Vers une programmation par recherche locale : LocalSolver

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

[1] Bouygues e-lab, Paris, France. {tbenoist,fgardi}@bouygues.com

[2] Laboratoire d'Informatique Fondamentale – CNRS UMR 6166, Faculté des Sciences de Luminy – Université Aix-Marseille II, Marseille, France. {bertrand.estellon,karim.nouioua}@lif.univ-mrs.fr

Mots-Clés : *recherche locale, optimisation combinatoire, solveur/logiciel.*

En optimisation combinatoire, les techniques de recherche arborescente consiste en une exploration de l'espace des solutions par instantiation itérative des variables composant un vecteur solution. Leur efficacité en pratique repose sur leur capacité à élaguer l'arbre de recherche (de taille exponentielle dans le pire des cas). Fondée sur ces techniques, la programmation linéaire en nombres entiers (PLNE) est sans doute un des "outils" les plus puissants de la recherche opérationnelle. Pourtant limitée face à des problèmes fortement combinatoires et de grande taille, l'engouement qu'elle suscite s'explique par la simplicité d'utilisation des solveurs de PLNE : l'ingénieur modélise son problème sous la forme d'un PLNE, et le solveur le résout par *branch & bound (& cut)*.

Par opposition, les techniques de recherche locale consiste à appliquer de façon itérative des transformations à une solution dans le but d'en trouver de meilleures. Bien qu'incomplètes, ces techniques sont appréciées des chercheurs opérationnels car celles-ci permettent d'obtenir des solutions de qualité dans 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 sur la question de la couche algorithmique d'évaluation incrémentale des mouvements, qui reste entièrement à la charge de l'utilisateur. Deux logiciels font néanmoins état d'une avancée sur cette question : iOpt (British Telecommunications) et Comet (Van Hentenryck et Laurent). Malheureusement, le formalisme proposé dans les deux cas (inspiré de la programmation par contraintes) reste complexe et peu adapté selon nous au paradigme de la recherche locale.

Nous nous sommes engagés il y a deux ans dans un projet de recherche visant à définir un formalisme déclaratif rendant possible une "programmation par recherche locale", c'est-à-dire où l'utilisateur modélise son problème et le solveur le résout par recherche locale. Notre but à terme est d'obtenir un formalisme simple (accessible aux ingénieurs non informaticiens), générique (permettant d'aborder l'ensemble des problèmes d'optimisation combinatoire), efficace (adapté aux techniques algorithmiques de la recherche locale). Dans cette note, nous présentons le logiciel LocalSolver 1.0, première concrétisation de nos travaux sur le sujet. Disponible en licence BSD sur simple demande auprès des auteurs, cette version permet de traiter une classe restreinte, mais néanmoins importante, des problèmes d'optimisation combinatoire : les problèmes de *partitioning*, *packing*, *covering*. Le logiciel peut être utilisé de deux façons : comme un programme exécutable prenant en entrée le fichier dans

lequel est déclaré le problème à résoudre, ou bien comme une librairie C++ avec laquelle l'utilisateur peut programmer aisément un algorithme de recherche locale.

Le formalisme sur lequel repose LocalSolver 1.0 est de type fonctionnel, tout en restant proche des syntaxes usitées en programmation mathématique. Voici un petit problème artificiel de type *bin-packing* permettant d'appréhender celui-ci. Nous avons 3 objets x, y, z de taille 1, 2, 2 respectivement et 3 boîtes A, B, C sachant que C contient déjà un objet de taille 3. Notre but est de placer les objets dans les boîtes de façon à minimiser le produit des tailles de la plus petite et de la plus grande des boîtes si aucune n'est vide, et la taille de la plus grande sinon. Comme objectif second, nous souhaitons que les objets x et y apparaissent ensemble dans une même boîte, que x ou z soit placé dans C mais pas les deux, et que y n'apparaisse pas dans B.

```
xA <- bool(); yA <- bool(); zA <- bool();
xB <- bool(); yB <- bool(); zB <- bool();
xC <- bool(); yC <- bool(); zC <- bool();
constraint booleansum(xA, xB, xC) = 1;
constraint booleansum(yA, yB, yC) = 1;
constraint booleansum(zA, zB, zC) = 1;
display tailleA <- sum(1xA, 2yA, 2zA);
display tailleB <- sum(1xB, 2yB, 2zB);
display tailleC <- sum(1xC, 2yC, 2zC, 3);
tailleMin <- min(tailleA, tailleB, tailleC);
tailleMax <- max(tailleA, tailleB, tailleC);
o1 <- or(and(xA, yA), and(xB, yB), and(xC, yC));
o2 <- xor(xC, zC);
o3 <- not(yB);
minimize if(tailleMin > 0, product(tailleMin, tailleMax), tailleMax);
maximize booleansum(o1, o2, o3);
```

L'opérateur `bool` crée une variable booléenne : la variable `xA` est vraie si l'objet x est placé dans la boîte A. Le préfixe `constraint` permet de forcer une expression booléenne à vrai, ici les relations assurant que chaque objet est affecté à une et une seule boîte. Ensuite, l'opérateur `<-` est utilisé pour définir des variables intermédiaires : la taille de chaque boîte ainsi que les deux objectifs. L'ordre de définition des objectifs importe : il correspond à l'ordre lexicographique dans lequel ils seront optimisés. Le préfixe `display` permet d'afficher en console les valeurs prises par les variables `tailleA`, `tailleB`, `tailleC` durant la recherche locale.

Le problème ainsi déclaré est transformé par le logiciel en un graphe acyclique orienté (DAG), dont les noeuds racines sont les variables de décision et les noeuds feuilles les objectifs et les contraintes. L'évaluation d'un mouvement se fait par une propagation incrémentale dans ce DAG, en exploitant les invariants induits par les différents opérateurs arithmétiques (`min`, `max`, `sum`, `booleansum`, `product`) et logiques (`not`, `and`, `or`, `xor`, `if`). Ainsi, que ce soit en mode autonome, où l'heuristique et les mouvements sont créés automatiquement, ou bien en mode librairie, où ceux-ci sont implémentés par l'utilisateur par héritage de classes, l'évaluation est réalisée par le solveur. Notons que l'utilisateur peut s'il le désire implémenter ses propres opérateurs.

LocalSolver 1.0 a été testé avec succès sur une dizaine de problèmes académiques (*car sequencing*, *social golfer*, *steel mill slab design*, *spot 5*, *eternity*) et industriels (gestion des stocks de banches chez Bouygues Construction, planification des séminaires de formation chez Bouygues SA, Challenge ROADEF 2005 posé par RENAULT). Pour la majorité des instances traitées, LocalSolver permet d'obtenir en quelques secondes des solutions qu'un solveur par PLNE n'obtient pas après plusieurs heures de calcul. Comme preuve marquante de son efficacité, les résultats obtenus par LocalSolver nous aurait permis de figurer parmi les 18 finalistes du Challenge ROADEF 2005.