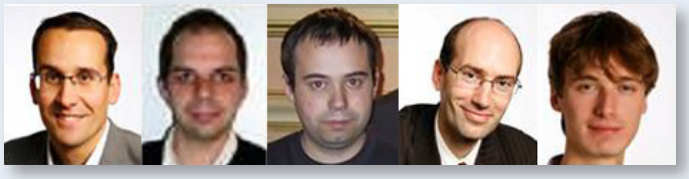
TUTORIAL

Integrating Local Search Techniques Into A Mathematical Programming Solver



Thierry Benoist <tbenoist@localsolver.com>
Julien Darlay <jdarlay@localsolver.com>
Bertrand Estellon
Frédéric Gardi <fgardi@localsolver.com>
Romain Megel <rmegel@localsolver.com>

Introduction

Mixed Integer Linear Programming (MIP) is undoubtedly one of the most powerful tools of Operations Research (OR). Its ease of use appeals to OR professionals: the user models the problem as an integer linear program and the MIP engine solves it by branch & bound & cut. This ``model & run" approach, when effective, reduces considerably development and maintenance of optimization software, and other tree search-based technologies like Constraint Programming (CP) are following the way.

Faced with situations such as large-scale nonlinear optimization problems, OR practitioners often find MIP or CP solvers ineffective. They address this problem by implementing local search heuristics. In contrast with tree search techniques, Local Search (LS) involves applying iterative changes (called moves) to improve the objective function. This technique allows operations researchers to obtain good-quality solutions in a reasonable time (in the order of minutes). However, designing and implementing local search algorithms is not straightforward, even with frameworks that have been designed to help the programmer.

The algorithmic layer dedicated to the evaluation of moves is particularly difficult to engineer, because it requires an expertise both in algorithms and computer programming. (see [1] for a survey on the LS paradigm and its applications.)

This observation motivated the development of LocalSolver , a mathematical programming solver based on local search. Started in 2007, the project aims to combine the simplicity of use of a model-and-run solver and the power of local search techniques for combinatorial optimization. It thus enables OR practitioners to focus on the modeling of the problem using a simple formalism, and leave its actual resolution to a solver based on efficient and reliable local search techniques.

Modeling

LocalSolver includes an innovative math modeling and scripting language for fast prototyping called LSP (Local Search Programming) language. This language is used in our examples although lightweight object-oriented APIs are also available for full integration (C++, Java, .NET). LocalSolver's modeling language is close to that of classical mathematical programming but its use of a larger set of common mathematical operators makes it more intuitive and easy to learn for OR practitioners. For example the following lines define the model of a knapsack problem with n objects of given weights and values.

```
function model()  {
        for [i in 1..n] x[i] <-

      knapsackWeight <- sum[i in 1..n](weights[i] * x[i]);
      constraint knapsackWeight <= kn

       knapsackValue <- sum[i in 1..n](values[i] * x[i]);
       maximize knapsackValue;
 }
```

In this basic example, binary decision variables x[i] are introduced with the bool( ) statement. Then, the weight in the knapsack is introduced as a sum expression and a constraint is set on its value. Finally, the value in the knapsack is defined and set as a maximization objective. Note that several objective functions could be added, which would be interpreted as a lexicographic objective function. The crucial point here is that nothing more needs to be defined and in particular no neighborhoods are specified. Only this model is given to the solver. The solver is left to apply a local search algorithm made of ``general-purpose" moves to work on the abstract combinatorial structure induced by the user model, first for finding a feasible solution and then for iteratively improving this solution. The key principles of these moves will be given in the next section.

Although this simple knapsack example uses only linear expressions, the underlying solving techniques allow the use of highly nonlinear operators including conditional expressions (*if A then B else C* written as A ? B : C) or even array lookups (the expression A[N] coding for the Nth element in array A). Table 1 gives the list of available operators.

Table 1. Mathematical operators available in LocalSolver.

Arithmetic	Logical	Relational
sum, min, max	not	==
prod, div, mod	and	!=
log, exp, pow	or	<=
sin, cos, tan	xor	>=
floor, ceil, round	if	<
abs, dist, sqrt	at	>

Introducing logical, arithmetic, or relational operators has two important benefits of being intuitive and efficient in a local search context. With such low-level operators, modeling is easier than with basic MIP syntax, even for beginners (in particular, for those who are not comfortable with computer programming). Besides, the invariants induced by these operators can be exploited by the internal algorithms of the LS solver to speed up local search.

For example, we can consider the P-median problem [2], of selecting a subset S of P cities among N, for instance for locating public facilities, so as to minimize the sum of distances from each city to the closest city in S. In the model below the use of conditional and min operators allows formulating the model almost as expressed in the above sentence. This simplicity also yields a model focused only on the relevant decision variables, namely the selection or not of each city: x[i]. After constraining the sum of these variables to equal P, the minDistance[i] from each city i to the closest city in S is written as the minimum of distances to other cities, the distance to cities outside of S being counted as infinite (InfiniteDistance can be set to the maximum value in the distance matrix). The objective function is the sum of these minDistance[i].

```
function model()  {
        x[1..N] <- bool();
        constraint sum[i in 1..N](x[i]) == P;

        minDistance[i in 1..N] <- min[j in 1..N]
                (x[j] ? distance[i][j] : InfiniteDistance);

        minimize sum[i in 1..N] (minDistance[i]);
 }
```

Here again this definition of the problem is all that the solver needs to find high-quality solutions in seconds (an average gap of 0.6 % on the 40 instances of the OR-Library with a time limit set to 1 minute). In the next section, we will describe the internal mechanisms making this possible.

Solving

Our approach to autonomous Local Search was guided by the following fundamental principle: the LS solver must work in the same way that an LS practitioner works. This implies that LocalSolver performs structured moves to maintain the feasibility of solutions at each iteration, whose evaluation is accelerated by exploiting invariants induced by the structure of the model. Components that make the algorithm efficient are:

- An incremental machinery that quickly evaluates the impact of a transformation of the solution;
- Multiple autonomous moves exploiting the combinatorial structure of the model to explore feasible neighborhoods of a solution;
- A global adaptive search strategy guiding the search towards high-quality solutions.

The incremental machinery is based on a representation of the model as a directed acyclic graph (DAG), the roots of which are the decisions and the leaves, the constraints and objectives. The inner nodes of this DAG are the operators listed in Table 1. With this representation, a solution is a complete instantiation of the root variables. Applying moves to the current solution consists of modifying the current values of the decisions (roots) and evaluating constraints and objectives (leaves) by propagating these modifications along the DAG. Designing a highly optimized propagation algorithm allows evaluating millions of moves per minute on real-life models.

The most simple generic move is the change of the value of a decision, for example, changing the value of a x[i] decision in our P-median example. However, if we want to move from one feasible solution to another, as in most hand-made local search algorithms, more structured moves are required. The solver builds such moves by analyzing the structure of the model. For instance, the cardinality constraint in the P-median problem will be exploited to design moves preserving the number of selected cities and consequently the feasibility of the solution. More generally, following ejection chains or ejections cycles in the constraint hypergraph (Figure 1) yields very powerful moves. In addition to these small neighborhoods, larger neighborhoods can be explored by combining smaller neighborhoods within a destroy & repair mechanism. Finally, the identification of special combinatorial structures can trigger the activation of specific neighborhoods. Note that the neighborhoods can be explored almost randomly or by targeting moves with higher probability of success.

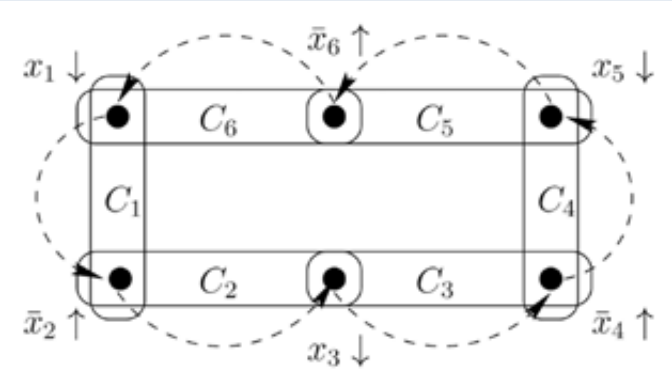


Figure 1. An ejection cycle involving six boolean variables x1, x3, x5 (whose current value is 1) and x2,x4, x6 (whose current value is 0), and six constrained sums C1, …, C6. Each variable belongs to two sums (for example, x1 belongs to C1 and C6). Now, x1, x3, x5 are decreased while x2, x4, x6 are increased. This move preserves the values of the sums, and thus the feasibility of the constraints.

The exploration of the search space is distributed on several threads with periodic synchronization. The global diversification of the search is ensured through simulated annealing with reheating and restart mechanisms. Statistics on the performance of the moves are dynamically exploited to improve the overall performance along the search.

Benchmarks and conclusions

Despite their apparent conceptual simplicity, the principles given in the previous section yield remarkably good results in practice, in particular for large-scale combinatorial optimization problems, which are out-of-scope of state-of-the-art mathematical programming solvers. Even on some of the hardest MIPLIB instances, LocalSolver recently outperformed the best MIP solvers. Another typical example is the car sequencing problem , which consists of scheduling cars along painting and assembly lines subject to sequencing constraints. The solutions obtained by LocalSolver in 10 seconds are far better that the ones obtained by the best MIP solvers after running for 1 day. This ability to tackle large-scale combinatorial problems in a model-and-run fashion was vividly illustrated during the EURO/ROADEF Challenge 2012, which involved the reassignment of processes on Google servers subject to various resources and dependency constraints. Ranked 24th over 82 participating teams, LocalSolver was the sole model-and-run, general-purpose mathematical programming solver to qualify for the final round using a 100-line model, written in one day. More computational results on both academic and industrial problems can be found in [3] or on our website .

In 2012, LocalSolver was mature enough to move from a research project to a commercial product that is now used in various industries around the world, from the maximization of TV advertising revenue in France to the optimization of bakery supply chain in Japan (a nonlinear problem involving 3 million 0-1 decisions!).

For the end of 2013, the first step toward an all-in-one mathematical programming solver for large-scale mixed-variable non-convex optimization is planned. This new version will offer several new important features from both functional and technical points of view: small-neighborhood moves to optimize over continuous or mixed decisions; exploration of large, exponential-size neighborhoods over 0-1 or mixed decisions using some tree search techniques (for example, rounding heuristics based on linear relaxation); exploration of large neighborhoods over continuous decisions by revisiting successive linear programming techniques for nonlinear programming (based on a simplex algorithm); computation of lower bounds combining constraint propagation and dual linear relaxation.

Bibliography

[1] Aarts E, Lenstra J (1997) Local search in combinatorial optimization. John Wiley & Sons
[2] Beasley J (1985) A note on solving large p-median problems. Eur J Oper Res 21(2):270–273
[3] Benoist T, Estellon B, Gardi F, Megel R, Nouioua K (2011) Localsolver 1.x: a black-box local-search solver for 0-1 programming. 4OR-Q J Oper Res 9(3):299–316