LocalSolver

Math programming by local search
LocalSolver

A solver aligned with enterprise needs
• Provides high-quality solutions in seconds
• Scalable: to tackle problems with millions of decisions
• Proves optimality when possible (best effort)

A solver aligned with practitioner needs
• « Model & Run »
  • Simple mathematical modeling formalism
  • Direct resolution: no need of complex tuning
• A simple and transparent pricing
LocalSolver

A new-generation solver
- **Computing good-quality solutions by local search**
- Computing lower bounds separately (relaxation, inference, cuts)

A quality software
- An innovative modeling language for fast prototyping
- Lightweight object-oriented APIs: a few classes only
- Reliable and robust: quality assurance through continuous integration
- Fully portable: Windows, Linux, Mac OS (x86, x64)
- Reactive support, realized by developers themselves (even for academics)
Why local search?

Weaknesses of tree search

• Not suited to reach quickly good “integer feasible solutions”
• Designed to prove optimality
• Exponential time: not scalable (the best MIP solvers still fail to find feasible solutions for real-life instances with 10,000 binaries)

Benefits of local search

• Provides good-quality solutions in short running times
• Scalable (each iteration done in sublinear or even constant time)
• Able to optimize in nonconvex and nonsmooth spaces
Why local search?

- Time to obtain high-quality solutions

- MIP/CP solvers
- LocalSolver
- Specific Local Search

Instance size
LocalSolver

Quick tour
Knapsack

8 items to pack in a sack: maximize the total value of items while not exceeding a total weight of 102 kg

```plaintext
function model() {
  // 0-1 decisions
  x_0 <- bool(); x_1 <- bool(); x_2 <- bool(); x_3 <- bool();
  x_4 <- bool(); x_5 <- bool(); x_6 <- bool(); x_7 <- bool();

  // weight constraint
  knapsackWeight <- 10*x_0 + 60*x_1 + 30*x_2 + 40*x_3 + 30*x_4 + 20*x_5 + 20*x_6 + 2*x_7;
  constraint knapsackWeight <= 102;

  // maximize value
  knapsackValue <- 1*x_0 + 10*x_1 + 15*x_2 + 40*x_3 + 60*x_4 + 90*x_5 + 100*x_6 + 15*x_7;
  maximize knapsackValue;
}
```

The user writes the model: nothing else to do!

declarative approach = model & run
function model() {
    // 0–1 decisions
    x[0..7] <- bool();

    // weight constraint
    constraint knapsackWeight <= 102;

    // maximize value
    maximize knapsackValue;

    // secondary objective: minimize product of minimum and maximum values
    knapsackMinValue <- min[i in 0..7](x[i] ? values[i] : 1000);
    knapsackMaxValue <- max[i in 0..7](x[i] ? values[i] : 0);
    knapsackProduct <- knapsackMinValue * knapsackMaxValue;
    minimize knapsackProduct;
}
### Mathematical operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>prod</td>
<td>not</td>
</tr>
<tr>
<td>min</td>
<td>max</td>
<td>and</td>
</tr>
<tr>
<td>div</td>
<td>mod</td>
<td>or</td>
</tr>
<tr>
<td>abs</td>
<td>dist</td>
<td>xor</td>
</tr>
<tr>
<td>sqrt</td>
<td>log</td>
<td>if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
function model() {
  // 0-1 decisions
  x[1..nbItems] <- bool();
  // weight constraint
  knapsackWeight <= sum[i in 1..nbItems](weights[i] * x[i]);
  constraint knapsackWeight <= knapsackBound;
  // maximize knapsack value
  knapsackValue <- sum[i in 1..nbItems](values[i] * x[i]);
  maximize knapsackValue;
}
LocalSolver

Real-life applications
Scheduling cars along an assembly line

- Each car requires some options
- Each option induces a ratio constraint \( P/Q \)
- A class is a set of cars requiring the same options

Objective: to space options over the line

- We wish no more than 2 sunroofs over 5 consecutive cars
- For any window of 5 cars, a penalty is computed as \( \max(n-2, 0) \) with \( n \) the number of cars requiring a sunroof
LSP model

\[ x_{cp} = 1 \iff \text{car of class } c \text{ is in position } p \]

\[
\begin{align*}
& x[1..\text{nbClasses}][1..\text{nbPositions}] \leftarrow \text{bool}(); \\
& \text{for } [c \text{ in } 1..\text{nbClasses}] \\
& \quad \text{constraint } \sum[p \text{ in } 1..\text{nbPositions}](x[c][p]) = \text{card}[c]; \\
& \text{for } [p \text{ in } 1..\text{nbPositions}] \\
& \quad \text{constraint } \sum[c \text{ in } 1..\text{nbClasses}](x[c][p]) = 1; \\
& \text{op}[o \text{ in } 1..\text{nbOptions}][p \text{ in } 1..\text{nbPositions}] \leftarrow \text{or}[c \text{ in } 1..\text{nbClasses} : \text{options}[c][o]](x[c][p]); \\
& \text{nbVehicles}[o \text{ in } 1..\text{nbOptions}][j \text{ in } 1..\text{nbPositions} - Q[o]+1] \leftarrow \sum[k \text{ in } 1..Q[o]](\text{op}[o][j+k-1]); \\
& \text{violations}[o \text{ in } 1..\text{nbOptions}][j \text{ in } 1..\text{nbPositions} - Q[o]+1] \leftarrow \text{max}(\text{nbVehicles}[o][j] - P[o], 0); \\
& \text{obj} \leftarrow \sum[o \text{ in } 1..\text{nbOptions}][p \text{ in } 1..\text{nbPositions} - Q[o]+1](\text{violations}[o][p]); \\
& \text{minimize } \text{obj};
\end{align*}
\]
Car sequencing at Renault

Additional constraints: no more than 10 consecutive cars with the same color

Decision variables remain the same

Additional objective: minimize the number of paint color changes

Problem posed by Renault as 2005 ROADEF Challenge
LocalSolver competitive with finalists (ranked 16/55)
Reassignment of processes to machines, with different kinds of constraints (mutual exclusion, resources, etc.)

More than 100,000 binary decisions
Only 1 day of work
LocalSolver qualified for final round (ranked 25/80)

http://www.localsolver.com