théories utilisant des formalismes mathématiques, les calculs suivants sont prévus pour des CSP binaires. Appelons m le nombre minimal d'instanciations partielles inconsistantes dans un sous-problème. Ces instanciations partielles sont composées d'exactement 2 affectations puisque le CSP est binaire.

Considérons à présent la contrainte d'indice j. Elle concerne les variables x_{j1} et x_{j2}, dont les cardinalités des domaines sont respectivement notées k_{j1} et k_{j2}. Appelons z_j le nombre de couples de valeurs interdits par la contrainte d'indice j parmi tous les couples de valeurs possibles de x_{j1} et x_{j2}.

Le quotient $\frac{z_j}{k_{j1} \cdot k_{j2}}$ sera appelé *taux de contrainte* de la contrainte d'indice j.

Supposons, dans un problème donné, qu'aucune des valeurs restantes pour les variables non-instanciées n'est *directement* inconsistante avec les variables déjà instanciées (par exemple, un *Forward-Checking* a été effectué, *cf. chapitre 2.1.2.3 : Méthodes à test avant (Look-Ahead).*) Dans ce cas, les seuls contraintes pouvant encore mener à un échec sont celles qui ne concernent *que* des variables non-instanciées. Appelons C l'ensemble des indices de ces contraintes.

On peut alors poser : $m = \sum_{j \in C} z_j$

Notons que z_j est lié au *taux de contrainte* défini plus haut, et que ce taux peut être aisément déterminé pour beaucoup de contraintes binaires (notamment, il est étudié pour MAXIME dans le chapitre *3.1.1 : MAXIME*). Nous pouvons donc déterminer m.

Pour maintenir la valeur de m tout au long de l'élaboration de l'arbre des solutions, il suffit de procéder comme suit :

- Initialiser une table donnant, pour chaque contrainte, les variables associées et le z de cette contrainte.
- m est initialisée à la somme de tous les z
- A chaque sauvegarde d'une instanciation partielle, sauvegarder m ; à chaque restauration d'une instanciation partielle, restaurer m
- A chaque instanciation d'une variable x_i, m est recalculé en retranchant les valeurs de z des contraintes où x_i apparaît, et où la seconde variable impliquée n'est pas encore instanciée.

2.2.2.2 Calculs

Dans la suite, nous considérerons un sous-problème composé de n variables non-instanciées.

Posons à présent N comme étant le nombre maximal possible d'instanciations partielles minimum inconsistantes [Hogg9x] :

$$N = \sum_{i=1..n, h=1..n, i \neq h} k_i \cdot k_h = \frac{1}{2}\left[(\sum_{i=1}^{n} k_i)^2 - \sum_{i=1}^{n} (k_i)^2 \right]$$

La probabilité p_n qu'une instanciation des n variables du sous-problème soit consistante (c'est-à-dire que ce soit une solution) est définie par :

$$\ln p_n \approx m \cdot \ln\left(1 - \frac{\binom{n}{2}}{N}\right)$$

En utilisant une méthode à base de backtracking simple comme celle de l'algorithme ***(1)***, le coût (mesure du temps) de recherche pour trouver la première solution est :

$$C_{1st} \approx \frac{\sum_{h=1}^{n} \left(e^{\sum_{i=1}^{h} k_h - \frac{mh^2}{2N}} \right)}{\max\left(1, e^{\sum_{i=1}^{n} k_n - \frac{mn^2}{2N}}\right)}$$

Avec ces deux indications (probabilité de trouver une solution, temps mis pour trouver une solution), on peut donc construire de nouvelles heuristiques mêlant l'un ou l'autre facteur, afin de maximiser les chances de trouver une solution, ou alors de minimiser le temps de recherche dans un sous-problème supposé sans solution.

2.2.2.3 Discussion sur les performances

Les points suivants, pouvant fortement influencer les performances, sont à noter concernant cette méthode :

- Le modèle mathématique est prévu pour des CSP binaires. Dans le cas de CSP non-binaires, une approximation devra être faite afin de se ramener à un modèle binaire.
- z est censé décrire le nombre de couples de valeurs interdits par une contrainte, parmi tous les couples formés par le produit cartésien des domaines de ces variables lorsqu'elles ne sont pas instanciées. Or, si les domaines des variables sont *a priori* réduits en raison d'une impossibilité à satisfaire la contrainte pour certaines valeurs de l'une ou de l'autre des variables, le sens de m par rapport aux tailles des domaines sera faussé. Ceci peut arriver lorsqu'une mise en consistance de niveau supérieur à 1 est effectuée par l'algorithme de construction.
- La formule donnant le coût de calcul d'une solution est élaborée pour une stratégie à base de backtracking simple, et est probablement peu fiable si elle est utilisée avec des algorithmes plus complexes. De plus, la précision intrinsèque de son évaluation n'est pas très bonne [Hogg9x].

2.2.3 Choix des variables

On peut classer ces heuristiques en deux catégories, suivant que le choix de la prochaine variable à instancier est *dynamique* (effectué après l'instanciation de la variable précédente) ou *statique* (toutes les variables sont ordonnées en début d'algorithme). [FroDec94] note que les performances des heuristiques dynamiques sont nettement meilleures que les autres. Nous nous concentrerons donc sur cette catégorie de méthodes.

2.2.3.1 Heuristique dynamique basée sur les domaines

Parmi les heuristiques de [TsaKwa93], on retiendra celle qui consiste à choisir systématiquement la variable dont le domaine est le plus restreint. En outre, [FroDec94] départage les *ex æquo* en donnant la priorité aux variables qui sont impliquées dans le plus de contraintes. Cette heuristique est considérée comme donnant en général de très bons résultats.

En effet, les variables qui ont le moins de valeurs possibles et le plus de contraintes associées sont vraisemblablement celles pour lesquelles il est le plus difficile de trouver une valeur. Il est donc probable que ce soit en les instanciant en premier que l'algorithme trouvera une impasse au plus tôt. Ainsi, cela évitera de trouver

une impasse après avoir déjà instancié d'autres variables (et perdre alors du temps en backtracking sur ces variables.)

2.2.3.2 Heuristiques dynamiques basées sur les valeurs

[Hogg9x] utilise un principe qui peut se généraliser : pour construire une heuristique de choix de variables, on peut utiliser une heuristique de choix de valeurs et choisir la variable dont les valeurs donnent, via cette heuristique, la situation la plus favorable.

Ainsi, on peut déterminer les heuristiques suivantes :

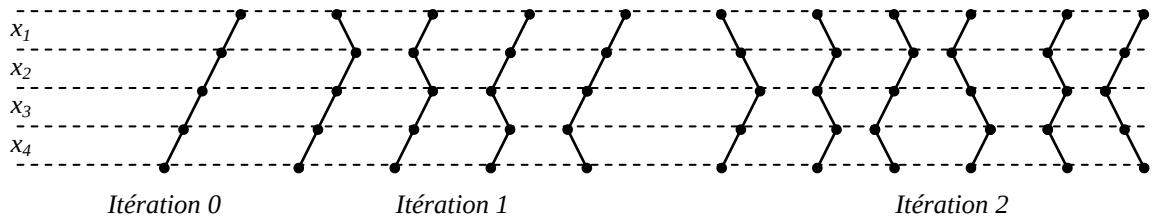
- Choisir la variable qui possède les valeurs qui ont le moins de chances de succès (*cf. chapitre 2.2.2 : Choix des valeurs et évaluations mathématiques*), afin de rencontrer les impasses au plus tôt.
- Choisir la variable qui possède les valeurs engendrant le plus long temps de calcul (utilisée dans [Hogg9x]) En fait, [Hogg9x] utilise une autre heuristique de choix de variables particulière à son problème, et prend celle-ci pour départager les *ex æquo*.

Il faut toutefois veiller à ce que le coût de l'heuristique de choix de valeurs ne soit pas trop important : en effet, il faudra utiliser cette heuristique pour chaque valeur de chaque variable candidate à l'instanciation.

2.2.4 « Limited Discrepancy Search » : Une approche différente

Ce système présenté à l'origine dans [HarGin95] introduit une approche différente de l'algorithme ***(1)***. Dans un backtrack (ou backjump) classique, une heuristique de choix de valeurs aide, à chaque nœud de l'arbre des solutions, à explorer les différentes branches dans un ordre supposé avantageux. Mais un choix erroné (de l'heuristique) effectué près de la racine de l'arbre ne peut être corrigé avant d'avoir exploré tout le sous-arbre (souvent de grande taille puisque l'erreur a été commise tôt) correspondant. Or, la plupart des heuristiques sont moins fiables près de la racine de l'arbre que près des feuilles. Ceci est probablement particulièrement vrai lorsque les heuristiques se basent, soit sur les valeurs des variables instanciées précédemment (il n'y en a pas beaucoup au début de la recherche), soit sur du test avant (moins fiable lorsqu'on est éloigné des solutions.)

La méthode « Limited Discrepancy Search » est conçue à la base pour des variables booléennes. Elle propose d'explorer d'abord la solution donnée par les valeurs préconisées par l'heuristique à tous les nœuds de l'arbre (itération 0) ; puis, d'explorer toutes les solutions données par les valeurs préconisées par l'heuristique à tous les nœuds de l'arbre sauf un (itération 1) ; puis, tous sauf deux (itération 2) ; etc… Il faut *n+1* itérations pour explorer tout l'arbre d'un problème à *n* variables.

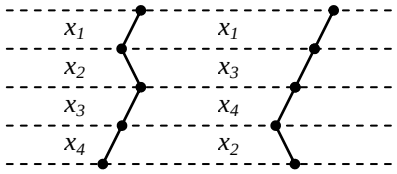


***(5)** : Exemple des 3 premières itérations pour un problème à 4 variables booléennes dont l'arbre est parcouru par un système à base de « Limited Discrepancy Search ».*
Les arcs représentent les variables. La valeur des variables préconisée par l'heuristique est toujours celle repésentée par la branche de gauche.

L'exemple du schéma ***(5)*** est pris avec un ordonnancement statique des variables. Dans ce cas, il est facile de voir que toutes les branches de l'arbre sont explorées après *n+1* itérations : à l'itération *k*,

$\binom{n}{k} = \frac{n!}{(n-k)!\cdot k!}$ solutions potentielles différentes sont explorées, donc après *n+1* itérations, on a exploré $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ solutions , ce qui correspond bien à toutes les branches de l'arbre.

En fait, le système fonctionne également avec un ordonnancement dynamique des variables, c'est-à-dire que toutes les solutions sont encore explorées lorsqu'une heuristique dynamique de choix des variables est mise en place. En effet l'algorithme construira toujours 2^n instanciations partielles de taille *n*. Pour qu'une solution ne soit pas explorée, il faudrait donc que 2 solutions identiques à l'ordre des variables près soient construites (***figure (5a)***) :



***(5a)** : « Limited Discrepancy Search » : situation **impossible** avec ordonnancement dynamique des variables.*

Le schéma montre deux solutions identiques à l'ordre des variables près, pour un problème à 4 variables.

Or, ceci est impossible. En effet, dans l'algorithme de base, pour une instanciation partielle (incomplète) donnée, la variable suivante choisie pour instanciation est *toujours* la même (ce qui n'est pas le cas dans la figure ***(5a)*** puisqu'après l'affectation de x_1 à la même valeur dans les deux cas exposés, c'est une fois x_2 qui est choisie, et une fois x_3). Cela implique qu'on ne peut pas changer à un niveau de l'arbre donné l'ordre d'instanciation des prochaines variables s'il n'y a pas eu, à un niveau précédent, un choix de valeur différent pour au moins une des variables. Donc, une situation comme celle de la figure ***(5a)*** ne peut se produire qu'en cours de construction de l'arbre, pour un problème à plus de *4* variables, et seulement si une des variables précédemment instanciées a une valeur différente dans les deux cas.

Donc deux solutions identiques ne sont jamais construites, même avec un ordonnancement dynamique des variables.

On peut étendre ce principe à des variables non-booléennes. Pour cela, il faut scinder le domaine de chaque variable en 2 sous-ensembles : un « préconisé » par l'heuristique, et un « rejeté » par l'heuristique (pas forcément de tailles égales ; d'ailleurs, il n'est pas absolument nécessaire que ces 2 sous-ensembles couvrent la totalité du domaine d'origine – cela constituerait alors une « coupe franche »). On peut alors explorer les valeurs d'un des sous-ensembles avec un algorithme à base de backtrack ou de backjump, tout en gardant le principe de sélection des sous-ensembles défini ci-dessus.

L'algorithme correspondant est celui-ci :

1. Initialiser *t* au nombre de choix « rejetés » autorisés (numéro de l'itération)
2. Créer une instanciation partielle *P* vide, et initialiser une pile vide
3. $D_1 \ldots D_n$ contiennent les domaines de valeurs pour les variables $x_1 \ldots x_n$
4. **Si** *P* est consistante
5. **Si** il y a encore une variable non-instanciée
6. Choisir un indice *i* correspondant à une variable x_i non-instanciée
7. Noter dans D_i les valeurs « préconisées » et « rejetées » suivant l'heuristique de choix de valeur
8. **Aller** à 17
9. **Sinon**
10. *P* est une solution
11. **Si** la pile est vide, **Arrêter**
12. Dépiler et restaurer les domaines $D_1 \ldots D_n$ précédents
13. *i* reçoit l'indice de la dernière variable affectée dans *P*
14. Enlever de D_i la valeur qui était affectée à x_i dans sa dernière affectation dans *P*
15. Dépiler et restaurer le *P* précédent
16. Dépiler et restaurer le *t* précédent
17. **Si** $t > 0$ **et** il y a encore une valeur « rejetée » dans D_i de la variable x_i
18. Choisir une valeur « rejetée » *v* dans le domaine D_i de la variable x_i
19. Empiler *t*
20. $t = t - 1$
21. **Sinon**
22. **Si** il y a encore une valeur « préconisée » dans le domaine D_i de la variable x_i
23. Choisir une valeur « préconisée » *v* dans le domaine D_i de la variable x_i
24. Empiler *t*
25. **Sinon**
26. Aller à 11
27. Empiler P et empiler les $D_1 \ldots D_n$
28. Affecter *v* à x_i (réduire le domaine D_i à $\{v\}$) et ajouter cette affectation à *P*
29. **Aller** à 4

***(6)** : Algorithme de construction de l'arbre des solutions en « Limited Discrepancy Search »*
*Les modifications par rapport à **(1)** sont indiquées par les bandes noires*

2.3 Autres méthodes

Les méthodes présentées dans ce chapitre sont « non-constructives » : elles ne construisent pas un *arbre des solutions* de manière exhaustive. Cependant, elles se basent toutes, à un certain moment, sur des méthodes constructives, ce qui les fait rentrer dans le cadre de cette étude. L'avantage de telles méthodes est de combiner la rapidité de recherche des méthodes incomplètes avec l'exhaustivité des méthodes complètes [LobLem97].

2.3.1 Hybride d'algorithme génétique

Il existe plusieurs façons d'utiliser des algorithmes génétiques dans le cadre de la résolution de CSP [Mic9x]. [BarBri99] propose une méthode originale, hybride entre un algorithme génétique et une résolution constructive, qui assure le respect des contraintes grâce à une *pénalisation des individus inacceptables* (selon la classification de [Mic9x]).

2.3.1.1 Introduction aux algorithmes génétiques

Les algorithmes génétiques tentent de simuler le processus d'évolution naturelle suivant le modèle darwinien dans un environnement donné [BarBri99]. Ils utilisent un vocabulaire similaire à celui de la génétique naturelle. On parlera ainsi d'individus dans une population. L'individu est représenté par un *chromosome* constitué de *gènes* qui contiennent les caractères héréditaires de cet individu. Les principes de *sélection*, de *croisement* et de *mutation* s'inspirent des processus naturels de mêmes noms.

Pour un problème d'optimisation donné, un individu représente de près ou de loin une solution. On lui associe la valeur, pour cette solution, du critère à optimiser : c'est son *adaptation*. On génère ensuite de façon itérative des populations d'individus sur lesquelles on applique des processus de sélection, de croisement et de mutation. La sélection a pour but de favoriser les meilleurs éléments de la population pour le critère considéré (les mieux *adaptés*), le croisement et la mutation assurant l'exploration de l'espace de recherche.

On commence par générer une population aléatoire d'individus, en général uniformément répartis dans l'espace de recherche. Pour passer d'une génération *k* à une génération *k+1*, les opérations suivantes sont effectuées :

- La population est reproduite par *sélection* où les bons individus se reproduisent mieux que les mauvais.
- On applique un *croisement* aux paires d'individus (les *parents*) d'une certaine proportion de la population (probabilité P_c, généralement autour de *0.6*) pour en produire des nouveaux (les *enfants*).
- On applique un opérateur de *mutation* à une certaine proportion de la population (probabilité P_m, généralement très inférieure à P_c).
- L'*adaptation* des nouveaux individus est évaluée et ils sont intégrés à la population de la génération suivante.

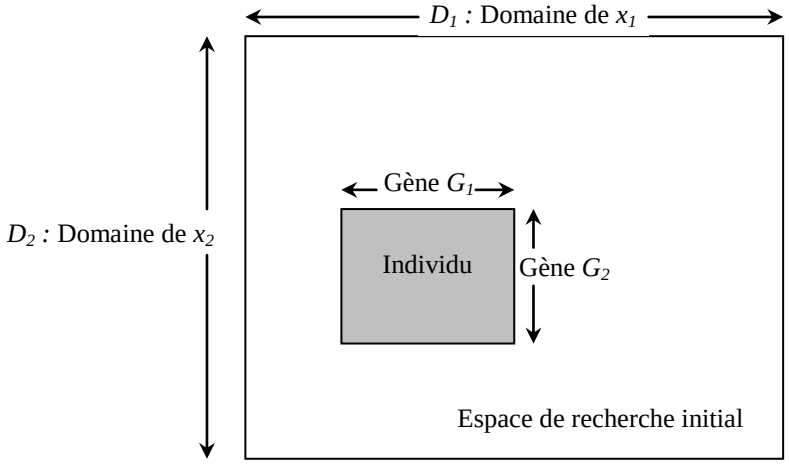
En général, on fait en sorte de garder une population de taille constante (ce qui définit le taux de reproduction).

Plusieurs critères d'arrêt de l'algorithme sont possibles : le nombre de générations peut être fixé *a priori* (temps constant) ou l'algorithme génétique peut être arrêté lorsque la population n'évolue plus assez rapidement.

2.3.1.2 Description de la méthode

La base de cette méthode est constituée par un algorithme génétique où chaque *individu* (ou *chromosome*) représente une *région* de l'espace de recherche initial. L'espace de recherche initial est le produit cartésien des domaines des *n* variables du CSP (une solution au problème est donc représentée par un point de cet espace ; une instanciation partielle est un point dans un sous-espace de cet espace.) Une *région* de l'espace de recherche initial est définie comme étant le produit cartésien, pour les *n* variables, d'une fraction du domaine de chacune.

Un *gène* de l'individu *A* est représenté par l'ensemble des valeurs contenues dans la fraction de domaine associée à l'individu *A*, pour une des variables.



***(7)** : Représentation schématique des entités utilisées par la méthode [BarBri99] pour un problème à 2 variables x_1 et x_2*

Un individu représente donc un sous-problème du CSP initial : il travaille sur des variables dont les domaines sont réduits, et il ne comporte que les contraintes influencées par ces sous-domaines.

Le paramètre ρ permet de définir la taille des individus en posant : $\rho = \frac{|D_i|}{|G_i|} \quad \forall i \in \{1,...,n\}$. Notons que l'on peut passer d'une résolution purement constructive ($\rho = 1$) à une résolution purement génétique ($|G_i| = 1$). Il peut être intéressant d'avoir plusieurs valeurs de ρ suivant les variables, si les tailles de leur domaine sont très différentes.

L'*adaptation* d'un individu est obtenue par résolution constructive du CSP qu'il représente : on utilise pour cela la valeur de la fonction objectif pour la solution trouvée. Un *time-out* est fixé sur cette résolution, afin de laisser un maximum de temps de calcul à l'approche « génétique » du problème. L'optimisation de la solution du problème global est d'ailleurs confiée à l'aspect « génétique » de la résolution, et il n'est donc pas nécessaire que la résolution constructive cherche à améliorer la première solution qu'elle trouve.

Une valuation pénalisante (par exemple *0*) est affectée à chaque individu pour lequel aucune solution n'est trouvée, afin qu'il n'intervienne que peu dans le processus d'amélioration de la population. Cependant, une bonne partie des individus risquent de se retrouver sans solution si toutes les contraintes sont gardées. Il est donc vraisemblablement préférable de travailler sur une version relaxée du problème (*cf. chapitre 2.1.3.2 : Valeurs fictives et relaxation de contraintes*).

2.3.1.3 Opérateurs génétiques

Les opérateurs présentés ici sont des opérateurs génériques : dans un cas particulier donné, ils peuvent se révéler moins efficaces que des opérateurs spécialement conçus pour ce cas particulier. Néanmoins, ils peuvent servir de base de travail pour de premières optimisations.

Ces opérateurs ne sont pas tout à fait les opérateurs « habituels » utilisés dans les algorithmes génétiques de base : ils ont été mis au point dans le cadre de cette méthode hybride, et évalués expérimentalement par [BarBri99] comme étant supérieurs aux opérateurs habituels.

2.3.1.3.1 Croisement

Appelons *locus* l'endroit dans un chromosome où se situe un gène (le *locus* peut être vu comme étant l'indice de la variable dont le domaine est associé au gène). L'opérateur de croisement proposé entre deux individus parents $_{père}{}^{1}$ et $_{père}{}^{2}$ fonctionne de la manière suivante :

1. Pour chaque locus i ($i \in [1...n]$), on réalise l'union $G_{i\,père}{}^{U} = G_{i\,père}{}^{1} \cup G_{i\,père}{}^{2}$
2. On choisit aléatoirement une partie de $G_{i\,père}{}^{U}$ de taille appropriée, qui constituera $G_{i\,fils}{}^{1}$
3. On prend les valeurs restantes de l'union des gènes des pères $G_{i\,père}{}^{U} - G_{i\,fils}{}^{1}$ pour constituer la première partie de $G_{i\,fils}{}^{2}$
4. Si nécessaire, on complète $G_{i\,fils}{}^{2}$ avec des valeurs choisies aléatoirement dans $G_{i\,fils}{}^{1}$

Ce croisement peut également être modifié afin d'inclure un certain « guidage », fonction de la solution obtenue pour les parents. En effet, il peut être profitable de tenter d'influencer les gènes des enfants en les faisant se rapprocher des valeurs qui constituaient les solutions des parents. Attention toutefois à ne pas abuser de telles méthodes car elles minimisent la diversité de la recherche en contraignant artificiellement l'espace exploré.

Pour mettre en œuvre cette modification, il suffit par exemple, avant l'étape *2*, d'inclure *a priori* dans $G_{i\,fils}{}^{1}$ et $G_{i\,fils}{}^{2}$ les valeurs des solutions (s'il y en avait) des deux parents.

2.3.1.3.2 Mutation

Dans notre cas, l'opération de mutation reçoit comme rôle de fournir à un gène de l'individu la possibilité de parcourir l'ensemble du domaine de la variable à laquelle il est associé.

Pour cela, et puisque la taille d'un gène est toujours constante, la seule méthode est de sélectionner au hasard des valeurs dans le domaine de la variable considérée.

Concernant ce principe, on peut également formuler la même remarque que pour l'opérateur de croisement : si l'on souhaite « guider » les gènes mutés vers les solutions des gènes d'origine, il suffit d'insérer *a priori* dans le nouveau gène la solution de l'ancien, et ensuite seulement de compléter aléatoirement. Notons toutefois qu'au moment de l'application de l'opérateur de mutation, on ne connaît pas forcément la solution de l'individu concerné (en effet, le cycle classique d'un algorithme génétique est *sélection – croisement – mutation - évaluation*), et cela supposerait donc de procéder au préalable à une évaluation. Par ailleurs, les mêmes réserves que pour l'opérateur de croisement sont applicables à ce procédé.

2.3.2 Méthodes parallèles

Une méthode de ce type est décrite dans [CleHogHub92]. Elle vise à utiliser plusieurs solveurs constructifs travaillant simultanément sur le même CSP. Lorsque la coopération entre ces solveurs est correctement assurée, le gain de performances est *plus* que linéairement fonction de la quantité d'acteurs.

Les principes généraux de cette méthode sont les suivant :

1. Les solveurs attaquent le problème par des bouts différents afin de créer une certaine diversité, nécessaire pour rendre profitable la coopération entre eux. Par exemple, on peut les obliger à utiliser un ordre d'instanciation des variables différent (il peut suffire pour cela de départager aléatoirement les *ex æquo* d'une heuristique de choix des variables).
2. Tous les solveurs ont accès à un « tableau » (*blackboard*) qui liste des « astuces » (*hints*). Ces astuces sont des instanciations partielles *recommandables* – c'est-à-dire des instanciations partielles qui ont une bonne probabilité de se retrouver dans une solution du problème.
3. Lorsqu'un solveur trouve une instanciation partielle qu'il considère comme *recommandable*, il l'inscrit sur le tableau.
4. Lorsqu'un solveur se trouve dans une situation où il pourrait bénéficier d'une des astuces du tableau, il choisit, avec une certaine probabilité, cette astuce.

Bien sûr, les points 2, 3 et 4 doivent être précisément définis au cas par cas. Néanmoins, on peut *a priori* dégager les indications suivantes :

- Un solveur peut utiliser une astuce lorsqu'il cherche à instancier une variable qui figure dans cette astuce. Ainsi, ce système peut être vu comme une sorte d'heuristique de choix de valeur. Le problème se pose de gérer les éventuels conflits entre l'instanciation partielle courante du solveur et les autres variables figurant dans l'astuce. On peut par exemple décider qu'une astuce n'est utilisable que si elle ne rentre pas en conflit avec plus d'un certain nombre de variables de l'instanciation partielle courante.
- Les « astuces » peuvent également être des « écueils » (ou *nogoods* : ce sont des instanciations partielles à éviter car elles ne mènent jamais à aucune solution.) Ces écueils sont parfois connus au fur et à mesure de la progression de la recherche. Dans ce cas, dès qu'un écueil est détecté, il est mis au tableau, et aucun solveur ne doit chercher à travailler sur une instanciation partielle comportant un des écueils du tableau.
- Un solveur peut considérer qu'une instanciation partielle est « recommandable » (et donc en faire une astuce) lorsqu'il travaille sur cette instanciation depuis un temps assez long, sans avoir fait de *backtrack*. Si un *backtrack* intervient par la suite, il peut retirer l'astuce du tableau.
- A la limite, cette astuce peut même ne comporter qu'une seule variable : dans ce cas, le principe mis en œuvre se rapproche d'un système à base de *fourmis* [Cost94], où les différents acteurs vont peu à peu chercher dans la voie qui a déjà été plébiscitée par le plus grand nombre.

Ce type de méthode demande à l'évidence un temps de mise au point considérable avant d'arriver à un résultat. Il a toutefois l'avantage de pouvoir se greffer sur n'importe quelle autre type de méthode de résolution.

2.3.3 Bornes inférieures par relaxation

Cette méthode [LobLem97] constitue plutôt une « astuce » qu'un réel système de résolution de CSP : elle peut être ajoutée à n'importe quel autre procédé existant afin de fournir une ouverture permettant éventuellement d'améliorer les performances.

Son principe repose sur la constatation qu'un problème relaxé est plus facile à résoudre que son original. Si l'on peut postuler qu'une très bonne solution du problème relaxé est peu éloignée (peu d'instanciations différentes) d'une bonne solution du problème complet, alors il est vraisemblable que, lors d'une résolution complète, le fait d'explorer en priorité les branches qui ont mené aux meilleures solutions du problème relaxé

constituera une bonne heuristique. On suppose également que le temps investi dans la résolution du problème relaxé est compensé par le gain de temps effectué lors de la résolution du problème complet.

Le principe peut également être étendu en un système à plusieurs phases : on relaxe de moins en moins le problème, en effectuant à chaque étape une résolution permettant de déterminer les chemins à essayer en premier lors de la résolution suivante.

2.4 Discussion / comparaison

2.4.1 Tableau récapitulatif des différentes méthodes

Abréviation	Nom de la méthode	Type de méthode	Caractéristiques pouvant rendre cette méthode inapplicable
VAL/TA	**Choix des valeurs à base de test avant**	Heuristique de choix des valeurs	• Domaine des variables trop restreint • Coût du « test avant » trop important
VAL/EM	**Choix des valeurs et évaluations mathématiques**	Heuristique de choix de valeurs	• CSP trop éloigné d'un CSP binaire • Mise en consistance opérée par l'algorithme de résolution trop forte • Domaine des variables trop restreint • Coût du « test avant » trop important
VAR/DOM	**Choix des variables dynamique basé sur les domaines**	Heuristique de choix des variables	
VAR/VAL	**Choix des variables dynamique basé sur les valeurs**	Heuristique de choix des variables	• CSP trop éloigné d'un CSP binaire • Mise en consistance opérée par l'algorithme de résolution trop forte • Coût du « test avant » trop important
LDS	**Limited Discrepancy Search**	Méthode constructive alternative	• Heuristique de choix des valeurs peu fiable en fin de construction de la solution
AG	**Hybride d'algorithme génétique**	Méthode incomplète utilisant une méthode constructive	• Eloignement des solutions de qualité similaire
PAR	**Méthodes parallèles**	Système d'optimisation pour les méthodes constructives	
INF	**Bornes inférieures par relaxation**	Système d'optimisation pour les autres méthodes	• Eloignement des solutions d'un problème relaxé et de l'original

2.4.2 Tableau donnant l'adéquation des méthodes entre elles

Le tableau suivant donne une appréciation de l'adéquation des méthodes entre elles, dans le cas d'une utilisation combinée de deux de celles-ci.

Adéquation entre les méthodes	**VAL/EM**	**VAR/ DOM**	**VAR/ VAL**	**LDS**	**AG**	**PAR**	**INF**
VAL/TA	Combinaison indispensable pour VAL/EM	Possible	Possible	Supposée bonne : VAL/TA devrait être plus fiable en fin d'arbre	Possible	Possible	Possible
VAL/EM		Possible	Supposée très bonne puisque VAL/EM réutilise les données de VAR/VAL	Supposée bonne : VAL/EM devrait être plus fiable en fin d'arbre	Possible	Possible	Possible
VAR/DOM			Impossible	Possible	Possible	Possible	Possible
VAR/VAL				Possible	Possible	Possible	Possible
LDS					Possible	Possible	LDS peut être particulièrement intéressant dans les premières phases de INF
AG						Possible	Possible
PAR							Possible

2.4.3 Décision sur les méthodes à implémenter

Les méthodes suivantes sont retenues pour être adaptées et évaluées dans le cadre de MAXIME:

1. Choix des valeurs à base de test avant : Cette méthode semble pouvoir être mise en œuvre assez rapidement, notamment parce que Ilog Solver utilise le *test avant* comme stratégie de base.
2. Limited Discrepancy Search : Cette méthode semble également aisée à implémenter car elle ne fait que modifier l'ordre d'instanciation des variables.
3. Hybride d'algorithme génétique : Ce système est original et il est intéressant de l'évaluer même s'il semble plus adapté à des problèmes peu contraints
4. Méthodes parallèles : Ce système, également original, permettra de juger de l'efficacité intuitive d'une méthode coopérative.

La méthode de *choix des valeurs par évaluation mathématique* semble poser trop de problèmes de modélisation dans notre cas, et ne sera donc probablement pas efficace.

Enfin, la méthode à *bornes inférieures par relaxation* est écartée car nous disposons déjà pour MAXIME d'heuristiques génériques performantes et qui ne nécessitent pas, elles, de résolution préalable d'un problème.

3 IMPLEMENTATION DANS MAXIME

3.1 Analyse de l'existant

3.1.1 MAXIME

3.1.1.1 Présentation

MAXIME est une application permettant d'effectuer la planification des personnels au sol des Aéroports De Paris (ADP).

La planification de tâches consiste à affecter des tâches sur des vacations. MAXIME procède pour cela en deux temps :

1. Affectation « brute » des tâches sur les vacations, en respectant certaines contraintes
2. Amélioration de l'emploi du temps ainsi constitué

La seconde phase, dont nous ne traiterons pas dans ce document, permet d'augmenter la qualité de l'emploi du temps « brut » en uniformisant les charges des différents agents ou en augmentant le nombre de plages « repas ». Les algorithmes utilisés effectuent ainsi de la recherche locale (Tabu, A*) autour de la solution de base.

La première phase est celle qui nous intéresse dans ce document. Elle utilise la résolution de problèmes sous contraintes (CSP), au travers de l'algorithme fourni par Ilog Solver 4.3 (*cf. chapitre 3.1.2 : ILOG SOLVER version 4.3*).

Chaque tâche est représentée par une variable, dont la valeur est le numéro de la vacation sur laquelle cette tâche est affectée. Le domaine de cette variable est donc un sous-ensemble de la liste des vacations. Réciproquement (pour faciliter certains calculs), chaque vacation est représentée par un ensemble de valeurs donnant la liste des tâches affectées à cette vacation. Cette seconde modélisation (modèle *dual*), n'a pas de rôle fonctionnel.

Les contraintes entre les différentes variables (tâches) sont complexes (problème des repas, des temps de déplacement géographiques, etc…) Toutefois, nous nous intéresseront essentiellement à deux types de contraintes, qui influent de façon particulièrement importante sur la difficulté de résolution du problème :

1. Les contraintes de *non-recouvrement*, qui interdisent à deux tâches de se trouver sur la même vacation (elles ne peuvent pas être effectuées par le même agent en même temps).
2. Les contraintes de « cumul », qui interdisent de trop charger une vacation, pour laisser la place à des tâches « repas », etc…

MAXIME utilise un système de relaxation des contraintes par ajout de « valeurs fictives » (*cf. chapitre 2.1.3.2 : Valeurs fictives et relaxation de contraintes*) qui permet de trouver rapidement une solution incomplète dans le cas d'un problème difficile. Pour cela, une vacation « fictive » différente est ajoutée à chaque tâche à traiter. Une tâche ne peut être affectée à cette vacation que lorsque toutes les autres possibilités ont été testées (dans l'état où se trouve le solveur au moment de procéder à l'affectation de la tâche, celle-ci ne peut être placée sur aucune vacation réelle). Par ailleurs, la fonction objectif pénalise (croît) les solutions qui utilisent le plus de vacations fictives. Notons qu'avec ce système, le solveur trouve toujours une solution dès la première série d'instanciations (sans défaire plus d'une affectation à la suite) puisqu'au moins *une* valeur est toujours disponible pour chaque variable. Cependant, la solution trouvée peut être « incomplète » : certaines tâches peuvent être placées sur des vacations fictives. Seul un second essai d'instanciation du solveur peut permettre d'obtenir une solution de meilleure qualité utilisant moins de vacations fictives.

Notons également que MAXIME utilise une heuristique de choix de variables qui traite en premier les variables dont les domaines sont les plus petits (*cf. chapitre 2.2.3.1 : Heuristique dynamique basée sur les domaines*).

La fonction objectif utilisée par MAXIME (fonction qui permet de « noter » la qualité d'une solution) réagit de la façon suivante :

1. Elle décroît (valorise la solution) lorsque certaines vacations restent totalement libres
2. Elle croît (pénalise la solution) lorsque certaines tâches restent non-affectées (affectées à des vacations fictives)

3.1.1.2 Evaluation des contraintes du CSP

Ce chapitre a pour but de définir les éléments nécessaires à l'application éventuelle du principe de [Hogg9x] (*cf. chapitre 2.2.2 : Choix des valeurs et évaluations mathématiques*).

Dans la suite, *K* définit la taille du domaine d'une variable. Cette valeur correspond plus ou moins au nombre de vacations (on néglige alors la valeur fictive de repli et les vacations « impossibles » pour cette tâche.)

Les contraintes exposées dans MAXIME sont les suivantes :

1. **Les contraintes de non-recouvrement de tâches.** Ces contraintes sont définies dans la méthode *postContrainteNonRecouvrement (planif_planning.cpp).* Ce sont des contraintes binaires d'inégalité générant donc, dans ce modèle, un taux de contrainte approximatif de $\frac{K}{K^2}$, soit $\frac{1}{K}$. La valeur exacte de ce taux est toutefois difficile à prévoir puisque l'intersection des domaines des deux variables impliquées n'est pas forcément vide. Il peut donc y avoir des valeurs de l'une des variables qui n'interdisent aucune valeur de l'autre.
2. **Les contraintes forçant certaines tâches sur des vacations connues à l'avance (verrous)**. Ces contraintes sont définies dans la méthode *postContraintesVerrou (planif_planning.cpp)*. Ces contraintes forcent une valeur constante prédéfinie pour certaines variables. Elles n'introduisent donc pas à proprement parler de *taux de contrainte*. Par contre, elles réduisent le domaine des variables concernées à la valeur constante imposée (le domaine devient de taille *1*).
3. **Les contraintes de maintien en cohérence du modèle dual**. Ces contraintes ne font qu'alimenter le modèle dual lors de leur propagation. Elles ne chargent pas le problème. Elles sont définies dans la méthode *postContrainteCohérence (planif_planning.cpp).*
4. **Les contraintes sur le minimum de tâches repas/dîner**. Ces contraintes ne sont pas binaires (elles effectuent la totalisation du nombre de variables ayant une certaine valeur) et ne seront donc pas prises en compte ici. Elles figurent dans la méthode *postContrainteRepas (planif_planning.cpp).*
5. **Les contraintes sur la faisabilité des tâches mobiles (contraintes de « cumul »)**. Ces contraintes sont définies à l'aide du modèle dual et ne seront donc pas prises en compte ici. Elles figurent dans la méthode *InitVariables (planif_planning.cpp).*

3.1.2 ILOG SOLVER version 4.3

Ilog Solver v. 4.3 est un ensemble de librairies C++ dédiées à la résolution constructive de CSP.

La méthode utilisée est du type *test avant*, accompagnée d'un *backtracking* simple. Le niveau de test avant exactement pratiqué n'est pas connu avec précision, mais l'algorithme utilisé est proche d'AC-3 et utilise la *propagation de contraintes*. Le résultat de ce système, après instanciation de chaque variable, est probablement de rendre le CSP proche de l'arc-consistance (consistance de niveau 2).

Le tableau suivant rassemble quelques points caractéristiques de cet outil, en tentant de montrer quelles sont les implications positives et négatives de ces aspects :

Caractéristique	Conséquences positives	Conséquences négatives
La propagation des contraintes est totalement câblée en interne, et inaltérable. Le déclenchement des échecs est automatique lorsque le domaine d'une variable devient vide.		• Impossibilité de savoir quelles variables et quelles valeurs déclenchent un échec • Impossibilité, donc, de connaître les instanciations partielles inconsistantes minimales • Impossibilité de réduire la propagation à un niveau moindre que celui câblé
Il est possible d'effectuer un *backjump*, sur demande, vers n'importe quel nœud déjà exploré de l'arbre (méthode *IlcManager::fail(IlcAny)*).	• Je ne vois pas d'application directe de cette fonctionnalité. Toutes les méthodes qui prédisent vers quel nœud il faut effectuer un *backjump* ont besoin de savoir quelle est, dans l'instanciation partielle courante, l'instanciation partielle minimale qui est inconsistante. Or, ceci est impossible…	
Les contraintes sont définissables de façon très fine, à l'aide de code C++ indiquant quand et comment réduire les domaines des variables. Ainsi, on bénéficie d'un pourvoir d'expression qui va bien au-delà d'un formalisme mathématique.	• Le modèle du problème se rapproche bien plus facilement de la réalité qu'avec des définitions de contraintes standards	• Il devient très difficile d'opérer des calculs probabilistes mathématiques sur une application déjà existante, car les contraintes ne peuvent plus être modélisées par des expressions mathématiques
Il est possible, à n'importe quel moment, de demander à poursuivre la résolution dans une voie d' « essai » qui sera ensuite annulée (ou pas), et qui aura permis de voir ce qui se serait passé « si … » (méthode *IlcManager::solve(IlcGoal)*)	• Ceci est très pratique pour faire de la prospective « en avant » et qui n'engage à rien.	• Lorsqu'on opère plusieurs essais, il reste impossible, si *une* des prospectives se révèle être fructueuse, de restaurer l'état où l'on était arrivé à la fin de celle-ci afin de ne pas avoir à refaire toutes les instanciations et propagations déjà effectuées

3.2 Standard de comparaison

Dans la suite, les expérimentations ont été effectuées sur les problèmes de la série « *pp* » utilisée par Antoine Charbonneau. Cette série propose des problèmes allant de 50 à 600 tâches à planifier, et de 18 à 216 vacations. Le ratio du nombre de tâches sur le nombre de vacations est toujours de 2,78.

Pour simuler des problèmes plus importants, les définitions de ces 600 tâches et 216 vacations ont été dupliquées plusieurs fois jusqu'à obtenir le nombre d'éléments désirés. Le ratio du nombre de tâches sur le nombre de vacations a toujours été maintenu à 2,78. Les définitions d'origine se basant elles-mêmes sur des répétitions d'une série de 300 tâches et de 100 vacations, ces duplications supplémentaires n'entraînent pas de déséquilibre dans la répartition des tâches et des vacations dans le temps (cette affirmation a été graphiquement vérifiée).

La série de problèmes de référence suivante a donc été créée selon ce principe :

Nombre de variables	Nombre de vacations
200	72
400	144
500	180
600	216
800	288
1000	400
1200	432

Dans la suite de ce document, les problèmes sont désignés par le nombre de tâches qu'ils mettent en jeu. Il s'agit systématiquement des problèmes figurant dans ce tableau.

Les solutions considérées dans le cadre de ce document sont toujours les **premières solutions trouvées par les algorithmes**. Ce choix a été fait pour les raisons suivantes : ce dossier se propose d'évaluer de nouvelles méthodes dont nous ne connaissons donc pas a priori les performances, tant en terme de qualité des résultats qu'en terme de temps de calcul. Nous ne pouvons donc pas stopper nos algorithmes « lorsqu'ils arrivent à une solution d'une certaine qualité » ou alors « après un certain temps de calcul » puisqu'il est vraisemblable que certains de ces algorithmes trouveront une *solution médiocre en un temps très court* (puis ne trouveront plus rien) tandis que d'autres trouveront une *très bonne solution en un temps très long* (alors qu'ils n'avaient rien trouvés avant). De plus, il est impossible de « poser une limite » dans le cadre d'une application concrète comme MAXIME où l'on ne connaît pas à l'avance la difficulté du problème posé – et donc le temps nécessaire (et acceptable) pour le résoudre : dès lors, un algorithme qui proposerait successivement plusieurs solutions en laissant planer le doute sur une éventuelle amélioration de leur qualité serait pratiquement inutile.

3.3 Nature des ajouts

3.3.1 Choix des valeurs à base de test avant

3.3.1.1 Génération de l'arbre des solutions

Il faut, avant chaque instanciation d'une nouvelle variable, essayer l'ensemble des valeurs possibles et les trier suivant leur « qualité ». Ainsi, la « meilleure » valeur est essayée en premier, puis, en cas d'échec, la suivante est prise, etc... Ce système n'est pas directement utilisable avec Solver, car le principe de sélection de valeur proposé permet uniquement de choisir la valeur à utiliser parmi celles restantes, au moment où l'instanciation doit être effectuée. Il faudrait donc, après chaque échec, ré-évaluer toutes les valeurs restantes pour en sélectionner une, alors que ce travail a déjà été fait (avec une valeur de plus) à l'étape précédente.

Pour parer à ce problème, il faut ré-écrire les *buts* (*goals)* de génération et d'instanciation (*IlcGenerate* et *IlcInstantiate*). Dans le but de génération, juste après le choix de la variable, on effectue alors les opérations suivantes :

- Evaluation de chacune des valeurs possibles pour la variable (à ce stade, *toutes* les valeurs du domaine d'origine sont encore présentes). On peut stocker les valeurs et les « notes » associées dans une liste.
- Classement des valeurs de la liste depuis la meilleure jusqu'à la moins bonne.

Dans le but d'instanciation, on essaie alors les valeurs dans l'ordre de la liste.

3.3.1.2 Evaluation des valeurs

L'évaluation d'une valeur se comporte normalement de la façon suivante : une valeur qui engendre peu de réductions de domaines dans le reste des variables est considérée comme bonne ; une valeur qui réduit beaucoup les domaines des autres variables est considérée comme mauvaise. Pour départager les *ex aequo*, l'heuristique de *tassement* déjà utilisée dans MAXIME (et qui était à l'origine la stratégie de base) à été choisie. Ce système a en outre été affiné en tenant plus précisément compte de certaines contraintes.

3.3.1.2.1 Propagation câblée

Dans un premier temps, cette évaluation a été implémentée en utilisant la propagation des contraintes câblée dans Ilog Solver : la méthode *IlcManager::solve* était utilisée pour affecter chacune des valeurs possibles à la variable, puis un second *but* permettait de totaliser la taille des domaines restants pour les variables non-encore instanciées. Entre temps, Solver effectuait de lui-même la propagation de l'affectation.

Cette méthode fonctionne mais offre de très mauvaises performances, à cause des temps de propagation énormes exigés par Ilog Solver. Rappelons que ce biais avait été mentionné comme possible lors de l'étude théorique (*cf. chapitre 2.4.1 : Tableau récapitulatif des différentes méthodes*) :

Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**Heuristique de choix des valeurs à base de test avant – propagation câblée**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution
200	1845	2114	**5,0**	-20	2290	265	**12,3**	-17
400	6929	7426	**33,1**	2393	8409	500	**91,3**	1964
500	10875	11524	**64,3**	-52	13998	647	**174,7**	-50
600	15490	16259	**110,5**	-62	19354	765	**305,7**	-60

Notons toutefois que le nombre de nœuds explorés diminue considérablement (de 8 à 21 fois) avec cette méthode, ce qui prouve son efficacité dans le parcours de l'arbre.

3.3.1.2.2 Forward-Checking par programmation

3.3.1.2.2.1 Contraintes de non-recouvrement

Dans MAXIME, une grande partie des contraintes sont des contraintes de *non-recouvrement* entre tâches : deux variables représentant des tâches ne peuvent pas prendre la même valeur – c'est-à-dire que les tâches ne peuvent pas se trouver sur la même vacation. Ces contraintes sont donc des contraintes binaires d'inégalité entre variables.

Dans une contrainte binaire d'inégalité, lorsque l'une des variables est instanciée, l'autre voit son domaine se réduire d'*une* valeur (si la valeur instanciée était dans le domaine de cette autre variable). Dans le cas (et seulement dans *ce* cas) où son domaine ne comportait plus que deux valeurs, elle est également instanciée. Enfin, si cette seconde variable était déjà instanciée, il ne se passe rien.

Ainsi, pour que l'instanciation d'une variable concernée par une contrainte binaire d'inégalité déclenche d'autres instanciations en cascade, il faut absolument que la seconde variable n'ait plus qu'exactement deux valeurs possibles – ce qui est relativement rare. Si l'on connaît, pour chaque variable, la liste des variables potentiellement en conflit via des contraintes d'inégalité, on peut très facilement obtenir une bonne approximation du nombre de valeurs éliminées dans les autres variables lors d'une instanciation : il suffit de compter le nombre de variables en conflit non-encore instanciées, et qui ont encore la valeur choisie dans leur domaine.

Ilog Solver ne permet pas de retrouver les variables impliquées dans les contraintes qui sont définies pour le problème. Il faudra donc rajouter un membre dans l'objet *PLANIF_Tache* : il comportera la liste des tâches en conflit avec l'objet considéré. Cette liste sera alimentée en même temps que les contraintes d'inégalité seront ajoutées au problème. Pour évaluer une valeur possible *v* d'une variable, on utilisera l'algorithme suivant :

1. Initialiser *note* avec le nombre de variables en conflit avec celle à traiter
2. **Pour** toutes les variables *x* en conflit avec celle à traiter, **faire**
3. **Si** *x* n'est pas encore instanciée
4. **Si** le domaine de *x* comporte *v*
5. *note* = *note* – 1

Finalement, *note* est donc d'autant plus faible qu'il y a de variables dont le domaine sera réduit d'*une* valeur – ce qui est bien le résultat attendu. Ce système approximatif ne traite pas les instanciations en cascade, mais n'utilise pas la propagation très coûteuse en temps d'Ilog Solver.

Les résultats suivants ont été obtenus :

Tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM), avec 1 échantillon								
Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**Heuristique de choix des valeurs à base de test avant – Forward-Checking pour les contraintes de non-recouvrement**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution
200	1845	2114	**4,977**	-20	1738	2003	**4,536**	-17
400	6929	7426	**33,619**	2393	6130	6630	**27,320**	1964
500	10875	11524	**60,207**	-52	9859	10506	**56,601**	-50
600	15490	16259	**113,443**	-62	14099	14864	**94,896**	-60
800	28333	29378	**273,774**	-95	24654	25695	**249,669**	-70

Les temps de calcul avec cette méthode sont toujours plus faibles qu'avec la stratégie de base. Pour les problèmes insolubles (où toutes les tâches ne peuvent pas être affectées, comme celui à 400 tâches), la qualité de la solution est meilleure. Pour les problèmes solubles, la qualité de la solution est moins bonne, mais il faut garder à l'esprit que la fonction objectif de MAXIME gratifie le fait que certaines vacations restent libres, ce qui avantage artificiellement l'heuristique de tassement puisque celle-ci a pour but de « tasser » les tâches sur les vacations.

3.3.1.2.2.2 Ajout de la contrainte de cumul

Dans MAXIME, une contrainte (*cf. chapitre 3.1.1.2 : Evaluation des contraintes du CSP*) vérifie la faisabilité de certaines tâches « mobiles » en calculant, à chaque instanciation, un indice de « cumul » sur la vacation. Cet indice ne doit pas être supérieur à un seuil fixé à l'avance (*100*), et augmente avec le nombre de tâches affectées à cette vacation.

On peut intégrer cette contrainte dans l'heuristique de test avant, de la façon suivante : pour chaque valeur possible de la variable à instancier, on calcule le cumul qui serait généré sur la vacation d'affectation avant et après l'instanciation de la variable. La préférence est donnée aux valeurs qui minimisent l'augmentation du cumul. Ainsi, la mesure *cumul avant – cumul après* peut être utilisée comme note pour une valeur (plus la note est élevée, plus la valeur est intéressante).

De plus, on peut directement éliminer du domaine de la variable les valeurs dont le cumul après instanciation est supérieur au seuil. Ceci permet également de retirer l'étape de calcul du cumul dans la contrainte qui la gérait jusque là (PLANIF_MyConstraintI).

Tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM), avec 1 échantillon								
Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**Heuristique de choix des valeurs à base de test avant – Forward-Checking pour les contraintes de non-recouvrement et de cumul**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution
200	1845	2114	**4,977**	-20	0	264	**1,462**	-18
400	6929	7426	**33,619**	2393	26	522	**6,970**	1963
500	10875	11524	**60,207**	-52	1	647	**11,507**	-52
600	15490	16259	**113,443**	-62	4	771	**18,898**	-58
800	28333	29378	**273,774**	-95	12	1056	**50,593**	-90
1000	44740	46001	**581,676**	3126	0	1265	**109,608**	3130
1200	63896	65439	**1240,100**	-128	6	1544	**203,343**	-122

Les performances sont excellentes par rapport à la méthode de référence (rapport de 1/3 à 1/6 pour les temps de calcul) pour des résultats pratiquement identiques. On peut noter que la qualité est toujours meilleure que celle obtenue en ne considérant que les contraintes de non-recouvrement (*cf. tableau précédent*). De plus, les bons résultats obtenus sur le problème insoluble se confirment.

3.3.1.2.2.3 Ajout d'un facteur hasard

Comme on peut le constater dans les résultats précédents, le nombre de nœuds visités a chuté de façon impressionnante : on observe une réduction d'un facteur 8 à 36 ! Toutefois, le temps de calcul ne diminue pas

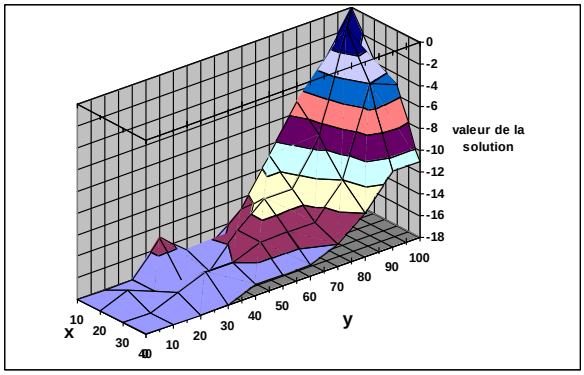
dans les mêmes proportions (facteur 3 à 6 seulement). Ceci est dû au temps requis par l'heuristique pour évaluer les différentes valeurs avant de les ordonner, qui est plus important puisque l'algorithme est plus complexe.

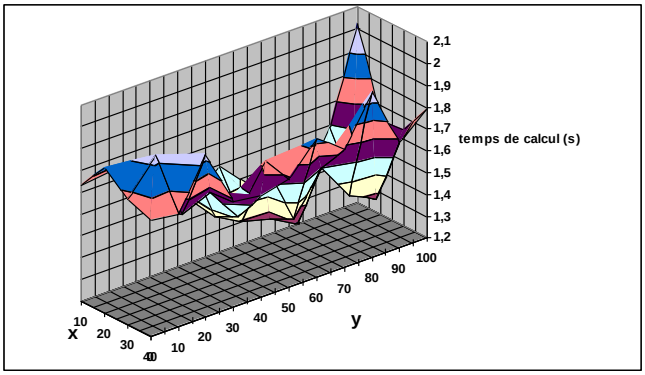
Dans la mesure où la taille des domaines des variables est importante à un certain moment de la résolution (et donc qu'il y a beaucoup de choix dans la façon d'instancier les variables, c'est-à-dire qu'il y a peu de chances d'arriver à une impasse – rappelons que les variables sont choisies par ordre croissant de taille des domaines), on peut se demander s'il est nécessaire d'utiliser une heuristique aussi sophistiquée et coûteuse alors qu'une valeur à peu près acceptable ferait probablement l'affaire. Nous allons donc introduire un paramètre entier *x* qui fixera une limite dans la taille des domaines, au-delà de laquelle les valeurs ne seront plus ordonnées par l'heuristique mais simplement au hasard. Une exception est toutefois faite pour la valeur fictive, qui est quand même systématiquement placée en dernier.

En appliquant simplement ce procédé, on s'aperçoit que certaines des valeurs choisies sont réellement catastrophiques pour le reste du processus (elle ne sont pas « à peu près acceptables » mais réellement mauvaises). On ne va donc plus ordonner *toutes* les valeurs au hasard, mais seulement un pourcentage fixé par un second paramètre *y*. La valeur fictive est toujours laissée en dernière position. Notons qu'avec ce principe, il faut conserver la contrainte « dure » vérifiant le cumul, car certaines valeurs prises au hasard peuvent la violer.

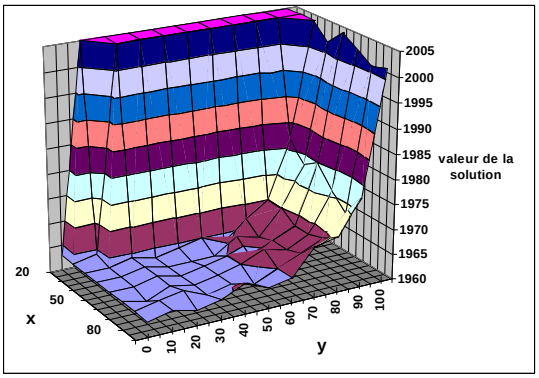
On obtient les résultats suivants, en faisant varier *x* et *y* (tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM), avec 1 échantillon) :

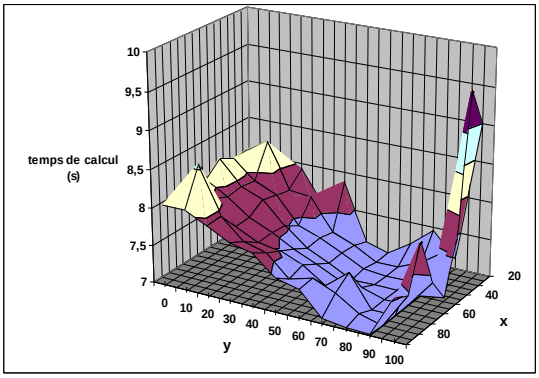
1. *Problème à 200 tâches*



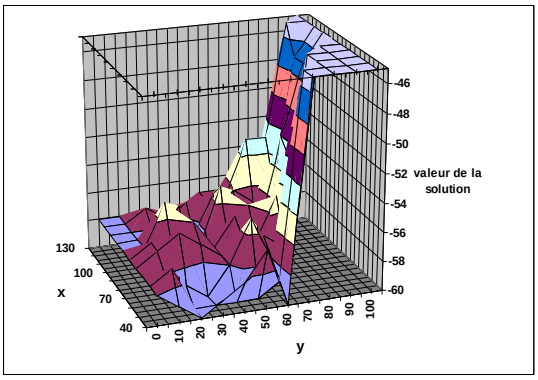


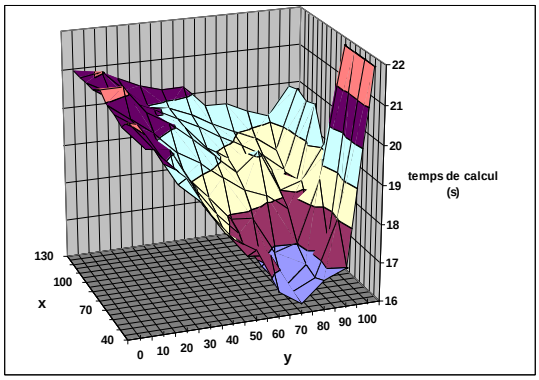
2. *Problème à 400 tâches*



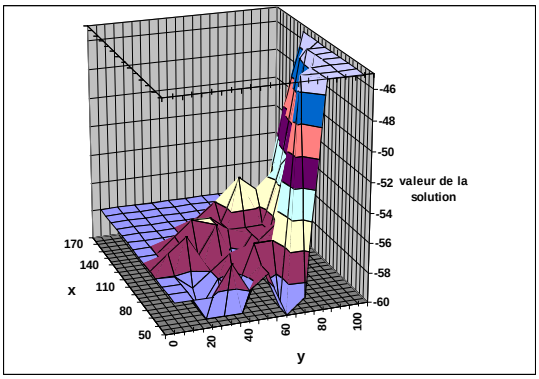


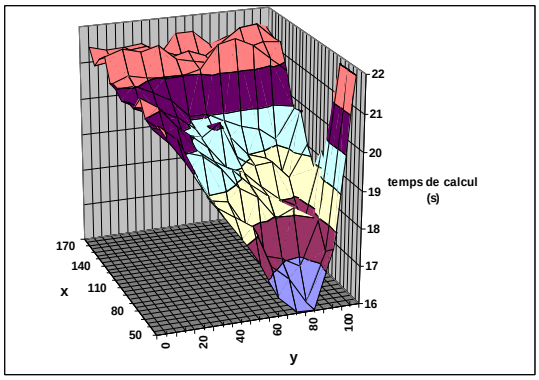
3. *Problème à 600 tâches*



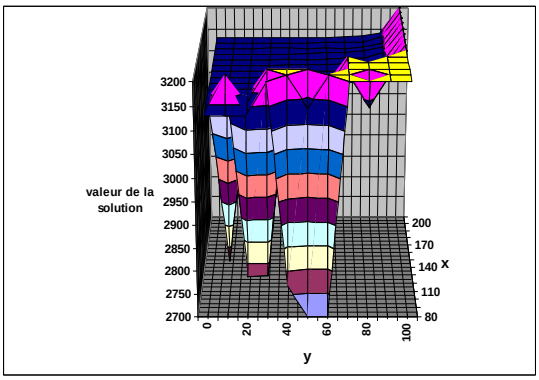


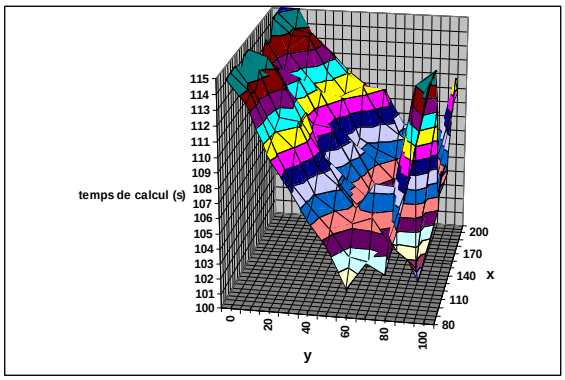
4. *Problème à 800 tâches*



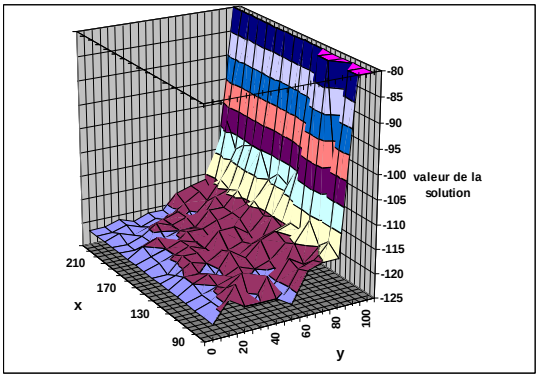


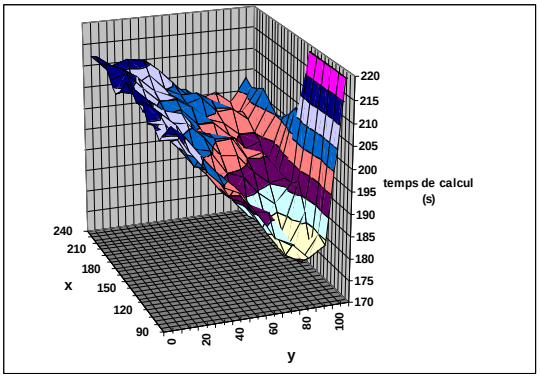
5. *Problème à 1000 tâches*





6. Problème à 1200 tâches





Les points suivants apparaissent :

- Pour des valeurs de *y* inférieures à un seuil visible de façon évidente, *x* influence assez peu la valeur de la solution (sauf s'il est radicalement trop petit, comme on peut le voir sur le problème à 400 tâches).
- Dans cette plage de valeurs de *y*, la qualité de la solution est d'autant plus stable par rapport à *y* que *x* est grand (autrement dit, la valeur exacte prise pour *y* a d'autant moins d'importance que *x* est grand).
- Lorsque *x* est assez petit, les temps de calculs passent clairement par un minimum en fonction de *y*. Ces temps croissent généralement avec *x* (mais de façon moins prononcée lorsque le problème n'a pas de solution, cf. les problèmes à 400 et 1000 tâches).
- La qualité des solutions est largement moins bonne que celle obtenue par l'heuristique « complète ». Cependant, les temps de calcul sont également plus faibles.

On peut donc en tirer les conclusions suivantes sur la stratégie à adopter :

- Il faut prendre un *x* le plus petit possible afin de minimiser les temps de calcul. Ce *x* doit toutefois être suffisamment grand pour qu'une « erreur » sur *y* ne soit pas potentiellement catastrophique pour la qualité de la solution. Il faut donc définir un seuil de stabilité acceptable.
- Il faut prendre *y* le plus proche possible de la valeur donnant le temps de calcul le plus faible. Toutefois, cette valeur doit être assez faible pour rester dans la plage des solutions de qualité acceptable.

La valeur *70* pour *y* convient dans toutes les situations. Pour *x*, on prendra le nombre de tâches du problème divisé par *10*. On obtient alors les résultats suivants :

Tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM), avec 1 échantillon								
Nombre de tâches du problème	**Heuristique de choix des valeurs à base de test avant – Forward-Checking pour les contraintes de non-recouvrement et de cumul**				**Heuristique de choix des valeurs à base de test avant – Forward-Checking pour les contraintes de non-recouvrement et de cumul, avec classement aléatoire**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution
200	0	264	**1,462**	-18	0	272	**1,292**	-13
400	26	522	**6,970**	1963	194	691	**6,820**	1967
600	4	771	**18,898**	-58	15	788	**16,154**	-58
800	12	1056	**50,593**	-90	0	773	**17,966**	-56
1000	0	1265	**109,608**	3130	1122	2391	**105,722**	3852
1200	6	1544	**203,343**	-122	102	1646	**185,627**	-115

Les temps de calcul sont à peine plus faibles en rajoutant ce système (de 2% à 15%), sauf pour le problème à 800 tâches (amélioration de 64%), qui fait figure d'exception.

Comme l'introduction d'un facteur hasard amène une instabilité au niveau des performances, nous considérerons que ce dernier système n'est pas intéressant vu le gain de performances négligeable dans le cas général.

3.3.2 Hybride d'algorithme génétique

Ce système se base sur le principe exposé en 2.3.1. A l'origine, il en reprend toutes les caractéristiques en matière d'initialisation de la population et de comportement des opérateurs.

3.3.2.1 Conception du système

Deux nouveaux objets sont créés au sein de l'application :

- *PLANIF_Indigen* représente un individu génétique. Il possède les caractéristiques suivantes :
    - Il peut être construit d'après les données initiales du problème. Ce mécanisme est utilisé pour générer la population initiale.
    - Il peut également être construit d'après deux autres individus. Ce constructeur joue alors le rôle d'opérateur de croisement (*cf. ci-dessous*).
    - Il peut se *muter* via une méthode (sans arguments).
    - Il peut s'*évaluer* via une méthode (sans arguments). Cette méthode effectue alors la résolution du CSP représenté par l'individu.

    Par ailleurs, chaque individu stocke ses gènes – c'est-à-dire les valeurs qui lui sont attribuées dans chaque domaine des variables.
- *PLANIF_Genetique*, qui joue le rôle de *manager* pour l'ensemble de la population génétique. Il permet de :
    - Construire la population initiale (dans son propre constructeur)
    - Lancer la résolution génétique pour un certain nombre de générations. Il effectue alors les tâches suivantes en boucle, chaque cycle représentant une génération :
        - Sélection des meilleurs individus afin de garder une population de taille constante. Les individus étant toujours triés par ordre de qualité décroissante, il suffit d'éliminer les derniers de la liste.
        - Croisement de certains individus, par construction de nouveaux individus à partir de leurs deux parents
        - Mutation de certains individus
        - Evaluation de l'ensemble de la population et classement des individus par ordre de qualité décroissante

3.3.2.2 Opérateurs génétiques

3.3.2.2.1 Opérateur de mutation

L'opérateur retenu fonctionne de la manière suivante : il prend un certain nombre de gènes (tirés au hasard) de l'individu et en change radicalement le contenu. Pour cela, il sélectionne, à nouveau au hasard dans le domaine entier de la variable, de nouvelles valeurs. Au bout de quelques générations, la population a en effet tendance à s'uniformiser et ce mécanisme (même s'il peut paraître radical) permet de diversifier les individus.

Des essais ont été faits en ne sélectionnant, dans les domaines des variables, que les valeurs dont l'indice de *productivité* (indice d'évaluation de la qualité utilisé pour l'heuristique de *tassement*) est le plus élevé. Ces tests se sont soldés par des échecs car la population a alors tendance à s'uniformiser conformément à ce que préconise l'heuristique de tassement. L'algorithme devient alors incapable de trouver de nouvelles solutions.

3.3.2.2.2 Opérateur de croisement

Si l'on utilise l'opérateur présenté initialement en 2.3.1.3.1, l'algorithme est incapable de trouver régulièrement une solution au problème et se bloque, au bout d'un certain nombre de générations (dépendant du problème), autour d'une solution de mauvaise qualité.

De meilleurs résultats sont obtenus en apportant à l'opérateur les modifications suivantes :

- Le croisement n'engendre plus qu'un seul enfant. Ceci permet de limiter la quantité de nouveaux individus produits à chaque génération, et qui sont longs à évaluer.
- La valeur de la solution obtenue pour chacun des parents est conservée dans le gène de l'enfant (cette modification est proposée en 2.3.1.3.1). L'enfant a donc une bonne probabilité d'être de meilleure qualité que le meilleur de ces deux parents.
- Les valeurs constituantes du gène de l'enfant ne sont plus choisies aléatoirement parmi celles des parents : seules les valeurs ayant l'indice de *productivité* le plus élevé sont retenues.

Cet opérateur modifié permet, en général, de faire converger la population vers une solution.

3.3.2.3 Implémentation avec Ilog Solver

L'implémentation de cette méthode avec Ilog Solver nécessite de réécrire les *goals* de génération et d'instanciation afin de pouvoir choisir quelles valeurs du domaine sont utilisées pour instancier chaque variable. Il suffit alors, lors de la résolution du CSP correspondant à un individu, de ne proposer pour une variable donnée que les valeurs du domaine qui sont attribuées à l'individu (au lieu du domaine entier de la variable). Le problème se résume en outre à une simple heuristique de choix de valeur puisque la dernière valeur autorisée (qui est la valeur fictive du domaine) est toujours utilisable : il n'est donc pas nécessaire de se préoccuper du cas où toutes les valeurs attribuées à l'individu ont été épuisées sans succès (ce qui devrait engendrer un *backtrack* même si le domaine de la variable n'est pas vide).

D'autre part, la résolution consécutive de plusieurs CSP correspondant à plusieurs individus différents est obtenue en utilisant la méthode *IlcManager::restart* après l'évaluation de chaque individu. Ainsi, un seul exemplaire de MAXIME (et du solveur) est utilisé pour l'ensemble de l'algorithme.

3.3.2.4 Résultats

Ces résultats sont obtenus avec les paramétrages suivants : population de *8* individus ; $\rho = 9$; *6* croisements effectués à chaque génération (*PC = 1,5*) ; probabilité de mutation à *0,8* ; probabilité qu'un gène mute sachant que l'individu est mutant fixée à *0,1*.

Notons que cet algorithme ne permet pas de traiter les problèmes insolubles de façon simple, puisque aucune résolution du problème global n'est effectuée – alors que c'est le seul moyen d'affirmer que le problème n'a pas de solution.

Tests sur machine P2400 (iPII 400 – 160 Mo RAM), en moyenne sur 5 échantillons									
Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**2 solveurs en parallèle – résolution par les extrémités**				
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. échecs (total pour tous les individus)	Nb. nœuds explorés (pour l'individu solution)	**Temps (s)**	Valeur de la solution	Génération de l'individu solution
200	1845	2114	**5,648**	-20	85784	147179	**373,945**	-10	19
500	10875	11524	**66,966**	-52	86199	113660	**864,253**	-46	3
600	15490	16259	**114,745**	-62	368740	443396	**4679,860**	-59	8

Ces mauvais résultats s'expliquent de la manière suivante : pour que ce système soit compétitif, il faudrait pouvoir effectuer l'évaluation de 80 individus environ (8 individus sur 10 générations) dans un temps similaire à celui qu'une stratégie standard utilise pour l'évaluation d'1 seul individu. Or, la réduction des domaines des variables d'un facteur 9 ne permet pas d'améliorer les temps de calcul d'un facteur 80, et cela parce que MAXIME effectue en réalité peu de *backtracks* par rapport à la taille de l'arbre des solutions – et donc ne profite pas réellement du fait que le nombre total de nœuds de l'arbre d'un individu est quand même réduit d'un facteur $9^{200}=7*10^{190}$ (pour le problème à 200 tâches par exemple) par rapport à l'arbre initial.

Cette stratégie est probablement performante dans le cadre de problèmes d'*optimisation*, où les solutions abondent et où le but est de trouver *la meilleure*. Dans ce cas, une stratégie classique oblige à parcourir une grande partie de l'arbre des solutions, et les temps de calcul bénéficient donc pleinement de la réduction des domaines des individus génétiques.

3.3.3 Résolution parallèle

Le but de cette méthode (appelée aussi *résolution concurrentielle*) est de démarrer plusieurs solveurs simultanément, chacun prenant le problème par un bout différent. Le gain de performances doit être obtenu grâce à la communication entre les solveurs : chaque solveur résout localement les problèmes qui surviennent mais s'inspire également des solutions trouvées par les autres. Dans un premier temps, nous allons étudier ce principe pour 2 solveurs seulement.

3.3.3.1 Instanciations et communication

Dans le cas de solveurs concurrentiels, il faut toujours garder en mémoire que les instanciations faites par un des solveurs ne sont que des *propositions* pour la solution finale tant que ce solveur n'a pas traité l'ensemble du problème. De la même façon, les valeurs rejetées par ce solveur ne sont que des rejets *préférentiels*. En effet, les interactions entre les variables font que la liste définitive des valeurs permettant de résoudre le problème ne peut être connue qu'une fois l'*ensemble* des instanciations effectuées.

Pratiquement, les cas suivants se présentent lorsqu'un solveur (appelons-le le solveur 1) veut instancier une variable :

- Le solveur 2 n'a pas traité cette variable et n'a traité aucune autre tâche ayant un impact sur la liste des vacations possibles pour cette variable. Dans ce cas, le comportement du solveur 1 est « classique » : il doit choisir une valeur parmi celles figurant dans le domaine de la variable à instancier.
- Le solveur 2 n'a pas traité cette variable mais a traité une autre tâche ayant un impact sur la liste des vacations possibles pour cette variable. Comme nous l'avons vu précédemment (*cf. chapitre 3.3.1.2.2.1 : Contraintes de non-recouvrement*), cela arrive presque uniquement lorsque le solveur 2 a affecté une autre tâche, potentiellement en conflit, à une vacation qui était admissible pour la variable à traiter par le solveur 1. Dans ce cas, cette vacation devient *déconseillée* pour la variable du solveur 1, et ne doit être essayée qu'en dernier recours. **Lorsqu'il instancie une variable, un solveur doit donc signifier que la valeur utilisée devient déconseillée à toutes les autres variables potentiellement en conflit avec celle-ci, pour tous les autres solveurs**.
- Le solveur 2 a déjà traité cette variable. Dans ce cas, le solveur 1 doit de préférence utiliser la valeur qui avait été sélectionnée par le solveur 2. **Donc lorsqu'un solveur instancie une variable, il doit communiquer la valeur qu'il a choisie, pour la conseiller aux autres solveurs**. Les valeurs qui avaient été rejetées par le solveur 2 sont à présent à nouveau utilisables : si elles avaient été rejetées car incohérentes avec les instanciations précédentes, elles sont déjà *déconseillées* ; si elles avaient été rejetées à cause d'une impossibilité d'instancier de futures variables (après backtrack), ces rejets n'ont pas lieu d'être pour le solveur 1 puisque celui-ci n'utilise pas forcément le même ordre d'instanciation.
- Le solveur 2 est en train de traiter cette variable. On pourrait imaginer que les deux solveurs s'informent alors sur les rejets de valeurs qu'il effectuent chacun. Cependant, l'instanciation d'une variable isolée est un processus relativement rapide, et il est probable que le fait de partager ces informations ne procure pas un bénéfice suffisant pour compenser les temps de communication. Nous assimilerons donc ce cas au cas où le solveur 2 n'a pas encore traité la variable.

De plus, la règle suivante apparaît : **lorsqu'un solveur effectue un backtrack, il doit indiquer quelle est la variable dont il supprime l'instanciation (et à quelle valeur)**, afin d'enlever les directives de « *déconseil* » qui avaient été formulées concernant cette valeur pour d'autres variables ; de même, il faut supprimer le *conseil* qui avait été donné pour cette valeur et cette variable.

La gestion des conflits *conseil*/*déconseil* est opérée de la façon suivante : en prévision d'une extension du parallélisme à plus de 2 solveurs, nous dirons que **l'attitude à adopter par rapport à une valeur pour une variable donnée (*conseil* ou *déconseil*) est représentée par un entier relatif, qui sera d'autant plus grand que la valeur est conseillée**, et qui est calculé en faisant la différence du nombre de solveurs conseillant cette valeur et de ceux la déconseillant.

En fait, dans le cas de 2 solveurs, le « conflit » ne se présente jamais : si une valeur a été à la fois *conseillée* et *déconseillée* pour le solveur 1, c'est que c'est lui-même (le solveur 1) qui a prodigué le *déconseil* (s'il avait prodigué le *conseil*, il aurait déjà instancié cette variable.) Or, s'il a lui-même *déconseillé* la valeur, il ne peut de

toutes façons pas l'utiliser (elle n'est plus dans le domaine de la variable : pour le solveur 2, cette valeur est simplement « à éviter si possible », mais pour le solveur 1, elle viole directement une contrainte avec une instanciation précédente).

Toujours dans le cas de 2 solveurs, une particularité apparaît dans le mécanisme d'instanciation : lorsque toutes les variables du problème ont été instanciées, chacune au moins une fois (par un solveur), on peut dire que tous les *conseils* et *déconseils* utiles ont été exprimés. En effet :

1. Chaque solveur ne peut plus qu'émettre des *conseils* sur les variables que l'autre a déjà instanciées
2. Chaque solveur ne peut plus qu'émettre des *déconseils* bénéfiques pour l'instanciation de varaibles que l'autre a déjà instanciées

On voit donc que, dans ce cas, un des solveurs peut être arrêté afin de concentrer toute la puissance de calcul sur l'autre.

On laissera donc continuer le solveur le plus « avancé » (celui qui, à ce moment-là, a déjà instancié le plus de variables – c'est lui qui est susceptible de terminer la résolution en premier) . L'autre aura contribué à la solution, mais sans l'atteindre.

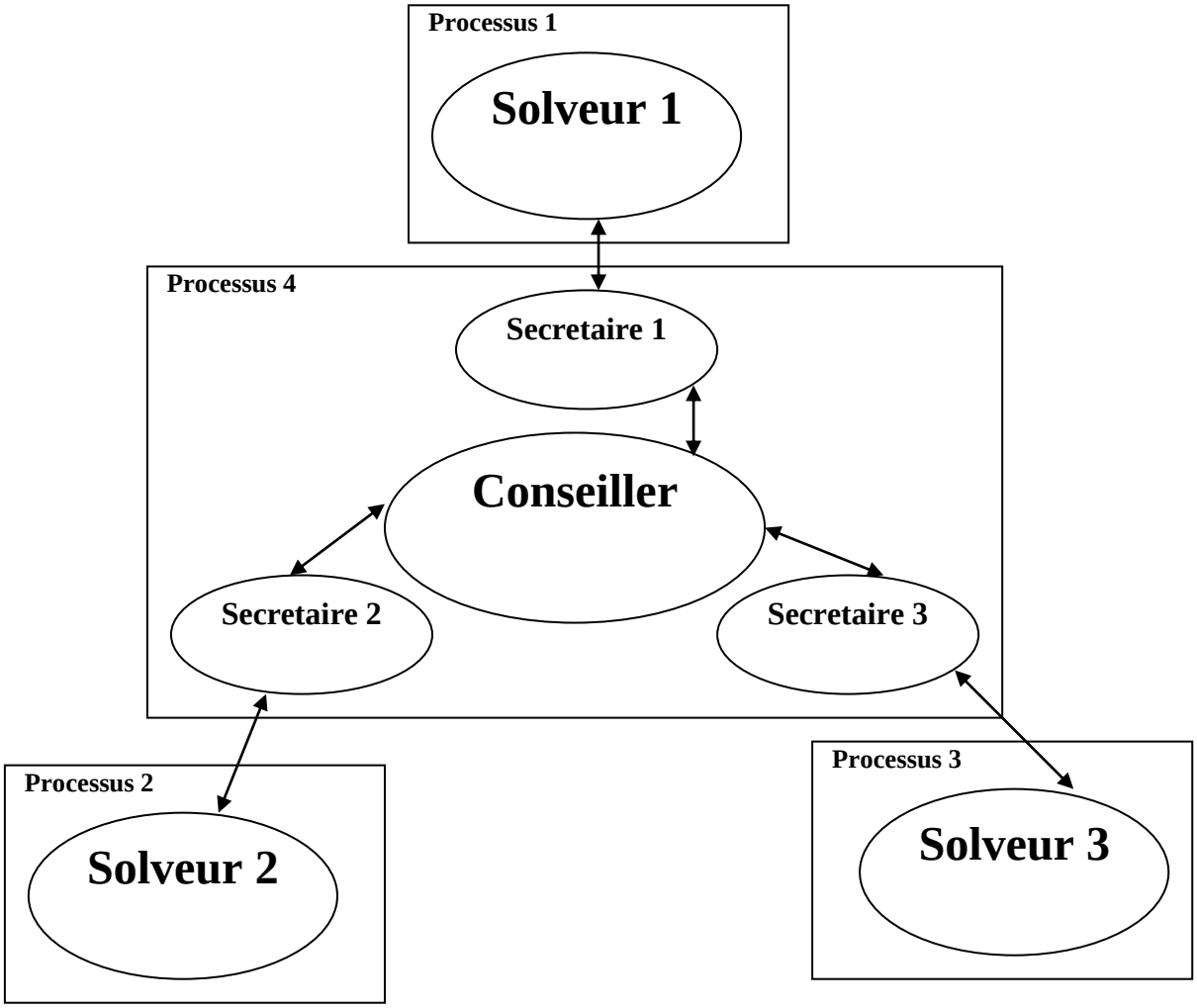
3.3.3.2 Implémentation technique générale

Afin de mieux découper le problème, et dans l'optique d'ouvrir le système à plus de 2 solveurs, la gestion des *conseils/déconseils* sera assurée par une entité « centrale » séparée appelée « conseiller ». Les deux solveurs fonctionneront dans des processus séparés (et non pas des threads - il est difficile de dire si l'application MAXIME est *thread-safe* ; de plus, il est difficile de lister exactement les données partageables par les solveurs à l'intérieur de cette application ; enfin, le cas où les processus sont séparés est plus général). Le conseiller aura lui aussi son processus.

Toujours dans un but d'ouverture, le conseiller sera implémenté sous forme d'objet Windows COM/DCOM, plutôt que sous forme d'exécutable utilisant la communication inter-processus du système. Ainsi, tous les solveurs et le conseiller pourront être répartis sur des machines séparées de façon entièrement transparente : pour communiquer, DCOM utilise de la *mémoire partagée inter-processus* si les processus sont détectés comme étant sur le même système physique ; sinon, il utilise des appels RPC (dans ce cas, et uniquement dans celui-ci, le système de « mise en rang » (*marshaling*) est déclenché afin de convertir les adresses mémoires désignées par les différents processus en données réellement transmissibles.)

Dans la pratique, DCOM construit l'objet C++ correspondant à un composant COM à chaque ouverture de dialogue avec celui-ci. Dans notre cas, l'objet conseiller devant être unique, les vrais objets COM seront donc des objets d'interface assurant la mise à disposition des méthodes du conseiller. Ces objets d'interface seront appelés « secretaires ». Ces objets seront créés, détruits et gérés par un serveur COM dans un mode *mono-thread* afin que tous les accès au conseiller soient parfaitement séquentiels. Ceci permet de régler directement l'ensemble des problèmes de synchronisation des tâches par rapport au conseiller (qui est une ressource critique).

Le schéma ci-dessous décrit le système ainsi obtenu (pour 3 solveurs) :



(8) *: Représentation schématique de l'architecture technique utilisée pour la résolution concurrentielle*

3.3.3.3 Spécification générale des solveurs

Tous les solveurs sont strictement identiques. Ce sont des exécutables MAXIME modifiés de façon à incorporer les fonctions suivantes :

- Initialisation DCOM avant de commencer la résolution
- Information du conseiller à chaque instanciation
- Information du conseiller à chaque *backtrack*
- Demande de l'avis du conseiller à chaque nouvelle variable rencontrée, afin d'en tenir compte dans l'ordre d'essai des valeurs
- Si le solveur n'est pas le plus avancé, arrêt de la recherche lorsque toutes les variables ont été instanciées, chacune au moins par 1 solveur.
- Libération DCOM à la fin de la résolution

3.3.3.4 Spécification générale du conseiller

Les points suivants apparaissent lorsqu'on examine le rôle du conseiller dans le cadre de l'application MAXIME :

- C'est un objet passif : il n'a pas besoin d'initier de communication avec les solveurs ; il se contente d'exécuter les ordres (conseil ; déconseil ; demande de conseil) de ceux-ci.

- Il est préférable qu'il connaisse, pour chaque variable, la liste de toutes les autres variables potentiellement en conflit avec elle (c'est-à-dire la liste de toutes les variables qui ne doivent pas prendre la même valeur qu'elle.) De cette façon, lorsqu'un solveur instancie une variable et communique sa valeur au conseiller , celui-ci peut déduire les « déconseils » à appliquer. Ainsi, il n'est pas nécessaire que ces déconseils soient communiqués entre les processus, ce qui constitue un gain de temps sensible.
- Le conseiller sera conçu de façon à être indépendant de la façon dont le problème est posé et résolu. En particulier, il ne fera pas d'hypothèses sur les variables en conflits. Ainsi, pour modifier la méthode de résolution, il suffira de modifier les solveurs (le code de tous les solveurs peut être identique), le conseiller restant inchangé. Ceci implique qu'une phase d' « initialisation » permette à un des solveurs (ici, le solveur numéroté « zéro ») de communiquer au conseiller la liste des variables en conflit, avant d'amorcer réellement la résolution du problème.
- Les arguments passés aux méthodes du conseiller sont volontairement triviaux (entiers, tableaux – pas de listes ou de dictionnaires) afin de faciliter le *marshaling* dans le cadre d'une utilisation distribuée. Cela n'empêche pas d'utiliser des types évolués dans l'implémentation des méthodes.

3.3.3.4.1 Méthodes d'interface du conseiller

```
/////////////////// METHODES D'INITIALISATION

// Renvoie true si les conflits ont ete definis
// Les solveurs ne doivent commencer la recherche que lorsque les conflits ont ete definis
bool isInitFait () const

// Affirme que les conflits ont ete definis
// Cette methode ne doit etre utilisee que par le solveur 0
void setInitFait()

// Permet à un solveur de recuperer son identifiant
// Les identifiants sont delivres en ordre croissant depuis 0
unsigned long getNoSolveur ();

// Definit le nombre de variables a traiter
// Cette méthode ne doit être utiliseé que par le solveur 0, avant appel de setConflits
void setNbVar (unsigned long nombre_var);

// Definit les conflits de non-recouvrement entre les variables
// Cette méthode ne doit être utiliseé que par le solveur 0, apres appel de setNbVar
//// conflits n'est pas recopie / il est detruit
void setConflits (unsigned long no_var, unsigned long nb_conflits,
                  unsigned long * conflits)

/////////////////// METHODES D'EXECUTION

// Informe le conseiller qu'une instantiation a ete operee
void instantiation (unsigned long no_solveur, unsigned long no_var, unsigned long val)

// Informe le conseiller qu'il y a eu desinstantiation de la variable no_var
void backtrack (unsigned long no_solveur, unsigned long no_var)


// Le solveur no_solveur demande un conseil pour la variable no_var, et pour chacune
//   des nb_val valeurs passée dans val
// note prend une valeur relative
// Le nombre d'elements de note est nb_val
//// note doit etre alloue avec suffisamment d'espace
//// val n'est pas recopie / il n'est pas detruit
void conseil (unsigned long no_solveur, unsigned long no_var, unsigned long nb_val,
              unsigned long * valeurs, long * note) const;

// Indique si toutes les variables ont ete instantiees, chahcune par au moins 1 solveur
// Dans ce cas, s vaut CONS_STOP pour tous les solveurs sauf le plus avance
//  (celui qui a instantie le plus de variables).
// Tous ces solveurs sont alors pries de s'arreter (ils ne peuvent plus conseiller
// le solveur le plus avance sur de nouvelles variables) afin de concentrer la
// puissance de calcul.
// Pour le solveur le plus avance, s vaut alors CONS_FINAL, indiquant qu'il n'a
//   plus besoin d'informer le conseiller sur ces instantiations / backtrack
// En temps normal (certaines variables non-instantiees), s vaut CONS_NULL
void etat (unsigned long no_solveur, short * s) const;
```

3.3.3.4.2 Données gérées par le conseiller

Le conseiller a besoin de contrôler les données suivantes :

- Pour chaque solveur qui a demandé un identifiant : la liste des variables avec la valeur affectée à chacune par ce solveur (ou « vide » si la variable n'a pas encore été instanciée par ce solveur)
- Pour chaque variable, la liste des variables potentiellement en conflit avec elle

3.3.3.4.3 Algorithme de base du conseil

L'algorithme utilisé pour implémenter la méthode `conseil` sera celui-ci (cette méthode demande, pour le solveur *no_solveur*, une « note » pour chacune des valeurs passées dans un tableau *valeurs*, pour une variable d'indice *no_var*) :

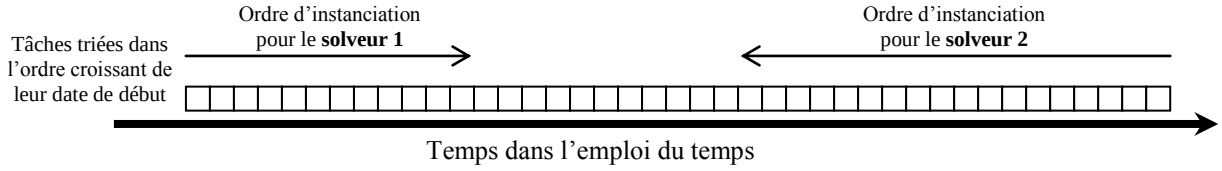
```
1.  Pour chaque valeur val de valeurs
2.      Affecter 0 à conseil
3.      Pour chaque solveur s différent de no_solveur
4.          Si la variable no_var a été affectée à val par le solveur s
5.              Incrémenter conseil
6.          Sinon
7.              v est la première variable en conflit avec no_var
8.              Affecter faux à fin
9.              Tant que v existe et fin est faux
10.                 Si v a été affectée à val par le solveur s
11.                     Décrémenter conseil
12.                     Affecter vrai à fin
13.                 Sinon
14.                     v est la prochaine variable en conflit avec no_var
15.     conseil est la note attribuée à val par le conseiller
```

***(9)** : Algorithme de base donnant un conseil en vue d'une résolution concurrentielle utilisant un « conseiller » central*

3.3.3.5 Stratégies de résolution

3.3.3.5.1 Stratégie de départ : résolution par les extrémités

Dans notre cas, la stratégie suivante a d'abord été retenue : comme les tâches à affecter ont d'autant moins de chances d'être en conflit (par rapport à une vacation) qu'elles doivent être effectuées dans des plages horaires éloignées, nous posons que les deux solveurs vont chacun commencer par traiter une extrémité temporelle de l'emploi du temps. Le premier solveur traitera les tâches dans l'ordre croissant de leur date de commencement ; le second prendra l'ordre décroissant. Ainsi, les conflits (instanciations effectuées par un solveur, incompatibles avec celles effectuées par l'autre) se situeront essentiellement au niveau de la jonction des deux traitements (c'est-à-dire vers le milieu de l'emploi du temps.) Ces conflits devraient par ailleurs diminuer plus on s'éloigne de cette jonction.



***(10)** : Ordre d'instanciation des variables dans la stratégie de résolution par les extrémités*

Le gain de temps est censé être obtenu grâce à l'usage des *déconseils* : pendant qu'un solveur traite une moitié de l'emploi du temps, le second s'occupe de la deuxième partie en choisissant judicieusement ses valeurs de façon à pouvoir :

1. Etablir rapidement la « jonction » avec l'autre solveur vers le milieu de l'emploi du temps
2. Pouvoir utiliser sans difficulté, le moment venu, les instanciations préalablement effectuées par l'autre solveur sur la seconde partie de l'emploi du temps

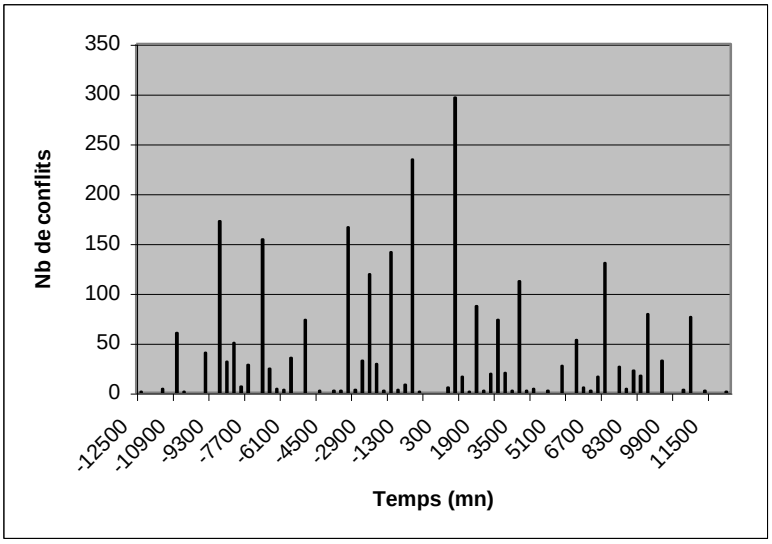
Lors de ces tests, les solveurs utilisent l'heuristique de tassement pour départager deux valeurs conseillées de façon identique. Les résultats obtenus sont les suivants :

Tests sur machine P2400 (iPII 400 – 160 Mo RAM), en moyenne sur 5 échantillons
Cette stratégie, non finalisée, n'introduit pas les valeurs fictives (et ne traite donc pas les problèmes insolubles) car l'heuristique utilisée permet trop facilement l'usage de ces valeurs pour les problèmes ayant une solution.

Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**2 solveurs en parallèle – résolution par les extrémités**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. Echecs (pour 1 solveur)	Nb. nœuds explorés (pour 1 solveur)	**Temps (s)**	Valeur de la solution
200	1845	2114	**5,648**	-20	845	1120	**6,662**	-15
500	10875	11524	**66,966**	-52	5872	6535	**72,200**	-47
600	15490	16259	**114,745**	-62	7560	8340	**117,180**	-57
800	28333	29378	**309,955**	-95	13916	14973	**309,347**	-82
1200	63896	65439	**1237,180**	-128	43329	44898	**1536,760**	-111

Cette stratégie amène donc les conclusions suivantes :

- La résolution parallèle n'est efficace que si un solveur peut bénéficier, lors de ses instanciations, des essais effectués par les autres solveurs. S'il n'y a pas (ou peu) d'essais (backtracks) à faire, l'implémentation de cette stratégie amène une perte de temps considérable. Ainsi, les heuristiques développées en *3.3.1.2.2* (qui « donnent » la solution pratiquement sans aucune recherche) ne sont pas utilisables avec la méthode de résolution parallèle. Elles donnent toujours un meilleur résultat lorsqu'elles sont utilisées seules. Rappelons cependant que ces heuristiques sont « particulières » à l'application MAXIME et qu'elles ne peuvent prétendre au titre de méthode « générique » que peut revendiquer la résolution concurrentielle. **Les résultats de la résolution parallèle seront donc comparés à ceux obtenus avec la stratégie de « référence » utilisant l'heuristique de tassement uniquement.**
- Globalement, les temps de calcul (ainsi que la valeur des solutions) sont à peu près identiques que l'on utilise la stratégie de référence ou cette nouvelle stratégie « concurrentielle », ce qui, compte tenu de l'infrastructure logicielle nécessitée par le multi-tâches, ne constitue pas une bonne performance. Néanmoins, il faut garder à l'esprit que l'initialisation de l'application MAXIME (définition des variables et des contraintes) prend plusieurs secondes (voire plusieurs dizaines de secondes) et qu'elle est réalisée 2 fois de façon absolument identique lorsqu'on utilise 2 solveurs. Cet inconvénient technique, dû à l'absence de *clonage de process* sous Windows NT, handicape artificiellement la méthode « concurrentielle ». L'usage de cette méthode commence dès lors à se justifier.
- Les performances limitées de cette méthode s'expliquent de la manière suivante : le gain de temps devait être dû aux « déconseils » prodigués par un solveur pour le second. Or, il se trouve qu'une tâche ne peut entrer en conflit avec une autre que si cette dernière se situe dans une certaine fenêtre temporelle située autour de la première tâche. Cette fenêtre, que nous appellerons *zone d'influence* de la tâche, a une taille limitée à 25000 minutes environ (soit environ ¼ de l'emploi du temps total). Comme les solveurs traitent des tâches temporellement très éloignées, ils s'échangent dès lors peu d'information : il est donc normal de ne pas constater de réelle amélioration des temps de calcul.



***(11)** : Nombre de conflits en fonction du temps séparant les deux tâches (problème à 500 tâches)*

3.3.3.5.2 Stratégie de résolution avec des méthodes différentes

Pour parer à ce manque de communication, une solution simple est d'utiliser un ordre d'instanciation des variables différent pour chaque solveur. Toutefois, on prendra cette fois soin de choisir ces stratégies de façon à ce que les solveurs partagent un certain nombre de variables.

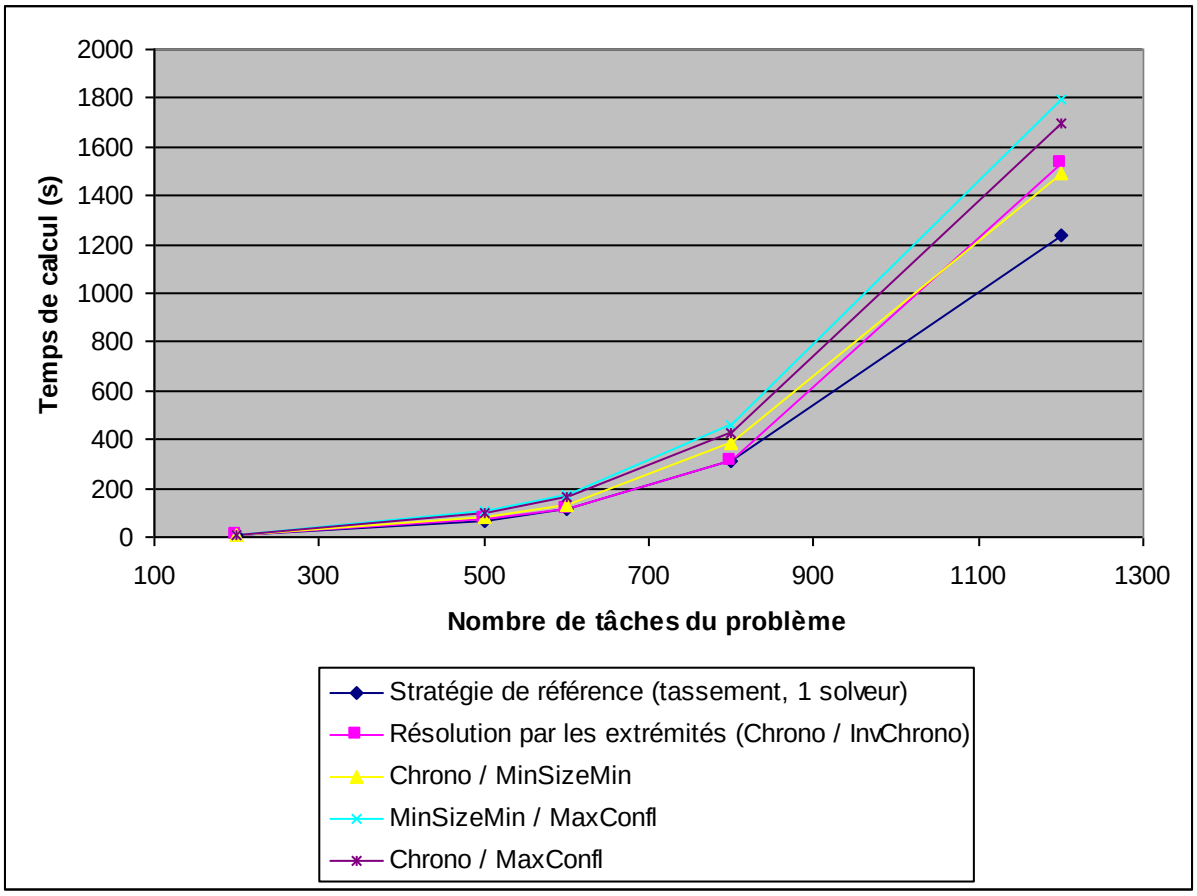
Les stratégies testées sont les suivantes :

- *Chrono/InvChrono* est la stratégie de résolution exposée ci-dessus (résolution par les extrémités temporelles)
- *Chrono/MinSizeMin* : un solveur effectue l'instanciation dans l'ordre chronologique des tâches ; l'autre prend en premier les variables-tâches dont le domaine est le plus réduit
- *MinSizeMin/MaxConfl* : un solveur prend en premier les variables-tâches dont le domaine est le plus réduit ; l'autre prend en premier les tâches qui engendrent le plus de conflits
- *Chrono/MaxConfl* : un solveur effectue l'instanciation dans l'ordre chronologique des tâches ; l'autre prend en premier les tâches qui engendrent le plus de conflits

Tests sur machine P2400 (iPII 400 – 160 Mo RAM), en moyenne sur 5 échantillons
Ces stratégies, non finalisées, n'introduisent pas les valeurs fictives (et ne traitent donc pas les problèmes insolubles) car l'heuristique utilisée permet trop facilement l'usage de ces valeurs pour les problèmes ayant une solution.

Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**2 solveurs en parallèle – résolution par les extrémités (Chrono / InvChrono)**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. Echecs (pour 1 solveur)	Nb. nœuds explorés (pour 1 solveur)	**Temps (s)**	Valeur de la solution
200	1845	2114	**5,648**	-20	845	1120	**6,662**	-15
500	10875	11524	**66,966**	-52	5872	6535	**72,200**	-47
600	15490	16259	**114,745**	-62	7560	8340	**117,180**	-57
800	28333	29378	**309,955**	-95	13916	14973	**309,347**	-82
1200	63896	65439	**1237,180**	-128	43329	44898	**1536,760**	-111
	2 solveurs en parallèle – Chrono / MinSizeMin				**2 solveurs en parallèle – MinSizeMin / MaxConfl**			
200	1122	1394	**7,931**	-12	982	1253	**8,813**	-17
500	6343	6998	**84,824**	-50	7193	7852	**107,100**	-52
600	8636	9427	**134,710**	-58	10322	11100	**174,731**	-59
800	15806	16866	**381,779**	-86	18552	19606	**461,792**	-89
1200	36507	38077	**1492,530**	-113	43814	45357	**1793,170**	-122
	2 solveurs en parallèle – Chrono / MaxConfl							
200	1267	1539	**8,678**	-13				
500	7339	7996	**96,104**	-49				
600	10566	11344	**160,557**	-55				
800	18780	19840	**426,517**	-81				
1200	42823	44384	**1693,065**	-117				

Ces stratégies sont en général moins bonnes que celle utilisant la résolution par les extrémités – et toujours moins bonne que l'heuristique de référence à un seul solveur. En effet, un défaut que l'on ne peut maîtriser avec ces méthodes vient de la contrainte de cumul (notons que le nombre d'échecs augmente sensiblement, ce qui indique le viol d'une contrainte) : chaque solveur « tasse » les tâches sur certaines vacations – parfois les mêmes que celles utilisées par l'autre solveur. Or, les tâches ne peuvent être ainsi tassées que dans la limite tolérée par la contrainte de cumul. Lors de la mise en commun des deux demi-résultats, il apparaît donc des vacations sur lesquelles se retrouvent beaucoup trop de tâches (celles proposées par le premier solveur *plus* celles proposées par le second) : ceci fait échouer la contrainte de cumul et les demi-résultats s'avèrent donc être incompatibles. On constate alors plus de déchets dans les conseils prodigués par un solveur.



***(12)** : Performances de deux solveurs utilisant des stratégies de résolution sans séparation des domaines*

Mesures effectuées sur machine P2400 (iPII 400 – 160 Mo RAM), en moyenne sur 5 échantillons, sans valeurs fictives

3.3.3.5.3 Stratégie de résolution par « tranches » avec séparation des domaines

Ces constats amènent à effectuer deux modifications au processus de résolution : une séparation des domaines de valeurs et une division de la liste des tâches.

3.3.3.5.3.1 Séparation des domaines de valeurs

Tout d'abord, et sans introduire de critère trop particulier à l'application MAXIME (contrairement aux heuristiques décrites en *3.3.1.2.2*), on peut modifier le conseiller de telle façon que les deux solveurs travaillent *de préférence* avec des vacations différentes. Ainsi, même si le tassement opéré par chaque solveur peut être important, la mise en commun est plus aisée. Pour cela, il suffit de modifier l'algorithme de conseil de la façon suivante :

```
1.  Pour chaque valeur val de valeurs
2.      Affecter 0 à conseil
3.      Pour chaque solveur s différent de no_solveur
4.          Si la valeur val a été affectée à la variable no_var par le solveur s
5.              Ajouter VAL_CONSEIL à conseil
6.          Sinon
7.              Pour toutes les variables r
8.                  Si val a été affectée à r par le solveur s
9.                      Retrancher VAL_SEPARATION à conseil
10.             v est la première variable en conflit avec no_var
11.             Affecter faux à fin
12.             Tant que v existe et fin est faux
13.                 Si val a été affectée à v par le solveur s
14.                     Retrancher VAL_DECONSEIL à conseil
15.                     Affecter vrai à fin
16.                 Sinon
17.                     v est la prochaine variable en conflit avec no_var
18.     conseil est la note attribuée à val par le conseiller
```

***(13)** : Algorithme de conseil intégrant la séparation des domaines de valeurs pour les solveurs.*
*Les modifications par rapport à **(9)** sont indiquées par les bandes noires*

L'attitude d'un solveur est donc la suivante :

- Il place les tâches de préférence sur les vacations non-utilisées ou peu utilisées par l'autre solveur (ainsi, il prévient l'augmentation de l'indice de « cumul »)
- Il place les tâches de préférence sur les vacations les plus utilisées globalement (ainsi, il bénéficie de la rapidité et de la qualité de l'heuristique de tassement)

Pour une tâche, chaque vacation possible est notée suivant ces deux critères et les valeurs obtenues sont additionnées (avec une pondération). Les vacations sont alors essayées dans l'ordre décroissant des notes.

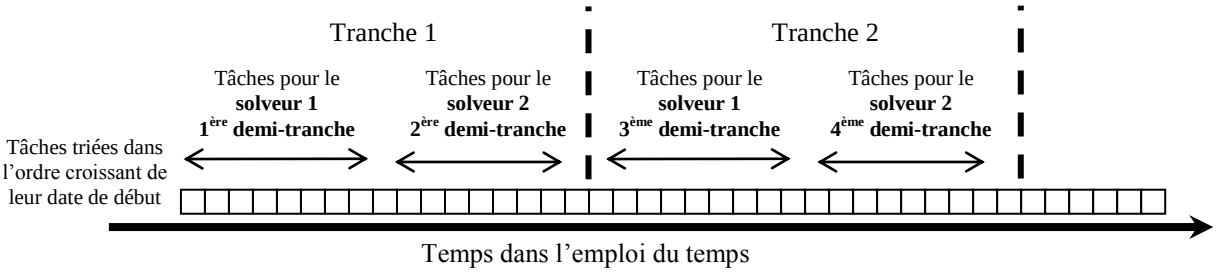
Après essai, les valeurs numériques retenues pour les pondérations et les constantes de l'algorithme sont les suivantes :

- VAL_CONSEIL = VAL_DECONSEIL = 1000
- VAL_SEPARATION = 1
- L'indice de « productivité » qui sert de base à l'heuristique de tassement est multiplié par 10 dans la note attribuée aux vacations (sa valeur est habituellement de quelques unités au maximum).

3.3.3.5.3.2 Division de la liste des tâches

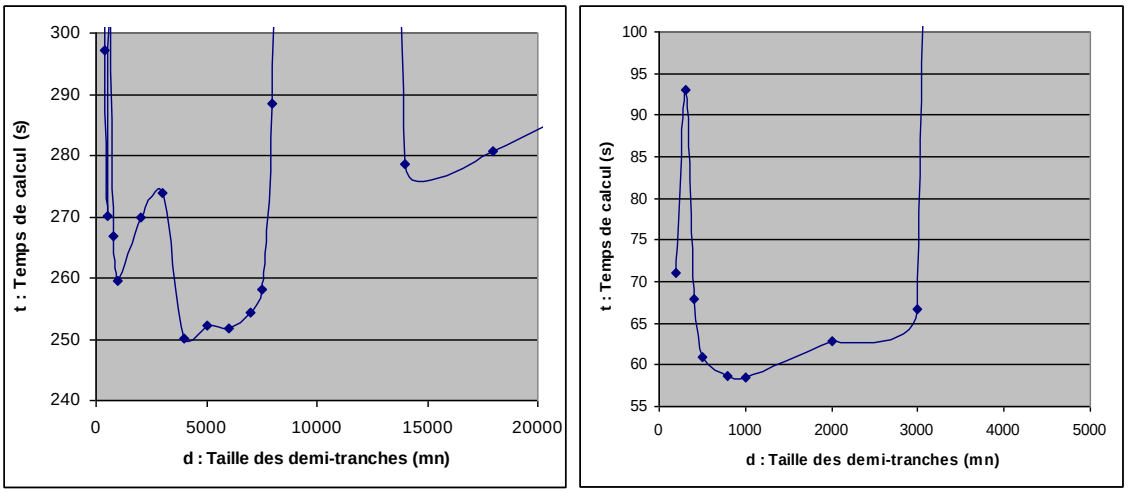
Si l'on laisse les deux solveurs commencer chacun à une extrémité du problème, les résultats sont décevants car il n'y a presque plus d'interaction entre eux (les domaines des variables sont assez différents entre les deux extrémités et les solveurs ont tendance à utiliser des vacations différentes – donc les *déconseils* ne vont plus jouer). Une idée est alors de diviser le problème en « tranches temporelles » consécutives, chaque solveur traitant une des deux moitiés de chaque tranche (les tâches situées exactement aux extrémités de chaque tranche sont traitées par les deux solveurs). Deux « demi-tranches » consécutives comportent en effet des variables (tâches) ayant des domaines assez similaires, ce qui redonne un sens aux *déconseils*.

A l'intérieur d'une tranche, les variables dont la taille du domaine est la plus faible sont instanciées en premier, ce qui constitue une heuristique habituellement performante (*cf. chapitre 2.2.3.1 : Heuristique dynamique basée sur les domaines*).



***(14)** : Ordre d'instanciation des variables dans la stratégie de résolution par tranches*

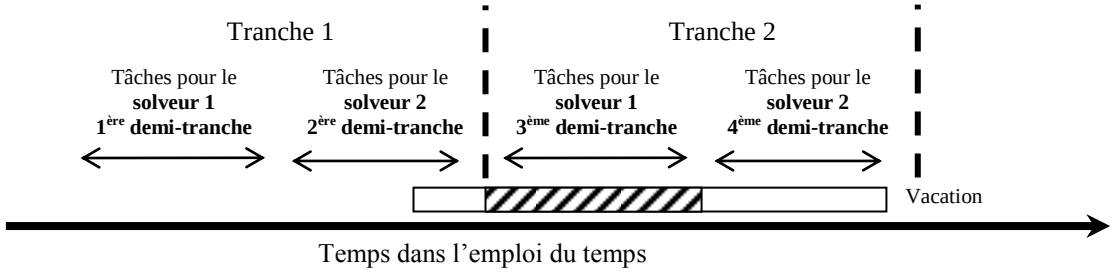
La taille des demi-tranches influe de la façon suivante sur les performances des solveurs :



***(15)** : Influence de la taille des tranches sur les performances de résolution*
Exemples pour les problèmes à 800 tâches, à gauche, et à 500 tâches, à droite
Mesures effectuées sur machine P2400 (iPII 400 – 160 Mo RAM), en moyenne sur 5 échantillons, sans valeurs fictives

L'allure générale de ces courbes s'explique de la manière suivante :

- Lorsque *d* est très faible (quelques centaines de minutes), le comportement est erratique car certaines tranches ne comportent aucune tâche. De plus, les deux solveurs traitent simultanément un grand nombre de tâches identiques : ces tâches sont celles situées aux extrémités des tranches. **Ce cas n'est donc pas fonctionnel**.
- Lorsque *d* est élevé (plusieurs milliers, mais inférieur à 15000 mn environ), le système est incapable de trouver une solution au problème car il optimise mal le placement des tâches sur les vacations. Pour s'en convaincre, considérons les chiffres suivants : une vacation possède une taille de quelques dizaines de milliers de minutes (18000 à 43200 mn) ; une tâche de quelques milliers (300 à 12600 mn). Lorsque la taille des tranches complètes est de l'ordre de grandeur de celle des vacations, une bonne partie de chaque vacation est dans un premier temps inutilisée car le solveur s'occupant d'une des moitiés va y « tasser » ses tâches, tandis que l'autre solveur va de préférence placer les siennes sur une *autre* vacation.



***(16)** : Utilisation d'une vacation dont la taille avoisine la taille d'une tranche*
La zone hachurée correspond à la partie de la vacation réellement utilisée au début de la résolution

Cette mauvaise optimisation engendre au bout d'un certain temps une impossibilité de placer les tâches restantes sans faire échouer la contrainte de cumul.
Notons que ce problème n'apparaît pas si la taille des tranches est sensiblement plus petite (lorsque la taille d'une demi-tranche est de l'ordre de grandeur de la taille d'une tâche) car dans ce cas, les tâches situées aux extrémités des demi-tranches, sur les vacations, peuvent effectivement occuper l'espace des demi-tranches adjacentes.

- Lorsque *d* est très élevé (supérieur à 15000 mn environ), la taille des tranches devient supérieure à la taille des vacations. Dans ce cas, chaque vacation est utilisée normalement par un des solveurs en majorité, et le problème peut à nouveau être résolu. Par contre, la grande taille des tranches rend la collaboration entre les solveurs assez rare, et les performances sont donc modestes (rappelons que la taille de la zone d'influence d'une tâche n'est que de 25000 minutes environ).
- Enfin, si *d* vaut quelques milliers de minutes, le problème est soluble et la coopération entre les solveurs est bonne, ce qui assure des résultats de bonne qualité. **La taille des demi-tranches doit donc être prise dans cette plage**.

3.3.3.5.3.3 Résultats

Ces résultats ont été obtenus avec une taille de tranche fixée à 2000 minutes (demi-tranches de 1000 minutes).

Tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM), en moyenne sur 5 échantillons								
Nombre de tâches du problème	**Heuristique de tassement uniquement (stratégie de référence)**				**2 solveurs en parallèle – résolution par « tranches »**			
	Nb échecs	Nb. nœuds explorés	**Temps (s)**	Valeur de la solution	Nb. Echecs (pour 1 solveur)	Nb. nœuds explorés (pour 1 solveur)	**Temps (s)**	Valeur de la solution
200	1845	2114	**4,977**	-20	683	958	**4,871**	-6
400	6929	7426	**33,619**	2393	3145	3641	**30,482**	2535
500	10875	11524	**60,207**	-52	3808	4467	**50,603**	-26
600	15490	16259	**113,443**	-62	5431	6208	**86,426**	-35
800	28333	29378	**273,774**	-95	9852	10914	**229,852**	-48
1000	44740	46001	**581,676**	3126	24460	20754	**534,674**	2272
1200	63896	65439	**1240,100**	-128	23587	25156	**977,321**	-87

Les résultats sont plutôt bons comparés à la méthode de référence : les temps de calcul sont réduits de 2% à 24 % tandis que la qualité des solutions reste correcte.

Ces résultats permettent de valider l'approche même s'ils ne parviennent pas à surclasser une heuristique plus particulière.

3.3.4 « Limited Discrepancy Search »

Cette stratégie présentée en *2.2.4* a pour but de focaliser la recherche sur les branches de l'arbre préconisées par une heuristique. Ainsi, on choisit en premier lieu (jusqu'à aboutir à un échec, ou jusqu'à trouver une solution) uniquement les valeurs préconisées par l'heuristique, puis toutes les valeurs préconisées par l'heuristique sauf une, puis toutes sauf deux, etc… Ce mécanisme, à l'origine élaboré pour des variables binaires (2 valeurs par domaine), peut être adapté à des variables quelconques en partageant chaque domaine en une série de valeurs « conseillées » et une autre de valeurs « déconseillées ».

3.3.4.1 Implémentation avec Ilog Solver

L'adaptation du « Limited Discrepancy Search » à Ilog Solver peut se réaliser aisément de la façon suivante (en supposant que les *goals* d'instanciation et de génération ont été réécrits manuellement) :

- Le *goal* de génération doit être pourvu d'un argument supplémentaire, appelé ici *t*, qui donne le nombre de variables pour lesquelles il faut encore prendre une valeur « déconseillée », à un certain niveau de l'arbre. Cet argument doit aussi pouvoir décrire le cas particulier où la variable à instancier ne possède plus de valeur déconseillée dans son domaine. Ce cas est figuré par la valeur DECONSEIL_VIDE de *t*.
- A la fin du *goal* de génération, le domaine de la variable considérée doit être réduit à la série de valeurs à utiliser dans le cadre de la recherche, c'est-à-dire :
  - Les valeurs « conseillées » si *t* vaut 0 ou DECONSEIL_VIDE
  - Les valeurs « déconseillées » si *t* est strictement supérieur à 0
- 3 cas se présentent alors pour achever ce *goal*
  - Si *t* vaut 0, le comportement est « classique » : il faut appeler le *goal* d'instanciation (qui donnera à la variable une valeur « conseillée ») puis rappeler le *goal* de génération avec *t* = 0 pour instancier la variable suivante.
  - Si *t* est strictement supérieur à 0, il faut effectuer un « ou » (*IlcOr*) pour prendre l'une des deux options suivantes :
    - Instancier la variable avec une valeur « déconseillée ». Pour cela, appeler le *goal* d'instanciation, puis le *goal* de génération avec t = t-1 pour instancier la variable suivante.
    - Si ce n'est pas possible, instancier la variable avec une valeur « conseillée ». Pour cela, rappeler d'abord le *goal* de génération avec t = DECONSEIL_VIDE puis, le rappeler une seconde fois avec t = t. Ces appels permettent, dans l'ordre, de reconstruire le domaine de la variable avec les valeurs « conseillées », puis d'effectuer l'instanciation de cette variable (*cf. ci-dessous*), et enfin de passer à la variable suivante sans modifier *t* puisqu'aucune valeur « déconseillée » n'a été utilisée.
  - Si *t* vaut DECONSEIL_VIDE, appeler simplement le *goal* d'instanciation.

3.3.4.2 Résultats

Les performances obtenues sont extrêmement mauvaises, principalement en raison des deux points suivants :

- Le système de *look-ahead* utilisé par Ilog Solver induit le comportement suivant : lorsqu'une variable est instanciée, le domaine des variables suivantes se trouve modifié. En particulier, il faut toujours re-départager les valeurs du domaine d'une variable (en valeurs « conseillées » et « déconseillées »), même si cette variable a déjà été traitée précédemment, avant un *backtrack*. Or, les calculs nécessaires à ce classement sont gourmands en temps, même pour une heuristique simple. De plus, les *backtracks* sont extrêmement fréquents puisque le principe même de la méthode veut que les valeurs « déconseillées » soient utilisées le plus près possible de la racine de l'arbre (les performances de l'heuristique sont supposées moins bonnes lorsque peu de variables sont instanciées.)
- Cette stratégie correspond assez mal au problème posé par MAXIME. En effet, lorsqu'il y a échec d'une instanciation, le « Limited Discrepancy Serach » en impute la cause à un mauvais choix de l'heuristique, effectué très en amont dans le processus de résolution. Or, il s'avère qu'un échec dans MAXIME est très souvent lié à un problème local : plus assez de vacations possibles pour les

variables, tassement trop important sur une vacation… Il est donc en général inutile de modifier une instanciation trop éloignée du niveau où l'échec intervient.

Avec le problème à 200 tâches, aucune solution n'est obtenue aux itérations 0, 1, 2 et 3. Le calcul à l'itération 4 a été arrêté après plusieurs dizaines de minutes de recherche infructueuse sur la machine P2400 (iPII 400 – 160 Mo RAM). Rappelons que ce problème est traité en quelques secondes avec l'heuristique de tassement (heuristique simple).

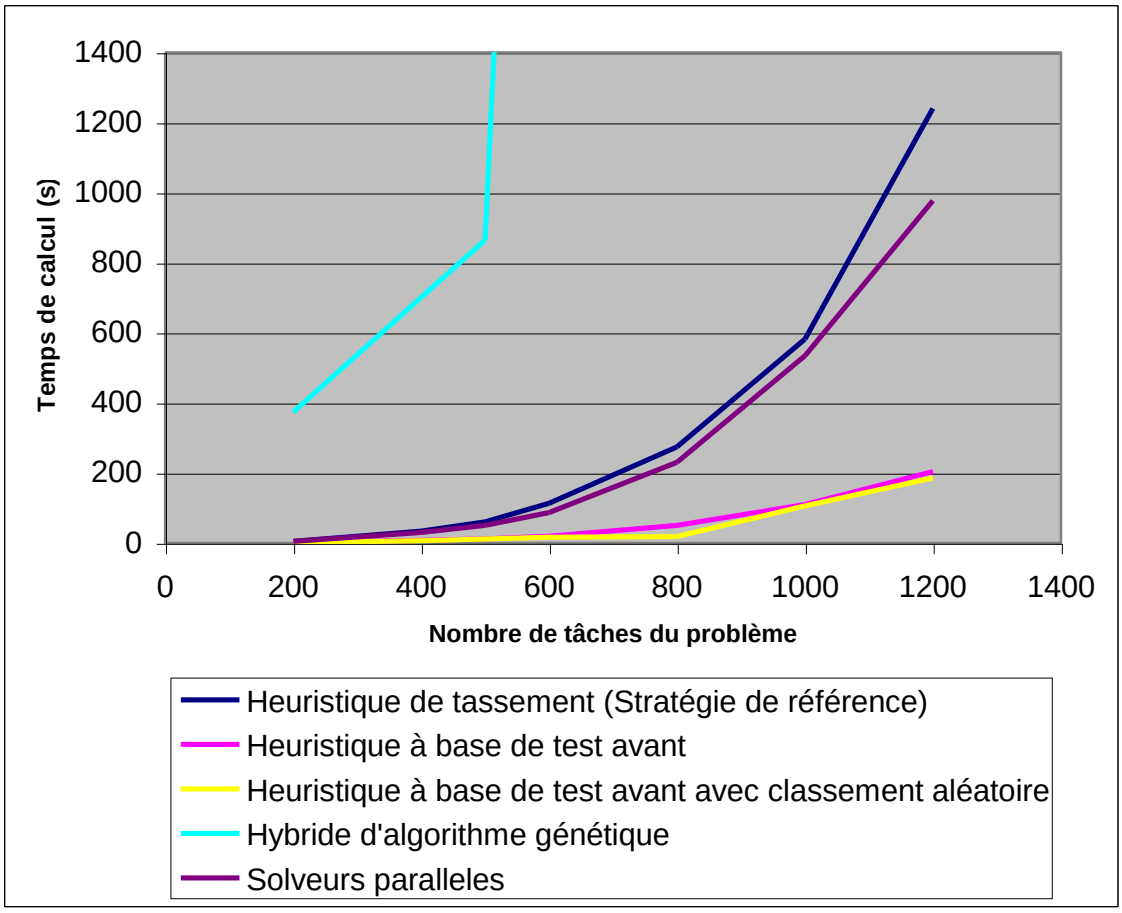
De la même façon, le problème à 400 tâches n'a pas pu être résolu dans un délai raisonnable**. Cette méthode est donc inadaptée à notre environnement technique et à notre problème.**

4 RESULTATS

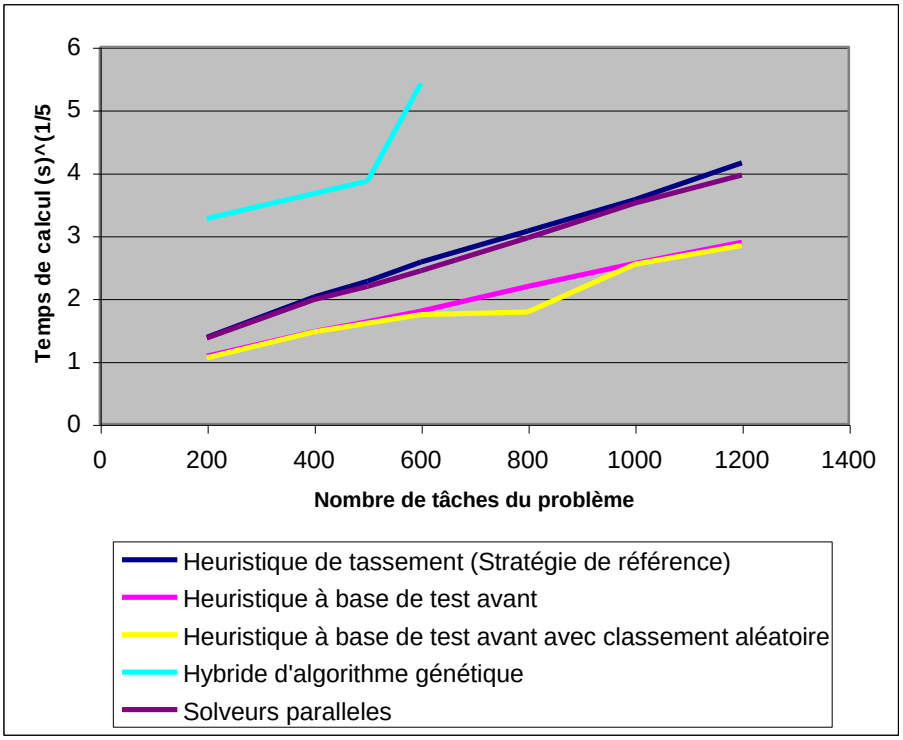
La stratégie « Heuristique à base de test avant » fait référence aux tests menés en 3.3.1.2.2.2 : « Heuristique de choix des valeurs à base de test avant – Forward-Checking pour les contraintes de non-recouvrement et de cumul »

La stratégie « Solveur parallèles » fait référence aux tests menés en 3.3.3.5.3.3 : « 2 solveurs en parallèle – résolution par « tranches ».

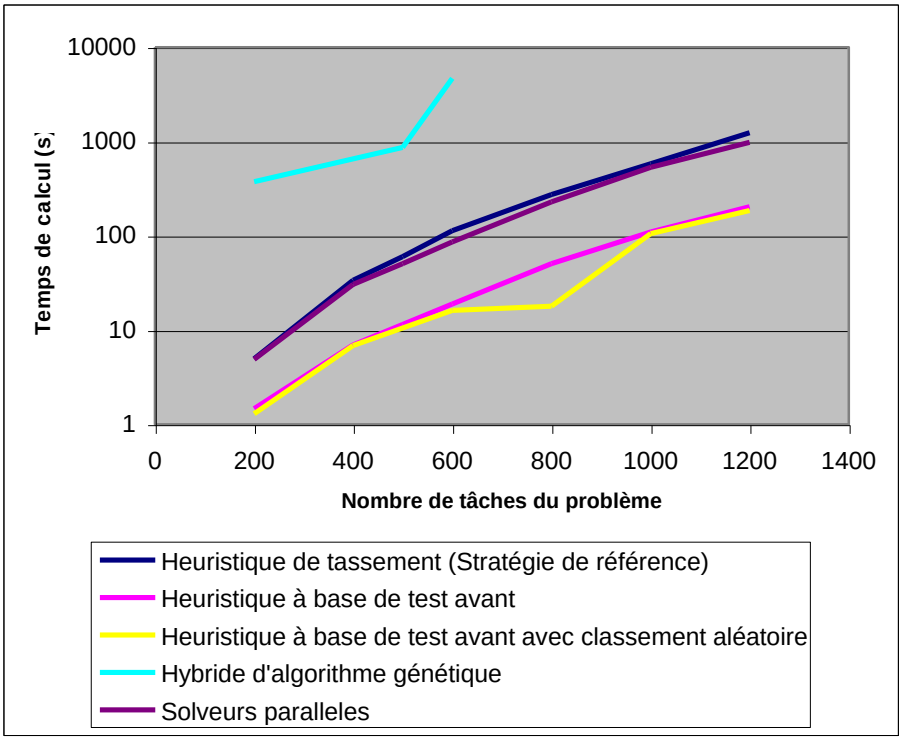
Tests sur machine Petit Pierre (iPIII 450 – 128 Mo RAM)



Temps de calcul avec une échelle y linéaire



Les temps de calcul représentés par des droites sur ce graphique croissent comme le nombre de tâches à la puissance 5



Temps de calcul avec une échelle y logarithmique

5 CONCLUSION

Ce document a présenté plusieurs méthodes récentes visant à améliorer les temps de calcul dans le cadre de la résolution de grands problèmes de satisfaction de contraintes (CSP) à valeurs discrètes. L'application de certaines de ces méthodes au projet MAXIME de génération d'emplois du temps a ensuite été traitée.

Il ressort de cette étude que les heuristiques les plus particulières à une application sont les plus performantes. Typiquement, on peut ainsi constater que l'heuristique de *test avant* agrémentée de l'évaluation préventive du taux de *cumul* (propre à MAXIME) donne de bien meilleurs résultats que la même heuristique en version « standard ». De même, la résolution concurrentielle est bien plus efficace lorsque la *séparation des domaines* est envisagée, mais cette amélioration reste propre à une certaine classe de problèmes.

Cette constatation montre qu'il faut poser le problème du compris entre coût de développement et performances.

Toutefois, l'usage de l'heuristique de *test avant* (même, éventuellement, en version standard) semble un bon choix pour de nombreux problèmes de grande taille, notamment en raison de sa simplicité et de sa grande flexibilité. On peut en effet définir rapidement et au cas par cas les indices à calculer pour anticiper le viol des principales contraintes.

Enfin, il apparaît que Ilog Solver 4.3 est un outil puissant mais dont l'interface laisse peu de possibilités de personnalisation. Ainsi, les stratégies de recherche à base de *backjumping* ne peuvent pas être implémentées de manière efficace. De plus, il se pose un problème de performances lorsque l'utilisation de la mise en consistance (propagation des contraintes) n'est pas souhaitée et vient ralentir le mécanisme d'instanciation des variables.

6 REFERENCES

[BarBri99] **Barnier N., Brisset P**. *Optimisation par algorithme génétique sous contraintes,* Technique et science informatiques, Vol. 18 – n°1, 1-29, 1999 (barnier/search.pdf)

[CleHogHub92] **Clearwater S.H., Hogg T., Huberman B.A**. *Cooperative Problem Solving* dans *Computation : The Micro and the Macro View*, B. A. Huberman, ed. World Scientific, 33-70, 1992 (clearwater/cryptarithmetic 92.ps)

[Cost94] **Costa D**. *An evolutionary tabu search algorithm and the NHL scheduling problem*, Ecole Polytechnique Fédérale de Lausanne Dpt. Mathématiques, ORWP 92-11, 1994

[Cost95] **Costa D**. *Méthodes de résolution constructives, séquentielles et évolutives pour des problèmes d'affectation sous contraintes*, Ecole Polytechnique Fédérale de Lausanne Dpt. Mathématiques, Thèse n°1411, chap. 3, 1995

[FroDec94] **Frost D., Dechter R**. *In search of the best constraint satisfaction search*, pour The Twelfth National Conference on Artificial Intelligence, 301-306, 1994 (dechter/search-for-best-search(1).ps)

[FroDec95] **Frost D., Dechter R**. *Look-ahead value ordering for constraint satisfaction problems*, Dept. of Information and Computer Science, University of California, Irvine, 1995 (dechter/look-ahead-value-ordering.ps)

[FroDec99-1] **Dechter R., Frost D**. *Evaluating Constraint Processing Algorithms*, Dept. of Information and Computer Science, University of California, Irvine, 1999 (dechter/evaluatingconstR74.ps)

[FroDec99-2] **Dechter R., Frost D**. *Backtracking algorithms for constraint satisfaction problems*, Dept. of Information and Computer Science, University of California, Irvine, 1999 (dechter/backtrackingR56.ps)

[Fros97] **Frost D.H.** *Algorithms and Heuristics for Constraint Satisfaction Problems*, University of California, Irvine, 1997. (frost/R69.ps)

[Gins93] **Ginsberg M.L.** *Dynamic Backtracking*, Journal of Artificial Intelligence Research 1, 25-46, 1993 (ginsberg/ginsberg93a.pdf)

[HarGin95] **Harvey W.D., Ginsberg L.M.** *Limited Discrepancy Search*, pour IJCAI-95, 607-613, 1995 (harvey/lds.ps)

[Hogg9x] **Hogg T**. *Exploiting Problem Structure as a Search Heuristic*, Xerox Palo Alto Research Center, >1993 (hogg/hardnessHeuristic.ps)

[Kond94] **Kondrak G.** *A.Theoretical Evaluation of Selected Backtracking Algorithms*, Dpt. of Computing Science thesis, University of Alberta, 1994 (kondrak/kondrak.ps)

[Kuma92] **Kumar V.** *Algorithms for Constraint Satisfaction Problems : A Survey*, AI Magazine 13(1) : 32-44, 1992 (kumar/kumar.ps)

[LobLem97] **Lobjois L., Lemaître M.** *Coopération entre méthodes complètes et incomplètes pour la résolution de (V)CSP : une tentative d'inventaire*, ONERA/CERT/DERI Toulouse, pour JNPC'97, 1997 (lobjois/coop-jnpc97.ps)

[Mic9x] **Michalewicz Z., Dasgupta D., Le Riche R., Schoenauer M.** *Evolutionary Algorithms for Constrained Engineering Problems*, >1994 (michalewicz/eng.ps)

[TsaKwa93] **Tsang E., Kwan A**. *Mapping Constraint Satisfaction Problems to Algorithms and Heuristics*, Technical Report CSM-198, Dpt. of Computer Science, University of Essex, 1993 (tsang/tsang.ps)

[WeaBurEll95] **Weare R., Burke E., Elliman D.** *A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems*, Dpt. of Computer Science, University of Nottingham, UK, 1995 (weare/8_ps.ps)

[WilHog92] **Williams C., Hogg T.** *The Typicality of Phase Transitions in Search*, Xerox Palo Alto Research Center SSL technical report, 1992. (hogg/searchLONG.ps). Une version réduite est dans *Computational Intelligence* 9, 221-238 (1993)

7 ANNEXES

7.1 Détails des références bibliographiques

7.1.1 [BarBri99]

Titre : Optimisation par algorithme génétique sous contraintes	**Référence biblio. :** [BarBri99]
Auteur : Barnier N., Brisset P.	**Année de parution :** 1999
Editeur / Organisme de publication / Titre du journal : Technique et science informatiques. Vol. 18 n°1/1999	
ISBN / Référence complémentaire :	**Pages / Chapitres :** 1-29
Fichier associé : barnier/search.pdf	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Cet article présente un système original d'hybridation de PPC avec un algorithme génétique. Ce système associe chaque « individu » génétique avec un sous-espace de l'espace de recherche. Ainsi, chaque gène est associé à une dimension de l'hyper-cube que forme ce sous-espace, c'est-à-dire à une variable du problème. Le découpage en sous-espaces est réalisé en affectant à un gène certaines valeurs du domaine de sa variable associée.
- Pour chaque individu, l'adaptation est mesurée en fonction de la valeur de la fonction objectif obtenue par la première solution au problème (cette résolution se fait par PPC), dans le sous-espace de cet individu. Une pénalité forte est infligée à un individu qui n'a pas de solution dans son sous-espace.
- L'évolution est donc obtenue par modification des domaines de recherche
- Cette approche est générique et s'inscrit entre une démarche purement PPC (1 seul individu ayant pour sous-espace tout l'espace du problème) et purement génétique (plusieurs individus ayant pour sous-espace un « point » de l'espace, c'est-à-dire une valeur donnée pour chacune des variables du problème). La modification de la taille des sous-espaces permet de naviguer entre ces deux extrêmes.
- Les résultats obtenus sur le problème du *VRP* (*Vehicle Routing Problem*) et *RLFAP* (*Radio Link Frequency Assignment Problem*) sont très bons.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Cette stratégie semble très intéressante pour les raisons suivantes :

- Elle ne remet pas en cause la modélisation du problème du point de vue de la PPC
- Elle permet subdiviser l'espace de recherche (temps de calcul PPC exponentiellement décroissant)
- Elle n'empêche pas d'appliquer d'autres algorithmes d'amélioration, tant pour la PPC que pour la macro-résolution (résolution concurrentielle, etc…)
- Sa couverture très large entre PPC et algorithme génétique permet de doser les effets de l'une et de l'autre des méthodes

En outre, cette méthode est un bon exemple d'hybride génétique/PPC où la satisfaction des contraintes est garantie par pénalité des individus inacceptables.

7.1.2 [CleHogHub92]

Titre : Cooperative Problem Solving	**Référence biblio. :** [CleHogHub92]
Auteur : Clearwater S. H., Hogg T., Huberman B. A.	**Année de parution :** 1992
Editeur / Organisme de publication / Titre du journal : *Computation : The Micro and the Macro View*, B. A. Huberman, ed. World Scientific	
ISBN / Référence complémentaire :	**Pages / Chapitres :** 33-70
Fichier associé : clearwater/cryptarithmetic 92.ps	**Support papier disponible :** oui

Points abordés et principes généraux :

- Etude des mécanismes de résolution parallèles, avec et sans coopération
- L'étude porte sur les problèmes *cryptarithmétiques* (DONALD + GERALD = ROBERT, etc…)
- La coopération s'effectue par partage d'*astuces* (*hints*) et non par échange de *nogoods* (cette dernière solution est souvent utilisée mais n'est pas implémentable avec Ilog Solver)

Le résultat de la coopération de *n* agents est meilleur que *n* fois le résultat d'un agent (*Super-linear Speed-up*)

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Ce document semble assez complet en ce qui concerne les méthodes de résolution constructives parallèles. Toutefois, les exemples traités sont assez éloignés de notre problème.

7.1.3 [Cost94]

Titre : An evolutionary tabu search algorithm and the NHL scheduling problem	**Référence biblio. :** [Cost94]
Auteur : Costa D.	**Année de parution :** 1994
Editeur / Organisme de publication / Titre du journal : Ecole Polytechnique Fédérale de Lausanne, Dpt. Mathématiques	
ISBN / Référence complémentaire : ORWP 92-11	**Pages / Chapitres :**
Fichier associé :	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Présentation générale des algorithmes génétiques en utilisation classique (2 p.)
- Présentation générale de l'algorithme du Tabu en utilisation classique (2 p.)
- Présentation d'un concept d'hybridation entre algorithme génétique et Tabu (1 p.), ou la phase de mutation est remplacée par une phase d'optimisation Tabu

Application détaillée de ce principe au planning de la NHL. La façon dont les contraintes sont triées entre *essential* et *relaxed* est intéressante. Un certain nombre de contraintes importantes sont passées dans le groupe *relaxed* afin d'alléger le taux de contrainte du problème.

La résolution générale passe par des améliorations successives d'une population de solutions de base. Trouver cette population initiale ne semble pas poser beaucoup de problèmes, et n'influence pas significativement la solution finale.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

La méthode (comme tous les algorithmes génétiques standards) semble plutôt être applicable à des problèmes peu contraints, où la génération de la population de base est aisée.

La façon de relaxer certaines contraintes pour obtenir des problèmes moins contraints est intéressante.

L'hybridation de l'AG peut être retenue afin d'améliorer la phase de mutation.

7.1.4 [Cost95]

Titre : Méthodes de résolution constructives, séquentielles et évolutives pour des problèmes d'affectation sous contraintes	**Référence biblio. :** [Cost95]
Auteur : Costa D.	**Année de parution :** 1995
Editeur / Organisme de publication / Titre du journal : Ecole Polytechnique Fédérale de Lausanne, Dpt. Mathématiques	
ISBN / Référence complémentaire : Thèse n° 1411	**Pages / Chapitres :** Chap. 2 et 3
Fichier associé :	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Présentation générale de méthodes d'optimisation combinatoire constructives, séquentielles et évolutives.
- Application directe à des problèmes de confection d'horaires, assez identique à [Cost94]

Conclusion sur l'intérêt du document par rapport au sujet étudié :

On peut noter la présentation des algorithmes génétiques et surtout de l'algorithme de la fourmi, qui peuvent ouvrir de bonnes perspectives.

7.1.5 [FroDec94]

Titre : In search of the best constraint satisfaction search	**Référence biblio. :** [FroDec94]
Auteur : Frost D., Dechter R.	**Année de parution :** 1994
Editeur / Organisme de publication / Titre du journal : In Proceedings of the Twelfth National Conference on Artificial Intelligence	
ISBN / Référence complémentaire :	**Pages / Chapitres :** p. 301-306
Fichier associé : dechter/search-for-best-search(1).ps	**Support papier disponible :** oui

Points abordés et principes généraux :

- Comparaison sur des Constraint Solving Problems générés aléatoirement de différents algorithmes et heuristiques de résolution : *backtracking* simple, *backmarking*, *forward checking* et *conflict-directed backjumping* pour les algorithmes et *min-width* (heuristique statique de choix des variables), *DVO* (*Dynamic Variables Ordering* : heuristique dynamique de choix des variables) et une méthode originale *sticking value* (choix des valeurs) pour les heuristiques.
- Les tests se veulent aussi génériques que possible (sans se focaliser sur une classe de problèmes)
- Les résultats montrent une très faible différence entre *backtracking* et *forward-checking* lorsqu'une heuristique dynamique de choix des variables est utilisée.
- Le *backmarking* apporte une amélioration considérable au *backjumping*
- L'heuristique de choix des valeurs *sticking value* améliore considérablement le *backjumping*.
- L'apport d'une méthode dynamique de choix des variables comme *DVO* est considérable quel que soit l'algorithme.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

- *DVO* est un système qui est implémenté « en standard » dans Ilog Solver (stratégie *IlcChooseMinSizeInt*)
- L'heuristique *sticking value* peut se révéler utile et pourrait être mise à profit dans le cas d'une résolution par un système utilisant des algorithmes parallèles. Telle qu'elle est présentée ici, cette heuristique n'est vraisemblablement pas utilisable, car elle n'a de sens qu'avec un algorithme de *backjumping.*

7.1.6 [FroDec95]

Titre : Look-ahead value ordering for constraint satisfaction problems	**Référence biblio. :** [FroDec95]
Auteur : Frost D., Dechter R.	**Année de parution :** 1995
Editeur / Organisme de publication / Titre du journal : Dept. of Information and Computer Science, University of California, Irvine	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : dechter/look-ahead-value-ordering.ps	**Support papier disponible :** oui

Points abordés et principes généraux :

- Description très sommaire des systèmes à base de *forward checking* (1/2 p.)
- Description d'un système d'ordonnancement des variables par taille croissante du domaine restant (*DVO*, comme dans [FroDec94]), et combinaison de ce système avec un système d'ordonnancement des valeurs
- Dans ce cadre, 4 heuristiques d'ordonnancement sont étudiées (chapitre 3.2). Ces heuristiques utilisent les informations données par l'application du *forward checking* pour déterminer la valeur qui amènera les domaines les plus grands sur les variables restant à instancier. La première (« min-conflicts » ou MC) est clairement la meilleure.
- Des résultats expérimentaux très précis sont présentés. Ils démontrent que l'heuristique MC est bénéfique pour des problèmes faisant plus de 100000 tests de consistance environ.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Le système semble être profitable dans notre cas (problème très contraint). Il devrait être utilisable (la propagation de contraintes est une sorte de *look-ahead*).

Avec Ilog Solver, ce type de système peut vraisemblablement être mis en place, mais ces performances ne sont pas assurées en raison de la mise en cohérence trop forte effectuée par Solver.

7.1.7 [Fros97]

Titre : Algorithms and Heuristics for Constraint Satisfaction Problems	**Référence biblio. :** [Fros97]
Auteur : Frost D. H.	**Année de parution :** 1997
Editeur / Organisme de publication / Titre du journal : University of California PhD Thesis, Irvine	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : frost/R69.ps	**Support papier disponible :** Partiel

Points abordés et principes généraux :

Description détaillée avec tests génériques et algorithmes explicites des points suivants :

- Algorithmes utilisant le Backtracking et le Look-Ahead
- Algorithmes de mise en consistance
- Algorithmes avec apprentissage des impasses
- Heuristiques de choix des valeurs et de choix des variables
- Hybrides (BJ + DVO + LRN + LVO)

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Ce document est extrêmement complet. Il présente pratiquement toutes les méthodes de résolution constructives.

Pour la partie théorique, il surclasse [FroDec94], [FroDec95], [FroDec99-1]. Ces documents apportent, par contre, d'autres tests et d'autres exemples.

[FroDec99-1]

[FroDec99-1]	Dechter R., Frost D. *Evaluating Constraint Processing Algorithms*, Dept. of Information and Computer Science, University of California, Irvine, 1999. (dechter/evaluatingconstR74.ps)

[FroDec99-1] est un document de synthèse de [Fros97] qui reprend beaucoup de résultats quantitatifs mais ne donne pas la théorie sous-jacente. Il fait également l'impasse sur les algorithmes.

[FroDec99-2]

[FroDec99-2]	Dechter R., Frost D. *Backtracking algorithms for constraint satisfaction problems*, Dept. of Information and Computer Science, University of California, Irvine, 1999. (dechter/backtrackingR56.ps)

[FroDec99-2] est un document de synthèse de [Fros97] qui reprend les algorithmes et un résumé de la théorie, mais ne présente aucun résultat expérimental.

7.1.8 [Gins93]

Titre : Dynamic Backtracking	**Référence biblio. :** [Gins93]
Auteur : Ginsberg M.L.	**Année de parution :** 1993
Editeur / Organisme de publication / Titre du journal : Journal of Artificial Intelligence Research	
ISBN / Référence complémentaire :	**Pages / Chapitres :** 1, 25-46
Fichier associé : ginsberg/ginsberg93a.pdf	**Support papier disponible :** Non

Points abordés et principes généraux :

- Présentation des algorithmes *BackJumping* (BJ) et d'un nouveau système « Dynamic Backtracking »
- Comparaisons avec preuves mathématiques
- Résultats pratiques

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Ce document ne fait que détailler une partie de [Kond94]

7.1.9 [HarGin95]

Titre : Limited Discrepancy Search	**Référence biblio. :** [HarGin95]
Auteur : Harvey W. D., Ginsberg L. M.	**Année de parution :** 1995
Editeur / Organisme de publication / Titre du journal *: In Proc. of* IJCAI-95	
ISBN / Référence complémentaire :	**Pages / Chapitres :** 607-613
Fichier associé : harvey/lds.ps	**Support papier disponible :** oui

Points abordés et principes généraux :

Ce document présente une méthode de recherche constructive inhabituelle, dans le cas de variables booléennes :

Dans un *backtrack* (ou *backjump*) classique, une heuristique « aide » à explorer les solutions (les « feuilles » de l'arbre des solutions) dans un ordre supposé avantageux. Mais un choix erroné (de l'heuristique) effectué près de la racine de l'arbre ne peut être corrigé avant d'avoir exploré tout le sous-arbre (de grande taille) correspondant. Or, les auteurs notent que la plupart des heuristiques sont moins fiables près de la racine de l'arbre que près des feuilles.

La solution proposée est d'explorer d'abord la solution donnée par les valeurs préconisées par l'heuristique à tous les nœuds de l'arbre ; puis, d'explorer toutes les solutions données par les valeurs préconisées par l'heuristique à tous les nœuds de l'arbre sauf un ; etc…

Les tests sont très encourageants mais ne portent que sur des problèmes booléens ayant une densité de solution élevée (1/1000) et une bonne heuristique (choisissant la bonne valeur dans au moins 90% des cas.)

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Notre problème bénéficie effectivement d'une bonne heuristique (*tassement*), qui est vraisemblablement moins efficace en début de résolution puisqu'elle se base sur les affectations déjà faites pour déterminer celles à venir. Notons également qu'Ilog Solver effectue systématiquement la propagation des affectations, ce qui renforce largement les chances qu'une valeur choisie par l'heuristique soit acceptable.

Ce système est facile à implémenter avec Ilog Solver.

Par contre, le document laisse ouverte la question de l'application de cette méthode à des problèmes non-booléens.

7.1.10 [Hogg9x]

Titre : Exploiting Problem Structure as a Search Heuristic	**Référence biblio. :** [Hogg9x]
Auteur : Hogg T.	**Année de parution :** >1993
Editeur / Organisme de publication / Titre du journal : Xerox Palo Alto Research Center	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : hogg/hardnessHeuristic.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Il s'agit d'une heuristique de choix des valeurs et des variables qui est basée sur la constatation suivante : on trouve rapidement (facilement) une solution pour un problème soluble peu contraint, et on trouve rapidement qu'il n'y a pas de solution à un problème insoluble très contraint. Entre ces deux extrêmes, il y a une transition aux environs de laquelle une catégorie intermédiaire de problèmes (solubles ou insolubles) existe. Pour ces problèmes, les temps de calcul sont longs. On tire alors de cette règle une heuristique qui ordonne les valeurs et les variables d'après une estimation de la difficulté du sous-problème restant à chaque étape de la résolution.
- On arrive également à connaître la probabilité de *trouver* une solution dans un certain sous-problème, ce qui permet de prendre également en considération ce paramètre.
- Les heuristiques finalement choisies sont les suivantes : les valeurs qui ont le plus de chance de succès sont choisies en premier (on peut ensuite les départager en utilisant une heuristique locale) ; les variables sont choisies en fonction d'une heuristiques locale (on départage ensuite en prenant en premier les variables qui amènent les temps de calcul les plus courts.)
- Le cas étudié est un problème de coloration de graphes à 3 couleurs (*3-coloring*)
- La théorie est basée sur la connaissance d'un paramètre *m* définissant le nombre d'assignations inconsistantes (*nogoods*) dans un sous-problème.
- Cette théorie s'applique à des CSP binaires uniquement.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Le système est intéressant et propose une théorie plus évoluée que la LVO de [FroDec95] puisqu'elle prend en considération non seulement la taille des domaines restant dans un sous-problème mais également le taux de contrainte restant dans ce sous-problème.

Le point délicat concerne la connaissance de la valeur du paramètre *m* : il faut savoir combien de couples de valeurs sont interdits dans le sous-problème restant (voir document de *notes*.)

7.1.11 [WilHog92]

- Une théorie plus précise sur l'origine de cette heuristique est dans :

[WilHog92]	Williams C., Hogg T. *The Typicality of Phase Transitions in Search*, Xerox Palo Alto Research Center SSL technical report, 1992. (hogg/searchLONG.ps). Une version réduite est dans *Computational Intelligence* 9, 221-238 (1993)

7.1.12 [Kond94]

Titre : A Theoretical Evaluation of Selected Backtracking Algorithms	**Référence biblio. :** [Kond94]
Auteur : Kondrak G.	**Année de parution :** 1994
Editeur / Organisme de publication / Titre du journal : Department of Computing Science Thesis, University of Alberta	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : kondrak/kondrak.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Présentation de divers algorithmes de test arrière. Notamment, détail des algorithmes « simples » (BT/BJ/CBJ/FC) et détail des algorithmes sophistiqués et hybrides (BMJ/GBJ/hybrides FC)
- Présentation de deux algorithmes améliorés BMJ2 et BM-CBJ2
- Comparaisons avec preuves mathématiques des différentes méthodes. Hiérarchisation des performances de ces méthodes.
- Résultats comparatifs expérimentaux sur des problèmes standards

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Il s'agit d'une bible sur le test arrière, avec les algorithmes complets de toutes les méthodes. Le problème reste que ces algorithmes nécessitent une connaissance plus ou moins précise des inconsistances après une affectation pour pouvoir backtracker efficacement. Celles-ci ne seront probablement pas possible à déterminer dans le cadre de ce projet, en raison de l'utilisation d'Ilog Solver.

Les comparaisons mathématiques présentées sont très claires.

7.1.13 [Kuma92]

Titre : Algorithms for Constraint Satisfaction Problems : A Survey	**Référence biblio. :** [Kuma92]
Auteur : Kumar V.	**Année de parution :** 1992
Editeur / Organisme de publication / Titre du journal : AI Magazine	
ISBN / Référence complémentaire :	**Pages / Chapitres :** 13(1) : 32-44
Fichier associé : kumar/kumar.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Inventaire des méthodes de résolution de Constraint Satisfaction Problems pour des CSP binaires : Backtracking simple, propagation de contraintes (notamment : détail des mises en consistance), Backtracking intelligent
- Discussion sur les heuristiques élémentaires dans le domaine de la PPC : ordonnancement des variables et des valeurs
- Discussion sur le dosage de la propagation de contraintes par rapport au Backtracking. Classification d'algorithmes de Backtracking « sophistiqués » en fonction de leur taux d'inclusion ou de ressemblance avec de la propagation de contraintes.

Les algorithmes ne sont pas présentés en détail (par contre, la bibliographie est énorme).
L'extrapolation de l'étude à des CSP non-binaires est présentée comme aisée.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Il s'agit d'un survol de toutes les méthodes de résolution de CSP par la PPC.
Les lignes directrices de l'amélioration des performances dans le cadre de la pure PPC sont faciles à déduire de ce document (dosage du backtracking et de la propagation, choix des valeurs et des variables).

7.1.14 [LobLem97]

Titre : Coopération entre méthodes complètes et incomplètes pour la résolution de (V)CSP : une tentative d'inventaire	**Référence biblio. :** [LobLem97]
Auteur : Lobjois L., Lemaître M.	**Année de parution :** 1997
Editeur / Organisme de publication / Titre du journal : ONERA/CERT/DERI, Toulouse**,** pour JNPC'97	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : lobjois/coop-jnpc97.ps	**Support papier disponible :** oui

Points abordés et principes généraux :

Ce document tente de recenser les méthodes de résolution de CSP qui mixent les approches *complètes* (constructives ; explorant à terme toutes les solutions) et *incomplètes* (explorant la fraction de l'espace des solutions supposée la plus favorable). Le but d'une telle méthode *hybride* est d'obtenir les avantages des deux catégories d'algorithmes : rapidité pour les méthodes incomplètes, et exhaustivité et facilité de paramétrage pour les méthodes complètes.

Les points suivants sont abordés :

1. Schémas basés sur des méthodes complètes
- Greffe de mécanismes incomplets sur une souche complète
- Affaiblissement des méthodes complètes
2. Schémas basés sur des méthodes incomplètes
- Résolution alternée
- Exploration du voisinage
- Orientation par renforcement de cohérence
3. Schémas de coopération par cohabitation
- Partage du travail
- Attaque simultanée
4. Schémas de coopération indirecte
- Bornes inférieures par relaxations
- Recherche de propriétés structurales

Conclusion sur l'intérêt du document par rapport au sujet étudié :

On peut faire les commentaires suivants sur les méthodes proposées :

1. *Schémas basés sur des méthodes complètes*
- *Greffe de mécanismes incomplets sur une souche complète* : dans cette catégorie rentre l'heuristique [Hogg9x]
- *Affaiblissement des méthodes complètes* : typiquement, cette section comporte la méthode [HarGin95]

2. Schémas basés sur des méthodes incomplètes

- *Résolution alternée* : cette catégorie vise à alterner *temporellement* les deux types de méthodes, c'est-à-dire à résoudre dans un premier temps une partie du problème avec une méthode, puis une autre avec l'autre méthode. Ceci constitue une stratégie très générique qui peut facilement s'adapter à tous les algorithmes existants.
- *Exploration du voisinage* : la méthode [BarBri97] correspond à ce schéma
- *Orientation par renforcement de cohérence* : ce schéma n'utilise pas à proprement parler de méthode « constructive ». Il sera abandonné dans le cadre de ce projet.

3. Schémas de coopération par cohabitation

- *Partage du travail* : Ce système suppose un « choix » entre l'une ou l'autre méthode en fonction du problème à traiter. Il semble clair que ce système ne s'applique qu'à la conception d'un solveur générique.
- *Attaque simultanée* : Le but est ici de lancer simultanément les 2 méthodes sur le même problème. Sur une machine unique, on peut raisonnablement penser qu'une forme de coopération est plus efficace qu'une résolution parallèle brute.

4. Schémas de coopération indirecte

- *Bornes inférieures par relaxations* : Voilà encore une méthode très générique qui pourra s'appliquer à tous les algorithmes : une solution du problème relaxé constitue une borne inférieure pour le problème de départ, et est donc susceptible d'orienter la recherche sur ce problème.
- *Recherche de propriétés structurales* : Pour appliquer ce genre de méthodes, il faut vraisemblablement être dans un cadre théoriquement plus simple et plus « modélisable » que celui de notre problème.

Ce document ne traite pas des méthodes uniquement constituées de systèmes constructifs comme, par exemple, la résolution *parallèle* par deux algorithmes constructifs s'attaquant simultanément au même problème.

7.1.15 [Mic9x]

Titre : Evolutionary Algorithms for Constrained Engineering Problems	**Référence biblio. :** [Mic9x]
Auteur : Michalewicz Z., Dasgupta D., Le Riche R., Schoenauer M.	**Année de parution :** >1994
Editeur / Organisme de publication / Titre du journal :	
ISBN / Référence complémentaire :	**Pages / Chapitres :**
Fichier associé : michalewicz/eng.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Description des principes de gestion des contraintes dans les résolutions avec des algorithmes génétiques.
- Etude de 3 exemples mettant en œuvre des stratégies différentes
- Les méthodes étudiées pour prendre en compte les contraintes sont les suivantes : rejet des individus inacceptables (c'est-à-dire qui ne respectent pas les contraintes), pénalisation des individus inacceptables, maintient de la recherche dans la population des solutions acceptables en utilisant des opérateurs spécifiques, réparation des individus inacceptables, remplacement des individus inacceptables, usage de décodeurs et séparation des contraintes avec optimisation multi-objectifs.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

L'utilisation d'algorithmes génétiques purs ne rentre pas dans le cadre de ce projet, c'est pourquoi une bonne partie de ce document n'est pas réellement utilisable. Néanmoins, une partie des méthodes décrites peuvent s'appliquer dans le cas d'une hybridation d'un algorithme génétique avec de la PPC classique.

Ainsi, les principes suivants sont à retenir, si on les considère, par exemple, dans le cadre de la solution décrite par [BarBri97] :

1. Rejet des individus inacceptables
2. Pénalisation des individus inacceptables
3. Usage de décodeurs

De plus, la méthode de maintien de la recherche dans la population acceptable en utilisant des opérateurs spécifiques correspond directement au principe mis en œuvre dans [WeaBurEll95].

7.1.16 [TsaKwa93]

Titre : Mapping Constraint Satisfaction Problems to Algorithms and Heuristics	**Référence biblio. :** [TsaKwa93]
Auteur : Tsang E., Kwan A.	**Année de parution :** 1993
Editeur / Organisme de publication / Titre du journal : Dpt. of Computer Science, University of Essex	
ISBN / Référence complémentaire : Technical Report CSM-198	**Pages / Chapitres :**
Fichier associé : tsang/tsang.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Inventaire et classification des algorithmes et des heuristiques de résolution pour les Constraint Satisfaction Problems (environ 30). Cet inventaire traite les algorithmes « complets », les algorithmes de recherche stochastique à but local (*Tabu*, *Hill Climbing*, *Simulated Annealing*) ou global (algorithmes génétiques, méthodes connectionnistes). Les algorithmes ne sont pas détaillés.
- Analyse de l'adaptation de toutes ces méthodes à différents types de problèmes, en fonction des critères suivants : nombre de solutions demandé, temps alloué, taux de contrainte du problème, type de graphe primaire du CSP.

Conclusion sur l'intérêt du document par rapport au sujet étudié :

Ce document a le mérite de faire un tour des différentes méthodes disponibles. Malheureusement, il est finalement peu détaillé en ce qui concerne les critères qui amènent le choix de telle ou telle méthode.

7.1.17 [WeaBurEll95]

Titre : A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems	**Référence biblio. :** [WeaBurEll95]
Auteur : Weare R., Burke E., Elliman D.	**Année de parution :** 1995
Editeur / Organisme de publication / Titre du journal : Dpt. of Computer Science, University of Nottingham, UK	
ISBN / Référence complémentaire : Technical Report NOTTCS-TR-95-8	**Pages / Chapitres :**
Fichier associé : weare/8_ps.ps	**Support papier disponible :** Oui

Points abordés et principes généraux :

- Traitement d'un problème de génération d'emploi du temps pour une université. Ce problème est très contraint mais est assez simple à aborder (types de contraintes limités)
- Le système utilise un algorithme génétique pour parcourir l'espace des solutions. La satisfaction des contraintes est assurée via l'incorporation de PPC à l'intérieur de l'opérateur (unique) de reproduction / mutation

Conclusion sur l'intérêt du document par rapport au sujet étudié :

C'est un bon exemple d'algorithme hybride génétique/PPC où la satisfaction des contraintes est garantie en n'utilisant, pour les « individus » génétiques, que des solution « admissibles » (satisfaisant les contraintes).

Le problème de ce type de méthode dans notre cas est de disposer de solutions initiales en nombre suffisant, et ensuite de pouvoir assez facilement générer de nouvelles solutions (toujours admissibles) par des interventions suffisamment simples sur les précédentes.