Toward Local Search Programming: LocalSolver 1.0

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

[1] Bouygues e-lab, Paris, France

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

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

Abstract. This paper introduces *Local Search Programming* (LSP), as a paradigm allowing the practitioner to focus on the modeling of the problem using a simple formalism, and then to let its actual resolution to a solver based on efficient and reliable local-search algorithms. In other words, our goal is to offer a model & run approach to combinatorial optimization problems which are out of reach of existing Integer/Constraint Programming autonomous solvers. In this paper, *LocalSolver 1.0* is presented, first software realization of our works on this subject.

1 Introduction

In combinatorial optimization, the tree-search techniques consists 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 (IP) is surely one of the most powerful tools of operations research. Although limited 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 & 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 this one. Although incomplete, these techniques are appreciated by operations researchers because they allow to obtain high-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 [1].

This paper introduces *Local Search Programming* (LSP), as a paradigm allowing the practitioner to focus on the modeling of the problem using a simple formalism, and then to let its actual resolution to a solver based on efficient and reliable local-search algorithms. In other words, our goal is to offer a model & run approach to combinatorial optimization problems which are out of reach of existing IP/CP autonomous solvers. In this paper, *LocalSolver 1.0* is presented, first software realization of our works on this subject. This version allows to tackle a restricted (but important) class of combinatorial optimization problems: *assignment, partitioning, packing, covering*. Distributed under a BSD licence[3], the software can be used in two ways: as an executable program taking in input a file where the problem is modeled, or as a C++ library for programming quickly efficient local-search algorithms by overriding some components of the LocalSolver framework (in particular, heuristics and moves).

The paper is organized as follows. Having reviewed related works in the literature, the LSP modeling formalism associated with LocalSolver 1.0 is presented. Then, the architecture and the principal features of the solver are detailed. 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 Related Works

A local-search heuristic is designed according to three layers [11]: search strategy & (meta)heuristic, moves, algorithms & implementation. Our past experiences in engineering high-performance local-search algorithms [3, 9–11] have convinced us that neglecting one of these three layers may yield a significant decrease in terms of performance.

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 [5, 7]). In 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. As example, the reader is referred to the work of Helsgaun [12] on the traveling salesman problem. Hence, these frameworks do not address the hardest issues of the engineering of local-search algorithms. To the best of our knowledge, there exists no LS-based solver yet, apart from the pioneering work of Comet [22] (and its ancestor Localizer [16]) and iOpt [25]. These solvers aim at simplifying the writing of local-search heuristics thanks to a CP-based language (Comet) or to a Java library (iOpt). Finally, note that some of the best solvers for Satisfiability Testing (SAT) or Pseudo-Boolean Programming rely on stochastic local search (see for example Walksat [21]).

[3] `http://e-lab.bouygues.com/?p=693`

Our approach to autonomous LS is guided by the following fundamental principle: *the LS solver must work as a LS practitioner works*. Two implications are: the moves performed by the LS solver (tend to) maintain the feasibility of solutions, and the evaluation of these moves is speeded up by exploiting the basic invariants induced by the structure of the problem. Then, compared to the above frameworks or solvers, the main specificities of LocalSolver 1.0 are to provide: a simple mathematical formalism to model the problem in an appropriate way for LS resolution, and an autonomous LS-based solver focused on the feasibility and the efficiency of moves, which can be simply launched and tuned in command line. In addition, LocalSolver could also be used as a library, allowing to override each component of the local search: the heuristic, the moves, or even the evaluation.

3 LSP Modeling Formalism

In the Local Search Programming (LSP) format, a program consists in: decision variables, intermediate variables, constraints and objectives. Note that, of course, this file format has a programming interface counterpart in the LocalSolver C++ library. As example, here is described an artificial toy problem which can be classified as a bin-packing problem. We have 3 objects x, y, z of size 1, 2, 2 respectively, and 3 bins A, B, C knowing that C already contains an object of size 3. Our goal is to put these objects in the bins so as to minimize the product of the sizes of the smallest and largest bins if none is empty, and the size of the largest one otherwise. As secondary objective, we hope that the objects x and y appear together in a same bin, that either x or z is placed in C but not both, and that y does not appear in B. In this case, this second objective makes sense because one shall observe that several solutions exist with optimal cost equal to 4 for the first objective.

```
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 sizeA <- sum(1xA, 2yA, 2zA);
display sizeB <- sum(1xB, 2yB, 2zB);
display sizeC <- sum(1xC, 2yC, 2zC, 3);
sizeMin <- min(sizeA, sizeB, sizeC);
sizeMax <- max(sizeA, sizeB, sizeC);
prefer1 <- or(and(xA, yA), and(xB, yB), and(xC, yC));
prefer2 <- xor(xC, zC);
prefer3 <- not(yB);
minimize if(sizeMin > 0, product(sizeMin, sizeMax), sizeMax);
maximize booleansum(prefer1, prefer2, prefer3);
```

The statement `bool()` creates a boolean decision variable. For instance, the variable `xA` is true when item x is assigned to bin A. In this 1.0 version, only

boolean decision variables are allowed, making the LSP formalism close to the one of 0-1 integer programming. Then, the operator `<-` is used to define intermediate variables (for example, the size of each bin), 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 bin. In the same way, the keyword `minimize` and `maximize` prefixe the two lexicographically-ordered objectives of the program. Finally, `display` is used to write at console the values of the intermediate variables `sizeA`, `sizeB`, `sizeC` during the local search.

Formally, the BNF syntax of a program is:

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

where $< \textit{expression} >$ 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 until now, only boolean variables are allowed as decision variables.

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

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

where $< \textit{scalar} >$ is a number and $< \textit{identifier} >$ 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 Fig. 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)`. Since

boolean expressions are actually 0/1 variables, they can be used in all integer operators.

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*	integer

operators	arity	input	output
`sum`	n	integer	integer
`booleansum`	n	boolean	integer
`min`	n	integer	integer
`max`	n	integer	integer
`product`	n	integer	integer

operators	arity	input	output
`=`	2	integer	boolean
`<=`	2	integer	boolean
`>=`	2	integer	boolean
`<`	2	integer	boolean
`>`	2	integer	boolean
`!=`	2	integer	boolean

Fig. 1. Mathematical operators available in LocalSolver 1.0 (*the `if` operator takes a boolean as first argument and some integers as second and third arguments).

Introducing logical, arithmetic, or relational operators has two important benefits in local search context: expressiveness and efficiency. With such low-level operators, modeling is easier than with basic IP syntax, while remaining quickly assimilable by beginners (in particular, for engineers which are not comfortable with computer programming). On the other hand, 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 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 its problem, the practitioner shall remind 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 dirty trick where a big coefficient is used to simulate the lexicographic order: `minimize 1000 ratio_violations + color_changes;`. Note that the number of objectives is not limited and can have different directions (minimization or maximization).

4 LocalSolver

The command line for solving the above toy problem, granting one second to LocalSolver 1.0 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 lspfile toy.lsp... [OK]
Number of nodes : 41 (9 booleans)
Compute initial feasible solution : *** Objective value = ( 5, 2 )
Set descent heuristic : [OK]
Set objective bounds : [OK]
Create autonomous structures : [OK]
Create moves : [OK]
Running localsolver during 1 seconds...
8Path [ 0 / 0 / 1781 ]
7Path [ 0 / 0 / 3587 ]
6Path [ 0 / 0 / 7286 ]
5Path [ 0 / 0 / 14483 ]
4Path [ 0 / 0 / 28889 ]
3Path [ 1 / 2 / 58064 ]
2Path [ 1 / 3 / 115799 ]
*** Total [ 2 / 5 / 229889 ] in 1 seconds
*** Objective value = ( 4, 2 ) ( Time = 0 seconds, Nb iterations = 26 )
sizeA = 4
sizeB = 0
sizeC = 4
Final solution : *** Objective value = ( 4, 2 )
Writing solution in file toy.sol : [OK]
```

The cost of the initial feasible solution found by LocalSolver is $(5, 2)$, meaning 5 for the first objective and 2 for the second one. The output solution, found by local search after 26 iterations, has cost $(4, 2)$. During the second of allocated time, LocalSolver has performed 229 889 iterations, which corresponds to the total number of moves attempted, and also to the number of (feasible or infeasible) solutions visited during the search. Among these moves, 5 have been committed and 2 have strictly improved the cost of the current solution. These statistics are detailed for each move used by the autonomous solver. All these indicators shall be of interest for tuning LocalSolver. Finally, it creates the file "toy.sol" with the solution:

```
xA=0; yA=1; zA=1; xB=0; yB=0; zB=0; xC=1; yC=0; zC=0;
```

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. 2). 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 [25] or Comet [22]. With this representation, a solution corresponds to an instantiation of the root variables. In this context, 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.

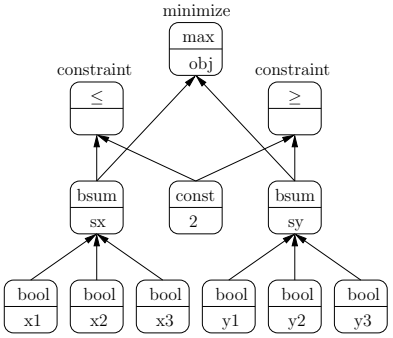

Fig. 2. The directed acyclic graph (DAG) induced by a simple model. For each node, the type (resp. name) of the node is given above (resp. below). Here "bsum" stands for `booleansum`.

LocalSolver is composed of 4 open components, corresponding each one to a software library: `dag`, `parser`, `solver`, `autonomous`. Roughly speaking, LocalSolver follows the three-layers methodology of the authors [11] for designing and engineering high-performance local-search heuristics. A sequence diagram describing the main interactions between the three layers (heuristic, moves, evaluation) of LocalSolver is sketched on Fig. 3. Resolutely oriented towards simplicity and efficiency, the design and implementation of LocalSolver have required a considerable effort in terms of software and algorithm engineering, which cannot be entirely detailed here. Hence, our presentation will be focused on the two crucial aspects of the solver: the evaluation algorithms and the autonomous moves.

4.1 Evaluation algorithms

Each node of the DAG must implement the following methods: `init`, `eval`, `commit`, `rollback`, `check`. 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

```
solver::Heuristic::run()
  *[!isTimeToStop()]
    ↪ solver::Heuristic::selectMove()
    ↪ solver::Move::modify()
    ↪ solver::Move::eval()
      ↪ solver::Propagation::eval()
        ↪ *[nbImpactedNodes] dag::Node::eval()
    ↪ isMoveAccepted := solver::Heuristic::decide()
    ↪ solver::Move::update()
      ↪ [isMoveAccepted] solver::Propagation::commit()
        ↪ *[nbModifiedNodes] dag::Node::commit()
      ↪ [else] solver::Propagation::rollback()
        ↪ *[nbModifiedNodes] dag::Node::rollback()
```

Fig. 3. Simplified sequence diagram of a local search in LocalSolver (`[isMoveAccepted]` defines a condition; `*[!isTimeToStop()]` defines a while-loop with stop condition `isTimeToStop()`; `*[nbImpactedNodes]` defines a for-loop over `nbImpactedNodes` iterations).

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, then 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. The reliability of the three previous routines, which are highly optimized through incremental calculations, is ensured by the `check` method which allows to check the correctness of all data structures of the node through brute calculations (in debug mode).

As mentioned above, the fast evaluation of moves is obtained by exploiting invariants related to each type of nodes. 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. The list T of a_i's whose value is true is maintained. 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.

4.2 Autonomous moves

As suggested in introduction, our ultimate goal is to autonomously perform the moves that a practitioner would have been designed to solve its problem. In this first version of LocalSolver, the autonomous moves can be viewed as a generalization of ejection chains [19] applied to the hypergraph induced by boolean variables and constraints. For example, let us consider the car sequencing problem [9, 10]: 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 successively the 4 boolean variables $x_{i,p}$, $x_{i,q}$, $x_{j,q}$, $x_{j,p}$ while preserving the feasibility of the 4 partition constraints where these variables appear. In a generic way, our autonomous moves are shown to simulate k-moves and k-swaps on packing/covering problems.

Define a *root sum* as a sum involving at least two binary decision variables either directly or multiplied by a scalar. A data structure is built listing all root sums in the DAG and for each binary decision variable, the list of root sums it belongs to. Besides, we maintain for each root 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 root 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 circularly repair modified sums, by applying an opposite change at each step and finally repairing the first and last sum simultaneously. Since in practice root sums are generally involved in constraints, k-Paths and k-Cycles tend to maintain the feasibility of the solution. For instance, when the root sums define a complete matching problem, any k-Cycle with k even can be completed (that is, closed without failing) in $O(k)$ time.

Changing the definition of root sums leads to variants which can be of interest for practically speeding up the convergence of the local search. For instance, we can focus on sums having at least one constraint among their successors in the DAG, or on sums involving only decisions variables. 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. Finally, a promising direction consists in flipping more than one variable per constraint. This extension follows the same logic as the generalization from ejection chains to ejection trees [6]. For instance, it allows ejecting two objects of size 1 when adding an object of size 2 in a set, in packing problems.

5 Experimental Results

LocalSolver 1.0 was tested on a benchmark mixing academic and industrial problems. We insist on the fact that our purpose is not to reach state-of-the-art results for all these problems. The primary goal of LocalSolver is to obtain some results similar to the ones obtained by standard local-search heuristics. The second goal is to obtain good-quality solutions with short running times, in particular when IP/CP solvers fail to find any solution. Each problem addressed in the benchmark is briefly described, and sample results are presented. Comparisons with IP approaches are done using the noncommercial solver GLPK 4.24 [15] (implementing a classical branch & bound & cut algorithm); this solver, widely used by OR practitioners, was reported to be one of the best non commercial IP solvers, in particular to quickly obtain good feasible solutions [14]. All numerical experimentations were performed on a standard computer equipped with the operating system Windows XP SP3 and the chip Intel Core 2 Duo T7600 (2.33 GHz, RAM 2 Go, L2 4 Mio, L1 64 Kio). Note that all results presented here have been rigorously checked; in particular, having recovered the business solution from the mathematical one, all the constraints and the objective values are verified.

Car sequencing. The car sequencing problem [9, 13] consists in ordering cars on an assembly line while minimizing violations on ratio constraints. Sample results are presented for 5 instances on Table 1 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 [13]; the fifth comes from a benchmark generated by Perron and Shaw [17]. The "state-of-the-art LS" corresponds to the high-performance local-search algorithm presented in [9], which currently owns the best results on all car sequencing benchmarks. "LocalSolver 1.0 (white)" means that LocalSolver 1.0 is used as library: here classical moves for car sequencing (swaps, shifts, inversions) [9] have been implemented in the LocalSolver framework, and their evaluation is let to the solver. "LocalSolver 1.0 (black)" means that LocalSolver 1.0 is used as black box (with default parameters, unless otherwise stated). "IP GLPK 4.24" corresponds to the IP solver provided by GLPK 4.24 [15]. 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, with in parenthesis the time at which this solution has been reached (the symbol "x" is used if no solution has been obtained within the time limit). Note that additional tests done with the state-of-the-art commercial solver FICO Xpress-Optimizer which confirm that the gap between commercial and noncommercial IP solvers is thin when tackling highly combinatorial problems like car sequencing. Indeed, for instance 10-93, Xpress obtains solutions of cost 30 (after 54 s) and 3 (after 136 s); for instance 200-01, it obtains solutions of cost 73 (after 41 s) and 67 (after 72 s).

time limit: 60 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art LS	3 (4)	0 (4)	0 (21)	1 (56)	0 (3)
LocalSolver 1.0 (white)	3 (28)	1 (42)	4 (47)	7 (59)	14 (55)
LocalSolver 1.0 (black)	7 (30)	10 (56)	21 (51)	28 (57)	51 (55)
IP GLPK 4.24	10 (49)	x (x)	x (x)	x (x)	x (x)
time limit: 600 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art LS	3 (4)	0 (4)	0 (21)	1 (56)	0 (3)
LocalSolver 1.0 (white)	3 (26)	0 (63)	1 (285)	2 (495)	0 (333)
LocalSolver 1.0 (black)	6 (68)	5 (551)	7 (484)	14 (365)	23 (467)
IP GLPK 4.24	10 (49)	18 (361)	11 (403)	x (x)	x (x)

Table 1. Sample results on the academic car sequencing problem.

time limit: 600 s	X2	X3	X4
state-of-the-art LS	0, 192, 66 (1/19)	0, 337, 6 (1/19)	0, 160, 407 (1/19)
LocalSolver 1.0 (white)	0, 249, 89 (16/19)	0, 445, 62 (10/19)	0, 192, 695 (13/19)
LocalSolver 1.0 (black)	0, 268, 212 (16/19)	36, 544, 187 (16/19)	2, 353, 692 (18/19)

Table 2. Sample results on the Renault's car sequencing problem: X2, X3, X4 correspond to instances of set X named 023-EP-RAF-ENP-S49-J2, 024-EP-RAF-ENP-S49-J2, 025-EP-ENP-RAF-S49-J1 respectively. The ranking of each result relatively to the 18 finalists of the ROADEF Challenge is given in parenthesis.

A real-world version integrating the constraints and objectives of paint workshop was proposed by the car manufacturer Renault as subject of the ROADEF 2005 Challenge [10] (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 2 contains sample results for 3 very large-scale instances intractable by any IP/CP solver [18]: X2 (1260 vehicles, 12 options, 13 colors), X3 (1319 vehicles, 18 options, 15 colors), X4 (996 vehicles, 20 options, 20 colors). The "state-of-the-art LS" corresponds to the local-search heuristic which won the challenge [10]. In white-box mode, the search is started with the initial solution provided by Renault and the same moves than the ones described above for the academic car sequencing are implemented. In black-box mode, the results are obtained with specifying the following options: `hr_timelimit=600 hr_objbounds=0:0,1:0,2:0 hr_descent=phase:90,99,100 mv_kpaths=4:1000,6:100, 8:10,10:1`. For instance X2, the resulting LS program contains 516 936 variables whose 374 596 are binary decision variables (450 Mo of RAM are allocated 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 (strictly) improving solutions. Observe that LocalSolver used as white (resp. black) box outperforms (resp. is comparable to) the hand-made

variable neighborhood search by Prandtstetter and Raidl [18] mixing classical moves and large neighborhood search by IP (using ILOG CPLEX).

Social golfer. The social golfer problem [13] consists in assigning persons to groups over several weeks so as to maximize the number of meetings. A real-life version is encountered at Bouygues SA for scheduling the managers' seminars. Sample results are presented on Table 3. Three classical instances are addressed: 10-10-3 (10 groups of 10 players for 3 weeks), 10-9-4, 10-3-13. In this case, the objective is to minimize the number of duplicate meetings. The fourth 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 the characters into each group, avoiding the undesired meetings, maximizing the number of (desired) meetings. For classical instances, the "state-of-the-art LS" corresponds to the tabu-search heuristic presented in [8], which currently owns the best results on almost all social golfer benchmarks. For the real-life instance, "state-of-the-art LS" corresponds to the local-search algorithm which was implemented by one of the authors as operational solution: a first-improvement descent performing efficient swaps. Note that due to the quadratic form of the objective function, such a problem is particularly difficult to tackle by IP techniques. For almost all instances, GLPK is not able to compute the initial LP relaxation during the time limit, because of the large number of variables (hundreds of thousands) implied by the linearization.

time limit: 60 s	10-10-3	10-9-4	10-3-13	seminar
state-of-the-art LS	0	0	0	1, 0, 1082
LocalSolver 1.0 (white)	0	5	3	1, 0, 1082
LocalSolver 1.0 (black)	0	7	3	1, 0, 1082
IP GLPK 4.24	x	x	x	x

Table 3. Sample results on the social golfer problem.

Steel mill slab design. The steel mill slab design problem [23, 13] 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. The classical instance of CSPLib with 111 orders [13] is solved to optimality (cost equal to 0) in less than 1 second by LocalSolver, which outperforms most of the previous approaches (see [23] for the state of the art). Note that GLPK is not able to obtain a feasible solution within 1 hour. The LSP treated by LocalSolver in this case contains 40 739 variables with 12 321 booleans; LocalSolver attempts more than 5 millions moves per minute.

Spot 5 photographs scheduling. The spot 5 daily photograph scheduling problem [24] 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. 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 4. The results reported by Vasquez and Hao [24] are given on the line "Vasquez-Hao" of the table. For single-orbit instances 5, 54 and 509, these ones are proven to be optimal; for multi-orbit instances 1407 and 1506, these ones are shown to be near-optimal (gap lower than 1 %). Note that these near-optimal results have been obtained by a tabu-search heuristic. The main conclusion of this experience is that LocalSolver remains competitive face to IP solvers, even for tackling problems tractable by tree-search techniques. Another interesting remark is that the autonomous moves performed by LocalSolver may lead to better solutions than moves implemented by hand (k-moves and k-swaps of photos).

time limit: 60 s	5	54	509	1407	1506
Vasquez-Hao	115	70	19 125	176 245	168 247
LocalSolver 1.0 (white)	103	69	19 107	165 243	140 260
LocalSolver 1.0 (black)	105	69	19 118	156 250	152 247
IP GLPK 4.24	114	70	13 110	161 230	156 234

Table 4. Sample results on the spot 5 photographs scheduling problem.

Minimum formwork stock. The minimum formwork stock problem [4], 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 scale makes it efficiently tractable by IP solvers. Sample results are presented on Table 5. Observe that the autonomous moves performed by LocalSolver lead to better solutions than hand-made moves (k-swaps of formworks).

time limit: 60 s	site1	site8b	site12b	site13b
LocalSolver 1.0 (white)	5 669 638	5 679 798	9 355 316	7 868 796
LocalSolver 1.0 (black)	5 640 326	5 640 398	9 223 040	7 729 336
IP GLPK 4.24	5 630 426	5 409 300	8 392 256	7 408 566

Table 5. Sample results on the minimum formwork stock.

Eternity II. The Eternity II problem [2, 20] 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 [20] 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. We obtain a solution with 51 violations in 1 day of running time by using LocalSolver as white box (by implementing a threshold heuristic with appropriate moves). In this case, LocalSolver performs nearly one million moves per minute while the LS program contains 262 144 binary decision variables.

6 Conclusion

The above results demonstrate that Local Search Programming is possible: a model & run paradigm for local search can be obtained by combining a simple modeling grammar and an efficient incremental solver based on appropriate autonomous 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 the notion of *sets* 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 threshold metaheuristics [1] to the top layer of the solver, in addition to the standard descent.

References

1. E. Aarts, J.K. Lenstra, eds (1997). *Local Search in Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, Chichester, England, UK.
2. T. Benoist, E. Bourreau (2008). Fast global filtering for Eternity II. *Constraint Programming Letters* 3, pp. 35–50.
3. T. Benoist, B. Estellon, F. Gardi, A. Jeanjean (2009). High-performance local search for solving real-life inventory routing problems. In *Proceedings of SLS 2009*, *LNCS* 5752, pp. 105–109.
4. T. Benoist, A. Jeanjean, P. Molin (2009). Minimum formwork stock problem on residential buildings construction sites. *4OR* 7, pp. 275–288.
5. S. Cahon, N. Melab, E.-G. Talbi (2004). ParadisEO: a framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics* 10(3), pp. 357–380.

6. Y. Caseau, F. Laburthe, G. Silverstein (1999). A meta-heuristic factory for vehicle routing problems. In *Proceedings of CP 1999*, *LNCS* 1713, pp. 144–158.
7. L. Di Gaspero, A. Schaerf (2003). EasyLocal++: an object-oriented framework for flexible design of local search algorithms. *Software - Practice & Experience* 33(8), pp. 733–765.
8. I. Dotú, P. Van Hentenryck (2005). Scheduling social golfers locally. In *Proceedings of CPAIOR 2005*, *LNCS* 3524 , pp. 155–167.
9. B. Estellon, F. Gardi, K. Nouioua (2006). Large neighborhood improvements for solving car sequencing problems. *RAIRO Operations Research* 40(4), pp. 355–379.
10. B. Estellon, F. Gardi, K. Nouioua (2008). Two local search approaches for solving real-life car sequencing problems. *European Journal of Operational Research* 191(3), pp. 928–944.
11. B. Estellon, F. Gardi, K. Nouioua (2009). High-performance local search for task scheduling with human resource allocation. In *Proceedings of SLS 2009*, *LNCS* 5752, pp. 1–15.
12. K. Helsgaun (2009). General k-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation* 1(2/3), pp. 119–163.
13. B. Hnich, I. Miguel, I.P. Gent, T. Walsh (2009). CSPLib: a problem library for constraints. `http://www.csplib.org`
14. J.T. Linderoth, T.K. Ralphs (2005). Noncommercial software for mixed-integer linear programming. In *Integer Programming: Theory and Practice* (J. Karlof, ed), CRC Press Operations Research Series, vol. 3, pp. 253–303. CRC Press.
15. A. Makhorin (2007). GLPK : Gnu Linear Programming Kit, version 4.24. `http://www.gnu.org/software/glpk`
16. L. Michel, P. Van Hentenryck (2000). Localizer. *Constraints* 5(1/2), pp. 43–84.
17. L. Perron, P. Shaw (2004). Combining forces to solve the car sequencing problem. In *Proceedings of CPAIOR 2004*, *LNCS* 3011, pp. 225–239.
18. M. Prandtstetter, G.R. Raidl (2008). An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research* 191(3), pp. 1004–1022.
19. C. Rego, F. Glover (2002). Local search and metaheuristics. In *The Traveling Salesman Problem and Its Variations* (G. Gutin, A. Punnen, eds), pp. 105–109. Kluwer Academic Publishers, Dordrecht, Netherlands.
20. P. Schaus, Y. Deville (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.
21. B. Selman, H. Kautz, B. Cohen (1996). Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: 2nd DIMACS Implementation Challenge* (D.S. Johnson, M.A. Trick, eds), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26. AMS, Providence, RI.
22. P. Van Hentenryck, L. Michel (2005). *Constraint-Based Local Search*. The MIT Press, Boston, MA.
23. P. Van Hentenryck, L. Michel (2008). The steel mill slab design problem revisited. In *Proceedings of CPAIOR 2008*, *LNCS* 5015, pp. 377–381.
24. M. Vasquez, Jin-Kao Hao (2003). Upper bounds for the Spot 5 daily photograph scheduling problem. *Journal of Combinatorial Optimization* 7(1), pp. 87–103.
25. C. Voudouris, R. Dorne, D. Lesaint, A. Liret (2001). iOpt: a software toolkit for heuristic search methods. In *Proceedings of CP 2001*, *LNCS* 2239, pp. 716–730.