LocalSolver 1.x *
Un solveur boîte noire à base de recherche locale pour la programmation 0-1

Thierry Benoist[1], **Bertrand Estellon**[2], **Frédéric Gardi**[1], **Romain Megel**[1], **Karim Nouioua**[2]

[1] Bouygues e-lab, Paris

[2] Laboratoire d'Informatique Fondamentale - CNRS UMR 6166,
Faculté des Sciences de Luminy - Université Aix-Marseille II, Marseille

{tbenoist,fgardi,rmegel}@bouygues.com {estellon,nouioua}@lif.univ-mrs.fr

Résumé

Cet article présente LocalSolver 1.x, un solveur boite noire à base de recherche locale pour la programmation 0-1 généralisée. Ce logiciel permet aux praticiens de la recherche opérationnelle de se concentrer sur la modélisation de leur problème dans un formalisme simple, pour ensuite en confier la résolution à un solveur basé sur des techniques efficaces et robustes de recherche locale ("*model & run*"). Après une présentation du formalisme de modélisation et du fonctionnement interne de LocalSolver, nous en démontrons l'efficacité par une vaste étude expérimentale.

Abstract

This paper introduces LocalSolver 1.x, a black-box local-search solver for general 0-1 programming. This software allows OR practitioners to focus on the modeling of the problem using a simple formalism, and then to leave its actual resolution to a solver based on efficient and reliable local-search techniques ("*model & run*").Having outlined the modeling formalism and the main technical features behind LocalSolver, its effectiveness is demonstrated through an extensive computational study.

*Le travail de B. Estellon et K. Nouioua est en partie financé par le projet ANR OPTICOMB (ANR BLAN06-1-138894). T. Benoist, F. Gardi, R. Megel remercient Etienne Gaudin, directeur du e-lab de Bouygues, pour son soutien et ses encouragements.

1 Introduction

En optimisation combinatoire, les techniques de recherche arborescente consistent à explorer l'espace des solutions par une instantiation itérative des variables composant le vecteur solution. Leur efficacité en pratique repose sur leur capacité à élaguer l'arbre de recherche, qui est de taille exponentielle dans le pire des cas. Fondée sur ces techniques, la Programmation Linéaire en Nombres Entiers (PLNE) est sans nul doute un des outils les plus puissants de la Recherche Opérationnelle. Bien que limitée face à des problèmes combinatoires de grande taille, son succès auprès des praticiens est notamment dû à la simplicité d'utilisation des solveurs PLNE : l'ingénieur modélise son problème comme un programme linéaire en nombres entiers et le solveur le résout par *branch & bound* (& *cut*). Ainsi, une tendance récente en Programmation par Contraintes (PPC) est de promouvoir la conception de solveurs PPC autonomes [19]. En effet, cette approche "*model & run*", lorsqu'elle s'avère effective, réduit considérablement les efforts de développement et de maintenance des logiciels d'optimisation.

D'un autre côté, la Recherche Locale (LS, de l'anglais *Local Search*) consiste à appliquer de façon itérative des modifications (appelés mouvements) à une solution de façon à améliorer celle-ci. Bien qu'incomplète, ces techniques sont appréciés des chercheurs opérationnels parce que permettant d'obtenir des solutions de qualité en des temps très courts (de l'ordre de la minute). Cependant, concevoir et implémenter

des algorithmes de recherche locale n'est pas chose facile. La couche algorithmique dédiée à l'évaluation des mouvements est particulièrement difficile à mettre en oeuvre, parce qu'elle requiert à la fois une expertise en algorithmique et une dextérité en programmation informatique. Pour une synthèse sur la recherche locale et ses applications, le lecteur est invité à consulter le livre édité par Aarts et Lenstra [1].

Cet article introduit LocalSolver 1.x, un solveur "boîte noire" pour la programmation 0-1 généralisée (non linéaire). Ce logiciel permet praticien de se concentrer sur la modélisation de son problème, et de laisser sa résolution à un solveur basé sur des algorithmes de recherche locale efficaces et fiables. Débuté en 2007, ce projet vise à offrir une approche de type "*model & run*" pour la résolution des problèmes d'optimisation combinatoire hors de portée des solveurs autonomes de PLNE ou PPC. La version actuelle (LocalSolver 1.1) permet d'attaquer une classe restreinte (mais importante) de problèmes d'optimisation combinatoire : *assignment, partitioning, packing, covering*. Distribuée gratuitement sous licence BSD [1], les binaires du logiciel sotn disponibles pour les architectures x86 et pour les trois systèmes d'exploitations Linux 2.6, Mac OS X 10.5 (Leopard), Windows XP. Le logiciel peut être utilisé pour l'enseignement, la recherche et même à des fins commerciales sans autorisation des auteurs.

Le papier est organisé comme suit. Après un état de l'art sur le sujet, le formalisme LSP associé à LocalSolver 1.x est présenté. Ensuite, le logiciel et les idées maîtresses sur lesquelles il repose sont détaillés. Afin de démontrer l'efficacité de notre solveur, nous présentons dans les grandes lignes les résultats d'une vaste étude expérimentale réalisée sur une dizaine de problèmes académiques et industriels.

2 Travaux connexes et contributions

Une heuristique de recherche locale se conçoit en trois couches [12] : stratégie de recherche et (méta)heuristiques, mouvements, algorithmique d'évaluation. Nos expériences passées dans la conception et l'implémentation d'algorithmes de recherche locale [3, 10, 11, 12] nous ont convaincus que négliger l'une de ces trois couches pouvait entraîner une baisse significative des performances de l'algorithme. Ainsi la conception et l'implémentation d'une recherche locale est une tâche longue et complexe pour les praticiens de la recherche opérationnelle.

La plupart des outils ou des composants réutilisables proposés pour faciliter l'implémentation de recherche locale prennent la forme d'un *framework* pour traiter la couche haute, c'est-à-dire les métaheuristiques (voir par exemple [5, 7]). Ainsi, les mouvements et les algorithmes d'évaluation incrémentale associés restent à la charge de l'utilisateur tandis que le rôle du *framework* est d'appliquer la métaheuristique sélectionnée. Malheureusement, concevoir les mouvements et implémenter les algorithmes d'évaluation représente la plus grande part du travail (et du code source résultant). En effet, nous avons observé que le travail relatif à ces deux couches correspondait respectivement à 30 % et 60 % des temps de développement. Par conséquent, ces *frameworks* ne répondent pas aux besoins majeurs des praticiens.

Deux logiciels visent à répondre à ces besoins : Comet Constraint-Based Local Search (CBLS) [24] (et son ancêtre Localizer [15]) et iOpt [28]. En effet, ces logiciels permettent une évaluation automatique des mouvements, l'implémentation de ces mouvements et de l'heuristique restant à la charge de l'utilisateur. *À notre connaissance, aucun solveur "boîte noire" efficace basé sur la recherche locale n'est disponible à ce jour pour traiter des problèmes réels d'optimisation combinatoire de grandes tailles, comme on en connaît en PLNE ou PPC.* Van Hentenryck et Michel [25] ont publié récemment un article décrivant un synthétiseur d'heuristiques de recherche locale à partir de modèles de haut niveau, mais cette fonctionnalité n'est pas encore disponible dans Comet. Un algorithme tabou générique à base de *swaps* [9, pp. 330–331] est disponible dans Comet CBLS 2.1, et peut être utilisé comme une boîte noire pour aborder des modèles entiers.

Notons également que les meilleurs prouveurs SAT reposent sur des techniques de recherche locale (voir par exemple *Walksat* [23] et WSAT(OIP) [29]), mais ces solveurs ne traite pas la programmation 0-1 généralisée et sont donc rarement utilisés par les praticiens de la recherche opérationnelle.

Notre approche de la recherche locale autonome est guidé par le principe fondamental suivant : *le solveur doit faire ce que le praticien ferait.* Ceci constitue une différence majeure avec les frameworks et solveurs cités plus haut : LocalSolver effectue des mouvements structurés tendant à maintenir la faisabilité des solutions à chaque itération, et dont l'évaluation est accélérée en exploitant les invariants induits par la structure du problème.

Ainsi, *les spécificités majeures de LocalSolver 1.x sont : un formalisme mathématique simple pour modéliser le problème de façon appropriée pour une résolution par recherche locale, et un solveur boîte-noire par recherche locale focalisé sur la faisabilité et l'efficacité des mouvements.*

1. `http://www.localsolver.com`

3 Le formalisme LSP

Le formalisme de modélisation de LocalSolver (appelé LSP pour "Local Search Programming") est proche de formalismes classiques de la programmation mathématique comme la programmation entière 0-1 ou la programmation pseudo booléenne mais est enrichi des opérateurs mathématiques usuels, ce qui le rend facile à prendre en main pour des praticiens de la recherche opérationnelle. Dans le format LSP (*Local Search Programming*), un programme consiste en : des variables de décision, des variables intermédiaires, des contraintes et des objectifs. Bien entendu, le langage de modélisation possède son équivalent en terme d'interface de programmation dans la librairie C++ LocalSolver. Comme exemple, nous donnons le modèle d'un problème jouet de type *bin packing*. Nous avons 3 paquets x, y, z de hauteur 2, 3, 4 respectivement à arranger en 2 piles A et B, sachant que cette dernière contient déjà un paquet de hauteur 5. L'objectif est de minimiser la hauteur de la plus grande pile.

```
xA <- bool(); yA <- bool(); zA <- bool();
xB <- bool(); yB <- bool(); zB <- bool();
constraint booleansum(xA, xB) = 1;
constraint booleansum(yA, yB) = 1;
constraint booleansum(zA, zB) = 1;
hauteurA <- sum(2xA, 3yA, 4zA);
hauteurB <- sum(2xB, 3yB, 4zB, 5);
objectif <- max(hauteurA, hauteurB);
minimize objectif;
```

si le paquet x est placé dans la pile A. Dans cette version 1.0, seuls les variables de décision booléennes sont autorisés, rendant le formalisme proche de la programmation entière 0-1. Ensuite, l'opérateur `<-` est utilisé pour définir des variables intermédiaires (par exemple, la hauteur de chaque pile), qui peuvent être booléennes ou entières. Le mot-clé `constraint` préfixe la déclaration des contraintes ; ici trois contraintes assurent que chaque paquet est affecté à une et une seule pile. De la même façon, le mot-clé `minimize` préfixe la déclaration de l'objectif du programme.

Formellement, la syntaxe BNF d'un programme LSP est :

$$
\begin{array}{rcl}
< lsp > & ::= & (line) \\
< line > & ::= & [< modifier >][< naming >] \\
 & & < expression > \texttt{;} \\
< modifier > & ::= & \texttt{minimize}|\texttt{maximize}| \\
 & & \texttt{constraint} \\
< naming > & ::= & < identifier > \texttt{<-}
\end{array}
$$

Les différents types d'expression sont détaillés dans la section suivante. Notons que l'ordre des lignes dans le programme est libre, excepté lorsque l'on définit plusieurs objectifs.

3.1 Variables décisionnelles et intermédiaires

Toutes les variables de décision doivent être déclarées dans le programme. Cela est fait à l'aide de l'opérateur `bool()`, déclarant une variable booléenne. Les variables booléennes sont traités comme des entiers, avec la convention faux=0 et vrai=1. Nous insistons sur le fait que seuls les variables booléennes sont autorisées comme variables décisionnelles dans cette version de LocalSolver.

Des expressions peuvent être construites à partir de ces variables en utilisant d'opérateurs logiques, arithmétiques et relationnels :

$$
\begin{array}{rcl}
< expression > & ::= & < identifier > \mid < scalar > \mid \\
 & & < scalar >< expression > \mid \\
 & & < operator > \texttt{(}[< arglist >]\texttt{)} \mid \\
 & & < expression >< comparator > \\
 & & < expression > \\
< arglist > & ::= & < expression > [\texttt{,} < arglist >] \\
< operator > & ::= & \texttt{bool} \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{not} \mid \texttt{if} \mid \\
 & & \texttt{sum} \mid \texttt{booleansum} \mid \texttt{min} \mid \texttt{max} \mid \\
 & & \texttt{product} \mid \texttt{divide} \mid \texttt{modulo} \mid \\
 & & \texttt{distance} \mid \texttt{abs} \mid \texttt{square} \\
< comparator > & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{!=}
\end{array}
$$

où $< scalar >$ est un entier et $< identifier >$ un nom de variable.

En résumé, LocalSolver utilise une syntaxe fonctionnelle (seuls les opérateurs relationnels ne sont pas préfixés), sans limitation sur l'imbrication des expressions. Des variables intermédiaires peuvent être introduites via l'opérateur `<-`, soit pour accroître la lisibilité du modèle ou pour réutiliser des expressions dans différentes lignes. Certains opérateurs ont des contraintes sur le nombre et le type de leurs opérandes. Par exemple, l'opérateur `not` ne prend qu'un seul argument de type booléen. L'opérateur `if` prend exactement trois arguments, le premier étant nécessairement booléen : `if(condition, value_if_true, value_if_false)`. Les expressions booléennes étant considérées comme des variables 0/1, elles peuvent être utilisées comme opérandes d'opérateurs entiers.

L'introduction des opérateurs logiques, arithmétiques et relationnels a deux avantages importants dans un contexte de recherche locale : l'expressivité et efficacité. Avec ce type d'opérateurs mathématiques de bas-niveau, modéliser est plus facile qu'avec la syntaxe basique de la PLNE, tout en restant rapidement assimilable par les moins expérimentés (en particulier, par les ingénieurs qui ne sont pas à l'aise en programmation informatique). D'un autre côté, les invariants induits par ces opérateurs peuvent être exploités par les algorithmes internes du solveur pour accélérer la recherche locale.

3.2 Contraintes et objectifs

Toute expression booléenne peut être contrainte en préfixant la ligne par `constraint`. Une instantiation des variables est valide si et seulement si toutes les expressions contraintes ont comme valeur 1, c'est-à-dire sont satisfaites. Lors de la modélisation d'un problème l'utilisateur doit garder à l'esprit que la recherche locale n'est pas adaptée pour traiter des problèmes fortement contraints. Si des contraintes métier sont susceptibles de ne pas être satisfaites, alors il est recommandé de les transférer dans la fonction objectif (comme contraintes molles) plutôt que de laisser comme contraintes dures. De plus, LocalSolver offre une fonctionnalité facilitant ce type de modélisation : les objectifs à optimiser dans un l'ordre lexicographique.

Au moins un objectif doit être défini, en utilisant le modificateur `minimize` ou `maximize`. Toute expression peut être utilisée comme objectif. Si plusieurs objectifs sont définis, ils sont interprétés comme une fonction objectif lexicographique. L'ordre lexicographique est induit par l'ordre dans lequel sont déclarés les objectifs. Par exemple, dans le problème d'ordonnancement de véhicules [11], lorsque l'objectif est de minimiser les violations sur les contraintes de ratio et en second lieu le nombre de changements de couleurs dans la séquence, la fonction objectif peut être directement spécifiée comme suit : `minimize ratio_violations; minimize color_changes;`. Ceci permet d'éviter le travers classique de la PLNE où un grand coefficient est utilisé pour simuler l'ordre lexicographique : `minimize 1000 ratio_violations + color_changes;`. Notons que le nombre d'objectifs n'est pas limité et que ceux-ci peuvent avoir différentes directions (minimisation ou maximisation).

4 Recherche locale autonome

La ligne de commande pour résoudre le problème jouet présenté ci-dessus, en laissant 1 seconde de temps de résolution à LocalSolver, est :

```
localsolver.exe io_lsp=toy.lsp hr_timelimit=1
io_solution=toy.sol
```

Ensuite, l'affichage de la console devrait ressembler à celui-ci :

```
Parsing LSP file toy.lsp...
25 nodes, 6 booleans
3 constraints, 1 objectives
1 phases, 2 threads
*** Initial feasible solution : obj = ( 9 )
*** Solve phase 1 over 1
Running phase for 1 sec and 4294967295 itr using
descent
* Thread 0 : [ 2 / 2 / 260000 ] moves in 0 sec,
obj = ( 7 ) in 0 sec and 3 itr
* Thread 1 : [ 3 / 4 / 260000 ] moves in 0 sec,
obj = ( 7 ) in 0 sec and 5 itr
*** Best solution : obj = ( 7 ) in 0 sec and 3
itr
Writing solution in file toy.sol...
```

Par défaut, LocalSolver 1.1 effectue une descente standard [1] en utilisant tous les mouvements autonomes disponibles. Une heuristique de recuit simulé peut également être sélectionnée via les options de la ligne de commande. Le nombre de threads est paramétrable (2 par défaut). Plusieurs résolutions sont alors exécutées en parallèle avec des graines différentes pour le générateur pseudo-aléatoire et leurs résultats sont périodiquement synchronisés. Finalement la meilleure solution est retournée une fois la limite de temps atteinte. Le parallélisme doit moins être vu comme accélérateur de la recherche que comme un moyen d'augmenter la robustesse du solveur. Quant aux mouvements autonomes, ils sont choisis aléatoirement sur la base d'une distribution non uniforme qui peut évoluer au cours de la recherche en fonction de leurs taux d'acceptation et d'amélioration.

Le coût de la solution initiale (admissible) trouvé par LocalSolver est 9. Cette solution est obtenue par un simple algorithme glouton randomisé. Comme expliqué plus haut, LocalSolver n'est pas conçu pour optimiser des problèmes fortement contraints. Par conséquent si cet algorithme d'initialisation ne trouve par de solution faisable, il appartient à l'utilisateur de transformer une de ses contraintes en un objectif de premier niveau. Notons qu'il s'agit là d'une différence fondamentale avec les approches CBLS, dans lesquelles une mesure de violation est définie sur chaque contrainte. Nous considérons que de telles relaxations sont de la responsabilité de l'utilisateur. Typiquement pour des problèmes d'affectation de fréquences, l'ingénieur pourra choisir entre affecter une fréquence à tous les liens tout en minimisant les interférences ou assurer l'absence d'interférence tout en minimisant le nombre de liens sans fréquence affectée.

La meilleure solution, trouvée par recherche locale après 3 itérations (et 0 secondes) par le thread 0, a un coût 7. Durant la seconde de temps alloué, LocalSolver a effectué 260 000 itérations, ce qui correspond au nombre total de mouvements tentés et de fait au nombre de solutions visitées durant la recherche. Parmi ces mouvements, 4 ont été validés sur le thread 1 et 3 ont été (strictement) améliorant. Ces statistiques

sont détaillés par mouvement, ce qui peut être utilisé pour ajuster le paramétrage du solveur. Enfin, LocalSolver crée le fichier "toy.sol" contenant la solution suivante : `xA=0; yA=1; zA=1; xB=1; yB=0; zB=0;`

```
x1 <- bool();
x2 <- bool();
x3 <- bool();
y1 <- bool();
y2 <- bool();
y3 <- bool();
sx <- booleansum(x1, x2, x3);
sy <- booleansum(y1, y2, y3);
constraint sx <= 2;
constraint sy >= 2;
obj <- max(sx, sy);
minimize obj;
```

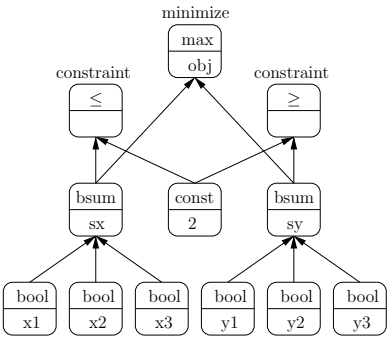

FIGURE 1 – Le graphe acyclique orienté (DAG) induit par un modèle simple. Pour chaque noeud, le type (resp. nom) du noeud est donné au dessus (resp. au dessous). Ici "bsum" signifie `booleansum`.

Un programme LSP, comme défini ci-dessus, peut être représenté par un graphe orienté acyclique (DAG, de l'anglais *Directed Acyclic Graph*), dont les racines sont les variables de décision et les feuilles sont les contraintes et objectifs (voir Fig. 1). Ensuite, les opérateurs utilisés pour modéliser le problème induisent les nœuds internes du DAG. Ces nœuds internes correspondent aux *invariants* ou *one-way constraints* dans des logiciels comme iOpt [28] ou Comet [24]. Selon cette représentation, une solution correspond a une instantiation des variables racines. Ainsi, appliquer des mouvements à la solution courante consiste à modifier les valeurs courantes des variables de décision (racines) et évaluer les valeurs des contraintes et objectifs (feuilles) par propagation des modifications dans le DAG.

L'architecture de LocalSolver est conçue en 3 couches suivant la méthodologie proposée par les auteurs [12] pour l'ingénierie d'heuristiques de recherche locale performantes : (méta)heuristiques, mouvements, évaluation. Résolument orienté vers la simplicité et l'efficacité, la conception et l'implémentation de LocalSolver a requis un effort considérable en terme d'ingénierie logicielle et algorithmique, ne pouvant être entièrement exposé ici. Ainsi, nous concentrerons notre présentation sur deux aspects cruciaux du solveur : l'évaluation des algorithmes et les mouvements autonomes.

4.1 Mouvements autonomes

Comme suggéré en introduction, notre but ultime est d'effectuer automatiquement les mouvements qu'un praticien aurait conçus pour résoudre son problème. Le mouvement le plus simple est le "K-flip" qui inverse aléatoirement les valeurs de K variables de décision (binaires). Cependant la structure du modèle permet souvent au solveur de concevoir des mouvements plus appropriés. Par exemple, lorsqu'une contrainte est posée sur une somme de variables binaires, un mouvement naturel consiste à modifier deux booléens de la somme dans des directions opposées, préservant ainsi la valeur de cette somme. Nous montrerons dans cette section que LocalSolver est également capable d'exploiter des structures plus complexes, en appliquant des mouvements autonomes qui peuvent être vus comme des chaînes d'ejection appliquées à l'hypergraphe induit par les variables booléennes et les contraintes (voir [20] pour plus de détail sur les chaînes d'éjection). Ces chaînes d'éjection sont spécialisées pour préserver la faisabilité des contraintes booléennes et sont un composant clé de l'efficacité de LocalSolver 1.x. Prenons l'exemple du problème d'ordonnancement de véhicules [10, 11] :

Par exemple, considérons le problème d'ordonnancement de véhicules [10, 11] : des voitures doivent être ordonnées sur une ligne de production de façon à minimiser un objectif non linéaire. Ce problème peut être modélisé comme un problème d'affectation (non linéaire) en définissant pour chaque voiture i et position p une variable booléenne $x_{i,p}$. Un voisinage basique pour ce modèle consiste à échanger les positions de deux véhicules. Au niveau du modèle, échanger les positions p et q de deux voitures i et j correspond à flipper successivement les 4 variables booléennes $x_{i,p}$, $x_{i,q}$, $x_{j,q}$, $x_{j,p}$, tout préservant la faisabilité des 4 contraintes de partition où ces variables apparaissent. D'une façon générique, nos mouvements autonomes correspondent à des *k-moves* ou *k-swaps* dans le cadre de problèmes de *packing/covering*.

Appelons une *somme racine* une somme dans laquelle apparaît au moins deux variables de décision binaires (éventuellement multipliées par un scalaire)

et qui est contrainte par un opérateur relationnel. Une structure de données est construite qui liste toutes les sommes racines dans le DAG et pour chaque variable décisionnelle, la liste des sommes racines la contenant. Ensuite, nous maintenons pour chaque somme racine l'ensemble des *booléens croissants*, c'est-à-dire les variables de décision dont la modification augmente la somme, et le complémentaire de cet ensemble (*booléens décroissants*). En utilisant cette structure, nous parvenons à effectuer des mouvements visant à trouver une chaîne alternée de booléens croissants et décroissants tel que deux booléens consécutifs dans cette chaîne appartiennent à la même somme racine. Pour obtenir un cycle alterné, comme décrit dans le précédent paragraphe, la propriété doit être vérifiée circulairement, entre la dernière et la première variable de la chaîne. l'idée maîtresse derrière ces mouvements, appelés *k-Chaînes* ou *k-Cycles*, est de réparer de façon alternée les sommes modifiés, en appliquant une modification opposée à chaque étape. En préservant la satisfaction des contraintes sur les sommes, les *k-Chaînes* et les *k-Cycles* tendent à maintenir la faisabilité de la solution, ce qui est crucial pour l'efficacité de la recherche. Par exemple, lorsque les sommes racines définissent une structure de couplage complet, tout k-Cycle avec k pair sera complété (c'est à dire fermé sans échec) en temps $O(k)$.

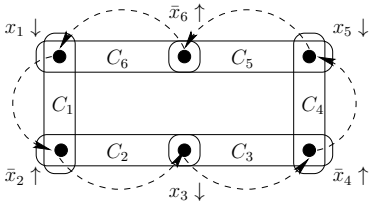

FIGURE 2 – Un 6-Cycle impliquant six variables binaires x_1, x_3, x_5 (dont la valeur actuelle est 1) et $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ (dont la valeur actuelle est 0), et six sommes contraintes $C_1, \ldots, C_6$. Chaque variable appartient à deux sommes (par exemple, x_1 appartient à C_1 et C_6). Diminuons x_1, x_3, x_5 ($\downarrow$) tandis que $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ sont augmentées ($\uparrow$). Ce mouvement préserve les valeurs des sommes et donc la faisabilité des contraintes.

Changer la définition des sommes racines conduit à des variantes qui peuvent être intéressantes en pratique pour accélérer la convergence de la recherche locale. Par exemple, nous pouvons nous concentrer sur des sommes ayant au moins une contrainte parmi leurs successeurs dans le DAG, ou bien sur des sommes dans lesquelles il n'y a que des variables de décision. Pour la sélection de la somme suivante à réparer, on peut également favoriser les sommes intervenant dans une contrainte d'égalité, parce que le mouvement ne peut pas réussir sans réparer ces sommes. Enfin, une autre idée consiste à flipper plus d'une variable par contrainte. Cette extension suit la même logique que la généralisation des chaînes d'éjection aux arbres d'éjection [6]. Cela permet par exemple d'éjecter 2 objets de taille 1 lorsque l'on ajoute un objet de taille 2 dans un ensemble, dans les problèmes de *packing*.

A notre connaissance, la conception de tels mouvements autonomes capables de préserver la faisabilité de la solution est une nouveauté. Ils forment un composant clé de LocalSolver en tant que boîte noire. En effet, par rapport aux voisinages classiques de type k-Flips employés dans les solveurs SAT/PB [23, 29], ils améliorent très largement l'efficacité de la recherche (c'est à dire sa convergence vers des solutions de qualité) sur des problèmes combinatoires structurés de très grande taille (du type de ceux rencontrés en pratique dans les applications de la RO). Notons que les auteurs ont appris récemment que des mouvements autonomes spécifiques sont implémentés dans le logiciel IBM ILOG Transportation PowerOps (TPO) afin de résoudre des problèmes de tournées de véhicules par recherche locale en mode boîte noire [13].

4.2 Algorithmes d'évaluation

Les premiers algorithmes d'évaluation incrémentale ont été introduits dans Localizer [15], l'ancêtre de Comet [24], et iOpt [28]. Ils se basent sur l'exploitation d'invariants sur les opérateurs combinatoires. Bien que ce mécanisme soit connu dans la littérature, nous le présentons ici dans un souci de complétude, et nous illustrons en fin de section les spécificités de notre implémentation.

Chaque nœud du DAG doit implémenter les méthodes suivantes : `init`, `eval`, `commit`, `rollback`. La méthode `init` est responsable de l'initialisation de la valeur du nœud en fonction des valeurs de ses parents, avant la recherche locale. Les structures de données spécifiques attachées au nœud, utilisées pour accélérer son évaluation incrémentale, sont aussi initialisées par cette méthode. Après application d'un mouvement sur les variables de décision, la méthode `eval` est appelée pour réévaluer de façon incrémentale la valeur du nœud, quand celui-ci est impacté durant la propagation. Ensuite, si le mouvement est accepté par l'heuristique, alors la méthode `commit` est appelée sur chaque nœud modifié pour valider les modifications induites par le mouvement. Sinon, le mouvement est rejeté, et la méthode `rollback` est utilisé à la place.

Comme mentionné précédemment, l'évaluation des mouvements est accélérée en exploitant les invariants relatifs à chaque type de nœud [15]. La propagation des modifications est faite par recherche en largeur dans le

DAG, garantissant que chaque nœud est visité au plus une fois. S'appuyant sur un patron de conception du type *observer*, la propagation est réduite aux nœuds impactés : un nœud est dit impacté si l'un de ses parents a été modifié. Par exemple, considérons le nœud $z \leftarrow a < b$ avec une valeur courante égale à vrai. Celui-ci ne sera pas impacté si la valeur de a diminue ou celle de b augmente. Ensuite, à chaque nœud est associé une méthode *eval* appelée pour calculer la nouvelle valeur du nœud impacté. Cette méthode prend en entrée la liste des parents modifiés (c'est-à-dire les nœuds parents dont la valeur a changé). Pour un opérateur linéaire comme `sum`, l'évaluation est simple : si k termes de la somme sont modifiés, alors sa nouvelle valeur est calculée en temps $O(k)$. Mais pour d'autres opérateurs (arithmétiques ou logiques), des accélérations significatives peuvent être obtenues en pratique. Par exemple, considérons le nœud $z \leftarrow \text{or}(a_1, \ldots, a_k)$ avec M la liste des a_i modifiés et T la liste (maintenue) des a_i dont la valeur courante est vrai. Nous pouvons observer que si $|M| \neq |T|$, alors la nouvelle valeur de z est nécessairement vrai. La validité de cette condition permet alors une évaluation en temps $O(1)$, et non plus $O(k)$. En effet, si $|M| < |T|$, alors au moins un parent reste avec une valeur égale à vrai ; sinon, il existe au moins un parent dont la valeur est modifiée de faux à vrai. Notre implémentation est focalisée sur la complexité pratique et expérimentale et non pas seulement sur la complexité au pire cas. Les facteurs constants ont une grande importance : de bons algorithmes et une optimisation du code améliorent significativement la vitesse d'évaluation. Par exemple la propriété mentionnée ci dessus pour maintenir l'opérateur `or` n'est, à notre connaissance, pas employée dans les systèmes Comet ou iOpt. De la même manière, dans Localizer [15, p. 67] maintient l'opérateur `min` en temps $O(\log k)$ avec k le nombre d'opérandes, en utilisant classiquement un tas binaire. Dans LocalSolver nous distinguons deux cas. Si la valeur minimum parmi les opérandes modifiés est inférieure ou égale à la valeur actuelle de l'opérateur `min` ou bien si un support demeure inchangé, alors l'évaluation est faite optimalement en temps $O(|M|)$ avec $|M|$ le nombre de valeurs modifiées. Sinon l'évaluation est faite en temps $O(k)$ time. En pratique le premier cas est de loin le plus fréquent et et nombre d'opérandes modifiés ne dépende pas de la taille du problème ($|M| = O(1)$), ce qui assure une complexité amortie en temps constant pour l'évaluation.

5 Résultats expérimentaux

LocalSolver a été testé sur un benchmark mixant problèmes académiques et industriels, sélectionnés avant de débuter le projet. Nous insistons sur le fait que notre but n'est pas d'atteindre (encore moins dépasser) les meilleurs résultats de la littérature pour tous ces problèmes. *Le but de LocalSolver est d'obtenir en mode boîte noire de bonnes solutions en des temps d'exécution courts (comme le ferait des heuristiques standards de recherche locale), en particulier quand les solveurs arborescents sont incapables d'en trouver une.* Le but de ces expérimentations est de comparer LocalSolver aux solveurs boîte noire existants : IBM ILOG CPLEX 12.2 (le solveur de référence pour la programmation entière) et Comet CBLS 2.1 [9, pp. 330-331] qui, bien que n'étant pas conçu a priori pour un usage boîte noire, offre une recherche tabu générique à base de swaps. IBM ILOG CP Optimizer a été testé également mais n'a pas conduit à des résultats compétitifs sur ces problèmes. Les résultats de solveurs SAT ou PB, inappropriés au traitement de ce type de problèmes structurés d'optimisation, sont également omis.

Pour chacun de problèmes de ce benchmark, les solveurs sont comparés sur la même machine avec le même modèle standard. Toutes les expérimentations ont été conduites sur un ordinateur standard équipé du système Windows XP 32 bits et du processeur Intel Core 2 Duo T7600 (2.33 GHz, RAM 2 Go, L2 4 Mo, L1 64 ko). Notons que deux coeurs seulement sont disponible sur cet ordinateur. Quant au modèle il est simplement adapté à la grammaire de chaque solver. Ainsi les modèles LSP et PLNE sont exprimés à l'aide de variables de décision binaires alors que les modèles CP et CBLS utilisent des variables entières et les contraintes globales disponibles. De même un opérateur *max* par exemple est natif en CBLS et LocalSolver mais sera exprimé comme un ensemble d'inégalités dans le modèle PLNE équivalent. Il est important de souligner qu'aucun élement spécifique n'a été ajouté au delà de ces nécessaires transformations (par d'inégalités valides par exemple).

L'efficacité d'un solveur autonome est une combinaison de plusieurs facteurs comme son implémentation, ses algorithmes internes, sa stratégie de recherche, ses mouvements ou coupes par défaut, sa recherche d'une solution initiale, etc. Ici chaque solveur est lancé avec ses paramètres par défaut, sauf mention contraire. En particulier aucune aucune solution initiale n'est fournie à LocalSolver ou Comet et aucune stratégie de recherche ou mouvements ne sont spécifiés : ces choix sont la responsabilité des solveurs boîte noire.

Chaque problème traité dans ce benchmark est brièvement décrit et le modèle choisi est cité ou esquissé. Les résultats obtenus par les différents solveurs sont donnés sur un ensemble représentatif d'instances, ainsi que (en tant que référence) les meilleurs résultats connus dans la littérature, généralement obtenus par

des heuristiques de recherche locale. Tous les résultats présentés ici ont été rigoureusement validés, en particulier le respect des contraintes et l'exactitude de l'objectif ont été vérifiés en dehors du modèle mathématique. Tout le matériel utilisé pour ce benchmark (code, modèles, résultats) est disponible sur demande.

Dans tous les tableaux ci-dessous, la ligne "LocalSolver 1.1" correspond aux résultats obtenus par LocalSolver 1.1 ; la ligne "CPLEX 12.2" correspond aux résultats obtenus par IBM ILOG CPLEX 12.2 et la ligne "CBLS Comet 2.1" correspond aux résultats obtenus par la recherche tabu générique de Comet 2.1. L'origine de la ligne "state of the art" sera donnée pour chaque problème.

5.1 Ordonnancement de véhicules

Le problème d'ordonnancement de véhicules [14] consiste à ordonner des véhicules sur une chaîne d'assemblage tout en minimisant les violations sur des contraintes de ratio. Pour LocalSolver et CPLEX, l'affectation des véhicules aux positions est modélisée par des variables booléennes et les pénalités sur chaque ratio sont additionnées (see [10]). En Comet nous utilisons des variables de décision entières et non booléennes. En outre une contrainte "sequence" est disponible dans ce language pour modéliser exactement les pénalités d'ordonnancement de véhicules. Cette contrainte globale a été nécessaire pour obtenir les résultats présentés ci-dessous (sans cette contrainte les résultats de Comet sont deux fois plus grands sur les plus grandes instances de ce problème de minimization).

Un échantillon de résultats est présenté pour 5 instances sur la Table 1 en fin de papier : 10-93 (100 véhicules, 5 options, 25 classes), 200-01 (200 véhicules, 5 options, 25 classes), 300-01 (300 véhicules, 5 options, 25 classes), 400-01 (400 véhicules, 5 options, 25 classes), 500-08 (500 véhicules, 8 options, 20 classes). Les 4 premières instances sont disponibles dans la CSPLib [14] ; la cinquième provient d'un benchmark généré par Perron *et al.* [17]. Dans la table, la ligne "state-of-the-art" correspond aux résultats obtenus avec l'algorithme de recherche locale des auteurs [10], qui obtient les meilleurs résultats sur toutes les instances. Les résultats présentés dans la table du haut (resp. bas) ont été obtenus avec un temps d'exécution limité à 60 (resp. 600) secondes. Le coût de la meilleure solution trouvée est donné et le symbole "x" est écrit si aucune solution n'a été obtenue dans le temps imparti. En résumé, nous observons que les résultats de LocalSolver dépassent ceux des solveurs de PLNE et CBLS, d'autant plus que l'échelle des instances grandit (notons que les instances à 400 et 500 véhicules induisent chacun 10 000 variables de décision booléennes). Nous ne détaillons pas les résultats des solveurs de PPC [16, 17], qui ne s'avèrent pas compétitifs pour traiter ce problème : Peron *et al.* [16, 17] obtiennent par *Large Neighborhood Search* un nombre de violations supérieur à 500 pour l'instance 500-08.

Une version industrielle du problème, intégrant les contraintes et objectifs de l'atelier de peinture, a été proposé par l'entreprise Renault comme sujet du Challenge ROADEF 2005 [11] (compétition internationale organisée tous les deux ans par la Société Française de Recherche Opérationnelle). Dans cette version, trois objectifs lexicographiques sont optimisés : EP = violations sur les contraintes de ratio prioritaires, ENP = violations sur les contraintes de ratio non prioritaires, RAF = le nombre de changements de couleur dans la séquence. La Table 2 contient un échantillon des résultats sur 3 instances : X2 = 023-EP-RAF-ENP-S49-J2 (1260 véhicules, 12 options, 13 couleurs), X3 = 024-EP-RAF-ENP-S49-J2 (1319 véhicules, 18 options, 15 couleurs), X4 = 025-EP-ENP-RAF-S49-J1 (996 véhicules, 20 options, 20 couleurs). Aucun solveur de PLNE/PPC/SAT n'est parvenu à traiter ces instances à ce jour [11]. Par exemple CPLEX 11.2 ne parvient pas à trouver de solution après plusieurs heures de calcul, de même pour Comet. La ligne 'Comet CBLS 2.1 (relaxed)" donne les résultats obtenus avec des modèles dans lesquels les contraintes de "paint limit" sont omises. Ici la recherche locale de l'état de l'art correspond à celle locale qui a remporté le challenge [11] ; notons que la conception et l'implémentation de cet algorithme a demandé près de 150 jours de travail à leurs auteurs. Pour l'instance X2, le modèle LSP contient 516 936 variables dont 374 596 sont des variables de décision binaires (450 Mo de RAM sont alloués durant l'exécution).

Localsolver réalise entre 1.5 et 4.5 million de mouvements par minute, avec un taux d'acceptation entre 5 et 20 % et presque un millier de solutions améliorantes. Observons que les résultats de LocalSolver utilisé sont comparables à ceux de la recherche à voisinages variables de Prandtstetter et Raidl [18] qui mixe mouvements classiques et voisinages larges explorés par PLNE.

5.2 Social golfer

Le problème *social golfer* [14] consiste à affecter des personnes à des groupes sur plusieurs semaines de façon à maximiser le nombre de rencontres uniques. Une fois modélisée la structure de partitionnement pour chaque semaine, la détection des rencontres entre golfeurs est très simple dans les différents solveurs considérés. Ce problème est rencontré chez Bouygues SA lors de la planification des séminaires des managers du Groupe Bouygues. Des échantillons de résultats

sont présentés sur la Table 3. Quatre instances classiques sont traitées (2 faciles et 2 difficiles) : 9-3-11 (9 groupes de 3 golfeurs sur 11 semaines), 10-10-3, 10-9-4, 10-3-13. Dans ce cas, l'objectif est de minimiser le nombre de rencontres en doublon. La cinquième instance, appelée "séminaire", est une instance réelle (120 personnes sur 3 semaines avec des tailles de groupe entre 7 et 9) avec comme des contraintes additionnelles portant sur les groupes et trois objectifs lexicographiques : équilibrer les caractères dans chaque groupe (un caractère peut être la filiale dans laquelle évolue la personne, son sexe, ou encore ses centres d'intérêt), éviter les rencontres indésirables, et maximiser le nombre de rencontres uniques (souhaitées). Pour les instances classiques, la recherche locale de l'état de l'art correspond à l'heuristique tabou implémentée en langage C dans [8], qui détient les records sur presque toutes les instances. Des approches par PPC dédiées et complexes (visant à casser les symétries) permettent d'obtenir des résultats similaires [8]. Pour l'instance réelle, l'état de l'art correspond à l'algorithme de recherche locale implémenté par un des auteurs comme solution opérationnelle : une heuristique de descente en *first-improvement* effectuant des échanges aléatoires à l'évaluation efficace (plus d'un million par seconde). Observons que du fait de la forme quadratique de la fonction objectif (le décompte des rencontres ne peut se faire qu'en utilisant un "et" logique), un tel problème est très difficile à traiter par des techniques de PLNE ; en effet, sa linéarisation induit un très grand nombre de variables (centaines de milliers). En résumé, nous observons qu'en dépit d'un temps d'exécution 10 fois plus élevé, les solveurs de PLNE ne parviennent pas à produire de bonnes solutions. Comme précédemment, LocalSolver ne parvient pas à atteindre les records pour les instances difficiles, mais en est proche.

5.3 Découpe de plaques d'acier

Le problème de découpe de plaques d'acier (connu sous l'appellation anglaise *Steel Mill Slab Design*) [14] est un problème de type *bin packing / cutting stock*, où des commandes d'acier de différentes tailles doivent être découpées dans des plaques de différents capacités tout en minimisant les chutes. En outre chaque commande a une "couleur" et le nombre de couleurs différentes dans une plaque est limité. Le modèle est basé sur l'affectation des commandes au plaques et la taille de chaque plaque est determinée par son contenu (même modèle que [22]). L'instance classique de la CSPLib avec 111 plaques [14] est résolue à l'optimum (coût égal à 0) en moins d'une seconde par LocalSolver, dépassant ainsi les approches proposées jusqu'à présent (PLNE, PPC, SAT) [26]. Notons que le LSP traité par LocalSolver dans ce cas contient 40 739 variables dont 12 321 booléens ; LocalSolver visite près de 100 000 solutions par seconde. Un échantillon des résultats obtenus sur les nouvelles instances proposées par Schaus [2] est donné Table 4. Les 200 instances numérotées de 11-0 à 20-19 ne sont pas mentionnées, car toutes sont résolues à l'optimum en moins d'une seconde. Les meilleurs résultats connus sont obtenus par Heinz [2], en utilisant judicieusement une décomposition de Dantzig-Wolfe qui peut être directement résolue par un solveur de PLNE grâce au nombre raisonnable de colonnes (fortement filtrées grâce aux contraintes de couleur et qui peuvent donc être toutes générées d'emblée). Notons que cette approche est dédiée au instances de la CSPLib : des limites moins basses sur le nombre de couleurs rendraient le nombre de colonnes bien plus grand, imposant alors une exploration de type branch-and-price. LocalSolver parvient ici à rester relativement proches des solutions optimales alors que CPLEX et CBLS en sont très loin. De même, les meilleures approches par PLNE ne sont pas competitives [22].

5.4 Prise de vues par le satellite Spot 5

Le problème de prise de vues par le satellite Spot 5 [27] consiste à sélectionner un sous-ensemble de photos à prendre par le satellite Spot 5 ; le but est de maximiser une fonction de profit sujet à des contraintes de sac-à-dos et d'exclusion mutuelle. Ce programme linéaire en variables binaires s'écrit de la même manière pour CPLEX et pour LocalSolver. Les plus grandes instances abordées dans la littérature (cas multi-orbites) contiennent au plus un millers de photos en entrée, ce qui les rend aujourd'hui efficacement abordables par des solveurs de PLNE. Un échantillon de résultats est présenté Table 5. L'état de l'art correspond ici à l'heuristique tabou de Vasquez et Hao [27] ; de plus, ces auteurs ont montré que leurs résultats se situés à moins d'1 % de l'optimum. La conclusion principale de cette expérience est que LocalSolver reste compétitif face aux solveurs de PLNE lorsque l'échelle des instances, plus petite, devient favorable aux techniques de recherche arborescente.

5.5 Minimum de matériel de coffrage

Le problème de minimisation du matériel de construction (en anglais *minimum formwork stock problem* [4], rencontré chez Bouygues Construction, consiste à minimiser le matériel de coffrage utilisé par un chantier. Une fois décomposé le problème peut être vu comme un problème de couverture dont l'échelle permet une résolution efficace par un solveur de PLNE.

2. `http://becool.info.ucl.ac.be/steelmillslab`

Quelques résultats sont présenté en Table 6. Comme dans le problème précédent, on observe que LocalSolver reste compétitif face à CPLEX.

5.6 Eternity II

Le problème Eternity II [2, 21] est un puzzle aussi ludique que difficile édité par la société TOMY en 2007. Le puzzle consiste à remplir une grille de taille 16×16 avec 256 pièces carrées, dont les quatre côtés sont colorés. Le but est de trouver un placement des pièces sur le plateau de façon à ce que les côté adjacents de toute paire de pièces voisines soient de la même couleur. Un tel problème peut être modélisé comme un problème d'optimisation : affecter toutes les pièces sur le plateau tout en minimisant le nombre de paires violant ces contraintes d'adjacences de couleur. A notre connaissance la meilleure solution connue a 13 violations (sur 480). [21] obtiennent une solution à 22 violations à l'aide d'une recherche tabou à grand voisinage. Notons qu'ils obtiennent des solutions avec presque 70 violations en effectuant uniquement des échanges de pièces au sein d'un recherche tabou. Les variables de décision pour chaque pièce sont sa rotation et sa position sur le plateau (exprimées par des variables booléennes en CPLEX et LocalSolver) et les arêtes incompatibles sont simplement détectées à l'aide des opérateurs disponibles dans chaque solver. Nous obtenons une solution avec 70 violations en une journée de calcul en utilisant LocalSolver 1.1. Dans ce cas, LocalSolver réalise presque un million de mouvements par minute alors que le modèle LSP contient 262 144 variables de décision binaires. Notons que CPLEX ne fournit aucune solution après un jour de calcul alors que la recherche générique de Comet obtient une solution avec 107 violations. Les approches purement PPC ne parviennent pas à descendre sous la barre des 80 violations [21].

6 Conclusion

Les résultats ci-dessus démontre qu'une "programmation par recherche locale" est possible : une paradigme du type "*model & run*" pour la recherche locale peut être obtenu en combinant un langage de modélisation simple et un solveur incrémental efficace basé sur des mouvements autonomes appropriés. Ainsi, la prochaine version de LocalSolver est envisagée suivant plusieurs directions de recherche. Tout d'abord, le formalisme LSP est loin d'être complet : notre préoccupation principale est d'y ajouter la notion d'ensembles sans perdre en simplicité et généricité. Cette étape est cruciale pour attaquer des problèmes complexes d'ordonnancement et de routage. Ensuite, le concept de mouvements autonomes maintenant la faisabilité, qui est une des clés de notre approche, doit être renforcé et développé. Enfin, nous avons planifié d'ajouter plus de métaheuristiques [1] dans la couche supérieure du solveur, en sus de la descente standard et du recuit simulé.

Références

[1] E. Aarts and J.K. Lenstra. Local search in combinatorial optimization. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, England, UK, 1997.

[2] T. Benoist and E. Bourreau. Fast global filtering for Eternity II. *Constraint Programming Letters*, 3 :35–50, 2008.

[3] T. Benoist, B. Estellon, F. Gardi, and A. Jeanjean. High-performance local search for solving inventory routing problems. In T. Stützle, M. Birattari, and H. Hoos, editors, *Proceedings of SLS 2009, the 2nd International Workshop on Engineering Stochastic Local Search Algorithms*, volume 5752 of *Lecture Notes in Computer Science*, pages 105–109. Springer, 2009.

[4] T. Benoist, A. Jeanjean, and P. Molin. Minimum formwork stock problem on residential buildings construction sites. *4OR-Q J Oper Res*, 7 :275–288, 2009.

[5] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO : a framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3) :357–380, 2004.

[6] Y. Caseau, F. Laburthe, and G. Silverstein. A meta-heuristic factory for vehicle routing problems. In *Proceedings of CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 1999.

[7] L. Di Gaspero and A. Schaerf. EasyLocal++ : an object-oriented framework for flexible design of local search algorithms. *Software - Practice & Experience*, 33(8) :733–765, 2003.

[8] I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In *Proceedings of CPAIOR 2005*, volume 3524 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 1999.

[9] Dynadec Decision Technologies Inc., Providence, RI. *Comet 2.1 Tutorial*, March 2010. 581 pages, `http://www.dynadec.com`.

[10] B. Estellon, F. Gardi, and K. Nouioua. Large neighborhood improvements for solving car sequencing problems. *RAIRO Operations Research*, 40(4) :355–379, 2006.

[11] B. Estellon, F. Gardi, and K. Nouioua. Two local search approaches for solving real-life car sequencing problems. *European Journal of Operational Research*, 191(3) :928–944, 2008.

[12] B. Estellon, F. Gardi, and K. Nouioua. High-performance local search for task scheduling with human resource allocation. In *Proceedings of SLS 2009*, volume 5752 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

[13] D. Fernandez Pons, 2010. Personnal communication.

[14] B. Hnich, I. Miguel, I.P. Gent, and T. Walsh. CSPLib : a problem library for constraints, 2009. `http://www.csplib.org`.

[15] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1/2) :43–84, 2000.

[16] L. Perron and P. Shaw. Combining forces to solve the car sequencing problem. In *Proceedings of CPAIOR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2004.

[17] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In *Proceedings of CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer, 2004.

[18] M. Prandtstetter and G.R. Raidl. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research*, 191(3) :1004–1022, 2008.

[19] J.F. Puget. Constraint programming next challenge : Simplicity of use. In *Proceedings of CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8, 2004.

[20] C. Rego and F. Glover. Local search and metaheuristics. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, pages 105–109. Kluwer Academic Publishers, Dordrecht, Netherlands, 2002.

[21] P. Schaus and Y. Deville. Hybridation de la programmation par contraintes et d'un voisinage à très grande taille pour Eternity II. In *Proceedings of JFPC 2008*, pages 115–122, 2008.

[22] P. Schaus, P. Van Hentenryck, J.-N. Monette, C. Coffrin, L. Michel, and Y. Deville. Solving steel mill slab problems with constraint-based techniques : CP, LNS, CBLS. to appear in Constraints, 2011.

[23] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D.S. Johnson and M.A. Trick, editors, *Cliques, Coloring, and Satisfiability : 2nd DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, Providence, RI, 1996.

[24] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Boston, MA, 2005.

[25] P. Van Hentenryck and L. Michel. Synthesis of constraint-based local search algorithms from high-level models. In *Proceedings of AAAI 2007*, pages 273–279, 2007.

[26] P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. In *Proceedings of CPAIOR 2008*, volume 5015 of *Lecture Notes in Computer Science*, pages 377–381, 2008.

[27] M. Vasquez and J.-K. Hao. A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Computational Optimization and Applications*, 20(2) :137–157, 2001.

[28] C. Voudouris, R. Dorne, D. Lesaint, and A. Liret. iOpt : a software toolkit for heuristic search methods. In *Proceedings of CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 716–730, 2001.

[29] J. Walser, R. Iyer, and N. Venkatasubramanyan. An integer local search method with application to capacitated production planning. In *Proceedings of AAAI 1998*, pages 373–379, 1998.

TABLE 1 – Quelques résultats pour le problème d'ordonnancement de véhicule (minimisation).

time limit : 60 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art	3	0	0	1	0
LocalSolver 1.1	8	8	8	13	18
CPLEX 12.2	6	11	27	17	x
Comet CBLS 2.1	7	8	16	18	91
time limit : 600 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art	3	0	0	1	0
LocalSolver 1.1	6	5	4	6	6
CPLEX 11.2	3	3	11	16	104
Comet CBLS 2.1	7	6	10	18	47

TABLE 2 – Quelques résultats pour le problème d'ordonnancement des véhicules Renault (minimisation). le classement de chaque résultat relativement au 18 finalistes du challenge Roadef est donné entre parenthèses.

time limit : 600 s	X2			
state-of-the-art	0,	192,	66	(1/19)
LocalSolver 1.1	0,	268,	212	(16/19)
Comet CBLS 2.1 (relaxed)	799,	1069,	481	(19/19)

time limit : 600 s	X3			
state-of-the-art	0,	337,	6	(1/19)
LocalSolver 1.1	36,	544,	187	(16/19)
Comet CBLS 2.1 (relaxed)	1447,	1100,	309	(19/19)

time limit : 600 s	X4			
state-of-the-art	0,	160,	407	(1/19)
LocalSolver 1.1	2,	353,	692	(18/19)
Comet CBLS 2.1 (relaxed)	1055,	1888,	651	(19/19)

TABLE 3 – Quelques résultats pour le problème du social golfer (minimisation).

time limit : 60 s	9-3-11	10-10-3	10-9-4	10-3-13	seminar
state-of-the-art	0	0	0	0	1, 0, 1082 = 11 082
LocalSolver 1.1	0	0	4	1	1, 0, 1082 = 11 082
Comet CBLS 2.1	2	0	8	6	x
time limit : 600 s	9-3-11	10-10-3	10-9-4	10-3-13	seminar
CPLEX 12.2	94	140	218	125	3 629 775
Comet CBLS 2.1	1	0	5	3	x

TABLE 4 – Quelques resultats pour le *Steel Mill Slab Design Problem* (minimisation).

time limit : 60 s	2-0	3-0	4-0	5-0	6-0
state-of-the-art	22	5	32	0	0
LocalSolver 1.1	31	5	34	4	8
CPLEX 12.2	178	511	x	x	x
Comet CBLS 2.1	136	135	69	65	42
time limit : 300 s	2-0	3-0	4-0	5-0	6-0
state-of-the-art	28	6	34	0	0
LocalSolver 1.1	40	34	35	3	7
CPLEX 12.2	94	65	x	63	x
Comet CBLS 2.1	124	110	43	58	33

time limit : 60 s	7-0	8-0	9-0	10-0
state-of-the-art	0	0	0	0
LocalSolver 1.1	2	0	0	0
CPLEX 12.2	275	226	229	201
Comet CBLS 2.1	30	26	21	20
time limit : 300 s	7-0	8-0	9-0	10-0
state-of-the-art	0	0	0	0
LocalSolver 1.1	1	0	0	0
CPLEX 12.2	189	226	97	64
Comet CBLS 2.1	33	17	17	15

TABLE 5 – Quelques résultats pour le problème Spot 5 de planification de prises de vues (maximisation).

time limit : 60 s	54	414	509	1401	1403
state-of-the-art	70	22 120	19 125	176 056	176 140
LocalSolver 1.1	69	22 115	19 118	167 064	169 127
CPLEX 12.2	70	22 119	19 125	176 056	176 138
time limit : 60 s	1405	1407	1502	1504	1506
state-of-the-art	176 179	176 245	61 158	124 243	168 247
LocalSolver 1.1	160 177	162 253	61 158	123 240	152 247
CPLEX 12.2	174 181	176 237	61 158	124 243	168 246

TABLE 6 – Quelques résutlats sur le *Minimum Formwork Stock* (minimisation).

time limit : 60 s	site1	site8b	site12b	site13b
LocalSolver 1.1	5 640 326	5 640 398	9 223 040	7 729 336
CPLEX 12.2	5 409 158	5 409 240	8 392 196	7 408 436