Learning parameters of the Wedelin heuristic with application to crew and bus driver scheduling[☆]

Sara Maqrot[a], Simon de Givry[a], Marc Tchamitchian[b], Gauthier Quesnel[a,*]

[a] *UR 875 MIAT, Université de Toulouse, INRAE, Castanet-Tolosan, France*
[b] *UR 767 Ecodéveloppement, INRAE, Avignon, France*

Abstract

Heuristics are important techniques designed to find quickly good feasible solutions for hard integer programs. Most heuristics depend on a solution of the relaxed linear program. Another approach based on Lagrangian relaxation offers a number of advantages over linear programming, namely it is extremely fast for solving large problems. Wedelin heuristic is such a Lagrangian based heuristic, initially developed to solve airline crew scheduling problems. The performance of this method depends crucially on the choice of its numerous parameters. To adjust them and learn which ones have important influence on whether a solution is found and its quality, we propose to conduct a sensitivity analysis followed by an automatic tuning of the most influential parameters.

We have implemented a C++ open-source solver called baryonyx which is a parallel version of a (generalized) Wedelin heuristic. We used the Morris method to find useful continuous parameters. Once found, we fixed other parameters and let a genetic optimization algorithm using derivatives adjust the useful ones in order to get the best solutions for a given time limit and training instance set. Our experimental results done mostly on crew and bus driver scheduling benchmarks tackled as set partitioning problems show the significant improvements obtained by tuning the parameters and the good performances of our approach compared to state-of-the-art exact and inexact integer programming solvers.

Keywords: integer programming, heuristics, Lagrangian relaxation, sensitivity analysis, parameter tuning, crew and bus driver scheduling, set partitioning problem
2010 MSC: 90C10, 90C59

1. Introduction

Heuristics are important techniques designed to find quickly good feasible solutions for hard integer programs. Most heuristics depend on a solution of the relaxed linear program. Another approach based on Lagrangian relaxation offers a number of important

[☆] Open-source solver baryonyx available at `https://github.com/quesnel/baryonyx`.
[*] Corresponding author
Email address: `gauthier.quesnel@inrae.fr` (Gauthier Quesnel)

advantages over linear programming [13], namely it is extremely fast for solving large problems.

Wedelin heuristic [32] is such a Lagrangian based heuristic. Wedelin [33] described the basic principle of the algorithm and its application in the Carmen system for scheduling crew rotations for airlines. Alefragis et al. [2] presented a scalable parallelization of the original algorithm used in the Carmen system. Grohe and Wedelin [15] and [34] introduced a similar algorithm for the max-sum problem. Ernst et al. [12] described a variation of the Wedelin algorithm and applied it to the staff planning problem. Bastert et al. [4] presented many extensions and generalizations of Wedelin algorithm with various improvements. They evaluated the performance of their variant on a set of instances from different sources, where the results were favorable compared to FICO Xpress, a commercial optimization software. Starting from [4], we propose an extension performing multiple runs in parallel.

A major difficulty in the use of optimization methods is the parameter setting. It is important for each problem to find a set of parameter values that leads to optimal performances. Choosing the best values manually requires a lot of experimentation. There are several methods for automatic configuration of parameter values [10, 16]. Eiben et al. [10] classify methods into two categories according to the manner of use, before running the optimization algorithm (parameters tuning) and during its execution (parameter control).

The control methods include self-adjust the parameter values for the resolution and dynamically control these values to improve the solution search. These techniques are widely applied to self-adjust the parameters of evolutionary algorithms [16], such as crossover rate [35], mutation rate [28] and population size [11] for genetic algorithms. These methods can be subdivided into two branches: deterministic and (self-)adaptive [18]. Deterministic methods are based on deterministic rules, which do not change during the execution of the algorithm. Their goal is to calculate approximate values of parameters and adjust them according to the problem [3]. Adaptive methods use information on the current state of the search to change parameter values. Adaptation is effected by changing the objective function, by increasing or decreasing the penalty coefficients for violations of constraints, from one generation to another. This prevents poor settings to conduct future generations [17]. These control methods are developed in an automated framework for setting the parameters of a specific problem (one instance). If the goal is to solve different instances, these methods can be costly in terms of computing time, given the number of parameters and instances.

The principle of parameter tuning methods is simpler. Parameters do not change values during search. They are adjusted before the execution of the algorithm, and remain unchanged after. The settings obtained by learning on a subset of instances is used to solve all instances of a problem. Different parameter tuning strategies exist in the literature and depend on the type of parameters: discrete, continuous or categorical. Two famous examples are CALIBRA [1] and ParamILS [20] parameter tuning methods for the discrete case. They explore the parameter configuration search space using (iterated) local search. CALIBRA limits the number of parameter configuration evaluations using partial statistical designs. ParamILS saves computation time by doing partial evaluations of the training set.

A sensitivity analysis can facilitate parameter tuning by focusing on important parameters for a problem. The goal is to reduce the time and effort in resolving sensitive parameters. In the literature, the work on the use of both techniques (sensitivity analysis to reduce the parameter search space and automatically adjusting their values) is not much discussed.

Kim et al. [23] perform a sensitivity analysis on the dynamic parameters of sea ice model, using automatic differentiation. Information on the gradient provided by the latter are used in a parameter optimization algorithm based on quasi-Newton method[37]. Teodoro et al. [30] combine sensitivity analysis and automatic parameter tuning for an image segmentation problem. This approach permits to identify the least influential parameters and reduce the parameter configuration search space (100 points instead of trillion points).

In these previous works, parameter tuning is performed for a single instance. In this paper we propose a protocol to generate a universal set of parameter values from a subset of instances of a given problem to be solved by our multi-run Wedelin algorithm. The basic idea is to i) select a subset of instances, ii) investigate the sensitivity of parameters for fixing the values of the non-sensitive parameters to their default values, iii) automatically adjust the values of the significant parameters by black-box optimization, and iv) apply the learned parameter setting on all instances of the problem.

In Section 2, we describe the Wedelin heuristic and our multi-run extension for parallelism. In Section 3, we present our protocol for parameter tuning. In Section 4, we give experimental results and conclude.

2. A Parallel Version of Wedelin Heuristic

We are interested in minimizing 0/1 linear programs of the following form:

$$\begin{array}{ll} \min & cx \\ s.t. & Ax = b \\ & x \in \{0,1\}^n \end{array}$$

where $c \in \mathbb{R}^n$ is a vector of n costs, $b \in \mathbb{N}^m$ a column vector of constant terms, and $A \in \{0,1\}^{m \times n}$ a coefficient matrix for the constraints. In the following, constraint index i refers to the ith row in equation $Ax = b$, and index j to the jth column in A associated to variable x_j. Let $J = \{1, \dots, n\}$ be the set of column indices, and $J(i) = \{j : a_{ij} = 1\} \subseteq J$ the subset of column indices occurring in constraint i. Similarly, let $I = \{1, \dots, m\}$ be the set of row indices, and $I(j) = \{i : a_{ij} = 1\} \subseteq I$ the subset of constraint indices involving x_j.

In this paper we adopt an approximate method, called Wedelin algorithm or In-the-middle algorithm. This method is designed for solving linear problems on Boolean variables [32] and uses the Lagrangian relaxation.

2.1. The Lagrangian relaxation

The Lagrangian relaxation consists in moving into the objective function all the constraints. These constraints are built into the objective function as linear combinations where the coefficients are Lagrangian multipliers. They penalize the objective function if one of the integrated constraint is violated. This relaxation has the advantage over linear relaxation to directly provide integer solutions and to tackle large size problems.

2.2. The Wedelin Heuristic

In this paper, we are particularly interested in a specific heuristic based on the Lagrangian relaxation known as Wedelin heuristic. This method solves linear programming problems with Boolean variables with a specific mathematical form such as the set partitioning problem. This heuristic tries to directly solve the linear problem:

$$\begin{aligned} \min_{x} \quad & cx - \pi(Ax - b) \\ s.c. \quad & x \in \{0,1\}^n \end{aligned} \tag{1}$$

Where $\pi = \{\pi_1, \pi_2, ..., \pi_m\} \in \mathbb{R}^m$ represent the Lagrangian multipliers or dual variables. The resolution is trivial when π_i elements in π are fixed to $\hat{\pi}$:

$$\begin{aligned} \hat{\pi}b + \min_{x} \quad & (c - \hat{\pi}A)x \\ s.c. \quad & x \in \{0,1\}^n \end{aligned} \tag{2}$$

The solution $\hat{x}_j$ is constructed with the reduced cost sign with $(c_j - \hat{\pi}a_j)$ and a_j the associated column in the coefficient matrix A.

$$\hat{x}_j = \begin{cases} 1 & \text{if} \quad (c_j - \hat{\pi}a_j) < 0, \\ 0/1 & \text{if} \quad (c_j - \hat{\pi}a_j) = 0, \\ 0 & \text{if} \quad (c_j - \hat{\pi}a_j) > 0. \end{cases} \tag{3}$$

If the reduced cost $\bar{c} = (c - \hat{\pi}A)$ is non-zero, the solution is unique. However, if one or a few elements of $\bar{c}$ are zero, it will be difficult to find a feasible solution for the not relaxed problem. The goal of this algorithm is to find a $\hat{\pi}$ where all elements are non-zero and where the single solution of the relaxed problem is realizable for the primal problem.

We note that the change of the value of a single component $\hat{\pi}_i$ modifies the values (and signs) of the reduced costs $\bar{c}$, and therefore affects the $\hat{x}$ solution.

The main idea of the Wedelin algorithm (see Algorithm 1) is to consider iteratively a single constraint i, updating the element $\hat{\pi}_i$ such that the $\hat{x}$ vector satisfies the i constraint of the primal problem. There is always a selection range of $\hat{\pi}_i$. The algorithm chooses the value of $\hat{\pi}_i$ in the middle of this interval. This corresponds to the dual search for descent by coordinates.

Algorithm 1 represents the core of the solver. It takes as input a definition of the problem: the matrix A and the vectors b and c. It expects an output vector of Boolean, $\hat{x}$, a solution of the problem. The algorithm starts at line 1 with the construction of an initial $\hat{x}$ vector which permits to build the list of violated constraints R. It uses the Bastert et al. [4] proposal to penalize variables with positive costs. The main loop begins at line 2 and run until the loop limit p and/or the $\kappa_{\max}$ is reached. For each violated constraints (line 3), it (i) decreases the preferences $\hat{p}$, (ii) produces the reduced cost vector r and sort it according to the reduced costs, (iii) updates the Lagrangian multipliers at line 4, (iv) affects the $\hat{x}$ vector and the preference matrix $\hat{p}$. Finally, at line 6, we update the list of violating constraints and exit with the solution found or adjust the κ adjustment and the iteration l.

```
   Input  : $A \in \{0,1\}^{m\times n}$, $b \in \mathbb{N}^m$, $c \in \mathbb{R}^n$
   Output: $\hat{x} \in \{0,1\}^n$ with $A\hat{x} = b$ and $c\hat{x}$ small or message no solution
1  for $j \in \{1,\dots,n\}$ do                                   // Build an initial variable assignment
       if $c_j \le 0$ then $\hat{x}_j \leftarrow 1$
       else $\hat{x}_j \leftarrow 0$
   end
   Let $\hat{\pi} \in \mathbb{R}^m$, $\hat{\pi} \leftarrow 0$, $\hat{p} \in \mathbb{R}^{m\times n}$, $\hat{p} \leftarrow 0$, $l \leftarrow 1$, $\kappa \leftarrow \kappa_{min}$,
    $R \leftarrow \{i : \sum_{j=1}^{n} a_{ij}\hat{x}_j \neq b_i\}$                       // List of violated constraints
2  while $l \le \rho$ and $\kappa \le \kappa_{max}$ do
3      for $i \in R$ do                                          // Update every violated constraint
           for $j \in J(i) = \{j : a_{ij} \neq 0\}$ do $\hat{p}_{ij} \leftarrow \theta \times \hat{p}_{ij}$        // History exp. decay
           for $j \in J(i)$ do                                       // Build reduced costs
               $r_j \leftarrow c_j - \sum_{i\in I(j)} a_{ij}\hat{\pi}_i - \sum_{i\in I(j)} \hat{p}_{ij}$
           end
           $r_{\varphi[1]} < r_{\varphi[2]} < ... < r_{\varphi[|J(i)|]}$             // Sort variables by increasing red. costs
4          $\hat{\pi}_i \leftarrow \hat{\pi}_i + \frac{1}{2}(r_{\varphi[b_i+1]} + r_{\varphi[b_i]})$                  // Update Lagrangian multipliers
           if $l \le \omega$ then $\Delta \leftarrow 0$           // Do not perturb reduced costs during warmup
5          else $\Delta \leftarrow \frac{\kappa}{1-\kappa}(r_{\varphi[b_i+1]} - r_{\varphi[b_i]}) + \delta$
           for $j \in \{1,\dots,b_i\}$ do           // Update variables and preferences positively
               $\hat{p}_{i\varphi[j]} \leftarrow \hat{p}_{i\varphi[j]} + \Delta$ ; $\hat{x}_{\varphi[j]} \leftarrow 1$
           end
           for $j \in \{b_i+1,\dots,|J(i)|\}$ do                                 // or negatively
               $\hat{p}_{i\varphi[j]} \leftarrow \hat{p}_{i\varphi[j]} - \Delta$ ; $\hat{x}_{\varphi[j]} \leftarrow 0$
           end
       end
6      $R \leftarrow \{i : \sum_{j=1}^{n} a_{ij}\hat{x}_j \neq b_i\}$               // Update the list of violated constraints
       if $R = \emptyset$ then return $\hat{x}$                        // If empty, exit with solution found
       $\kappa \leftarrow \kappa + \kappa_{step}(\frac{|R|}{m})^{\alpha}$                          // Adaptive adjustment of step size
       $l \leftarrow l + 1$                                                        // Next iteration
   end
   return no solution found
```

Algorithm 1: Wedelin algorithm with local preferences and adaptive step size.

2.3. The Parallel Solver Baryonyx

In this paper, we introduce a new integer and Boolean linear programming solver called baryonyx. This solver is largely based on the algorithms provided in Bastert et al. [4]. We have introduced several modifications to the proposed algorithms to (i) allow the reuse of previous solutions found, (ii) diversify the search, and also (iii) exploit todays symmetric multiprocessor CPUs.

Baryonyx accepts two modes: solver and optimizer. In the solver mode, it runs once trying to satisfy all the constraints (the exact implementation of Algorithm 1). In the optimizer mode, it runs in parallel according to the number of processors, and tries to satisfy all the constraints and to optimize the solution at each run, reporting the best solution found for all runs when it reaches its time limit.

The Wedelin heuristic depends only on the δ parameter (line 5 in Algorithm 1) to diversify the search. To improve diversification, we develop several random processes (see the technical documentation of baryonyx). The most important process of diversification is the initialization mechanism of $\hat{x}$. Indeed, $\hat{x}$ is used to determine violated constraints before any computation. Default, a deterministic initialization is proposed by Bastert et al. [4] where $\hat{x}_j$ equals 1 if $c_j \leq 0$ otherwise 0. We propose to extend this part by combining a random process and several different algorithms. We use the Bernoulli's law and its parameter $p \in [0, 1]$ to provide random Boolean.

random Each $\hat{x}_i$ are initialized with the Bernoulli law.

bastert For each $\hat{x}_i$, either the variable is initialized by the cost variable c_i (See section 1 in Algorithm 1) or by a random Boolean.

best For each $\hat{x}_i$, either the variable is initialized by the best solution found previously $\hat{x}_i$ or by random Boolean.

Since Wedelin's algorithm runs several times in baryonyx, we changed the initialization of $\hat{x}$ for each iteration. The default baryonyx choice is the best-cycle policy. It starts with the *bastert* policy. If is fails to find a solution, it restarts with the *random* policy, else it uses the best solution found as initial solution and it tries to improve this solution with the *best* policy three times. If it fails it switches to the first step of the algorithm (*bastert* policy) otherwise it keeps the *best* policy. Figure 1 shows the diagram of the best-cycle policy.

To further increase the diversification, we exploit the symmetric multiprocessor CPUs. Each core shares matrices A, vectors b and c. The vector $\hat{x}$ is local for each process. Each core have its own pseudo-random number generator initialized with different random seed.

3. Learning Continuous Parameter Values

In this section, we present our protocol for tuning continuous parameters of Wedelin heuristic automatically.

Algorithm 1 has eleven numerical parameters as described in Table 1. Parameters $\tau, \Gamma, \rho, \omega$ are integers whereas the others are continuous. We assume a fixed time limit τ and a fixed number of cores Γ, and treat the other integer parameters as continuous values

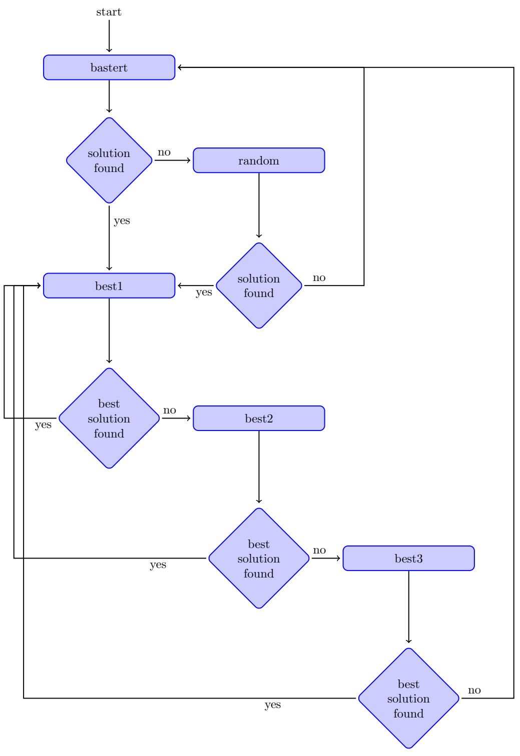

Figure 1: The best-cycle algorithm is designed to both improve diversification and to look for a better solution when a solution is found. Each box represents a run of the complete Wedelin algorithm. The solver stops when time limit is reached.

Table 1: baryonyx numerical parameters with their typical domain ranges and descriptions.

Parameter	Range	Description
τ	$[1, +\infty[$	CPU time limit (in seconds)
Γ	$[1, +\infty[$	Number of parallel runs
ρ	$[10^2, 10^5]$	Number of iterations inside a run
ω	$[0, 100]$	Number of warmup iterations before using κ
κ_{min}	$[0, 0.5]$	Minimum value for κ approximation
κ_{step}	$[10^{-4}, 10^{-2}]$	Step value for κ approximation
κ_{max}	$[0.6, 1]$	Maximum value for κ approximation
α	$[0, 2]$	Adaptive adjustment of κ based on the number of violated constraints [4]
δ	$[10^{-3}, 10^{-1}]$	Random perturbation on reduced costs
θ	$[0, 1]$	Strength of history of local preferences $\hat{p}$ [4]
p	$[10^{-4}, 1 - 10^{-4}]$	Bernoulli distribution with success probability p

using a rounding function. So, we have nine continuous parameters to tune. The Wedelin heuristic is viewed as a complex function with $K = 9$ input parameters $(x_1, \dots, x_K)$ or factors and a single output value $y(x_1, \dots, x_K)$ which corresponds to the quality of the parameter configuration over a set of training instances. For each benchmark category, we select an evenly distributed subset (20%) of instances to be part of the training set.

3.1. Quality of a parameter configuration

The quality of a parameter configuration $(x_1, \dots, x_K)$ is computed as follow. First, we execute our parallel baryonyx solver on every training instance using a modified Wedelin-Good configuration[1]. Each execution uses Γ' cores[2] in parallel during 2τ time limit. We collect the best l_e^{init} and worst u_e^{init} solution objective values returned by Algorithm 1 for each instance e. If no solution is found we remove this instance from the training set. During the training phase, the *normalized* quality of a parameter configuration is obtained from the mean over N valid training instances of the relative distance gap to the best initial solutions:

$$y(x_1, \dots, x_K) = 1 - \frac{1}{N} \sum_{e=1}^{N} \frac{l_e - l_e^{init}}{10u_e^{init} - l_e^{init}}$$

If no solution is found for a particular instance and parameter configuration then we assume $l_e = 10u_e^{init}$. By multiplying u_e^{init} by 10, we assume not finding a solution is at least ten times worse than finding a good solution close to the best initial solution. When $y > 1$, we have found a better configuration than the modified WedelinGood configuration.

3.2. Selection of important parameters by sensitivity analysis

We used the Morris method [26, 27] to determine which inputs have important effects on the output. The goal is to discover which parameters are important and cannot be ignored. These parameters will be fine tuned later on (see Section 3.3). For that a factorial sampling plan is built from individually randomized one-factor-at-a-time designs. The Morris method randomly selects L initial configurations within a regular K-dimensional d-level grid. Each parameter is discretized into d levels including its bounds (given in Table 1). Starting from each initial configuration, a *configuration trace* is constructed by changing the value of one parameter at a time until all the parameters have been modified. The step value is usually chosen as $\Delta = (d-1)/2$. See an example in Figure 2(left). Each trace corresponds to $K+1$ configuration evaluations, resulting in a total of $L \times (K+1)$ evaluations.The Morris method performs a limited amount of configuration evaluations which is a linear function of the number of parameters K.

The Morris method provides qualitative sensitivity measures allowing to rank the input factors in order of importance, but not to quantify by how much one given factor is more important than another [27]. In Figure 2(right) we show estimated means ($\mu_k^* =$

[1]We set $\alpha = 1$, $p = 0.5$ with the best-cycle strategy, and a randomized order of violated constraints at each iteration.

[2]In our experiments, we used a smaller number of cores $\Gamma' = 3$ during the training phase than during the test phase ($\Gamma = 30$) in order to save computation time.

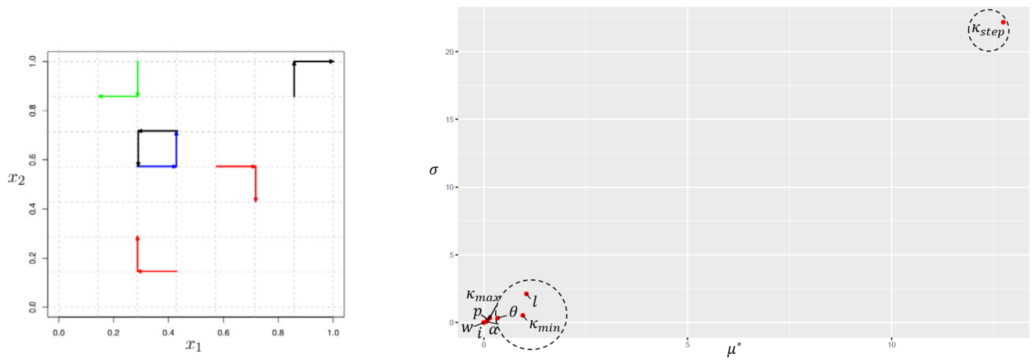

Figure 2: (Left) Example of $L = 6$ traces in the Morris design for $K = 2$ parameters discretized into $d = 8$ levels. The step value here is $\Delta = 1$. (Right) Sensitivity measures of the elementary effects for the CSPLib022 benchmark.

$\sum_{l=1}^{L} |\delta y_k^l|/L$) and standard deviations ($\sigma_k = \sqrt{\sum_{l=1}^{L}(\delta y_k^l - \mu_k)^2/L}$, $\mu_k = \sum_{l=1}^{L} \delta y_k^l/L$) of the distribution of (absolute values of) elementary effects δy_k^l found by the Morris method for some benchmark category with:

$$\delta y_k^l = \frac{y(x_1, \ldots, x_{k-1}, x_k + \Delta, x_{k+1}, \ldots, x_K) - y(x_1, \ldots, x_k, \ldots, x_K)}{\Delta}$$

When μ_k^* is large but not σ_k, parameter x_k has an important overall influence on the output. If both measures are large, it corresponds to a non-linear effect on the output or an input involved in interaction with other factors [27]. This is the case for parameter κ_{step} in Figure 2(right). We use μ_k^* to rank the input factors.

In our experiments, we have $K = 9, L = 50, d = 10$. We selected the four most important factors, $\kappa_{min}, \kappa_{step}, \delta, \theta$, which was the same set of parameters found by the Morris method in all our benchmark categories.

3.3. Optimization of Selected Parameter Values

genoud (GENetic Optimization Using Derivatives) [25] is a black-box optimization method for solving nonlinear, nonsmooth, and even discontinuous functions. It combines a genetic algorithm with a quasi-Newton method. The quasi-Newton method is the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method [14] using a built-in numerical derivative. It helps to quickly find a local optimum when the current parameter configuration is "in a smooth neighborhood of the local optimum point" [25]. It may also prevent to find the global optimum if used too early or too aggressively. In practice, a number of initial *warmup* generations are explored before BFGS is applied on the best individual in the current population.

genoud starts with a random population including the modified WedelinGood configuration. Then, next generations are built from 8 genetic algorithm operators (used in equal proportion) dedicated to continuous parameters: *Cloning, Uniform Mutation, Boundary Mutation, Non-Uniform Mutation, Polytope Crossover, Simple Crossover, Whole Non-Uniform Mutation, and Heuristic Crossover* [25]. For instance, the Polytope Crossover computes a new parameter configuration that is a convex combination of as many individuals as there are parameters to tune.

In our experiments, a (hard) maximum number of 10 generations is performed with a population size of 100. BFGS is applied on the best individual at each generation after the eighth generation. We observed between 803 to 1157 evaluations of the output value y. It can be more than 10×100 due to BFGS evaluations and less because genoud will not evaluate the same configuration twice.

4. Experimental Results

Baryonyx is a free software (MIT license) in C++17 for solving Boolean and integer linear programming problems. It is provided as a command line program, as a shared library, and has an encapsulation to the R software. Following results were achieved using the version 0.4 of baryonyx built with gcc-7.2.

The baryonyx wrapper for R is called Rbaryonyx. It relies on the rcpp package to facilitate exchanges between the two libraries. Rbaryonyx can read lp files, solves the problem and returns a list of solution(s) and solving information data to easily link with other packages such as R sensitivity [22] for the Morris method and rgenoud [25].

All computations were performed on a cluster of 32-physical-core Intel Xeon CPU E5-2683 v4 at 2.1 GHz and 4 GB of RAM per core. In order to speed up the parameter tuning process, we parallelize the evaluation of the training instances, by taking care that the actual number of executions of baryonyx multiplied by Γ' is less than the available number of cores. Depending on the size of the training set, each parameter configuration evaluation took between 1 to 1.5 minute. The Morris method took between 9 to 13 hours per benchmark family. The genoud method took from 14 to 28 hours.

In the following, we compare baryonyx against IBM ILOG cplex version 12.8 and LocalSolver[3] version 8.0. cplex is a state-of-the-art exact MIP solver. LocalSolver is a mathematical programming local search solver using a simulated annealing based on ejection chain moves specialized for maintaining the feasibility of Boolean constraints and an efficient incremental evaluation using a directed acyclic graph [6]. Every solver ran in parallel mode using 30 cores. The solving time limit is 60 seconds (except for VCS where it is $1,800$ sec.). cplex and LocalSolver use their default parameter configurations[4]. baryonyx ran with three different static parameter configurations (Supplementary Table .7) plus the one found by rgenoud (Supp. Table .8, where the corresponding generation of the best configuration found is also mentioned).

When available, we also report results from other publications [7, 9, 4, 31], but we must treat these results with care as they do not correspond to the same computer, nor time limit, and were obtained on a sequential machine.

All the instances have been preprocessed by cplex. A direct translation of 0/1 linear programming lp file format into LocalSolver lsp format has been done.

[3]`https://www.localsolver.com`

[4]For cplex, we set $EPAGAP = EPGAP = EPINT = 0$ to avoid premature stop.

4.1. SPP benchmark

This test set consists in 55 instances of airline crew scheduling problems expressed as set partitioning problems. These problems are obtained from the OR-library [5][5]. SPP instances were provided by four different airline corporations where a subset of these problems was initially solved in [19], and further experimented by other exact [7] and local search methods [8, 31].

These instances are easily solved by cplex within the 60-second time limit. The same optimum values were also reported in [7] and [31] (when available). LocalSolver could not find a solution on four instances, whereas all baryonyx configurations always found a solution. Using default or optimized by genoud configurations were the best options, respectively at 0.06% and 0.01% to the optimum.

Table 2: Computational results (relative distance to best-known solutions for solved instances and in parentheses number of solved instances) on SPP instances [5].

Instances	BARYONYX				CPLEX	Local-Solver	[7]	[31]
	DEFAULT	FAST	GOOD	GENOUD				
aa	0.23 %	0.80 %	1.19%	0.08%	0.00%	23.02%	0.00%	1.64%
(6)	(6)	(6)	(6)	(6)	(6)	(3)	(6)	(6)
us	0.09 %	1.48%	2.60%	0.02%	0.00%	0.00%	0.00%	0.04%
(4)	(4)	(4)	(4)	(4)	(4)	(3)	(4)	(4)
nw	0.00%	2.64%	3.96%	0.00%	0.00%	8.53%	0.00%	
(43)	(43)	(43)	(43)	(43)	(43)	(43)	(43)	N/A
kl	0.69%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	
(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)	N/A
Mean	0.06%	2.26%	3.41%	0.01%	**0.00%**	8.54%	**0.00%**	
(55)	(55)	(55)	(55)	(55)	(55)	(51)	(55)	N/A

4.2. Telebus benchmark

Telebus[6] [7] is a scheduling problem to program tour vehicles for disabled persons in Berlin. The goal is to provide a one-off service with minimum costs, while respecting a set of constraints, such as vehicle capacity and mandatory breaks. The problem was modeled in two steps. The first step, "clustering", identifies all possible bus circuits that can carry several people at a time. The goal is to select a set of controls with a minimal vehicle travel distance. In the second step, "chaining", the selected commands are chained to generate bus circuits that respect all the constraints. The objective is to minimize the total distance traveled by the vehicles. These two steps represent 28 instances of set partitioning problems divided into two equal-size subfamilies (14 v/clustering, 14 t/chaining) corresponding to different periods in the year 1996. t/chaining instances are more difficult than v/clustering instances and their optimum is mostly unknown (except for t0415, t0420, t0421).

cplex solved all v/clustering and t0415 instances within the time limit, doing slightly better than the original branch-and-cut method of [7] and the 4-flip neighborhood local search algorithm of [31]. baryonyx and LocalSolver were a few percent below.

[5]`http://people.brunel.ac.uk/~mastjjb/jeb/orlib/sppinfo.html`

[6]`http://www.zib.de/opt-long_projects/TrafficLogistics/Telebus/index.en.html`

baryonyx^{rgn} performed extremely well on t/chaining instances, obtaining the best results among all tested methods except on three instances where cplex and 4-flip local search found better solutions. In particular, for two open instances (t1717/t1722), it improved MIPLIB 2017 best reported solutions (from 184271/114245 to 165881/167523).

Looking at our training set, we observed it does not include any v-instances because the initial modified WedelinGood evaluation (see Section 3.1) failed to produce any solutions.

Table 3: Computational results (relative distance to best-known solutions for solved instances and in parentheses number of solved instances) on telebus instances [7].

Instances	BARYONYX				CPLEX	Local-Solver	[7]	[31]
	DEFAULT	FAST	GOOD	GENOUD				
v0415-0421	0.14%	0.12%	0.11%	0.30%	0.00%	0.04%	0.00%	0.00%
(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
v1616-1622	0.43%	1.08%	1.45%	1.43%	0.00%	6.21%	0.01%	0.09%
(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
t0415-0421	0.02 %	0.01 %	0.03%	0.03%	0.61%		1.88%	0.95%
(7)	(7)	(7)	(7)	(7)	(7)	(0)	(7)	(6)
t1716-1722	16.05%	30.37%	9.29%	0.00%	11.19%	150.25%	9.70%	10.71%
(7)	(7)	(7)	(7)	(7)	(7)	(4)	(7)	(7)
Mean	4.16%	7.89%	2.72%	**0.44%**	2.95%	35.82%	2.90%	3.01%
(28)	(28)	(28)	(28)	(28)	(28)	(18)	(28)	(27)

We also compared our solver using WedelinFast and WedelinGood static configurations with the original version developed by Bastert [4]. Our methods were able to find a solution for all the instances whereas the original approach could not for at least six instances. It demonstrates the robustness of doing multiple runs in parallel compared to a single run (using a longer 600-second time limit) as done in [4].

4.3. CSPLib022 benchmark

CSPLib022[7] is a library of 12 bus driver scheduling problems reformulated as set partitioning problems. The problems come from different bus companies: Reading (r1 to r5a), Centre West Ealing area (c1, c1a, c2), the former London Transport (t1 and t2). These problems are relatively small and easy. cplex took less than 7.7 seconds to solve its most difficult instance c1a. baryonyx^{rgn} found all the optima in less than 2.2 seconds (solving time) for the largest instance r3. LocalSolver did not find the optimum for this instance, neither found a solution for two other instances. The iterative repair local search method GENET [9] got the worst results.

4.4. VCS benchmark

VCS instances are randomly generated bus and driver scheduling problems. VCS1200 is a medium-size instance ($1,200$ constraints, $130,000$ variables), and VCS1600 is a large problem ($1,600$ constraints, $500,000$ variables) [36].

With a $1,800$-second time limit, cplex did not find any integral solution for the largest instance whereas baryonyx with its default setting or the one found by genoud produced

[7]`http://www.csplib.org/Problems/prob022`

Table 4: Computational results (relative distance to best-known solutions for solved instances and in parentheses number of solved instances) on CSPLib022 instances [29].

Instances	baryonyx				cplex	LocalSolver	[9]
	DEFAULT	FAST	GOOD	GENOUD			
CSPLib022 (12)	6.87% (12)	**0.00%** (12)	**0.00%** (12)	**0.00%** (12)	**0.00%** (12)	1.54% (10)	26.24% (8)

relatively good solutions (5.3% to the optimum for VCS1600). A first solution at 5.7% to the optimum for VCS1600 was found by default baryonyx in 280 seconds (solving time).

Table 5: Computational results (relative distance to best-known solutions for solved instances and in parentheses number of solved instances) on VCS instances [36].

Instances	baryonyx				CPLEX
	DEFAULT	FAST	GOOD	GENOUD	
VCS (2)	9.25% (2)	(0)	(0)	**7.32%** (2)	12.68% (1)

In order to find the optimum values, we also ran cplex without any time limit. It took 1 hour for VCS1200 and 4.3 hours for VCS1600 to be solved to optimality by cplex version 12.7.1 on a 4-core Intel CPU i7-4600U at 2.1 GHz.

4.5. Nqueens benchmark

Nqueens[8] instances represent the weighted n-queen problem where the goal is to place n queens on a chessboard $n \times n$ so that none of them attack each other (exactly one queen in each row and each column, at most one queen in each ascending diagonal and each descending diagonal). There are n^2 0/1 variables $x_{i,j}$. The objective function is to minimize $\sum_{i,j} c_{i,j} x_{i,j}$ with $c_{i,j}$ a random value uniformly sampled from $[1, n]$. The largest instance among 8 has 1 million variables.

This problem combines set partitioning linear constraints (for rows and columns) and set packing linear constraints (for diagonals). It shows the ability of baryonyx to tackle efficiently a larger class of integer linear programs, with better results than cplex and LocalSolver thanks to parameter tuning by genoud.

Table 6: Computational results (relative distance to best-known solutions for solved instances and in parentheses number of solved instances) on nqueens instances.

Instances	baryonyx				CPLEX	LOCALSOLVER
	DEFAULT	FAST	GOOD	GENOUD		
nqueens (8)	4.33% (6)	8.11% (7)	5.44% (8)	**0.61%** (8)	453.26% (8)	36.34% (8)

[8]`https://forgemia.inra.fr/thomas.schiex/cost-function-library/tree/master/random/wqueens`

5. Conclusion

baryonyx is a parallel multi-start meta-heuristic which offers good results on large crew and bus driver scheduling problems expressed as set partitioning problems in a relatively short amount of time. It can be used just after preprocessing to provide a good initial upper bound for a complete branch-and-bound solver. Depending on the usage context and time available, when off-line tuning of the parameters is allowed, we show a methodological process to select the important factors and optimize them. Even using a small training set, we could significantly improve the results. This methodology is readily available as supplementary R scripts next to baryonyx source code. Concerning the comparison with the other solvers, it is important to note that we did not try to tune their parameters and better results could be obtained as reported in [21]. Concerning the set partitioning problems (all our benchmarks except nqueens), more preprocessing rules could be applied.We made some preliminary experiments with set covering problems but the results were not as good as with set partitioning or set packing problems. It remains as future work to evaluate our solver on a larger set of benchmarks.

We plan to exploit the Lagrangian multipliers to give dual bound information which could make our solver exact in some cases, a feature already available in LocalSolver using linear relaxation. Dealing with a quadratic objective function as done in [15] is also an important topic we would like to work on and apply to planning problems in agronomy [24].

Acknowledgments. We are grateful to the GenoToul (Toulouse) Bioinformatic platform for providing us computational support for this work.

References

[1] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations research*, 54(1):99–114, 2006.

[2] Panayiotis Alefragis, Peter Sanders, Tuomo Takkula, and Dag Wedelin. Parallel integer optimization for crew scheduling. *Annals of Operations Research*, 99(1-4):141–166, 2000.

[3] Laura Barbulescu, Jean-Paul Watson, and L. Darrell Whitley. Dynamic representations and escaping local optima: Improving genetic algorithms and local search. *AAAI/IAAI*, 2000:879–884, 2000.

[4] Olivier Bastert, Benjamin Hummel, and Sven de Vries. A generalized Wedelin heuristic for integer programming. *INFORMS Journal on Computing*, 22(1):93–107, 2010.

[5] J. E. Beasley. Or-library: distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41:1069–1072, 1990.

[6] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. Localsolver 1. x: a black-box local-search solver for 0-1 programming. *4OR: A Quarterly Journal of Operations Research*, 9(3):299–316, 2011.

[7] Ralf Borndörfer. *Aspects of set packing, partitioning, and covering.* PhD thesis, Berlin, Techn. Univ., 1998.

[8] P. C. Chu and J. E. Beasley. A genetic algorithm for the set partitioning problem. *Imperial College, London*, 1995.

[9] Suniel D Curtis, Barbara M Smith, and Anthony Wren. Constructing driver schedules using iterative repair, 2000. Presented at The Practical Application of Constraint Technologies and Logic Programming Conference (PACLP'2000).

[10] Ágoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on evolutionary computation*, 3(2):124–141, 1999.

[11] Agoston Endre Eiben, Elena Marchiori, and V. A. Valko. Evolutionary algorithms with on-the-fly population size adjustment. In *International Conference on Parallel Problem Solving from Nature*, pages 41–50. Springer, 2004.

[12] Andreas Ernst, Houyuan Jiang, and Mohan Krishnamoorthy. A new Lagrangian heuristic for the task allocation problem. In *Industrial Mathematics*, pages 137–158. Oxford, 2006.

[13] Marshall L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management science*, 50(12supp.):1861–1871, 2004.

[14] PE Gill, W Murray, and MH Wright. *Practical Optimization*. Academic Press, San Diego, 1981.

[15] Birgit Grohe and Dag Wedelin. Cost Propagation–Numerical Propagation for Optimization Problems. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 97–111. Springer, 2008.

[16] Youssef Hamadi. Autonomous search. In *Combinatorial Search: From Algorithms to Systems*, pages 99–122. Springer, 2013.

[17] Georges R. Harik and Fernando G. Lobo. A parameter-less genetic algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pages 258–265. Morgan Kaufmann Publishers Inc., 1999.

[18] Robert Hinterding, Zbigniew Michalewicz, and Agoston E. Eiben. Adaptation in evolutionary computation: A survey. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 65–69. IEEE, 1997.

[19] Karla L. Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management science*, 39(6):657–682, 1993.

[20] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *AAAI*, volume 7, pages 1152–1157, Vancouver, Canada, 2007.

[21] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 186–202. Springer, 2010.

[22] Bertrand Iooss and Paul Lemaître. A review on global sensitivity analysis methods. In *Uncertainty Management in Simulation-Optimization of Complex Systems*, pages 101–122. Springer, 2015.

[23] Jong G. Kim, Elizabeth C. Hunke, and William H. Lipscomb. Sensitivity analysis and parameter tuning scheme for global sea-ice modeling. *Ocean Modelling*, 14(1):61–80, 2006.

[24] Sara Maqrot, Simon de Givry, Gauthier Quesnel, and Marc Tchamitchian. A Mixed Integer Programming Reformulation of the Mixed Fruit-Vegetable Crop Allocation Problem. In *Advances in Artificial Intelligence: From Theory to Practice*, Lecture Notes in Computer Science, pages 237–250. Springer, Cham, June 2017.

[25] Walter R. Mebane and Jasjeet Singh Sekhon. R version of genetic optimization using derivatives, version 5.8-2.0, 2012.

[26] Max D. Morris. Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33(2):161–174, 1991.

[27] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. John Wiley and Sons, 2004.

[28] Jim Smith and Terence C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 318–323. IEEE, 1996.

[29] Curtis Suniel. CSPLib problem 022: Bus driver scheduling, 1999.

[30] George Teodoro, Tahsin M. Kurç, Luís FR Taveira, Alba CMA Melo, Yi Gao, Jun Kong, and Joel H. Saltz. Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines. *Bioinformatics*, 33(7):1064–1072, 2016.

[31] Shunji Umetani. Exploiting variable associations to configure efficient local search algorithms in large-scale binary integer programs. *European Journal of Operational Research*, 263:72–81, 2017.

[32] Dag Wedelin. An algorithm for large scale 0/1 integer programming with application to airline crew scheduling. *Annals of operations research*, 57(1):283–301, 1995.

[33] Dag Wedelin. The design of a 0/1 integer optimizer and its application in the Carmen system. *European Journal of Operational Research*, 87(3):722–730, 1995.

[34] Dag Wedelin. Revisiting the in-the-middle algorithm and heuristic for integer programming and the max-sum problem. *preprint*, 2013.

[35] Shengxiang Yang and Bedau Abbass. Adaptive crossover in genetic algorithms using statistics mechanism. *Artificial Life*, 8:182–185, 2003.

[36] A. Zaghrouti, F. Soumis, and I. El Hallaoui. Integral simplex using decomposition for the set partitioning problem. *Operations Research*, 62:435–449, 2014.

[37] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

Table .7: Static parameter configurations for baryonyx.

Parameters	Configurations		
	WedelinFast	WedelinGood	baryonyx default
τ	60 seconds (1800 for VCS)		
Γ	30 parallel runs (only 3 for training)		
ρ	10^5 iterations		
ω	20 warmup iterations		
κ_{min}	0		
κ_{step}	10^{-3}	2×10^{-4}	10^{-3}
κ_{max}	0.6		
α	0		1
δ	0.01		$(1-\theta)\frac{\min_{j=1}^{n} \lvert c_j \rvert, c_j \neq 0}{\max_{j=1}^{n} \lvert c_j \rvert}$
θ	0.4	0.6	0.5
p	0		0.5
Violated constraint order	*none*		*random*
Initialization policy	*bastert*		*best-cycle*

Table .8: Learnt parameter configurations for baryonyx^{rgn}. Remaining parameters use baryonyx default configuration. After every benchmark name, we put between square brackets the generation number where this configuration has been found by rgenoud.

	baryonyx^{rgn}			
	κ_{min}	κ_{step}	δ	θ
SPP (VCS) [10]	0.00	1.88×10^{-4}	3.19×10^{-3}	2.85×10^{-1}
telebus [1]	1.05×10^{-1}	4.10×10^{-4}	1.13×10^{-2}	3.54×10^{-1}
CSPLib022 [0]	0.00	2.00×10^{-4}	1.00×10^{-2}	6.00×10^{-1}
nqueens [1]	3.37×10^{-2}	7.45×10^{-3}	9.04×10^{-3}	7.89×10^{-1}

Table .9: Computational results (objective value) on SPP instances (in bold: training set and best solutions, "-": no solution found)

Instances	Columns	Rows		BARYONYX			cplex	LocalSolver	Beasley[5]	Umetani[31]
			baryonyx	WedelinFast	WedelinGood	baryonyx^{rgn}				
sppaa01	7564	614	56255	57439	57396	56178	**56137**	-	**56137**	**56137**
sppaa02	3875	362	**30494**	**30494**	**30494**	**30494**	**30494**	**30494**	**30494**	**30494**
sppaa03	6776	561	49695	49876	50213	**49649**	**49649**	-	**49649**	**49649**
sppaa04	6139	343	26518	26636	27007	26485	**26374**	44591	**26374**	**26374**
sppaa05	6269	536	54012	54368	54307	**53839**	**53839**	-	**53839**	**53839**
sppaa06	5963	510	27093	27069	27176	**27040**	**27040**	**27040**	**27040**	**27040**
sppnw01	50069	135	**114852**	**114852**	115536	**114852**	**114852**	**114852**	**114852**	N/A
sppnw02	85258	145	**105444**	105684	107934	**105444**	**105444**	202986	**105444**	N/A
sppnw03	38956	53	**24492**	26517	27363	**24492**	**24492**	36402	**24492**	N/A
sppnw04	46189	35	**16862**	19036	18004	**16862**	**16862**	27610	**16862**	N/A
sppnw05	202593	62	132950	133180	133376	132920	**132878**	142110	**132878**	N/A
sppnw06	5956	38	**7810**	8592	9392	**7810**	**7810**	**7810**	**7810**	N/A
sppnw07	3105	34	**5476**	6200	6200	**5476**	**5476**	**5476**	**5476**	N/A
sppnw08	352	21	**35894**	**35894**	**35894**	**35894**	**35894**	**35894**	**35894**	N/A
sppnw09	2301	38	**67760**	**67760**	68402	**67760**	**67760**	**67760**	**67760**	N/A
sppnw10	655	21	**68271**	68286	68286	**68271**	**68271**	**68271**	**68271**	N/A
sppnw11	6482	34	**116256**	116856	117651	116259	**116256**	**116256**	**116256**	N/A
sppnw12	451	25	**14118**	14124	14190	**14118**	**14118**	**14118**	**14118**	N/A
sppnw13	10903	50	**50146**	50158	51934	**50146**	**50146**	**50146**	**50146**	N/A
sppnw14	95172	70	61846	62046	62452	61850	**61844**	66158	**61844**	N/A
sppnw15	441	29	**67743**	67746	67746	**67743**	**67743**	**67743**	**67743**	N/A
sppnw16	138947	135	**1181590**	1181638	**1181590**	**1181590**	**1181590**	**1181590**	**1181590**	N/A
sppnw17	78173	54	**11115**	13698	13083	**11115**	**11115**	27546	**11115**	N/A
sppnw18	8438	108	340162	341518	343538	**340160**	**340160**	**340160**	**340160**	N/A
sppnw19	2134	32	**10898**	11888	11888	**10898**	**10898**	**10898**	**10898**	N/A
sppnw20	536	22	**16812**	17556	17556	**16812**	**16812**	**16812**	**16812**	N/A
sppnw21	426	25	**7408**	**7408**	7436	**7408**	**7408**	**7408**	**7408**	N/A
sppnw22	521	23	**6984**	7144	7244	**6984**	**6984**	**6984**	**6984**	N/A
sppnw23	473	18	**12534**	**12534**	**12534**	**12534**	**12534**	**12534**	**12534**	N/A
sppnw24	926	19	**6314**	6568	6568	**6314**	**6314**	**6314**	**6314**	N/A
sppnw25	844	20	**5960**	**5960**	6286	**5960**	**5960**	**5960**	**5960**	N/A
sppnw26	516	23	**6796**	**6796**	**6796**	**6796**	**6796**	**6796**	**6796**	N/A
sppnw27	817	22	**9933**	**9933**	11091	**9933**	**9933**	**9933**	**9933**	N/A
sppnw28	599	18	**8298**	**8298**	**8298**	**8298**	**8298**	**8298**	**8298**	N/A
sppnw29	2034	18	**4274**	4392	4666	**4274**	**4274**	**4274**	**4274**	N/A
sppnw30	1884	26	**3942**	4450	4294	**3942**	**3942**	**3942**	**3942**	N/A
sppnw31	1823	26	**8038**	8484	8106	**8038**	**8038**	**8038**	**8038**	N/A
sppnw32	227	15	**14877**	**14877**	15120	**14877**	**14877**	**14877**	**14877**	N/A
sppnw33	2415	23	**6678**	6724	6724	**6678**	**6678**	**6678**	**6678**	N/A
sppnw34	736	20	**10488**	**10488**	10701	**10488**	**10488**	**10488**	**10488**	N/A
sppnw35	1403	23	**7216**	7316	**7216**	**7216**	**7216**	**7216**	**7216**	N/A
sppnw36	1408	20	**7314**	7824	8272	**7314**	**7314**	**7314**	**7314**	N/A
sppnw37	639	19	**10068**	10233	**10068**	**10068**	**10068**	**10068**	**10068**	N/A
sppnw38	908	21	**5558**	5688	5592	**5558**	**5558**	**5558**	**5558**	N/A
sppnw39	565	25	**10080**	**10080**	10758	**10080**	**10080**	**10080**	**10080**	N/A
sppnw40	336	19	**10809**	10848	10896	**10809**	**10809**	**10809**	**10809**	N/A
sppnw41	177	17	**11307**	**11307**	11766	**11307**	**11307**	**11307**	**11307**	N/A
sppnw42	893	22	**7656**	7684	7684	**7656**	**7656**	**7656**	**7656**	N/A
sppnw43	982	17	**8904**	9012	9012	**8904**	**8904**	**8904**	**8904**	N/A
sppkl01	5957	47	1091	**1086**	**1086**	**1086**	**1086**	**1086**	**1086**	N/A
sppkl02	16542	69	221	**219**	**219**	**219**	**219**	**219**	**219**	N/A
sppus01	351018	86	10051	10176	10101	10038	**10036**	-	**10036**	**10036**
sppus02	8433	45	5977	6015	6316	**5965**	**5965**	**5965**	**5965**	**5965**
sppus03	23207	50	**5338**	5544	5544	**5338**	**5338**	**5338**	**5338**	**5338**
sppus04	4422	98	**17854**	**17854**	**17854**	17862	**17854**	**17854**	**17854**	**17854**

Table .10: Computational results (objective value) on telebus instances (in bold: training set and best solutions, "-": no solution found)

Instances	Columns	Rows	BARYONYX				cplex	LocalSolver	Borndörfer[7]	Umetani[31]	Bastert [4]	
			baryonyx	WedelinFast	WedelinGood	baryonyx^{rgn}					WedelinFast	WedelinGood
t0415	3172	870	5346798	**5339422**	5349625	5346798	**5339422**	-	5590096	5572626	-	5593065
t0416	3152	974	**6088264**	**6088264**	**6088264**	**6088264**	6093843	-	6130271	**6088264**	-	6095736
t0417	3623	897	**5952247**	5955570	**5952247**	**5952247**	5955570	-	6043157	6024760	-	6034848
t0418	3921	999	**6442906**	**6442906**	**6442906**	6447571	6442908	-	6550898	6446019	-	6470351
t0419	3168	904	5908538	5908538	5908538	5908538	**5907874**	-	5916956	5910913	5910913	5910913
t0420	1847	562	**4153696**	**4153696**	**4153696**	**4153696**	4276444	-	4276444	-	-	-
t0421	1656	557	**4290809**	**4290809**	**4290809**	**4290809**	4340929	-	4354411	**4290809**	-	-
t1716	11952	467	173161	220247	171481	**146871**	169429	330789	161636	165972	181488	306453
t1717	16428	551	192123	196599	178810	**165881**	190181	397845	184692	180757	199024	178948
t1718	16310	523	175772	206736	164318	**153166**	169806	-	162992	174338	197136	175517
t1719	15846	556	195103	223397	189027	**167523**	187717	474975	187677	184354	213194	200215
t1720	16195	538	183560	217548	176376	**154881**	175643	-	172752	181868	183673	188880
t1721	9043	357	135647	134842	128337	**117602**	123567	296830	127424	130047	154951	138819
t1722	6581	338	128574	136612	118822	**113213**	121025	-	122472	114508	132369	125586
v0415	4012	598	2431155	2432482	2432378	2436076	**2429415**	2429420	**2429415**	2429568	2434157	2450287
v0416	10723	812	2732616	2730395	2731091	2739534	**2725602**	2726930	**2725602**	2726156	2731356	2737662
v0417	55232	715	2619345	2613689	2615766	2624641	**2611518**	2617710	**2611518**	**2611518**	2614749	2623710
v0418	4411	742	2847522	2847305	2847075	2852267	**2845425**	2845790	**2845425**	**2845425**	2865326	2864652
v0419	7356	650	2595906	2596932	2591893	2598319	**2590326**	2590330	**2590326**	**2590326**	2597016	2602660
v0420	2350	417	1697940	1697803	1698449	1700336	**1696889**	1696890	**1696889**	**1696889**	1714155	1720504
v0421	823	286	1853951	1854929	1855285	1855170	**1853951**	1853950	**1853951**	**1853951**	1857033	-
v1616	52775	1230	1009279	1016815	1021077	1019797	**1006460**	1066280	**1006460**	1007402	1022969	1029986
v1617	85300	1409	1108113	1119114	1121974	1126315	**1102586**	1216370	**1102586**	1103651	1129989	-
v1618	90805	1396	1159231	1170565	1175678	1173355	**1153871**	1237340	1154458	1155986	1179617	-
v1619	85565	1424	1162669	1169928	1176576	1178951	**1156338**	1248790	**1156338**	1157537	1184010	1189709
v1620	89367	1365	1145978	1154107	1159002	1158319	**1140604**	1233050	**1140604**	1141976	1159246	-
v1621	16606	807	830329	829829	832269	830175	**825563**	838979	**825563**	825605	835860	844014
v1622	10990	736	794530	799151	801465	799438	**793445**	811134	**793445**	793708	801475	-

Table .11: Computational results (objective value) on CSPLib02 instances (in bold: training set and best solutions, ”-”: no solution found).

Instances	Columns	Rows		BARYONYX			cplex	LocalSolver	Curtis [9]
			baryonyx	WEDELINFAST	WEDELINGOOD	baryonyx^{rgn}			
c1a	7543	186	29	**26**	**26**	**26**	**26**	30	34
c1	3829	186	29	**26**	**26**	**26**	**26**	**26**	33
c2	14771	205	33	**29**	**29**	**29**	**29**	**29**	N/A
r1	2503	53	12	**11**	**11**	**11**	**11**	**11**	15
r1a	4273	53	12	**11**	**11**	**11**	**11**	**11**	16
r2	3001	54	**14**	**14**	**14**	**14**	**14**	**14**	17
r3	19091	160	**16**	**16**	**16**	**16**	**16**	-	20
r4	2484	203	27	**25**	**25**	**25**	**25**	**25**	31
r5a	14764	242	31	**28**	**28**	**28**	**28**	**28**	N/A
r5	2202	242	30	**29**	**29**	**29**	**29**	-	N/A
t1	77	24	**7**	**7**	**7**	**7**	**7**	**7**	**7**
t2	3015	125	20	**19**	**19**	**19**	**19**	**19**	N/A

Table .12: Computational results (objective value) on VCS instances (using SPP parameter configuration for baryonyx^{rgn}). "-": no solution found.

Instances	Columns	Rows	baryonyx (1800s)				cplex (1800s)	cplex (36000s)
			baryonyx	WedelinFast	WedelinGood	baryonyx^{rgn}		
VCS1200	127014	1180	849	-	-	819	844	**749**
VCS1600	537412	1576	1449	-	-	1451	-	**1378**

Table .13: Computational results (objective value) on nqueens instances (in bold: training set and best solutions, ”-”: no solution found).

Instances	Columns	Rows		BARYONYX			cplex	LocalSolver
			baryonyx	WEDELINFAST	WEDELINGOOD	baryonyx^{rgn}		
8queens	64	42	**18**	**18**	**18**	**18**	**18**	**18**
10queens	100	54	**25**	**25**	**25**	**25**	**25**	**25**
20queens	400	114	**63**	**63**	**63**	**63**	**63**	**63**
30queens	900	174	108	103	103	103	**101**	144
40queens	1600	234	165	161	164	155	**154**	284
50queens	2500	294	197	190	194	180	**176**	340
500queens	250000	2994	-	8343	6856	**5864**	96859	24509
1000queens	1000000	5994	-	-	20013	**17667**	384140	191532