4OR manuscript No.
(will be inserted by the editor)

LocalSolver 1.x

A black-box local-search solver for 0-1 programming

Thierry Benoist · Bertrand Estellon · Frédéric Gardi · Romain Megel · Karim Nouioua

Received: date / Accepted: date

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 defer its actual resolution to a solver based on efficient and reliable local-search techniques. Started in 2007, the goal of the LocalSolver project is to offer a model-and-run approach to combinatorial optimization problems which are out of reach of existing black-box tree-search solvers (integer or constraint programming). Having outlined the modeling formalism and the main technical features behind LocalSolver, its effectiveness is demonstrated through an extensive computational study. The version 1.1 of LocalSolver can be freely downloaded at `http://www.localsolver.com` and used for educational, research, or commercial purposes.

Keywords Combinatorial optimization · 0-1 programming · Local search · Black-box solver · OR software

Mathematics Subject Classification (2000) 90C27 · 90C10 · 90C90 · 90B90

1 Introduction

In combinatorial optimization, the tree-search techniques consist in exploring the solution space by iteratively instantiating variables composing a solution vector. Their practical efficiency relies on their ability to prune the tree search, which has an exponential size in the worst case. Founded on these techniques, Integer Programming

The work of B. Estellon and K. Nouioua was supported in part by the ANR grant OPTICOMB (ANR BLAN06-1-138894). T. Benoist, F. Gardi, R. Megel extend special thanks to Dr. Etienne Gaudin, director of Bouygues e-lab, for his support and encouragements.

T. Benoist, F. Gardi, R. Megel
Bouygues e-lab, Paris, France
E-mail: {tbenoist, fgardi, rmegel}@bouygues.com

B. Estellon, K. Nouioua
Laboratoire d'Informatique Fondamentale – CNRS UMR 6166,
Université Aix-Marseille II – Faculté des Sciences de Luminy, Marseille, France
E-mail: {bertrand.estellon, karim.nouioua}@lif.univ-mrs.fr

(IP) is surely one of the most powerful tools of operations research. Although limited when faced with large-scale combinatorial problems, its success among practitioners is mainly due to the simplicity of use of IP solvers: the engineer models its problem as an integer program and the solver solves it by branch & bound (& cut). Following this observation, a recent trend in Constraint Programming (CP) aims to promote the design of effective autonomous CP solvers. Indeed, this "model-and-run" approach, when effective, reduces considerably the development and maintenance efforts of optimization softwares.

In contrast, Local Search (LS) consists in applying iteratively some changes (called moves) to a solution so as to improve the objective function. Although incomplete, this technique is widely appreciated because it allows operations researchers to obtain good-quality solutions in short running times (of the order of the minute). However, designing and implementing local-search algorithms is not straightforward. The algorithmic layer dedicated to the evaluation of moves is particularly difficult to engineer, because it requires both an expertise in algorithms and a dexterity in computer programming. For a survey on the LS paradigm and its applications, the reader is invited to consult the book by Aarts and Lenstra (1997).

This paper introduces LocalSolver 1.x, a black-box local-search solver for general 0-1 programming (with nonlinear constraints and objectives). This software allows OR practitioners to focus on the modeling of the problem using a simple formalism, and then to defer its actual resolution to a solver based on efficient and reliable local-search techniques. Started in 2007, the goal of the LocalSolver project is to offer a model-and-run approach to combinatorial optimization problems which are out of reach of existing IP/CP autonomous solvers. The current version (LocalSolver 1.1) is especially designed for tackling matching, partitioning, packing, covering problems. Distributed freely under a BSD licence[1], the binaries of the software are available for the architecture x86 and three operating systems Linux 2.6, Mac OS X 10.5 (Leopard), Windows XP. The software can be used for educational, research or even commercial purposes without permission from the authors.

The paper is organized as follows. After a review of related work in the literature, the modeling formalism associated with LocalSolver 1.x is presented. Then, the solver is presented and the main ideas on which it relies are outlined. In order to demonstrate the effectiveness of our solver, the results of an extensive computational study realized with a dozen of academic and industrial benchmarks are outlined.

2 Prior works and contributions

A local-search heuristic is designed according to three layers (Estellon et al 2009): search strategy, moves, evaluation machinery. Our past experiences in engineering high-performance local-search algorithms (Estellon et al 2006, 2008, 2009; Benoist et al 2009a) have convinced us that neglecting one of these three layers may yield a significant decrease in terms of performance. Then, designing and implementing local-search heuristics is a complex, time-consuming task for OR practitioners.

Most proposals made to offer tools or reusable components for local-search programmers take the form of a framework handling the top layer of the algorithm, namely metaheuristics (see for example Cahon et al (2004); Di Gaspero and Schaerf (2003)). In

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

this case, moves and associated incremental algorithms are implemented by the user, while the framework is responsible for applying the selected parameterized metaheuristic. However, designing moves and implementing incremental evaluation algorithms represent the largest part of the work (and of the resulting source code); from our observations, these two layers consume respectively 30 % and 60 % of the development times. Hence, these frameworks do not address the hardest issues of the engineering of local-search algorithms. Two softwares aim at answering to these needs: Comet Constraint-Based Local Search (CBLS) (Van Hentenryck and Michel 2005) (and its ancestor Localizer (Michel and Van Hentenryck 2000)) and iOpt (Voudouris et al 2001). These softwares allow an automatic evaluation of moves, but the implementation of these moves remains the responsibility of the user. Then, to the best of our knowledge, no effective black-box local-search solver is available today for tackling large-scale real-life combinatorial optimization problems, as known in IP. Van Hentenryck and Michel (2007) have recently described a synthesizer of local-search heuristics from high-level models, but this feature is not yet available in Comet (Deville and Schaus 2010); a generic swap-based tabu search procedure (Comet Tutorial 2010, pp. 330–331) is available in Comet CBLS 2.1, which can be used as black box for tackling integer models. Note also that some of the best solvers for Satisfiability Testing (SAT) or Pseudo-Boolean Programming (PB) rely on stochastic local search (see for example Walksat (Selman et al 1996) and WSAT(OIP) (Walser et al 1998)), but these solvers are not suited for real-life combinatorial optimization and thus rarely used by OR practitioners.

Our approach to autonomous LS is guided by the following fundamental principle: *the LS solver must work as a LS practitioner works*. This implies a major difference compared to the above frameworks or solvers: LocalSolver performs structured moves tending to maintain the feasibility of solutions at each iteration, whose evaluation is accelerated by exploiting invariants induced by the structure of the model. Then, the main specificities of LocalSolver 1.x are to provide: a simple mathematical formalism to model the problem in an appropriate way for LS resolution, and an effective black-box LS-based solver focused on the feasibility and the efficiency of moves.

3 Modeling formalism

LocalSolver's modeling formalism (named LSP for "Local Search Programming") is close to classical mathematical programming formalisms like 0-1 integer programming or pseudo-boolean programming but enriched with common mathematical operators, making it easy to understand by OR practitioners. In the LSP format, a program consists of: decision variables, intermediate variables, constraints and objectives. As an example, here is described an artificial toy problem which can be classified as a bin-packing problem. We have 3 items x, y, z of height 2, 3, 4 respectively to pack into 2 piles A, B knowing that B already contains an item of height 5. The goal is to minimize the height of the largest 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;
heightA <- sum(2xA, 3yA, 4zA);
heightB <- sum(2xB, 3yB, 4zB, 5);
objective <- max(heightA, heightB);
```

```
minimize objective;
```

The statement `bool()` creates a boolean decision variable. For instance, the variable `xA` is true when item x is assigned to pile A. In this 1.x version, only boolean decision variables are allowed. Then, the operator `<-` is used to define intermediate variables (for example, the height of each pile), which can be boolean or integer. The keyword `constraint` prefixes each constraint definition; here the three constraints ensure that each item is assigned to exactly one pile. In the same way, the keyword `minimize` prefixes the objective of the program.

Formally, the BNF syntax of a program is:

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

where $<$ *expression* $>$ shall be detailed in the following section. Then, below are described the different kind of lines. The ordering of lines in the program is free, except when defining lexicographic objective functions.

3.1 Decision and intermediate variables

All decision variables must be declared somewhere in the program. It is done with operators `bool()`, introducing boolean variables. Boolean variables are treated as integers, with the convention false=0 and true=1. We insist on the fact that so far only boolean variables are allowed as decision variables.

Expressions can be built upon these variables by using the native logical, arithmetic, or relational operators:

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

where $<$ *scalar* $>$ is a number and $<$ *identifier* $>$ a variable name.

In summary, LocalSolver uses a functional syntax (only comparators are infixed), with no limitation on the nesting of expressions. Intermediate variables can be introduced as well with operator `<-`, either to improve the readability of the model or to reuse expressions on different lines. Some operators apply only to a certain number of arguments or to a certain type of variables (see Table 1). For instance, the `not` operator takes only one argument whose type must be boolean. The `if` operator takes exactly three arguments, the first one being necessarily boolean: `if(condition, value_if_true, value_if_false)`. Operators are strongly typed, which explains the

Table 1 Mathematical operators available in LocalSolver 1.1.

operators	arity	input	output
`bool`	0	-	boolean
`and`	n	boolean	boolean
`or`	n	boolean	boolean
`xor`	n	boolean	boolean
`not`	1	boolean	boolean
`if`	3	mixed	mixed
`sum`	n	integer	integer
`booleansum`	n	boolean	integer
`min`	n	integer	integer
`max`	n	integer	integer
`product`	n	integer	integer
`square`	1	integer	integer
`divide`	2	integer	integer
`modulo`	2	integer	integer
`abs`	1	integer	integer
`distance`	2	integer	integer
`=`	2	integer	boolean
`<=`	2	integer	boolean
`>=`	2	integer	boolean
`<`	2	integer	boolean
`>`	2	integer	boolean
`!=`	2	integer	boolean

definition of `sum` and `booleansum`. On the other hand, since boolean expressions are actually 0/1 variables, they can be used in all integer operators.

Introducing logical, arithmetic, or relational operators has two important benefits in a local-search context: expressiveness and efficiency. With such low-level operators, modeling is easier than with basic IP syntax, while remaining quickly assimilable by practitioners. Besides, the invariants induced by these operators can be exploited by the internal algorithms of the LS solver to speed up local search.

3.2 Constraints and objectives

Any boolean expression can be made into a constraint by prefixing the line by `constraint`. An instantiation of decision variables is valid if and only if all constraints take value 1, coding for satisfied. When modeling a problem, practitioners should remember that local search is not suited for solving severely constrained problems: if some business constraints are not likely to be satisfied, it is recommended to define them in the objective function (as soft constraints) rather than as hard constraints. Moreover, LocalSolver offers a feature making this easy to do: lexicographic objectives.

At least one objective must be defined, using the modifier `minimize` or `maximize`. Any expression can be used as objective. If several objectives are defined, they are interpreted as a lexicographic objective function. The lexicographic ordering is induced by the order in which objectives are declared. For instance in car sequencing with paint colors, when the goal is to minimize violations on ratio constraints and, as a second criterion, the number of paint color changes, the objective function can be directly specified as: `minimize ratio_violations; minimize color_changes;`. This features allows avoiding the classical modeling workaround where a big coefficient is used to sim-

ulate the lexicographic order: `minimize 1000 ratio_violations + color_changes;`. The number of objectives is not limited and can have different directions (minimization or maximization).

4 Autonomous local search

The command line for solving the above toy problem, granting 1 second of running time to LocalSolver 1.1 is:

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

Then, the printout on standard output should look like this:

```
Parsing LSP file toy.lsp...
25 nodes, 6 booleans
3 constraints, 1 objectives
1 phases, 2 threads
strategy : descent
***** Initial feasible solution : obj = ( 9 )
***** Solve phase 1 during 1 sec and 4294967295 itr
* Thread 1 : obj = ( 7 ) in 0 sec, 3 itr
* Thread 2 : obj = ( 7 ) in 0 sec, 3 itr
*** After 1 sec, 260000 itr : best obj = ( 7 ) in 0 sec, 3 itr
Writing solution in file toy.sol...
```

By default, LocalSolver 1.1 uses a standard descent (Aarts and Lenstra 1997) as search strategy, with all available autonomous moves. A simulated annealing heuristic (Aarts and Lenstra 1997) is also available by specifying some options in command line. The chosen heuristic can be multithreaded (by default, two threads are launched). It consists in running several resolution instances with different seeds in parallel, synchronizing the best results regularly, and returning the best solution found when the time limit is reached. Multithreading shall not be seen as a way to speed up the search but rather as a way to increase the robustness of the solver. Autonomous moves are randomly chosen with a non uniform distribution; this distribution is dynamically tuned during the search in function of their accepting and improving rates.

The cost of the initial feasible solution found by LocalSolver on this example is 9. This solution is found by a basic randomized greedy algorithm. As explained above, LocalSolver is not designed for solving hardly-constrained optimization problems. Thus, if no initial feasible solution is found by this greedy algorithm, then the user should consider turning one of the constraints into a first-level objective function. By the way, this is a fundamental difference with CBLS approaches where a violation measure is defined for each constraint. We believe that such relaxations are the responsibility of the user. Typically, for frequency assignment problems, the practitioner can chose between assigning a frequency to each link while minimizing interferences or ensuring zero interferences while minimizing the number of unassigned links.

The best solution, found after 3 iterations (and 0 second), has cost 7. During the second of allocated time, LocalSolver has performed 260 000 iterations in each thread, which corresponds to the number of moves attempted and also to the number

of solutions visited during the search. Ultimately, it creates the file "toy.sol" with the solution: `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;
```

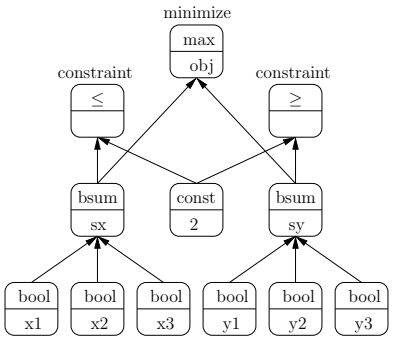

Fig. 1 The directed acyclic graph (DAG) induced by a simple model. For each node, the type (resp. name) of the node is given above (resp. below). Here "bsum" stands for `booleansum`.

A LSP program, as defined above, can be represented through a directed acyclic graph (DAG), whose roots are the decisions variables and whose leaves are the constraints and objectives (see Fig. 1). Then, the operators used to model the problem induce the inner nodes of the DAG. These inner nodes are related to "invariants" or "one-way constraints" in softwares like iOpt (Voudouris et al 2001) or Comet (Van Hentenryck and Michel 2005) (or its ancestor Localizer (Michel and Van Hentenryck 2000)). With this representation, a solution is a complete instantiation of the root variables. Applying moves to the current solution consists in modifying the current values of the decision variables (roots) and evaluating constraints and objectives (leaves) by propagating these modifications along the DAG.

Following the methodology of Estellon et al (2009), LocalSolver is composed of three layers: search strategy, moves, evaluation machinery. The design and implementation of LocalSolver have required a considerable effort in terms of software and algorithm engineering to reach high performances, which cannot be entirely detailed here. Hence, our presentation will be focused on the two crucial aspects of the solver: the autonomous moves and the incremental evaluation machinery.

4.1 Autonomous moves

As suggested in introduction, our ultimate goal is to autonomously perform the moves that a practitioner would have designed to solve its problem. The simplest possible move is the *k-Flips* which randomly flips the value of k (binary) decision variables. However, the structure of the model often allows to design more appropriate moves. For instance, when a constraint is set on a sum of boolean variables, a natural move consists in flipping two booleans of the sum in opposite directions, thus preserving the value of this sum. We will show in this section that LocalSolver is also able to exploit more complex patterns, applying autonomous moves that can be viewed as ejection chains applied to the hypergraph induced by boolean variables and constraints (see

Rego and Glover (2002) for more details on ejection chains). These ejection chains are specialized for maintaining the feasibility of boolean constraints and are a key component of the effectiveness of LocalSolver 1.x. For example, let us consider the car sequencing problem (Estellon et al 2006, 2008): cars must be ordered in the production line so as to minimize a non linear objective. This problem can be modeled as an assignment problem by defining for each car i and position p a boolean variable $x_{i,p}$. A basic neighborhood for this model consists in exchanging the positions of two vehicles. In terms of variables, exchanging the positions p and q of two cars i and j corresponds to flipping the 4 boolean variables $x_{i,p}$, $x_{i,q}$, $x_{j,q}$, $x_{j,p}$ which preserves the feasibility of the 4 partition constraints where these variables appear. In a generic way, our autonomous moves are equivalent to k-moves and k-swaps on packing/covering problems.

Define a *constrained sum* as a sum involving at least two binary decision variables either directly or multiplied by a scalar, whose value is constrained by a relational operator. A data structure is built listing all constrained sums in the DAG and for each binary decision variable, the list of constrained sums it belongs to. Besides, we maintain for each constrained sum the set of *increasing booleans*, namely decision variables whose change would increase the sum, and the complementary of this set (that is, the *decreasing booleans*). Using this structure, we can perform moves trying to find an alternating path of increasing and decreasing booleans such that two consecutive variables are involved in the same constrained sum. To obtain an alternating cycle, as in the above example, we can also enforce the same properties for the last and first variables in the path. The key idea of such moves, called *k-Paths* or *k-Cycles*, is to iteratively repair modified sums, by applying an opposite change at each step. By maintaining the feasibility of constrained sums, k-Paths and k-Cycles tend to maintain the feasibility of the solution, which is crucial for making the search effective. For instance, when the constrained sums define a complete matching problem, any k-Cycle with k even will be completed (that is, closed without failing) in $O(k)$ time.

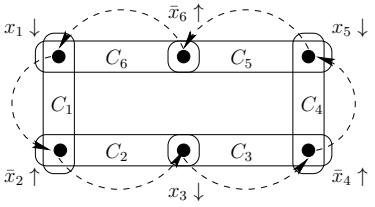

Fig. 2 A 6-Cycle involving six boolean variables x_1, x_3, x_5 (whose current value is 1) and $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ (whose current value is 0), and six constrained sums $C_1, \ldots, C_6$. Each variable belongs to two sums (for example, x_1 belongs to C_1 and C_6). Now, x_1, x_3, x_5 are decreased ($\downarrow$) while $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ are increased ($\uparrow$). This move preserves the values of the sums, and thus the feasibility of the constraints.

Changing the definition of constrained sums leads to variants which are of interest for practically speeding up the convergence of the local search. For instance, for the selection of the next sum to be repaired, we may also favor sums on which an equality constraint is set because the move cannot succeed without repairing these sums. Another variant consists in flipping more than one variable per constraint. This extension follows the same logic as the generalization from ejection chains to ejection trees

(Caseau et al 1999). For instance, it allows ejecting two objects of size 1 when adding an object of size 2 in a set, in packing problems.

At our knowledge, the design of such autonomous moves, specialized for maintaining the feasibility of the solution during the move, is novel. They form the key component of LocalSolver as black box. Indeed, they largely improve the effectiveness of the search (that is, the convergence toward high-quality solutions) on large-scale structured combinatorial problems (as frequently encountered in OR applications), relatively to the classical k-Flips neighborhood search employed in SAT/PB solvers (Selman et al 1996; Walser et al 1998). Note that we have recently been informed that specific autonomous moves are used in IBM ILOG Transportation PowerOps (TPO) software for solving vehicle routing problems by local search in a model-and-run fashion (Fernandez Pons 2010).

4.2 Incremental evaluation machinery

The first machinery for incremental evaluation was introduced in Localizer (Michel and Van Hentenryck 2000), the ancestor of Comet (Van Hentenryck and Michel 2005), and iOpt (Voudouris et al 2001). It is based on the exploitation of invariants induced by combinatorial operators. Although we claim no novelty for this mechanism, it is presented here for the sake of completeness, and the specificities of our implementation are illustrated at the end of the section.

Each node of the DAG implements the following methods: `init`, `eval`, `commit`, `rollback`. The method `init` is responsible of the initialization of the value of the node according to (the values of) its parents, before starting local search. The specific data structures attached to the node, used for speeding up its incremental evaluation, are also initialized by this method. Having applied a move on decision variables, the `eval` method is called for incrementally reevaluated the value of a node, when this one is impacted during the DAG propagation. Then, if the move is accepted by the heuristic, the `commit` method is called on each modified node for validating the changes implied by the move. Otherwise, the move is rejected, and the `rollback` method is used instead.

As mentioned above, the fast evaluation of moves is obtained by exploiting the invariants induced by each type of nodes (that is, operators) during the propagation (Michel and Van Hentenryck 2000). A breadth-first search propagation of the modifications is performed along the DAG, guarantying that each node is evaluated at most once. Following a classical observer pattern, the propagation is reduced to impacted nodes: a node is said to be impacted if some of its parents have been modified. For example, consider the node $z \leftarrow a < b$ with a current value equals to true. This one will not be impacted if a is decreased or b increased. Then, to each node is associated an `eval` method called for computing the new value of the node when impacted. This method takes in input the list of modified parents (that is, the parent nodes whose current value has changed). For a linear operator like `sum`, evaluation is easy: if k terms of the sum are modified, then its new value is computed in $O(k)$ time. But for other operators (arithmetic or logical), significant accelerations can be obtained in practice. For example, consider the node $z \leftarrow \text{or}(a_1, \ldots, a_k)$ with M the list of modified a_i's and T the list of a_i's whose current value is true. Thus, one can observe that if $|M| \neq |T|$, then the new value of z is necessarily true, leading to a constant-time evaluation. Indeed, if $|M| < |T|$, then at least one parent remains with value equals to true; otherwise, there exists at least one parent whose value is modified from false to true.

Here our implementation is focused on the practical, experimental complexity, and not only on the worst-case complexity. Constant factors do matter: fine algorithmic and code optimizations improve speed of evaluation by several orders of magnitude. For instance, the property mentioned previously for maintaining the `or` operator is, at our acquaintance, not employed in Comet or iOpt systems. In the same way, in Localizer, Michel and Van Hentenryck (2000, p. 67) maintains the `min` operators in $O(\log k)$ time with k the number of operands using classically a binary heap. In LocalSolver, we distinguish two cases. If the minimum value among the modified operands is lower than or equal to the current value of the `min` operator, or if one support remains unmodified, then the evaluation is optimally done in $O(|M|)$ time with $|M|$ the number of modified values. Otherwise, the evaluation is performed in $O(k)$ time. In practice, the former case is by far the most frequent and the number of modified operands is small ($|M| = O(1)$), ensuring an amortized constant-time evaluation.

5 Experimental results

LocalSolver was tested on a benchmark mixing academic and industrial problems, determined before starting the project. We insist on the fact that our purpose is not to achieve state-of-the-art results for all these problems. *The main goal of LocalSolver is to obtain, as black box, good-quality solutions with short running times (as it can be done with standard local-search heuristics), in particular when tree-search solvers fail to find any solution.*

The goal of these experiments is to compare LocalSolver to existing black-box solvers: IBM ILOG CPLEX (the state-of-the-art IP solver) and Comet CBLS (Comet Tutorial 2010, pp. 330-331) which, although not primarily designed to serve as a black box, offers a generic swap-based tabu search. IBM ILOG CP Optimizer (CPO) was tested as well but did not yield competitive results on these problems. The results of SAT or PB solvers, inappropriate to tackle such structured optimization problems, are also omitted.

For each problem of this benchmark, the solvers were compared on the same machine, with the same standard model. All numerical experimentations were performed on a standard computer equipped with the operating system Windows XP 32 bits and the chip Intel Core 2 Duo T7600 (2.33 GHz, RAM 2 GB, L2 4 MB, L1 64 kB). Note that only two cores are available on this computer. As for the model, it is merely adapted to the grammar of each solver. Thus, LSP and IP models are written using boolean decision variables, while CP and CBLS models are written using integer decision variables with the available global constraints. For instance, the *max* operator is native in Comet and LocalSolver but shall be expressed through inequalities in the MIP equivalent model. Note that no specific element was added beyond these necessary transformations (like IP valid inequalities). The efficiency of an autonomous solver is a combination of several factors including the search strategy (for finding a feasible solution, for optimizing a feasible solution), the bounding and cutting techniques (for IP solvers), the filtering techniques (for CP solvers), the moves and the incremental evaluation machinery (for LS solvers), and especially the internal algorithms and their implementation details. Here each solver is launched with its default parameter settings, unless explicit mention on the contrary. In particular, no initial solution is provided to LocalSolver or Comet, and no search strategy or moves are specified: these choices are the responsibility of the black-box solvers.

Each problem addressed in this benchmark is briefly described and the chosen model is cited or sketched out. Results obtained by the different solvers are reported on a representative set of instances and the state-of-the-art results found in the literature (generally obtained by local-search heuristics) are given as a baseline. All results presented here have been rigorously validated; in particular, having extracted the business solution from the mathematical one, all constraints and objective costs have been checked. All the material used for the benchmark (code, models, results) is available upon request from the corresponding author.

In all tables below, the line "LocalSolver 1.1" corresponds to the results obtained by LocalSolver 1.1, the line "CPLEX 12.2" corresponds to the results obtained by IBM ILOG CPLEX 12.2, the line "CPO 2.3" corresponds to the results obtained by IBM ILOG CPO 2.3, and the line "Comet CBLS 2.1" corresponds to the results obtained by the generic CBLS in Comet 2.1. The origin of the "State of the art" line will be detailed for each problem.

5.1 Car sequencing

The car sequencing problem (Hnich et al 2009) consists in ordering cars on an assembly line while minimizing violations on ratio constraints. In LocalSolver and CPLEX, the assignment of cars to positions is modeled with boolean variables and the violations on each ratio constraint are summed (see Estellon et al (2006)). In Comet, we use integer (and not boolean) decision variables. Besides, a "sequence" constraint is available in this language for modeling precisely the car sequencing violations. This global constraint was necessary to obtain the results given below (the performance of Comet is more than twice worse on large instances otherwise).

Sample results are presented for 5 instances on Table 2 below: 10-93 (100 vehicles, 5 options, 25 classes), 200-01 (200 vehicles, 5 options, 25 classes), 300-01 (300 vehicles, 5 options, 25 classes), 400-01 (400 vehicles, 5 options, 25 classes), 500-08 (500 vehicles, 8 options, 20 classes). The first 4 instances are available in CSPLib (Hnich et al 2009); the fifth comes from a benchmark generated by Perron et al (2004). The line "state-of-the-art" corresponds to the state-of-the-art results, here obtained by the high-performance local-search algorithm described in Estellon et al (2006). The results presented in the top (resp. bottom) table have been obtained with a time limit fixed to 60 (resp. 600) seconds. The cost of the best solution found is given (the symbol "x" is used if no solution has been obtained within the time limit). In summary, one can observe that LocalSolver outperforms IP and Comet solvers, especially as the scale of instances grows (instances with 400 and 500 vehicles induce more than 10 000 boolean decision variables). The results of CP solvers are not detailed, because not competitive for tackling this problem: Perron and Shaw (2004); Perron et al (2004) obtain by Large Neighborhood Search (LNS) a number of violations greater than 500 on instance 500-08.

A real-world version integrating the constraints and objectives of the paint workshop was proposed by the car manufacturer Renault as subject of the ROADEF 2005 Challenge (Estellon et al 2008), which is an OR competition yearly organized by the French Operations Research Society. Three lexicographic objectives have to be optimized in this version: EP = violations on high priority ratio constraints, ENP = violations on low priority ratio constraints, RAF = violations on paint color changes. Table 3 contains sample results for 3 instances: X2 = 023-EP-RAF-ENP-S49-J2 (1260

Table 2 Sample results for the academic car sequencing problem (minimization).

time limit: 60 s	10-93	200-01	300-01	400-01	500-08
LocalSolver 1.1	10	**7**	**11**	**13**	**46**
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
LocalSolver 1.1	6	**3**	**6**	**10**	**20**
CPLEX 12.2	**3**	**3**	11	16	104
Comet CBLS 2.1	7	6	10	18	47
State of the art	*3*	*0*	*0*	*1*	*0*

Table 3 Sample results for the Renault's car sequencing problem (minimization).

time limit: 600 s	X2			X3			X4		
State of the art	*0,*	*192,*	*66*	*0,*	*337,*	*6*	*0,*	*160,*	*407*
LocalSolver 1.1	**0,**	**289,**	**68**	**30,**	**452,**	**22**	**4,**	**244,**	**676**
Comet CBLS 2.1 (relaxed)	799,	1069,	481	1447,	1100,	309	1055,	1888,	651

vehicles, 12 options, 13 colors), X3 = 024-EP-RAF-ENP-S49-J2 (1319 vehicles, 18 options, 15 colors), X4 = 025-EP-ENP-RAF-S49-J1 (996 vehicles, 20 options, 20 colors). No IP/CP/SAT solver is able to tackle such instances today (Estellon et al 2008). For example, CPLEX is not able to find an integer solution after several hours of computing time. Comet is not able to find solutions too: the line "Comet CBLS 2.1 (relaxed)" gives the results obtained with models where paint limit constraints are omitted. Here the state of the art corresponds to the local-search heuristic which won the challenge (Estellon et al 2008); note that the design and the implementation of this algorithm required nearly 150 working days to its authors. For instance X2, the resulting LS program contains 516 936 variables whose 374 596 are binary decision variables (450 MB of RAM are allocated per thread during the execution). In both modes, Localsolver performs between that 1.5 and 4.5 million moves per minute, with an acceptation rate between 5 and 20 % and nearly a thousand improving solutions. Observe that LocalSolver's results are comparable to the hand-made variable neighborhood search by Prandtstetter and Raidl (2008) mixing classical moves and large neighborhood search by IP; according to its results, LocalSolver would have been ranked among the finalists of the 2005 ROADEF Challenge.

5.2 Social golfer

The social golfer problem (Hnich et al 2009) consists in assigning persons to groups over several weeks so as to maximize the number of meetings. Having modeled the partitioning structure for each week, the calculation of meetings between golfers is straightforward in each of the considered solvers. A real-life version is encountered at Bouygues SA for scheduling the managers' seminars. Sample results are presented on Table 4. Four classical instances are addressed (2 easy ones and 2 hard ones): 9-3-11 (9 groups of 3 players for 11 weeks), 10-10-3, 10-9-4, 10-3-13. In this case, the objective is to minimize the number of duplicate meetings. The fifth one, named "seminar", is a real-life instance (120 persons over 3 weeks with group sizes between 7 and 9) with additional constraints on groups and three lexicographic objectives: balancing some characters into each group (for example, men and women), avoiding the undesired

Table 4 Sample results for the social golfer problem (minimization).

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

meetings, maximizing the number of (desired) meetings. For classical instances, the tabu-search heuristic by Dotú and Van Hentenryck (1999) (implemented in C programming language) currently owns the best results on almost all social golfer benchmarks. Some dedicated and complex CP approaches (with an emphasis on breaking symmetries) obtain similar results (Dotú and Van Hentenryck 1999). For the real-life instance, the state of the art corresponds to the local-search algorithm which was implemented by one of the authors as operational solution: a first-improvement descent performing fast random swaps (one million per second). Due to the quadratic form of the objective function (a logical "and" must be used for counting meetings), such a problem is particularly difficult to tackle by IP techniques; indeed, its linearization induces a very large number of variables (hundreds of thousands). Then, despite a running time 10 times higher, IP solvers are not able to provide good solutions. Using an integer model and "atmost" global constraints to partition golfers, Comet CBLS obtains comparable solutions with longer running times, but it is not able to find admissible solutions for real-life instances. As previously, LocalSolver does not achieve the records for the hardest instances, but is close to.

5.3 Steel mill slab design

The steel mill slab design problem (Hnich et al 2009) is a variant of the celebrated cutting-stock problem, where orders of different sizes have to be packed onto slabs of different capacities such that the total slab capacity is minimized. In addition, two orders having the same color cannot be packed together into the a same slab (mutual exclusion constraints). The model is based on the assignment of orders to slabs, and the size of each slab is determined by its contents (Schaus et al 2011). The classical instance of CSPLib with 111 orders (Hnich et al 2009) is solved to optimality (cost equal to 0) in less than 1 second by LocalSolver, which outperforms most of the previous approaches (see Van Hentenryck and Michel (2008) for a state of the art). The LSP treated by LocalSolver in this case contains 40 739 variables with 12 321 booleans; LocalSolver visits nearly 100 000 solutions per second. Sample results obtained on new instances proposed by Schaus[2] are given on Table 5. The 200 instances numbered from 11-0 to 20-19 are not mentioned, because all are solved to optimality in less than one second. State-of-the-art results are obtained by Heinz[2], cleverly using a Dantzig-Wolfe decomposition of the problem which can be directly tackled by an IP solver due to the reasonable number of columns (heavily filtered thanks to the numerous mutual

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

Table 5 Sample results for the steel mill slab design problem (minimization).

time limit: 60 s	2-0	3-0	4-0	5-0	6-0	7-0	8-0	9-0	10-0
LocalSolver 1.1	**37**	**8**	**35**	**1**	**4**	**1**	**0**	**0**	**0**
CPLEX 12.2	136	288	x	126	x	232	226	163	133
CPO 2.3	90	65	58	50	54	46	28	29	20
Comet CBLS 2.1	136	135	69	65	42	30	26	21	20
time limit: 600 s	2-0	3-0	4-0	5-0	6-0	7-0	8-0	9-0	10-0
LocalSolver 1.1	**31**	**7**	**34**	**0**	**4**	**0**	**0**	**0**	**0**
CPLEX 12.2	94	65	x	63	x	189	226	97	64
CPO 2.3	62	38	40	42	36	36	21	23	18
Comet CBLS 2.1	124	110	43	58	33	33	17	17	15
State of the art	*22*	*5*	*32*	*0*	*0*	*0*	*0*	*0*	*0*

Table 6 Sample results for the Spot 5 daily photographs scheduling problem (maximization).

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	**70**	20 112	19 116	167 068	167 156
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	165 185	164 253	**61 158**	124 241	152 262
CPLEX 12.2	**174 181**	**176 237**	**61 158**	**124 243**	**168 246**

exclusion constraints). Note that this approach is dedicated to CSPLib instances: relaxing mutual exclusion constraints makes the number of columns greater, imposing a branch-and-price approach. LocalSolver keeps reasonably close to optimal solutions, whereas CPLEX and generic CBLS are clearly outperformed. Similarly, the best CP approaches are not competitive (Schaus et al 2011).

5.4 Spot 5 photographs scheduling

The spot 5 daily photograph scheduling problem (Vasquez and Hao 2001) consists in selecting the subset of photos to be shot by the Spot 5 satellite; the goal is to maximize a profit function subject to knapsack constraints and mutual exclusion constraints. This linear 0-1 model reads the same in CPLEX and LocalSolver. The larger instances addressed in the literature (multi-orbit case) contains at most one thousand photos in input, which makes them efficiently tractable by IP solvers today. Sample results are presented on Table 6. The state of the art corresponds to Vasquez-Hao's tabu heuristic (Vasquez and Hao 2001); moreover, these authors have proven that their results are nearly optimal (gap lower than 1 %). The main lesson of this experiment is that LocalSolver remains competitive with IP solvers when the scale of instances, smaller, becomes favorable to tree-search techniques.

5.5 Minimum formwork stock

The minimum formwork stock problem (Benoist et al 2009b), encountered at Bouygues Construction, aims at minimizing the shuttering material used on a construction site. Once decomposed, the master problem can be viewed as a covering problem, whose

Table 7 Sample results on the minimum formwork stock (minimization).

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**

scale makes it efficiently tractable by IP solvers. Sample results are presented on Table 7. As for the previous problem, one can observe that LocalSolver remains competitive in the face of CPLEX.

5.6 Eternity II puzzle

The Eternity II problem (Benoist and Bourreau 2008; Schaus and Deville 2008) is a very challenging (and recreative) edge-matching puzzle edited by the Tomy company in 2007. The puzzle consists in filling a 16×16 square board with 256 square tiles, the four sides of each tile being colored. The goal is to find an assignment of tiles to the board such that the sides of every adjacent pair of tiles have the same color. Such a problem can be modeled as an optimization problem: assign all tiles to the board while minimizing the number of pairs of tiles violating the color constraints. To our knowledge, the best solution found so far has 13 violations (over 480). Schaus and Deville (2008) report a solution with 22 violations, computed by large neighborhood tabu search with 1 day of running time. Interestingly, they obtain solutions with nearly 70 violations by using only swaps of tiles with tabu search. Decisions variables for each tile are its rotation and position on the board (expressed through boolean variables in CPLEX and LocalSolver) and mismatching edges are simply computing with the mathematical operators available in each solver. We obtain a solution with 70 violations with 1 day of running time by using LocalSolver 1.1. In this case, LocalSolver performs nearly one million moves per minute while the LS program contains 262 144 binary decision variables. Note that CPLEX does not provide any solution after 1 day of running time, whereas the generic CBLS of Comet obtains a solution with 107 violations. Pure CP approaches are not able to fall under the barrier of 80 violations (Schaus and Deville 2008).

6 Conclusion

The above results demonstrate that "Local Search Programming" is possible: an effective model-and-run paradigm for local search can be obtained by combining a simple modeling grammar and an efficient incremental solver based on autonomous structured moves. Hence, the next version of LocalSolver is envisaged following several research directions. First, the LSP formalism is far from being achieved: our main preoccupation is to add *set operators* without losing simplicity and genericity. This step is crucial for tackling complex scheduling and routing problems. Then, the concept of autonomous moves maintaining feasibility, which is a key of our approach, has to be reinforced and developed yet. Finally, we have planed to add other metaheuristics (Aarts and Lenstra 1997) to the top layer of the solver, in addition to the standard descent and the simulated annealing.

Acknowledgements We warmly thank all people who have contributed, directly or indirectly, to the LocalSolver project. Particularly: Antoine Jeanjean (Bouygues e-lab), Lucile Robin (École Centrale de Lyon), Guillaume Rochart (Bouygues e-lab), Michel Van Caneghem (LIF, Université Aix-Marseille II), Sofia Zaourar (ENSIMAG, Grenoble INP).

References

Aarts E, Lenstra J (1997) Local search in combinatorial optimization. In: Aarts E, Lenstra J (eds) Local Search in Combinatorial Optimization, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, Chichester, England, UK

Benoist T, Bourreau E (2008) Fast global filtering for Eternity II. Constraint Programming Letters 3:35–50

Benoist T, Estellon B, Gardi F, Jeanjean A (2009a) High-performance local search for solving inventory routing problems. In: Stützle T, Birattari M, Hoos H (eds) Proceedings of SLS 2009, the 2nd International Workshop on Engineering Stochastic Local Search Algorithms, Springer, Lecture Notes in Computer Science, vol 5752, pp 105–109

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

Cahon S, Melab N, Talbi EG (2004) ParadisEO: a framework for the reusable design of parallel and distributed metaheuristics. Journal of Heuristics 10(3):357–380

Caseau Y, Laburthe F, Silverstein G (1999) A meta-heuristic factory for vehicle routing problems. In: Proceedings of CP 1999, Springer, Lecture Notes in Computer Science, vol 1713, pp 144–158

Comet Tutorial (2010) Comet 2.1 Tutorial. Dynadec Decision Technologies Inc., Providence, RI, 581 pages, `http://www.dynadec.com`

Deville Y, Schaus P (2010) Personnal communication

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

Dotú I, Van Hentenryck P (1999) Scheduling social golfers locally. In: Proceedings of CPAIOR 2005, Springer, Lecture Notes in Computer Science, vol 3524, pp 155–167

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

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

Estellon B, Gardi F, Nouioua K (2009) High-performance local search for task scheduling with human resource allocation. In: Proceedings of SLS 2009, Springer, Lecture Notes in Computer Science, vol 5752, pp 1–15

Fernandez Pons D (2010) Personnal communication

Hnich B, Miguel I, Gent I, Walsh T (2009) CSPLib: a problem library for constraints. `http://www.csplib.org`

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

Perron L, Shaw P (2004) Combining forces to solve the car sequencing problem. In: Proceedings of CPAIOR 2004, Springer, Lecture Notes in Computer Science, vol 3011, pp 225–239

Perron L, Shaw P, Furnon V (2004) Propagation guided large neighborhood search. In: Proceedings of CP 2004, Springer, Lecture Notes in Computer Science, vol 3258, pp 468–481

Prandtstetter M, Raidl G (2008) 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

Rego C, Glover F (2002) Local search and metaheuristics. In: Gutin G, Punnen A (eds) The Traveling Salesman Problem and Its Variations, Kluwer Academic Publishers, Dordrecht, Netherlands, pp 105–109

Schaus P, Deville Y (2008) Hybridation de la programmation par contraintes et d'un voisinage à très grande taille pour Eternity II. In: Proceedings of JFPC 2008, pp 115–122

Schaus P, Hentenryck PV, Monette JN, Coffrin C, Michel L, Deville Y (2011) Solving steel mill slab problems with constraint-based techniques: CP, LNS, CBLS, to appear in Constraints

Selman B, Kautz H, Cohen B (1996) Local search strategies for satisfiability testing. In: Johnson D, Trick M (eds) Cliques, Coloring, and Satisfiability: 2nd DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol 26, AMS, Providence, RI

Van Hentenryck P, Michel L (2005) Constraint-Based Local Search. The MIT Press, Boston, MA

Van Hentenryck P, Michel L (2007) Synthesis of constraint-based local search algorithms from high-level models. In: Proceedings of AAAI 2007, pp 273–279

Van Hentenryck P, Michel L (2008) The steel mill slab design problem revisited. In: Proceedings of CPAIOR 2008, Lecture Notes in Computer Science, vol 5015, pp 377–381

Vasquez M, Hao JK (2001) 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

Voudouris C, Dorne R, Lesaint D, Liret A (2001) iOpt: a software toolkit for heuristic search methods. In: Proceedings of CP 2001, Lecture Notes in Computer Science, vol 2239, pp 716–730

Walser J, Iyer R, Venkatasubramanyan N (1998) An integer local search method with application to capacitated production planning. In: Proceedings of AAAI 1998, pp 373–379