Etude expérimentale de recherche locale pour la résolution de contraintes

An Experimental Study of Local Search for Constraint Solving

M. Jin-Kao Hao M. Jérôme Pannier

LGI2P/EMA-EERIE
Parc Scientifique Georges Besse
30000 Nîmes - France
email: {hao, pannier}@eerie.fr

Résumé

Nous présentons dans cet article une étude expérimentale de recherche locale pour la résolution de contraintes. En particulier, nous expérimentons deux algorithmes fondés sur le recuit simulé et la recherche tabou pour les problèmes de satisfaction maximale de contraintes. Ces deux algorithmes sont testés sur des instances de taille importante allant de 100 à 300 variables avec 10 à 30 valeurs par variable. Les résultats montrent que la recherche tabou domine le recuit simulé sur les jeux testés tant en terme de qualité des solutions trouvées qu'en terme de rapidité de résolution. Nous avançons des arguments empiriques pour expliquer cette différence de performance.

Mots Clef

Recherche locale, recuit simulé, recherche tabou, résolution de contraintes.

Abstract

In this paper, we present an experimental study of local search for constraint satisfaction. For this purpose, we experiment with two algorithms based on Simulated Annealing (SA) and Tabu Search (TS) for solving the maximal constraint satisfaction problem. These two algorithms were tested on various large instances going from 100 to 300 variables with 15 to 30 values per variable. Experimental results show that the TS algorithm dominates the RS algorithm on all the tested instances in terms of solution quality and solving speed. We propose empirical arguments to explain the difference of performances.

Keywords

Local Search, Simulated Annealing, Tabu Search, Constraint Satisfaction.

1 Introduction

La résolution pratique des problèmes sous contraintes, que ce soit la satisfaction ou l'optimisation, occupe une place très importante en Intelligence Artificielle (IA). Informellement, la résolution d'un tel problème consiste à trouver des affectations de valeurs à l'ensemble des variables du problème tout en respectant un ensemble de contraintes et éventuellement en optimisant une fonction de coût donnée.

Les problèmes de contraintes englobent des problèmes de référence tels que la coloration et la satisfiabilité en logique propositionnelle. Ils possèdent également un très grand nombre d'applications pratiques relevant de l'affectation de ressources, de la planification, du groupement, de l'emploi du temps ou encore de l'ordonnancement. Les problèmes de contraintes sont en général NP-complets (ou NP-difficiles pour l'optimisation) et donc ne possèdent pas de solution algorithmique efficace. Etant donné leur importance, les problèmes de contraintes ont fait l'objet de très nombreuses études ces dernières années et constituent actuellement un des domaines les plus actifs en IA.

Les méthodes connues à ce jour pour la résolution de contraintes peuvent être classées en deux grandes catégories : les méthodes complètes (exactes pour l'optimisation) qui garantissent la complétude de la résolution et les méthodes incomplètes (approchées pour l'optimisation) qui perdent la complétude pour gagner en efficacité.

Les méthodes complètes sont en général fondées sur le principe de construction successive combiné avec un mécanisme de retour arrière [10, 17, 20, 24]. Ces méthodes emploient de nombreuses heuristiques pour mieux guider le choix de la prochaine variable à instancier et le choix de sa valeur. Elles utilisent également des techniques puissantes de filtrage et de consistance. Le principal avantage d'une méthode complète est la complétude de la résolution : une telle méthode est théoriquement capable de démontrer l'existence (ou la non-existence) d'une solution d'une instance donnée. Elle est également capable de trouver une ou toutes les solutions sans contrainte de temps. En ce qui con-

cerne les méthodes exactes pour les problèmes d'optimisation de contraintes tels que la satisfaction maximale de contraintes, des méthodes sont souvent fondées sur le principe de séparation et évaluation progressive [4, 16, 25].

Les méthodes complètes (exactes) demandent en général un temps de calcul exponentiel en la taille du problème à résoudre et se limitent donc à des instances de taille raisonnable. Lorsque les instances à traiter sont de très grande taille, les méthodes incomplètes (approchées pour l'optimisation), comme la recherche locale, représentent une alternative très intéressante bien que la complétude de la résolution ne soit plus garantie.

La recherche locale a été appliquée à des classes particulières de problèmes sous contraintes comme par exemple la coloration de graphes [11, 1, 13] et la satisfiabilité maximale (Max-SAT) [8]. Le travail présenté dans [18] constitue un des premiers travaux en IA sur la recherche locale (appelée dans [18] "méthode de réparation") appliquée à la satisfaction de contraintes. L'heuristique "min-conflits" proposée correspond en réalité à une simple descente sans capacité à surmonter le problème d'optima locaux. De nombreuses améliorations et extensions ont alors été proposées [22, 26, 9].

Dans ce travail, nous nous intéressons à la résolution pratique de problèmes sous contraintes généraux avec la recherche locale. En particulier, nous adaptons deux méthodes bien connues en en Recherche Opérationnelle (RO), à savoir, le recuit simulé (RS) et la recherche tabou aux problèmes de satisfaction maximale de contraintes. Nous étudions le comportement de ces algorithmes sur des instances aléatoires de taille importante allant de 100 à 300 variables avec 10 à 30 valeurs par variable. Cette étude constitue la suite du travail décrit dans [5].

Après l'introduction des problèmes de contraintes (section 2), nous présentons la recherche locale et notamment les principes du recuit simulé et de la recherche tabou (section 3). Nous détaillons les algorithmes du RS et de la recherche tabou adaptés aux problèmes de contraintes (section 4). Nous présentons ensuite les jeux de tests, les résultats expérimentaux ainsi qu'une analyse de ces résultats (section 5). Nous terminons l'article par quelques conclusions.

2 Problèmes de contraintes

Les problèmes de contraintes peuvent être introduits à l'aide de la notion de réseau de contraintes. Un réseau de contraintes P est défini par un triplet $< V, D, C >$ où :

- V est un ensemble fini de variables ;
- D est une collection finie de domaines associés aux variables ;
- C est un ensemble de contraintes sur des sous-ensembles de variables ; une contrainte sur k variables est un sous-ensemble du produit cartésien des domaines des k variables spécifiant les combinaisons de valeurs autorisées (ou interdites).

Etant donné un réseau de contraintes $< V, D, C >$, le problème de satisfaction de contraintes (CSP - pour constraint satisfaction problem) associé consiste à trouver des assignations de valeurs dans D aux variables de V respectant toutes les contraintes de C [17] [1]. Le problème de satisfaction maximale de contraintes (MCSP - pour maximal constraint satisfaction problem) est un problème d'optimisation qui consiste à trouver des assignations respectant le maximum de contraintes [4]. Il est également possible d'associer un poids positif p_i à chaque contrainte $C_i \in$ C et de définir le problème de satisfaction maximale de contraintes pondérées (WMCSP - pour weighted maximal constraint satisfaction problem) pour lequel on cherche à maximiser la somme des contraintes satisfaites. Il est évident que les MCSP sont un cas particulier des WMCSP où la pondération de contraintes est égale à un.

Dans cette étude, nous nous intéressons essentiellement à la résolution de contraintes dans les MCSP. Il n'est pas difficile de constater que l'approche présentée s'applique aussi bien aux CSP qu'aux WMCSP. En effet, les CSP peuvent être traités comme un cas particulier de MCSP dont l'objectif recherché est d'atteindre un coût nul. De même, les WMCSP sont des MCSP avec pondération de contraintes supérieure à un.

3 Recherche locale

3.1 Présentation générale

La recherche locale représente une classe importante de méthodes heuristiques pour l'optimisation basées sur la notion de voisinage [21]. Traditionnellement, la recherche locale constitue une arme redoutable pour attaquer des problèmes réputés très difficiles tels que le voyageur de commerce. Depuis une dizaine d'années, des progrès importants ont été réalisés en matière de recherche locale grâce à la découverte de nouvelles métaheuristiques à la fois puissantes et générales. Ces nouvelles métaheuristiques de recherche locale sont fondées sur des principes généraux et sont adaptables à une large classe de problèmes. Le recuit simulé et la recherche tabou représentent deux exemples bien connus. Grâce à ces métaheuristiques, on peut traiter aujourd'hui des problèmes d'optimisation classiques de plus grande taille ainsi que de très nombreuses applications qui étaient intraitables auparavant [15].

Pour traiter avec la recherche locale un problème d'optimisation (S, f) défini par l'ensemble S des configurations et la fonction coût f, il est nécessaire de définir un voisinage.

Définition 1 : Soit S l'ensemble des configurations d'un problème, on appelle *voisinage* toute application N: $S \rightarrow 2^S$.

1. A cause de l'incomplétude des méthodes de recherche locale, on s'intéresse ici seulement à la recherche de solutions pour une instance donnée.

Une configuration s est un *optimum (minimum) local* pour le voisinage N si $f(s) \leq f(s')$ pour toute configuration $s' \in N(s)$.

Une procédure typique de recherche locale débute avec une configuration initiale, et réalise ensuite une série de *mouvements* qui consiste à remplacer la configuration courante par l'un de ses voisins en tenant compte de la fonction de coût. Ce processus itératif s'arrête et retourne la meilleure configuration trouvée quand la condition d'arrêt est réalisée. Cette condition d'arrêt concerne généralement une limite pour le nombre de mouvements ou un objectif à réaliser. Pour un voisinage donné, les méthodes de recherche locale diffèrent essentiellement entre elles par la stratégie de parcours de ce voisinage.

3.2 Recuit simulé

Le recuit simulé est une méthode de recherche locale avancée s'inspirant du processus de recuit physique [14]. Un algorithme du RS répète une procédure de réparation qui cherche des configurations de coût plus faible tout en acceptant de manière contrôlée des configurations qui dégradent la fonction coût.

Le déroulement est le suivant : à chaque nouvelle itération, un voisin $s' \in N(s)$ de la configuration courante s est généré de manière aléatoire. Selon les cas, ce voisin sera soit retenu pour remplacer celle-ci, soit rejeté. Si ce voisin est de performance supérieure ou égale à celle de la configuration courante, i.e. $f(s') \leq f(s)$, il est systématiquement retenu. Dans le cas contraire, s' est accepté avec une probabilité définie par $e^{(-\Delta)/t}$. Cette probabilité dépend de deux facteurs : d'une part l'importance de la dégradation $\Delta = f(s') - f(s)$ (les dégradations plus faibles sont plus facilement acceptées), d'autre part d'un paramètre de contrôle t, la température (une température élevée correspond à une probabilité plus grande d'accepter des dégradations). La température est contrôlée par une fonction décroissante qui définit un schéma de refroidissement. En pratique, l'algorithme s'arrête et retourne la meilleure configuration trouvée lorsqu'aucune configuration voisine n'a été acceptée pendant un certain nombre d'itérations à une température ou lorsque la température atteint la valeur zéro ou encore lorsqu'un nombre prédéfini de mouvements autorisés est réalisé.

La performance du RS dépend largement du schéma de refroidissement utilisé. De nombreux schémas théoriques et pratiques ont été proposés. Il existe des schémas qui garantissent la convergence asymptotique de l'algorithme. Cependant, ces schémas théoriques ont peu d'intérêt pratique car ils demandent un temps de calcul infini. Ainsi, on préfère des schémas relativement simples même s'ils ne garantissent pas la convergence de l'algorithme vers une solution optimale. Un schéma largement utilisé est la réduction par paliers : chaque température est maintenue égale pendant un certain nombre (assez élevé) d'itérations, et décroît ainsi par paliers.

3.3 Recherche tabou

La recherche tabou est une autre méthode de recherche locale avancée qui fait appel à un ensemble de règles et de mécanismes généraux pour guider la recherche de manière intelligente [7]. A l'inverse du recuit simulé, tabou est une méthode plus déterministe.

A chaque itération, tabou examine la totalité ou un sous-ensemble de $N(s)$ et retient la meilleure s' même si s' est plus mauvaise que s. La recherche tabou ne s'arrête donc pas au premier optimum trouvé. Pour empêcher des cycles infinis, tabou fait intervenir une mémoire à court terme, appelée *liste tabou* qui constitue l'élément le plus important de cette méthode.

Une liste tabou permet de mémoriser les k dernières configurations visitées ou plus généralement des caractéristiques pertinentes de ces configurations. Une utilisation simple de cette liste consiste à interdire tout mouvement qui conduit à une de ces configurations pour les k prochaines itérations. La valeur k s'appelle la longueur de la liste tabou et permet d'éviter tous les cycles de longueur inférieure ou égale à k. Cette valeur dépend du problème à résoudre et peut éventuellement évoluer au cours de la recherche.

Un autre composant de base de la recherche tabou est la fonction d'*aspiration* qui a pour objectif de lever le statut tabou de certains mouvements, sans pour autant introduire un risque de cyclage dans le processus de recherche. La fonction d'aspiration peut être définie de plusieurs manières. La fonction la plus simple consiste à révoquer le statut tabou d'un mouvement si ce dernier permet d'atteindre une solution de qualité supérieure à celle de la meilleure solution trouvée jusqu'alors.

Il existe d'autres composants intéressants pour améliorer la puissance de la méthode tabou, en particulier, *l'intensification* et *la diversification*. Toutes les deux se basent sur l'utilisation d'une mémoire à long terme et se différencient selon la façon d'exploiter les informations de cette mémoire.

Enfin il est à noter que tabou réalise une recherche très agressive car elle examine à chaque itération un nombre important de configurations pour effectuer un mouvement. Par conséquent, il est indispensable d'employer des structures de données performantes permettant une évaluation rapide des configurations voisines.

4 Algorithmes RS et tabou pour la résolution de contraintes

Pour présenter les algorithmes du recuit simulé et de la recherche tabou pour la résolution des problèmes de satisfaction maximale de contraintes, nous précisons d'abord les composants communs aux deux algorithmes : le codage des configurations et le voisinage.

4.1 Composants communs

Pour les problèmes de contraintes introduits dans la section 2, nous disposons d'un codage des configurations à la fois simple et naturel qui fait l'objet de la définition suivante.

Définition 2 : Etant donné un réseau de contraintes $< V, D, C >$, une *configuration* s est une assignation définie par $s = \{< X_i, v_i > \mid X_i \in \mathrm{V} \text{ et } v_i \in D_i\}$.

Ainsi, l'espace des solutions S du réseau de contraintes est composé de toutes les assignations possibles. Il est clair que le cardinal de S est égale au produit de la taille de tous les domaines, i.e. $\prod_{i=1}^{n} |D_i|$.

A partir de cette configuration, une fonction de voisinage très simple et générale peut être introduite.

Définition 3 : Soit s une configuration de S, le *voisinage* N: $S \rightarrow 2^S$ est une application telle que pour tout $s \in S$, $s' \in N(s)$ si et seulement si s et s' diffèrent uniquement par la valeur d'une seule variable en conflit [2].

Une configuration voisine de s peut donc être obtenue par le simple changement de la valeur courante d'une variable quelconque en conflit dans s. Un mouvement peut donc être caractérisé par le couple $< variable, valeur >$. Il est clair que la taille du voisinage varie au cours de la recherche.

Définition 4 : La *fonction de coût* est définie par :

$$f(s) = \sum_{i=1}^{|C|} p_i * \chi(C_i),\ C_i \in \mathrm{C} \text{ et } p_i = \text{le poids de } C_i$$

$$\chi(C_i) = \begin{cases} 1 \text{ si } C_i \text{ n'est pas satisfaite} \\ \\ 0 \text{ sinon} \end{cases}$$

Pour les MCSP (les WMCSP respectivement), *f(s)* correspond à la somme (pondérée pour les WMCSP) des contraintes violées par la configuration s. Un problème CSP peut être simplement traité comme un MCSP en recherchant une configuration de coût nul.

4.2 Algorithme du recuit simulé

Nous précisons maintenant les composants particuliers de l'algorithme du recuit simulé.

Evaluation des solutions: A chaque itération, après avoir choisi au hasard un voisin *s' ∈ N(s)*, l'algorithme doit calculer la différence de coût Δ = *f(s')* - *f(s)*. Soit $< X, v >$ le mouvement conduisant de s à s', pour faire ce calcul, il suffit alors de re-examiner localement les contraintes contenant une des variables adjacentes à X. Ce calcul local a une complexitéé en temps de *O(|V|*|D|)*.

2. Une variable est dite en conflit si elle est impliquée dans une contrainte non satisfaite.

Fonction de décroissance de la température: Après avoir testé plusieurs fonctions faisant décroître la température, nous avons choisi la fonction suivante D: $T \rightarrow T$ où $T \subset \mathbf{R}$ est l'ensemble des températures possibles. Plus précisément, soit $t \in T$ la température courante, on calcule la température suivante (à l'itération i) avec $D(t) = t(1 - \frac{A}{i})$ où A est un paramètre de contrôle. Cette fonction diminue progressivement le taux de réduction quand la recherche progresse.

Fonction de croissance de la longueur du palier: Après avoir testé plusieurs fonctions, nous avons choisi la fonction suivante P: $L \rightarrow L$ où $L \subset Z^+$ est l'ensemble des longueurs possibles. Formellement, soit l la longueur courante du palier, on calcule la longueur suivante (à l'itération i) avec la partie entière de la fonction $P(l) = l(1 + \frac{A}{i})$ où A est le même paramètre de contrôle que précédemment. Cette fonction augmente progressivement le nombre d'itérations par palier quand la recherche progresse.

La figure 1 présente l'algorithme du recuit simulé.

Algorithme RS

begin
  Déterminer une configuration aléatoire s_0;
  Choisir une température initiale t_0;
  Choisir une longueur de palier initiale l_0;
  $nb\_iter \leftarrow 0$; $nb\_mouv \leftarrow 0$;
  $s \leftarrow s_0$; $s^* \leftarrow s$; $t \leftarrow t_0$;
  $l \leftarrow l_0$; $compteur \leftarrow l$;
  while *((nb_mouv < max)* **and** *(f(s) ≠ 0))*
  do
    while *(compteur ≠ 0)* **do**
      Déterminer aléatoirement *s' ∈ N(s)* tel que
      s' ≠ s;
      Δ = *f(s')* - *f(s)*;
      if *((Δ ≤ 0)* **or**
      ((Δ > 0) **and** *(Proba.* $e^{(-\Delta)/t}$ *vérifiée)))* **then**
        $s \leftarrow s'$;
        if *(f(s) ≤ f(s*))* **then**
          $s^* \leftarrow s$;
        $nb\_mouv \leftarrow nb\_mouv + 1$;
      $nb\_iter \leftarrow nb\_iter + 1$;
      $compteur \leftarrow compteur - 1$;
    $t \leftarrow D(t)$;
    $l \leftarrow P(l)$;
    $compteur \leftarrow l$;
  output(s^*);
end

Figure 1 : *Algorithme du recuit simulé*

Notons que dans l'algorithme RS, une itération ne conduit pas nécessairement à un mouvement.

4.3 Algorithme de la recherche tabou

Nous précisons maintenant les composants particuliers de l'algorithme tabou.

Evaluation des solutions: A chaque itération, l'algorithme examine la totalité du voisinage pour effectuer un mouvement. Il est donc impératif de pouvoir évaluer rapidement les coûts des voisins d'une configu-

ration. Pour cela, nous utilisons une technique s'inspirant de [3]. Cette technique se fonde sur une matrice δ de $|V|*|D|$ cases où chaque case $\delta[i,j]$ représente la variation de la fonction coût qui devra être appliquée si le mouvement correspondant à la case de la matrice est effectué. Ainsi, le coût de $s' \in N(s)$ est obtenu immédiatement par la simple somme du coût de s et de la valeur de la case correspondante de δ ($O(1)$). Pour obtenir le meilleur voisin, il suffit de parcourir un sous-ensemble de la matrice δ en temps $O(|N(s)|)$. Après chaque mouvement, la matrice est mise à jour en temps $O(|V|*|D|)$ dans le pire des cas.

Liste tabou : Avec la configuration définie précédemment, un mouvement est caractérisé par le couple $< X, v >$, X et v étant respectivement une variable et une valeur possible pour la variable. Quand un mouvement qui affecte v à X en remplacement de v_0 est effectué, le couple $< X, v_0 >$ est mémorisé dans la liste tabou et la variable X se voit interdire de reprendre la valeur v_0 pour les k itérations suivantes.

Afin d'implémenter la liste tabou nous utilisons une matrice $|V|*|D|$, dont chaque case correspond à un mouvement et dont le contenu de la case est affecté avec le nombre d'itérations plus la longueur k de la liste tabou. Avec cette structure de données, une simple comparaison avec l'itération courante suffit donc pour savoir si un mouvement est tabou ou non.

Aspiration : Le critère d'aspiration utilisé est très simple : un mouvement tabou est accepté si celui-ci conduit à une configuration meilleure que toutes les configurations précédemment rencontrées.

La figure 2 présente l'algorithme tabou.

```
Algorithme tabou

begin
    Déterminer une configuration aléatoire s0;
    nb_mouv ← 0;
    s ← s0;
    s* ← s;
    while ( (nb_mouv < max) and (f(s) ≠ 0) )
    do
        Déterminer un meilleur voisin s' ∈ N(s)
        caractérisé par < X, v > parmi ceux non-tabou et
        ceux vérifiant l'aspiration;
        Introduire < X, v0 > dans la liste tabou;
        s ← s';
        Mettre à jour la matrice δ;
        if ( f(s) ≤ f(s*) ) then
            s* ← s;
        nb_mouv ← nb_mouv + 1;
    output(s*);
end
```

Figure 2 : *Algorithme de la recherche tabou*

On peut remarquer que contrairement à l'algorithme du RS, l'algorithme tabou réalise un mouvement à chaque itération.

5 Expérimentation et résultats

Cette section présente des résultats expérimentaux des deux algorithmes. Pour cela, il est nécessaire de présenter les jeux de tests, les critères de comparaison, et la procédure de réglage des paramètres.

5.1 Jeux de tests

Les jeux que nous avons utilisés pour mener nos études correspondent à des réseaux de contraintes binaires générés de manière aléatoire [3]. Chaque instance est tirée aléatoirement d'une classe de réseaux qui est définie par la donnée des 4 valeurs d'un quadruplet de paramètres : $< n, d, p_1, p_2 >$ où n représente le nombre de noeuds du réseaux, d le nombre de valeurs dans chacun des domaines, p_1 la densité, i.e. le pourcentage de contraintes appartenant effectivement au réseau parmi les $n(n-1)/2$ contraintes possibles, enfin p_2 représente la dureté, i.e. le pourcentage de couples interdits sur chacune des contraintes parmi les d^2 possibles. Pour obtenir des instances différentes d'une classe donnée, il suffit de choisir des grains aléatoires différents.

Un réseau de contraintes ainsi généré peut être sous-contraint ou sur-contraint. Une transition de phase concernant la satisfiabilité apparaît quand le réseau est contraint de manière critique [2, 12, 23] (notion de seuil). Les réseaux sous-contraints ont tendance à être facilement satisfiables ($f = 0$) et ne sont donc pas intéressants pour l'optimisation. Les réseaux sur-contraints sont en général non-satisfiables ($f > 0$). Les réseaux au seuil peuvent être satisfiables ou peuvent ne pas l'être. Ces réseaux sont en général difficiles á résoudre du point de vue de la satisfaction.

Ces différentes régions sont caractérisées par un facteur de *seuil* qui est défini par [6] :

$$\kappa = \frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2})$$

$\kappa = 1$ sépare les réseaux sous et sur-contraints. Les réseaux au seuil vérifient $\kappa \approx 1$.

Pour ce travail, seuls des réseaux sur-contraints ($\kappa > 1$) ou au seuil ($\kappa \approx 1$) sont intéressants. Cependant, du point de vue d'optimisation, nous ne savons pas distinguer un réseau facile d'un réseau difficile dans ces deux régions.

5.2 Critères de comparaisons

Nous avons retenu trois critères principaux pour effectuer l'étude comparative, ces critères sont par ordre décroissant d'importance :

- **Qualité de la solution** : la qualité du résultat final découvert par chacun des algorithmes, i.e. le minimum de la fonction de coût en terme de nombre de contraintes violées ;

3. Notre générateur est disponible auprès du premier auteur de ce papier.

- **Nombre de mouvements**: l'effort nécessaire à un algorithme pour atteindre une valeur de la fonction coût ; ce critère est indépendant de la machine utilisée ;
- **Temps d'exécution**: le temps CPU utilisé par un algorithme pour réaliser un nombre de mouvements donné (200 000 dans cette étude) ; ce critère est dépendant de la machine utilisée.

5.3 Réglage des paramètres

Le réglage des paramètres de RS et tabou est extrêmement important et conditionne la qualité des résultats obtenus. Pour obtenir les meilleures valeurs possibles, un protocole de réglage a été suivi pour les paramètres des algorithmes et pour chaque jeu de test. Pour chaque paramètre, on détermine d'abord grossièrement un intervalle de valeurs intéressantes. On recherche ensuite dans l'intervalle la valeur donnant le meilleur résultat avec un nombre limitée de 50 000 mouvements. Cette valeur est alors retenue comme étant la meilleure valeur du paramètre pour le jeu concerné et est utilisée pour les tests finaux.

Le principal paramètre de tabou est la longueur de la liste tabou. La première étape de l'étude paramétrique a permis de sélectionner l'intervalle [5 ; 50]. Les 10 valeurs allant de 5 à 50 avec un pas de 5 dans [5 ; 50] sont ensuite testées et la meilleure sélectionnée.

Le réglage des paramètres de RS s'avère plus délicat car il a plusieurs paramètres interdépendants : la température initiale t_0, la longueur initiale du palier l_0 et enfin le paramètre de contrôle A de la fonction de décroissance. Sur les jeux testés, des tests préliminaires croisés des 3 paramètres ont mis en évidence que $A = 1\,000$ est une valeur stable et satisfaisante quelque soit les valeurs des 2 autres paramètres. Cette valeur est donc retenue pour les tests suivants.

Afin de régler les paramètres t_0 et l_0, nous avons tout d'abord déterminé, au cours d'une série de tests préliminaires, des intervalles pour les deux paramètres donnant des résultats satisfaisants à savoir [1 ; 3] pour t_0 et [1600 ; 2500] pour l_0. Ensuite, nous avons réglé la température initiale, en partant de la valeur médiane de l_0 (2000) et de la valeur médiane de t_0 (2). On fait alors varier t_0 (avec un pas de 0.1) de part et d'autre de la valeur médiane tant que la valeur moyenne de la fonction coût et que le coût minimum rencontré diminuent sur 5 relances. La même procédure est suivie pour régler l_0, cependant la valeur précédemment déterminée de t_0 remplace la valeur médiane de ce paramètre.

5.4 Résultats et analyses

Lors de nos expérimentations, nous avons utilisé des instances individuelles de différentes classes mais aussi plusieurs instances d'une même classe. Les différentes classes utilisées sont séparées en 3 groupes caractérisés par le nombre de variables des réseaux. Les 3 tailles que nous avons testées sont 100, 200 et 300 avec 10 à 30 valeurs par variable.

Nous présentons dans cette partie les résultats comparatifs de RS et tabou. Pour obtenir ces résultats, chacun des deux algorithmes est exécuté 10 fois sur chaque instance avec un nombre maximum de 200 000 mouvements par exécution.

La table 1 présente les résultats obtenus sur 13 instances tirées au hasard dans 13 classes différentes (les résultats sur différentes instances d'une même classe sont présentés plus loin dans la table 3). Les paramètres de ces classes ont été choisis de manière à éviter des instances facilement satisfiables ($f = 0$).

Les colonnes 1-2 indiquent la classe dans laquelle l'instance a été tirée et la valeur de κ. Les colonnes 3 - 6 et 7 - 11 présentent respectivement les résultats de tabou et RS. Pour chaque algorithme, on indique la valeur utilisée pour chaque paramètre : la longueur de la liste tabou (long. colonne 3) pour l'algorithme tabou et la température initiale t_0, la longueur initiale du palier l_0 (colonnes 7 et 8) pour l'algorithme RS. On donne ensuite la valeur minimale, moyenne et maximale de la fonction coût, ainsi que le nombre de fois (le chiffre entre parenthèses) où la valeur a été atteinte. Enfin, les 2 dernières colonnes présentent l'écart du coût minimal et du coût moyen entre tabou et RS.

Plusieurs constations peuvent être tirées des résultats de la table 1. Tout d'abord, le *coût minimal* trouvé par tabou est meilleur que celui de RS pour 10 instances (sur 15) avec un écart de 1 à 4 pour des coûts de 4 à 25. Dans les 5 cas où RS parvient à trouver des solution de même qualité, le taux de réussite pour atteindre le coût minimal avec tabou est souvent supérieur ou égal à celui de RS. En terme de *coût moyen*, tabou donne de meilleurs résultats sur la totalité des jeux, avec un écart de 0,2 à 4,7.

On remarque également que le paramètre de tabou, à savoir, la longueur de la liste tabou admet d'importantes variations d'une instance à l'autre (une plage allant de 15 à 40 est utilisé par les jeux). La longueur *moyenne* de la liste tabou semble augmenter avec la taille du réseau. En effet, la longueur moyenne est de 24 pour les instances de 100 variables, 28,75 pour les instances de 200 variables et 33,3 pour celles de 300 variables. Cette constatation s'explique probablement ainsi. La liste tabou sert à éviter les cycles dont la taille est définie par la longueur de la liste tabou. Or si la taille du jeu augmente, il est probable que les cycles voient leur taille augmenter en même temps. Pour éviter ces cycles, il faut augmenter la longueur de la liste tabou.

Les valeurs des paramètres de RS semblent plus stables sur l'ensemble des jeux testés, ceci est particulièrement vrai pour la température initiale. Une explication possible est la suivante. Les paramètres de RS s'appliquent à la *variation* de coût de la fonction objective. Or cette variation diminue rapidement au cours de la recherche et devient très peu importante (de l'ordre de quelque unités de différence) et cela quelques soit la taille du jeu, c'est pourquoi les paramètres de RS sont plus "universels".

Problème	κ	Tabou				RS					Tabou-RS	
		long.	fonction coût			t_0	l_0	fonction coût				
			min.	moy.	max.			min.	moy.	max.	min.	moy.
100.10.15.25	0.93	30	0(4)	0.6	1(6)	2	2000	0(3)	0.8	2(1)	0	-0.2
100.10.20.25	1.24	25	19(10)	19	19(10)	2.5	2000	19(4)	20	22(1)	0	-1
100.15.10.45	1.64	25	11(4)	11.7	13(1)	2	2000	12(1)	13.3	14(4)	-1	-1.6
100.15.20.30	1.3	20	25(1)	26.4	28(2)	2.5	2000	26(3)	27.7	29(4)	-1	-1.3
100.15.30.20	1.22	20	18(2)	19	20(2)	2	2000	22(2)	23,7	25(3)	-4	-4.7
200.18.20.13	0.96	25	4(3)	5.6	8(1)	2	2200	5(4)	5.9	8(1)	-1	0.3
200.20.12.22	0.99	35	8(1)	9.6	11(2)	2	2000	11(2)	12.3	14(1)	-3	-2.7
200.20.18.14	0.9	15	0(1)	1.5	3(1)	2.5	1800	2(5)	2,8	4(3)	-2	-1.3
200.20.26.11	1	25	9(1)	11.6	14(1)	2	2300	11(3)	13	15(1)	-2	-1.4
200.20.20.15	1.08	30	21(1)	23.1	25(1)	2.3	2000	23(2)	26.7	29(1)	-2	-3.6
300.20.10.18	0.99	40	14(1)	17	20(1)	2	2100	16(1)	18,4	21(1)	-2	-1.4
300.25.18.10	0.92	25	2(2)	2.8	3(8)	2	2000	2(4)	3.3	5(3)	0	-0.5
300.28.12.16	0.94	35	9(1)	11	12(3)	2	2000	11(2)	12	14(5)	-2	-1
300.30.16.12	0.9	25	4(1)	5.7	7(2)	2.3	2000	4(1)	6,6	8(3)	0	-1.1
300.30.20.10	0.93	35	11(3)	12	14(1)	2	2000	11(2)	13.3	18(1)	0	-1.3

Table 1 : *Résultats comparatifs de RS et Tabou pour un nombre maximum de 200 000 mouvements*

La figure 3 donne une vue plus fine sur la différence de performance entre tabou et RS pour une instance de la classe *100.15.30.20*. En abscisse, on donne le nombre de mouvements de 0 à 200 000 avec un pas de 10 000 et en ordonné, le coût minimal trouvé par chaque algorithme à un nombre de mouvements donné.

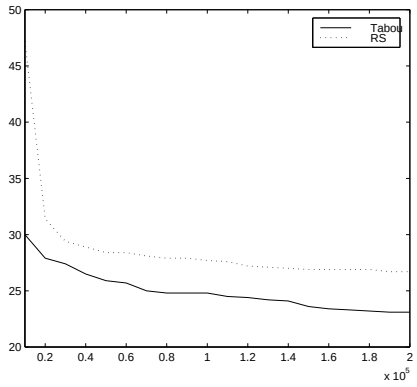

Figure 3 : *Comparaison du minimum trouvé par RS et tabou*

On observe que pour un même nombre de mouvements, tabou trouve toujours des solutions nettement meilleures que RS. L'écart en terme de qualité de solution semble s'élargir avec l'augmentation du nombre de mouvements effectués. On observe également que RS nécessite un nombre de mouvements plus important que tabou pour atteindre le même objectif (voir figure 4). Notons que ces remarques sont valides sur l'ensemble des jeux testés.

La table 2 présente les temps CPU moyens consommés par RS et tabou pour réaliser 200 000 mouvements pour atteindre la meilleure valeur de la fonction de coût. Les temps de la table 2 sont exprimés en seconde et sont obtenus sur une station de travail SPARC 5 (32Mo de RAM et 75MHz) [4]. Ces informations sont données à titre indicatif car les temps changent selon la machine utilisée. En plus, l'utilisation de structures de données plus performantes devrait réduire ces temps.

Problème	Temps de calcul moyen	
	Tabou	RS
100.10.15.25	208	263
100.10.20.25	361	691
100.15.10.45	585	1546
100.15.20.30	844	2440
100.15.30.20	1020	3952
200.18.20.13	1668	2039
200.20.12.22	1717	3256
200.20.18.14	1697	2182
200.20.20.15	1904	6538
200.20.26.11	2084	3163
300.20.10.18	3013	6571
300.28.12.16	7339	15981
300.30.16.12	9074	13224

Table 2 : *Temps CPU moyens en seconde de RS et Tabou pour effectuer 200 000 mouvements*

La table 2 montre la plus grande vitesse de tabou face à RS pour réaliser 200 000 mouvements. Or intuitivement une itération effectué avec tabou est plus coûteuse qu'une itération de RS car tabou examine la totalité du voisinage à chaque itération. Deux explications permettent de justifier malgré cela cette différence.

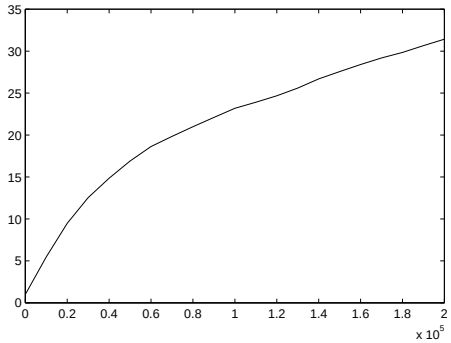

Figure 4 : *Evolution du ratio itérations/mouvements de RS durant la recherche*

4. Les deux algorithmes sont programmés en C++.

Problème	Tabou				RS					Tabou-RS	
	long.	fonction coût			t_0	l_0	fonction coût				
		min.	moy.	max.			min.	moy.	max.	min.	moy.
100.15.10.45.0	25	11(4)	11,7	13(1)	2	2000	12(1)	13,3	14(4)	-1	-1,6
100.15.10.45.1	25	10(2)	11.3	13(1)	2	2000	12(1)	13.3	15(1)	-2	-2
100.15.10.45.2	25	10(2)	11.2	12(4)	2	2000	11(1)	12.9	15(1)	-1	-1.7
100.15.10.45.3	25	8(1)	9.9	11(2)	2	2000	10(4)	11.2	13(1)	-2	-1.3
100.15.10.45.4	25	8(1)	10	11(3)	2	2000	8(1)	10	14(1)	0	0
100.15.10.45.5	25	9(4)	9.8	11(2)	2	2000	9(3)	10.4	13(1)	0	-1.6

Table 3 : *Résultats comparatifs de RS et tabou sur différentes instances d'une même classe*

D'abord, une itération de tabou conduit toujours à un mouvement alors que ce n'est pas le cas pour RS. En particulier, avec l'algorithme du RS, le nombre d'itérations nécessaires pour un mouvement augmente considérablement au cours de la recherche. La figure 4 montre l'évolution du ratio itérations/mouvements de RS durant le déroulement de l'algorithme. Le jeu qui a servi de base à cette courbe est une instance de la classe *200.20.20.15*, les points de la courbe sont une moyenne sur 10 exécutions.

On observe que le ratio *itérations/mouvements* augmente rapidement selon une fonction logarithmique. Par exemple, avant 10 000 mouvements, moins de 5 itérations sont nécessaires pour réaliser un mouvement alors que ce chiffre atteint 20 à 70 000 et 35 à 190 000. Par conséquent, le temps de RS pour un mouvement augmente au cours de la recherche.

Un autre point important contribuant à la rapidité de tabou est l'utilisation de la matrice δ. En effet, grâce à cette structure de données, tabou est capable de parcourir très rapidement l'ensemble des voisins à chaque itération (mouvement). Ainsi, le temps de RS pour un mouvement augmente durant la recherche, le temps de tabou reste quasi-constant.

Pour terminer cette section, nous présentons dans la table 3 les résultats sur 6 instances d'une même classe (la classe *100.15.10.45*, $\kappa = 1.64$). Le dernier chiffre de chaque instance indique le grain aléatoire initial utilisé pour générer l'instance. On constate encore une fois que tabou trouve de meilleurs résultats pour le même nombre de mouvements. D'après d'autres résultats qui ne sont pas montrés ici, les autres remarques sur les instances individuelles de différentes classes restent également valides ici, à savoir que tabou nécessite moins de mouvements pour obtenir des solutions de qualité égale et demande moins de temps pour effectuer le même nombre de mouvements.

5.5 Discussion

Influence des paramètres

Il est connu que les paramètres jouent un rôle primordial pour la performance des algorithmes de recherche locale. Ce point est confirmé lors de nos études. Les figures 5, 6 et 7 montrent sur un exemple (une instance de la classe *100.15.20.30*) comment les variations des paramètres des deux algorithmes se répercutent sur la qualité de solution. Les courbes sont obtenues avec 5 exécutions de chaque algorithme.

La figure 5 présente les variations des résultats (minimum, moyen et maximum) de la fonction de coût en fonction de la longueur de la liste tabou. On remarque qu'une longueur trop petite (<15) ou trop importante (>45) donne de mauvais résultats. Au contraire, les valeurs de l'intervalle entre 15 et 45 sont plus intéressantes. En particulier, la valeur 20 donne le meilleur coût de 25 de la fonction de coût.

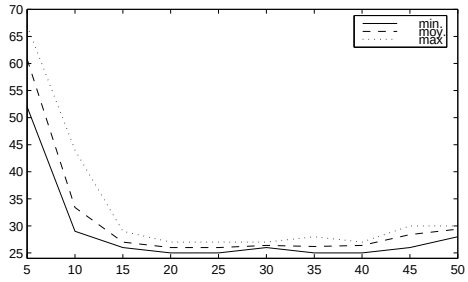

Figure 5 : *Influence de la longueur de la liste tabou sur le coût trouvé*

De manière similaire, les figures 6 et 7 présentent les mêmes résultats pour les variations de la température initiale et de la longueur initiale du palier de RS. Comme pour tabou, on observe toujours un intervalle de valeurs intéressantes pour ces paramètres. En revanche, il existe une plage de valeurs donnant le même minimum (26) de la fonction coût. Ceci s'explique probablement par le fait que l'aspect stochastique de RS rend ces paramètres moins sensibles à l'ensemble des valeurs d'un certain intervalle. Néanmoins, la détérminaison de cet intervalle n'est pas une tâche facile.

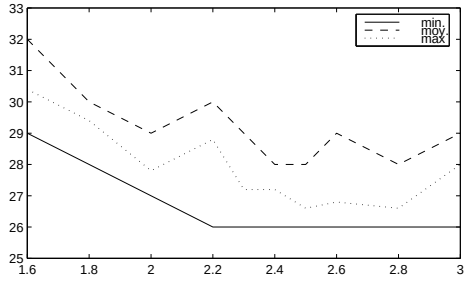

Figure 6 : *Influence de la température initiale de RS sur le coût trouvé*

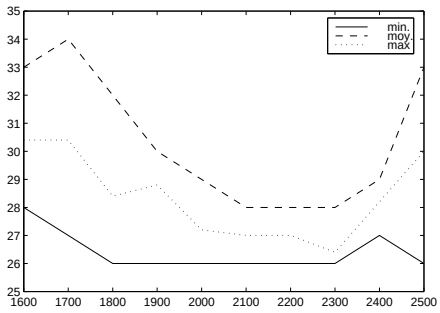

Figure 7 : *Influence de la taille initiale du palier de RS sur le coût trouvé*

Influence du voisinage

Le voisinage est un autre élément important pour la performance d'un algorithme de recherche locale. Dans cette étude, le voisinage utilisé (noté N_2) est défini sur un sous-ensemble particulier des variables du réseau de contraintes, à savoir, les variables en conflit. Un autre voisinage possible (noté N_1) consiste à inclure toutes les variables du réseau. Des expérimentations de ces deux voisinages ont conduit à des solutions très différentes pour les algorithmes de RS et tabou et ont confirmé l'intérêt du voisinage N_2 (voir les figures 8 et 9 pour un exemple).

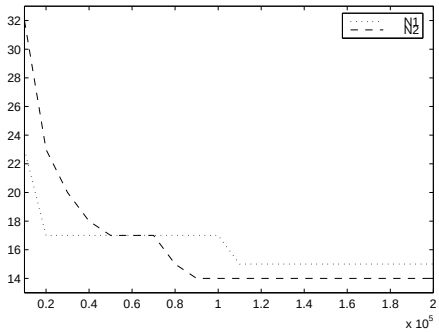

Figure 8 : *Comparaison de deux voisinages pour RS*

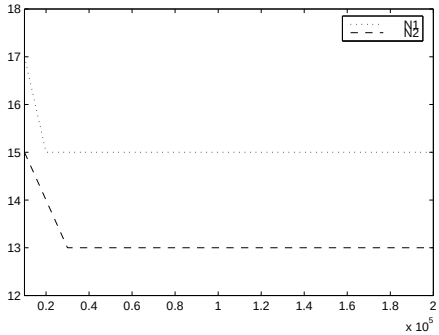

Figure 9 : *Comparaison de deux voisinages pour tabou*

En effet, N_2 possède certains avantages par rapport à N_1. D'abord, N_2 permet aux algorithmes de se concentrer sur les mouvements intéressants et d'éviter un grand nombre de mouvements qui ne conduisent pas à une amélioration de la fonction de coût. Deuxièmement, la taille de N_2 est plus petite que celle de N_1, ce qui est intéressant pour des méthodes comme tabou qui exploitent le voisinage en profondeur. Notons enfin que d'autres voisinages sont possibles. Par exemple, le mélange de N_2 et N_1 est une piste intéressante à exploiter.

Evaluation des configurations

Dans les deux algorithmes, l'évaluation de configurations voisines à chaque itération reste l'étape la plus coûteuse en temps. Cette étape est encore plus cruciale pour tabou car elle doit examiner un grand nombre de voisins chaque fois. Par conséquent, des structures de données appropriées sont indispensables pour avoir une bonne performance. La matrice δ utilisée par tabou en est un bon exemple. Sans cette structure, tabou serait beaucoup plus long à l'exécution et compromettrait sa capacité de trouver de très bonnes solutions. De même, le calcul local utilisé par l'algorithme de RS est aussi indispensable pour sa performance. Notons également que la matrice δ pourrait servir à RS.

6 Conclusion

Dans ce travail, nous avons effectué une étude expérimentale de recherche locale pour la résolution de contraintes. En particulier, nous avons présenté deux algorithmes fondés sur le recuit simulé et la recherche tabou pour les problèmes de satisfaction maximale de contraintes. Une configuration et un voisinage simples et efficaces ont été employés. Des expérimentations comparatives des deux algorithmes ont été réalisées sur des jeux de test aléatoires de taille importante tirées dans des classes variées. Les résultats empiriques ont montré les points suivants :

- pour le même nombre de mouvements, tabou produit de meilleures solutions en terme du coût minimal de la fonction objective ;
- pour obtenir des solutions de qualité égale, tabou nécessite moins de mouvements ;
- tabou est plus rapide en temps de calcul pour effectuer le même nombre de mouvements.

Les deux premiers points sont cohérents avec les études comparatives sur des classes spécifiques de problèmes de contraintes telles que la coloration [1, 11, 13] et le problème de Max-SAT [8]. Le dernier point doit être nuancé car le temps du RS pour effectuer un mouvement peut être amélioré avec des structures de données plus performantes.

Ce travail a également montré qu'un bon voisinage et des structures de données performantes sont indispensables pour obtenir une bonne efficacité des algorithmes. Enfin, ce travail confirme que le réglage des paramètres est crucial pour la qualité des solutions et qu'un réglage optimal reste une tâche difficile, en particulier, pour RS.

Il reste à savoir si ces résultats peuvent être généralisés à l'ensemble des MCSP, par exemple, aux instances dont $\kappa >> 1$ (largement sur-contraintes) et à d'autres classes de problèmes de contraintes.

Pour conclure, nous pensons que la recherche locale constitue une approche très puissante pour la résolution pratique des problèmes et des applications de contraintes et mérite d'être étudiée d'avantage.

Remerciement : Nous remercions J.C. Regin et C. Bessière pour leur assistance sur le générateur aléatoire. Nous remercions également les rapporteurs du papier pour leurs remarques. Ce travail a bénéficié d'une subvention de l'ANVAR sur le thème "Nouvelles heuristiques pour l'optimisation combinatoire".

References

[1] M. Chams, A. Hertz et D. de Werra, "Some experiments with simulated annealing for coloring graphs, European Journal of Operational Research 32, 260-266, 1987.

[2] P. Cheeseman, B. Kanefsky et W.M. Taylor, "Where the really hard problems are", Proc. of IJCAI'90, pp163-169, 1991.

[3] C. Fleurent et J.A. Ferland, "Genetic and hybrid algorithms for graph coloring", G. Laporte, I. H. Osman, et P. L. Hammer (Eds.), Special Issue of Annals of Operations Research, "Metaheuristics in Combinatorial Optimization".

[4] E.C. Freuder et R.J. Wallace, "Partial constraint satisfaction", Artificial Intelligence, Vol.58(1-3) pp21-70, 1992.

[5] P. Galinier et J.K. Hao, "Tabu search for maximal constraint satisfaction problems", Proc. of CP-97, LNCS 1330, pp196-208, Schloss Hagenberg, Austria, Oct-Nov, 1997.

[6] I.P. Gent, E. MacIntyre, P.Prosser, et T. Walsh, "Scaling effects in the CSP phase transition", Proc. of CP95, pp70-87, 1995.

[7] F. Glover et M. Laguna, "Tabu Search", dans C. R. Reeves (Ed.), Modern heuristics for combinatorial problems, Blackwell Scientific Publishing, Oxford, GB, 1993.

[8] P. Hensen et B. Jaumard, "Algorithms for the maximum satisfiability problem", Computing Vol.44, pp279-303, 1990.

[9] J.K. Hao et R. Dorne, "Empirical studies of heuristic local search for constraint solving", Proc. of CP-96, LNCS 1118, pp194-208, Cambridge, MA, USA, 1996.

[10] R.M. Haralick et G.L. Elliot, "Increasing tree search efficiency for constraint satisfaction problems", Artificial Intelligence, Vol. 14, pp263-313, 1980.

[11] A. Hertz et D. de Werra, "Using Tabu search techniques for graph coloring", Computing Vol.39, 345-351, 1987.

[12] B.A. Huberman, T. Hogg et C.P. Williams, Artificial Intelligence, Special Issue on Phase Transition and Complexity. Vol. 82(1-2), 1996.

[13] D.S. Johnson, C.R. Aragon, L.A. McGeoch, et C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning. Operations Research Vol. 39(3), pp378-406, 1991.

[14] S. Kirkpatrick, C.D. Gelatt Jr. et M.P. Vecchi, "Optimization by simulated annealing", Science No.220, pp671-680, 1983.

[15] G. Laporte et I.H. Osman, "Metaheuristics in combinatorial optimization", Annals of Operations Research 63, J.C. Baltzer Science Publishers, Basel, Switzerland, 1996.

[16] J. Larrosa et P. Meseguer, "Optimization-based heuristics for maximal constraint satisfaction", Proc. of CP-95, pp190-194, Cassis, France, 1995.

[17] A.K. Mackworth, "Constraint satisfaction", in S.C. Shapiro (Ed.) Encyclopedia on Artificial Intelligence, John Wiley & Sons, NY, 1987.

[18] S. Minton, M.D. Johnston et P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", Artificial Intelligence, Vol.58(1-3), pp161-206, 1992.

[19] P. Morris, "The Breakout method for escaping from local minima", Proc. of AAAI-93, pp40-45, 1993.

[20] N.B. Nadel, "Constraint satisfaction algorithms", Computional Intelligence, Vol 5, pp188-224, 1989.

[21] C.H. Papadimitriou et K. Steiglitz, "Combinatorial optimization - algorithms and complexity", Prentice Hall, 1982.

[22] B. Selman et H.Kautz, "Domain-independent extensions to GSAT: solving large structured satisfiability problems", Proc. of IJCAI-93, Chambery, France, 1993.

[23] B.M. Smith, "Phase transition and the mushy region in constraint satisfaction problems", Proc. of ECAI-94, pp100-104, 1994.

[24] E. Tsang, "Foundations of constraint satisfaction", Academic Press, 1993.

[25] R.J. Wallace, "Enhancements of branch and bound methods for the maximal constraint satisfaction problem", Proc. of AAAI-96, pp188-196, Portland, Oregon, USA, 1996.

[26] N. Yugami, Y. Ohta et H. Hara, "Improving repair-based constraint satisfaction methods by value propagation", Proc. of AAAI-94, pp344-349, Seattle, WA, 1994.