Article invité

LocalSolver

De la recherche locale en programmation mathématique

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

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

www.localsolver.com

1 Pourquoi LocalSolver ?

Un des plus beaux outils fournis par la programmation mathématique est sans nul doute l'algorithme du simplexe de Dantzig [5] pour la programmation linéaire. Le magazine *Computing in Science and Engineering* le citait parmi le top 10 des algorithmes du 20ème siècle dans son numéro de l'an 2000. En effet, qui n'a jamais été émerveillé en regardant défiler les itérations de son solveur de programmation linéaire préféré ? On modélise et "il" résout. C'est magique ! Nul n'a besoin de connaître les fondements mathématiques et encore moins toute l'ingénierie algorithmique sur lesquels repose l'efficacité du simplexe pour résoudre un programme linéaire. En revanche, l'utilisateur doit parvenir à modéliser le problème qui lui est posé, en des termes souvent bien éloignés des mathématiques. Mais cela, c'est l'affaire du chercheur opérationnel : variables de décision, contraintes et objectifs, et le tour est joué. Magique ! Pourtant, d'aucuns pourraient trouver à y redire : n'est-ce pas un comble que la star de la programmation mathématique – le simplexe – ne soit pas efficace au sens de la *théorie* de la complexité (puisqu'exponentiel dans le pire des cas) ? Le praticien, lui, n'en est pas moins heureux : que l'efficacité *empirique* du simplexe soit encore mal comprise n'est pas un souci pour lui, tant que ça marche !

Malheureusement, beaucoup des problèmes que l'on pose aux chercheurs opérationnels ne sont pas linéaires mais combinatoires. La programmation en variables entières (PLNE) est ainsi devenu un sujet de préoccupation majeur pour nombre de mathématiciens et d'informaticiens. Mais à ce jour, il n'existe point d'algorithme magique pour résoudre ce type de programmes (à moins que P = NP, il n'en existera sans doute jamais). Les algorithmes les plus performants pour traiter ce type de programmes sont fondés sur la technique de séparation et évaluation, connue sous le nom de *branch and bound* en anglais. Ces algorithmes utilisent généralement le simplexe comme routine d'évaluation, pour calculer la relaxation linéaire du sous-problème obtenu après branchement. Cette technique de *branch and bound* appartient à la grande famille des techniques de recherche arborescente, qui consistent en une exploration de l'espace des solutions par instantiation itérative des variables de décision. L'efficacité de ces approches en pratique repose sur leur capacité à élaguer l'arbre de recherche (de taille exponentielle dans le pire des cas). Grâce aux progrès algorithmiques et à l'accélération des processeurs, les meilleurs solveurs de PLNE peuvent aujourd'hui s'attaquer à des programmes comportant plus de 100 000 variables booléennes lorsque leur relaxation linéaire n'est pas trop mauvaise. Il n'en reste pas moins que ces solveurs butent encore sur des programmes fortement combinatoires à quelques milliers de variables.

Que font donc les praticiens lorsqu'aucun solveur de PLNE n'est capable de résoudre leur problème ? Beaucoup implémentent un algorithme de recherche locale. Les techniques de recherche locale consistent à appliquer de façon itérative des transformations (locales) à une solution dans le but d'en trouver de meilleures [1]. Bien que ne garantissant pas l'obtention d'un optimum, ces techniques sont appréciées des chercheurs opérationnels car elles permettent d'obtenir des solutions de qualité

1. Bouygues e-lab, Paris.
2. Laboratoire d'Informatique Fondamentale – CNRS UMR 6166, Aix-Marseille Université, Marseille-Luminy.

en des temps d'exécution de l'ordre de la minute. Toutefois, la conception et l'implémentation d'algorithmes de recherche locale performants n'est pas chose facile. La couche algorithmique dédiée à l'évaluation des transformations est particulièrement délicate à mettre en oeuvre, car elle demande une expertise en algorithmique et une dextérité dans la programmation que tous les ingénieurs ne possèdent pas. Plusieurs logiciels ont vu le jour dans le but de faciliter le développement d'algorithmes de recherche locale. Mais la majorité de ces logiciels n'offre pas de fonctionnalités notables quant à la couche algorithmique responsable de l'évaluation (incrémentale) des mouvements, qui reste entièrement à la charge de l'utilisateur.

Nous pensons qu'en dépit des progrès algorithmiques et de l'augmentation de la puissance de calcul, les approches par pure recherche arborescente resteront limitées face aux problèmes combinatoires de grande taille, notamment face aux grands problèmes industriels qui intègrent toujours plus de dimensions, de contraintes et d'objectifs. Qu'est-ce qui nous fait dire cela ? Tout d'abord, nous observons que bien souvent la relaxation continue n'apporte rien à la recherche alors que son calcul est très coûteux en temps. Dès lors, pourquoi perdre du temps à énumérer des solutions partielles ? Ensuite, pourquoi une recherche arborescente incomplète serait-elle plus performante qu'une recherche locale ? En effet, dès lors qu'elle ne termine pas, une recherche arborescente n'offre aucune garantie sur la qualité de la solution fournie, ni même seulement l'obtention d'une solution admissible. De plus, il est difficile d'explorer par recherche arborescente un espace de façon randomisée (non biaisée). C'est pourquoi les solveurs de PLNE les plus performants intègrent de plus en plus d'éléments de recherche locale [4, 8].

Si seule la recherche locale permet de passer à l'échelle, alors pourquoi ne pas envisager un solveur fondé sur la recherche locale ? Quelques travaux vont dans ce sens. Les meilleurs prouveurs SAT sont basés sur des techniques de recherche locale [10]. Mais en dépit de quelques succès [13], les tentatives d'extension de ces techniques à la PLNE se sont montrées peu convaincantes face à des problèmes industriels d'optimisation. iOpt [12] et Comet CBLS [11] sont des frameworks visant à faciliter l'implémentation d'heuristiques de recherche locale en s'appuyant sur une évaluation incrémentale automatique. Mais à ce jour, nous ne connaissons pas de solveur *boîte-noire* fondé sur la recherche locale pour l'optimisation combinatoire (en particulier aucun qui ne soit utilisé industriellement).

Fort de notre expertise et de nos expériences en recherche locale (notamment les Challenges ROADEF 2005 et 2007 [6, 7]), nous avons débuté en 2007 le projet LocalSolver. Ce projet à long terme, alliant des ingénieurs du Bouygues e-lab (Thierry Benoist, Frédéric Gardi, puis Romain Megel et Julien Darlay récemment recrutés) et des chercheurs du Laboratoire d'Informatique Fondamentale (LIF) de Marseille (Bertrand Estellon, Karim Nouioua), avait pour objectifs : 1) définir un formalisme de modélisation mathématique simple et générique adapté à une résolution par recherche locale (*model*) ; 2) développer un solveur efficace exploitant ce formalisme avec comme principe fondamental "que le solveur fasse ce qu'un expert en recherche locale ferait" (*run*).

2 LocalSolver 1.x

Début 2010, une première version de LocalSolver était diffusée permettant de résoudre des *programmes non linéaires en variables 0-1 de grande taille* (plus de 100 000 variables) issus de problématiques industrielles (par exemple, le problème d'ordonnancement de véhicules posé par Renault lors du Challenge ROADEF 2005). Une version 1.1 améliorée (recuit simulé, *multithreading*, *etc.*) a été présentée à l'occasion de ROADEF 2011 ; elle compte plus de 1000 téléchargements à ce jour. Ces versions 1.x sont gratuites pour toute utilisation, y compris à des fins commerciales. Le solveur est en exploitation dans plusieurs filiales du Groupe Bouygues (TF1, ETDE, Colas) mais aussi en dehors : la société de services Eurodecision l'a adopté pour résoudre certains de ses problèmes (par exemple, l'optimisation énergétique d'une ligne de métro pour Siemens).

Bien qu'il requière quelques 10 000 lignes de code C++, le cœur de LocalSolver se résume en deux points : 1) des mouvements structurés sur les variables booléennes, tels des chaînes ou cycles d'éjection, tendant à maintenir l'admissibilité de la solution courante ; 2) une évaluation de ces mouvements accélérée par l'exploitation des invariants induits par les opérateurs mathématiques employés au sein du modèle. LocalSolver est ainsi capable de réaliser près d'*un million de mouvements admissibles par minute*, assurant une convergence rapide vers des solutions de bonne qualité, même pour des modèles de grande taille. Pour plus de détails techniques, nous invitons le lecteur à consulter [3].

Comme tout algorithme de recherche locale, LocalSolver est conçu pour optimiser et non pour prouver. Nous n'insisterons jamais assez sur le fait qu'un bon modèle pour une résolution par recherche

locale (et donc par LocalSolver) doit être orientée "optimisation", laissant apparaître un très large espace de solutions admissibles. Pour cela, il est préférable de relâcher les contraintes dures en minimisant les écarts à celles-ci en priorité au sein de la fonction objectif (*goal programming*). LocalSolver a précisément été pensé pour optimiser un ensemble d'objectifs dans un ordre lexicographique (`minimize 100*x-y;` peut s'écrire simplement `minimize x; maximize y;`). Notons toutefois que les problèmes de satisfaction au sens strict du terme sont rarement rencontrés en entreprise. Une bonne modélisation doit au contraire garantir l'obtention d'une solution car un "*no solution found*" n'est généralement que de peu d'utilité pour les utilisateurs d'un logiciel d'aide à la décision.

3 LocalSolver 2.0

Convaincus de l'intérêt pratique de notre approche, ces premiers succès industriels nous ont permis d'envisager une commercialisation de la version 2.0 de LocalSolver. Cette commercialisation sera réalisée par Bouygues, en partenariat avec Aix-Marseille Université et le LIF. Ainsi, la prochaine version de LocalSolver, dont la sortie est prévue pour début 2012, restera gratuite pour la recherche et l'enseignement, mais deviendra payante pour toute utilisation commerciale. LocalSolver 2.0 sera composé d'un modeleur permettant le prototypage rapide d'applications d'optimisation ainsi que d'interfaces de programmation orientées objet (C++, Java, .NET) permettant une intégration totale de ces applications au sein des systèmes d'information. Les binaires du logiciel et du gestionnaire de licences seront fournis pour l'architecture x86 (i686) et les 3 systèmes d'exploitation Linux 2.6, Mac OS X 10.4 (Tiger), Windows XP. Ces binaires, la documentation du logiciel (guide de démarrage rapide, manuels de référence, tutoriels) et les conditions générales d'utilisation seront disponibles sur `www.localsolver.com`.

LocalSolver 2.0 comporte plusieurs nouveautés importantes, tant sur le plan fonctionnel que technique. Tout d'abord, un modeleur est fourni dans le nouveau "pack" LocalSolver. Le langage de modélisation mathématique proposé est pleinement adapté à une résolution par LocalSolver (il fournit de nombreux opérateurs mathématiques et facilite le *goal programming*). Mais au-delà, ce langage interprété intègre le meilleur des dernières technologies de script : un typage fort mais dynamique (comme Python), une déclaration implicite des variables (comme Lua ou PHP). De nombreuses fonctions peuvent être utilisées à la fois pour la modélisation et pour la programmation, facilitant ainsi la prise en main du langage. Par exemple, `c <- a * b` signifie que l'on déclare dans le modèle une expression c correspondant au produit des variables de modélisation a et b, alors que `c = a * b` signifie que l'on affecte à c le produit des variables de programmation a et b. Il en résulte des programmes moins verbeux et plus lisibles que ceux écrits avec les langages existants. Enfin, une singularité du langage est qu'il n'offre pas de fonction *main* : le flot d'exécution du programme est déterminé par l'appel de 5 fonctions principales *input*, *model*, *param*, *display*, *output*. Les fonctions ou variables intégrées au modeleur ont des valeurs par défaut (par exemple *param* qui permet de paramétrer la résolution du modèle par le solveur). Seule la fonction *model* doit être nécessairement programmée. L'idée est de réduire autant que possible les efforts de programmation du praticien. Voici un exemple de modèle LSP (*LocalSolver Programming*) permettant de résoudre le problème du sac-à-dos.

```
/* knapsack.lsp */

function input() {
  nbItems = 8;
  weights = {2, 20, 20, 30, 40, 30, 60, 10};
  values = {15, 100, 90, 60, 40, 15, 10, 1};
  sackBound = 102;
}

function model() {
  // 0-1 decisions
  x[0..nbItems-1] <- bool();

  // weight constraint
  sackWeight <-
    sum[i in 0..nbItems-1](weights[i] * x[i]);
  constraint sackWeight <= sackBound;

  // maximize value
  sackValue <-
    sum[i in 0..nbItems-1](values[i] * x[i]);
  maximize sackValue;
}

function param() {
  lsTimeLimit = 60;
  lsNbThreads = 4; // run 4 threads in parallel
}
```

Nous considérons qu'un modeleur est destiné au prototypage : il doit permettre d'aller vite par sa simplicité tout en cadrant le travail du praticien. Pour une utilisation avancée de LocalSolver (par exemple pour une intégration dans un système d'information), nous recommandons d'utiliser les API orientées objet du solveur (C++98, Java 5, .NET 2.0), qui n'en restent pas moins légères puisqu'exposant quelques classes seulement. Précisons une

fois encore que dans tous les cas, l'utilisateur n'a à fournir à LocalSolver que le modèle à résoudre ; en outre, il peut s'il le souhaite paramétrer la stratégie de recherche à l'aide de quelques "leviers" (nombre de *threads*, niveau de dégradation du recuit simulé, *etc.*).

Sur le plan technique, les performances de LocalSolver 2.0 ont été portées à un niveau encore supérieur, bien au-delà des capacités des meilleurs solveurs actuels en programmation 0-1 : *des modèles comportant plus de 10 millions de variables de décision peuvent être résolus en quelques minutes sur des ordinateurs standards.* Par exemple, LocalSolver 2.0 résout via un modèle LSP d'une centaine de lignes les instances A posées par Google dans le contexte du Challenge ROADEF 2012. Voici les solutions obtenues en 5 minutes pour ces instances (voir `http://e-lab.bouygues.com/?p=1179` pour plus de détails) :

instances	variables	binaires	contraintes	solutions
A1-1	6 020	400	503	44 306 501
A1-2	1 812 044	100 000	100 595	781 767 475
A1-3	1 423 438	100 000	26 097	583 006 134
A1-4	753 404	50 000	9 913	278 035 772
A1-5	229 213	12 000	13 905	727 578 312
A2-1	1 415 324	100 000	102 300	1 984 001
A2-2	3 769 381	100 000	19 770	1 268 279 367
A2-3	3 843 977	100 000	20 213	1 683 410 301
A2-4	1 537 771	50 000	13 373	2 035 401 379
A2-5	1 556 017	50 000	13 260	522 930 188

4 Et après ?

Le projet LocalSolver ne s'arrête pas là ! Des développements ambitieux sont envisagés à plus ou moins long terme. Un premier point concerne la *modélisation en nombres entiers* qui seule peut permettre de passer à des échelles encore supérieures. En effet, un gros défaut de la modélisation booléenne est qu'elle induit généralement un nombre quadratique de variables (si ce n'est plus). Ainsi, le modèle booléen utilisé pour résoudre le Challenge ROADEF 2012 ne tient plus en mémoire dès lors que l'on s'attaque à des instances de taille maximale (50 000 tâches à affecter à 5 000 processeurs, impliquant 250 millions de variables de décision 0-1). Une modélisation en nombres entiers permet d'obtenir un modèle dont la taille est linéaire en la taille de l'instance traitée. Nous disposons aujourd'hui d'un premier prototype permettant de réaliser une telle modélisation : celui-ci résout les instances de taille maximale du Challenge (avec 1 Go de mémoire vive allouée). Ces modèles "entiers" nous permettent d'attaquer des problèmes dont les modèles booléens équivalents contiendraient plus d'un milliard de variables 0-1. Nous travaillons encore à étendre ce formalisme de manière à pouvoir modéliser simplement et résoudre efficacement les problèmes d'ordonnancement et de routage. À plus long terme, nous souhaitons attaquer les *modèles en variables mixtes* (discrètes et continues) en nous appuyant sur la même méthodologie [2, 9]. Pour ce faire, nous envisageons une résolution des problèmes d'optimisation en variables continues par (pure) recherche locale.

Références

[1] E. Aarts, J.K. Lenstra (1997). *Local Search in Combinatorial Optimization.* John Wiley & Sons.

[2] T. Benoist, B. Estellon, F. Gardi, A. Jeanjean (2011). Randomized local search for real-life inventory routing. *Transport. Sci.* 45, pp. 381–398.

[3] T. Benoist, B. Estellon, F. Gardi, R. Megel, K. Nouioua (2011). LocalSolver 1.x : a black-box local-search solver for 0-1 programming. *4OR* 9, pp. 299–316.

[4] E. Danna, E. Rothberg, C. Le Pape (2005). Exploring relaxation induced neighborhoods to improve MIP solutions. *Math. Program. A* 102, pp. 71–90.

[5] G.B. Dantzig (1963). *Linear Programming and Extensions.* Princeton University Press.

[6] B. Estellon, F. Gardi, K. Nouioua (2008). Two local search approaches for solving real-life car sequencing problems. *Eur. J. Oper. Res.* 191, pp. 928–944.

[7] B. Estellon, F. Gardi, K. Nouioua (2009). High-performance local search for task scheduling with human resource allocation. In *SLS 2009*, *LNCS* 5752, pp. 1–15.

[8] M. Fischetti, A. Lodi (2003). Local branching. *Math. Program. B* 98, pp. 23–47.

[9] F. Gardi, K. Nouioua (2011). Local search for mixed-integer nonlinear optimization : a methodology and an application. In *EvoCOP 2011*, *LNCS 6622*, pp. 167–178.

[10] B. Selman, H. Kautz, B. Cohen (1996). Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability : 2nd DIMACS Implementation Challenge.* AMS.

[11] P. Van Hentenryck, L. Michel (2005). *Constraint-Based Local Search.* MIT Press.

[12] C. Voudouris, R. Dorne, D. Lesaint, A. Liret (2001). iOpt : a software toolkit for heuristic search methods. In *CP 2001*, *LNCS 2239*, pp. 716–730.

[13] J.P. Walser (1999). Integer optimization by local search : a domain-independent approach. *LNAI* 1637. Springer.