Automated Tuning of Optimization Software Parameters

University of Pittsburgh Department of Industrial Engineering
Technical Report 2007-7

Mustafa Baz, Brady Hunsaker
Department of Industrial Engineering, University of Pittsburgh, 1048 Benedum Hall, Pittsburgh, PA 15261, USA, {mub3@pitt.edu, hunsaker@engr.pitt.edu}

J. Paul Brooks
Department of Statistical Sciences and Operations Research, Virginia Commonwealth University, Richmond, VA 23284, USA, jpbrooks@vcu.edu

Abhijit Gosavi
Department of Industrial & Systems Engineering, University at Buffalo, State University of New York, Buffalo, NY 14260, USA, agosavi@buffalo.edu

We present a method to tune software parameters using ideas from software testing and machine learning. The method is based on the key observation that for many classes of instances, the software shows improved performance if a few critical parameters have "good" values, although which parameters are critical depends on the class of instances. Our method attempts to find good parameter values using a relatively small number of optimization trials.

We present tests of our method on three MILP solvers: CPLEX, CBC, and GLPK. In these tests, our method always finds parameter values that outperform the default values, in many cases by a significant margin. The improvement in total run time over default performance was generally from 31–88%. We also test similar instances that were not used for training and find similar improvements in run time. Our implementation of the method, Selection Tool for Optimization Parameters (STOP), is available under a free and open-source license.

Key words: parameter tuning, machine learning, design of experiments, artificial intelligence
History:

1. Introduction

Many OR software programs depend on parameters that must be set before the program is executed. The performance of the software often depends significantly on the values chosen for these parameters. In this paper, we present a method that can help users identify good parameter values for their instances. Our research is motivated by Mixed Integer Linear Programming (MILP) solvers in particular, which have a large number of parameters. In this paper, we consider three MILP solvers: GLPK 4.11, CBC 1.01 and CPLEX 9.0. GLPK has 11 algorithmic parameters. CBC has more than 20 parameters, including a growing number of cut parameters. There are dozens of algorithmic parameters in CPLEX 9.0, including 9 kinds of cuts alone.

The number of possible parameter combinations for these solvers is very high. When we consider a subset of just 6 important parameters for CPLEX—4 parameters with 3 values each and 2 parameters with 4 values each—there are 1296 possible combinations. Considering a subset of 10 important parameters yields over 100000 possible combinations. CPLEX provides this great flexibility because parameter settings that are good for one type of instance may be bad for another type of instance. This flexibility is important because it is impossible to find settings that perform well for every class of instances. Our experience is that for any particular class of instances, there are generally settings better than the default settings.

The flexibility provided by having many parameters poses the significant challenge of identifying parameter values that are good for the instances that a user wants to solve. We present a method to help users address this challenge. Our method tries a number of settings on representative instances and reports the best setting observed. We implemented this method in a tool which we call Selection Tool for Optimization Parameters (STOP).

Because it tries many settings, our method requires a significant investment of computer time, though it requires very little of the user's time. There are at least three scenarios when this initial investment of computer time makes sense:

- The user may want to run hundreds of similar instances, so the initial investment of computer time can be recovered from later time savings.
- The user may want to solve instances in "real time", when time is critical. In this situation it may be worth investing lots of "off-line" time with similar instances to better utilize the user's "on-line" time.
- The user may be comparing alternative algorithmic ideas. In this case, he or she may want a fair comparison where all the approaches have good parameter settings. In this case the time spent finding good parameter values is not critical.

This paper is organized as follows: Section 2 describes the problem more formally. Section 3 discusses the key observation that motivates our approaches. Our method and implementation are described in Section 4. In Section 5 computational tests and results are considered. Section 6 presents conclusions.

2. Problem Statement

We first describe the terminology we use in this paper: A software program has several *parameters* controlling its execution. Each parameter is assigned a particular *parameter value*. A set of parameter values—one for each parameter—is a *setting*. Informally, our goal is to find a good setting.

In order to use our method, the user specifies a set of instances to test. It is best to consider several instances with different sizes in order to represent the instance type better and avoid over-fitting the parameters. The user also states the number of settings that may be tried along with an optional solver time limit. These values depend on how much computer time the user is willing to use. The greater the number of settings tried, the more likely it is that better settings will be found. The optional solver time limit is the maximum time that each instance will be run with each setting; this limit avoids wasting a great deal of time on a setting that is clearly bad.

The parameters, parameter values to consider, and metric for comparison are specified in a solver interface file. The user may use an existing interface or may modify an interface to consider parameters and parameter values he or she believes to be important. It is possible to use our method with different metrics, such as solution time or solution quality. In this paper we use the metric of total solution time to prove optimality: the sum of the solution times of the test instances.

Our method assumes that each parameter has a small discrete set of values. This is the case for the MILP solvers we study. Another assumption is that good settings for the test instances will be good settings for similar types of instances. In Section 5 we present computational tests supporting this assumption.

Previous work on automated parameter tuning has often focused on parameters for particular algorithms using methods that do not generalize to the MILP branch-and-cut algorithms that motivate us. Examples of such work include Achard and De Schutter (2006), Haas et al. (2005), and Kohavi and John (1995). Of these, the most general is Kohavi and John (1995), which uses an idea similar to local search, testing settings in a neighborhood of the current setting. Audet and Orban (2006) provide a more general framework for parameter tuning, using mesh adaptive direct search to search through the space of settings. Both of these methods require a notion of parameter values that are "close" to one another, such as numeric parameter values, and the tuning is done by moving through the space of settings. In contrast, the MILP solvers we consider have many parameters that take discrete values with no clear order or relationship, so that moving through the space of settings does not seem natural. Our approach is instead based on an intelligent sampling of settings throughout the space.

3. Key Observation

In preliminary tests we found that there are typically a small number of parameters that significantly change the solution time of instances. However, which parameters are important depends on the instance class. If we can find good values for these few parameters, the solution time is relatively short. Our goal is to find a setting for which the critical parameter values are good, even though we may not know which parameters are critical.

Table 1: Key CPLEX parameters for circuit instances.

	Settings					
CPLEX (circuit)	1	2	3	4	5	Default
MIP emphasis	**1**	**1**	**1**	**1**	**0**	**0**
Node selection	**0**	**0**	**0**	**2**	**2**	**0**
Branching var. sel.	0	0	1	2	1	0
Dive type	1	0	0	3	3	0
Fractional cuts	**0**	**0**	**0**	**0**	**0**	**1**
MIR cuts	0	1	1	0	2	1
Time (s)	63.41	63.42	63.72	72.21	96.49	98.84

To demonstrate this idea, we present Table 1, which shows results for a set of related instances on CPLEX with six different parameter settings based on six parameters. The six parameters considered are MIP emphasis, node selection, branching variable selection, dive type, fractional cut generation (Gomory fractional cuts), and MIR cut generation. The exact purpose of these parameters is not important for this example. More information about them can be found in the CPLEX user documentation. The instances considered are from a circuit problem investigated by several of the authors.

As can be seen from the table, when we have the correct combination of the parameter values 1 (feasibility) for MIP emphasis, 0 (best-bound) for Node Selection and 0 (off) for Fractional Cuts, the solution time is low. When any of these three is not at its best value, the solution time is longer. When two of them or all three of them are not at their best value, the solution time is even longer.

Similarly, Table 2 shows results for a different set of instances on CBC. The seven parameters we explored are strong branching, cost strategy, Gomory cut generation, MIR cut generation, probing, clique cut generation, and feasibility pump. The instances are a subset of the ‘p’ instances from MIPLIB 3.0. In this case, cost strategy, Gomory cut generation, and feasibility pump are the three critical parameters that affect the solution time. The solution time is low for settings 1, 2 and 3 in Table 2 since the key parameters are at their best values. On the other hand, Gomory cut generation for setting 4, Gomory cut generation and feasibility pump for setting 5, and all three of the critical parameters for the default setting are not at their best values.

If the critical parameters (for an instance class) have good values, then the non-critical parameters may take any value and the resulting setting will be “good”. If the number of critical parameters is small, then this means that there are many “good” settings. Our goal is to find one of these settings without too many trials. Note, however, that we do not necessarily expect to determine which parameters are critical for the instance class, since that is not our primary goal.

Table 2: Key CBC parameters for p instances.

	Settings					
CBC (p)	1	2	3	4	5	Default
Strong Branching	0	2	1	2	1	1
Cost strategy	**1**	**1**	**1**	**1**	**1**	**3**
Gomory cuts	**0**	**0**	**0**	**1**	**3**	**3**
MIR cuts	0	3	2	3	2	3
Probing	3	2	0	0	3	3
Clique cuts	3	1	3	1	3	3
Feasibility pump	**0**	**0**	**0**	**0**	**1**	**1**
Time (s)	5.78	5.84	6.43	14.02	17.56	22.46

4. Algorithm

Below is the general framework of our method:

1. Select initial settings to try.
2. Run the solver on those settings.
3. Use some form of machine learning to find additional settings.
4. Run the solver on additional settings.
5. Output the best setting observed.

The first step of the framework is the selection of initial settings. For this, we consider three options: random, greedy heuristic and pairwise coverage. These are discussed in Section 4.1. The next step is to run the solver on those initial settings and record the solution times (or other metrics). Steps 3 and 4 are optional. In step 3, we use machine learning to guide our choice of additional settings based on the initial data. We consider two options for machine learning: regression trees and artificial neural networks. These are discussed in Section 4.2. At the end, we report the best setting observed.

4.1. Selection of Settings

Based on the key observation from Section 3, we would like to have a good "spread" of settings. A motivating idea for selecting settings comes from orthogonal arrays. An orthogonal array of strength t is an $N \times m$ matrix in which the N rows represent settings with m parameters (one per column) such that for any t columns, all possible combinations of parameter values appear equally often in the matrix (Wu and Hamada, 2000). In our case this would mean that every combination of t parameter values appear equally often. Unfortunately, orthogonal arrays do not exist for every combination of parameter and parameter value cardinalities and have not been well-studied in cases where parameters have different numbers of values.

We have tried three methods for selecting settings: pairwise coverage, a greedy heuristic, and random. Pairwise coverage is related to orthogonal arrays. It is widely used in software testing for identifying programming errors that are revealed when two parameter values interact. In our implementation, the goal is to have all pairs of parameter values appear at least twice in settings. An algorithm is used to identify settings that achieve this. If the number of settings needed is higher than the number the user has allowed, then we consider the easier requirement that all pairs appear at least once. We implemented an algorithm from Cohen et al. (1997) to generate the pairwise coverage test sets.

The second alternative is a greedy heuristic. The first setting is selected randomly. For each additional setting, all possible settings are compared with the already selected settings. For each possible setting, we consider the number of parameter values in common with each of the already-selected settings. We choose the new setting that minimizes the maximum number in common. Ties are broken based on the sum over the selected settings of the number of parameter values in common.

The final option for selecting settings is random selection. For each parameter, we select a value uniformly at random. This method is easy to implement, but it may miss some parameter interactions.

4.2. Machine Learning

Machine learning is an optional part of the algorithm. The goal of machine learning for our method is to extract useful information from the initial settings to guide our choice of additional settings by building good probabilistic models. We try two machine learning alternatives: regression trees and artificial neural networks.

4.2.1. Regression Trees

Regression trees are used as a non-parametric method (i.e., no assumptions are made about the distribution of the data) for regression. They are developed as part of classification and regression trees (CART) in Breiman et al. (1984). As in linear regression, regression trees are used to investigate the relationship between a continuous dependent variable and independent variables. The independent variables can be continuous or discrete.

Training data $(\boldsymbol{x_i}, y_i) \subseteq \mathbf{R}^d \times \mathbf{R}$ is used to generate rules for assigning a new observation $\boldsymbol{x} \in \mathbf{R}^d$ a value $y \in \mathbf{R}$. The regression tree algorithm recursively partitions data into observations with high y values and observations with low y values. Each partition is based on a single attribute. The partitions are selected such that the within-partition sums of squares is reduced the most; more precisely, a partition s places training observations in partitions t_L and t_H from partition t. The partition is selected such that

$$\Delta R(s,t) = R(t) - R_{t_L} - R_{t_H}$$

is maximized, where $R(t) = (1/n_t) \sum_{i=1}^{n_t} (y_i - \bar{y}(t))^2$ is the within-partition sum of squares for partition t.

In addition to prediction, regression trees can also be used for feature selection. Attributes that define partitions higher in the tree are likely to be important for determining whether an observation will have a high or low value.

Regression trees for our implementation are generated using the *rpart()* function in the *rpart* package in the R environment for statistical computing (R Development Core Team, 2007).

4.2.2. Artificial Neural Networks

Artificial neural networks use a simple model of multi-layered neurons as a basis for predicting the value of a function. In our case, the function of interest takes the solver parameter values as inputs and gives the total running time as output. The neural network is trained using a number of observed function values (total running times) and then provides predicted running times for other inputs.

One method of training a neural network is the backpropagation algorithm (Werbos, 1974). The backpropagation algorithm of a non-linear neural network essentially uses the concept of steepest gradient descent to minimize the sum of squared deviations of the predicted (fitted) values from the actual values. The power of the algorithm lies in its ability to produce a generic function form that can be evaluated at any given point. It employs the notion of sigmoidal functions that allow such a generic representation. We next describe the mathematics in some detail.

As before let $(\boldsymbol{x}_i, y_i)$ denote the ith piece of data. The neural network model consists of a graph with three layers of nodes: the input layer, the hidden layer, and the output layer. The number of input nodes is p—the number of optimization parameters—while the number of output nodes is one, since we are predicting a single value. Let H denote the number of nodes in the hidden layer (in our implementation we use $H = p$). The network is represented by scalar "weights," which are associated with the links from the input nodes to the hidden nodes and the links from the hidden nodes to the output node. The weight from the jth input node to the hth hidden node will be denoted by $w_1(j,h)$ and that from the hth hidden node to the output node by $w_2(h)$. We will assume that these weights are initialized to some arbitrary (usually small positive) values. Then the predicted value for the ith data piece can be represented according to the neural-network machinery as

$$o_i = w_2(0) + \sum_{h=1}^{H} w_2(h)\pi_i(h), \text{ for } i = 1, 2, \ldots, N \text{ where the sigmoidal functions are}$$

$$\rho_i(h) = \sum_{j=1}^{p} w_1(j,h)x_i(j) \text{ and } \pi_i(h) = \frac{1}{1+\exp(-\rho_i(h))}.$$

The neural network seeks to minimize, via the steepest descent method, the sum of squared errors, i.e.,

$$SSE = \sum_{i=1}^{N}(y_i - o_i)^2.$$

The resulting gradient-descent algorithm, which is implemented in an iterative style, is as follows: For $j = 1, 2, \ldots, p$ and $h = 1, 2, \ldots, H$,

$$w_1(j,h) \leftarrow w_1(j,h) + \mu \left[\sum_{i=1}^{N}(y_i - o_i)w_2(h)\pi_i(h)(1-\pi_i(h))x_i(j)\right],$$

$$w_2(h) \leftarrow w_2(h) + \mu \left[\sum_{i=1}^{N}(y_i - o_i)\pi_i(h)\right], \text{ and } w_2(0) \leftarrow w_2(0) + \mu \left[\sum_{i=1}^{N}(y_i - o_i)\right],$$

where μ denotes the step size.

For more details on artifical neural networks, see Mitchell (1997); Rumelhart et al. (1986). In our implementation we use the free and open-source Fast Artificial Neural Network (FANN) library (Nissen, 2003).

4.3. Implementation Details

We have implemented our approach in a tool we call Selection Tool for Optimization Parameters, or STOP. Given the framework presented at the beginning of Section 4, we now explain the details of our implementation. The solver interface is selected at compile time. It includes the parameters and parameter values to consider as well as the metric to use (in this paper the metric is always total solution time).

At run time, the user specifies the instances to test, the method for selecting settings, the number of initial settings to try, whether and what type of machine learning to use, and the number of additional settings to try if machine learning is being used. There are also several optional arguments, including a time limit on each solver run.

If pairwise coverage is the selection method, then the number of settings needed to cover every pair of values twice is determined. If this number is not more than the number of initial settings, then these settings are used. Otherwise, the number of settings needed to cover every pair of values just once is determined. If this number is not more than the number of initial settings, then these settings are used. In either case, additional settings are generated using the greedy heuristic to reach the specified number of initial settings. If single pairwise coverage still required more than the allowed number of initial settings, then the greedy heuristic is used for all of the initial settings. For example, assume that the pairwise coverage algorithm determines that double coverage requires 23 settings. If the number of initial trials had been specified as 32, then these 23 settings would be used and 9 additional settings would be generated with the greedy heuristic. On the other hand, if only 16 initial trials had been allowed, then the pairwise coverage algorithm would reject double coverage and find settings for single coverage.

If a regression tree is used for machine learning, then the first two branches of the regression tree are examined. These indicate the two most important parameters based on the initial trials. Additional tests are restricted to the parameter values that the regression tree indicates in the "good" branch for these first two branches. The choice of following the first two branches is arbitrary. It may be possible to improve this choice in the algorithm.

If a neural network is used for machine learning, then metric values are predicted for all possible settings. The top ten percent of predictions are made candidates for the additional tests. Again, the choice of ten percent is arbitrary. Additional tests in this case must either be determined by the greedy heuristic or randomly. It is not possible to use pairwise coverage to select the additional settings, since the pool of candidates may not even have every pair present. If pairwise coverage was used for initial settings, then we fall back to the greedy heuristic for the additional settings.

During execution, our implementation prints out the metric found for each test it does; therefore, the user may analyze all of the data found if that is desired. At the end of execution, the best setting is reported and its metric is compared with the metric for the default setting.

As a final option, the user may run a regression tree after all settings have been tested. This does not serve a predictive purpose since no more tests will be done. Instead, the final regression tree provides managerial insight by indicating what the most important parameters seem to be based on all of the settings that have been tested.

5. Computational Tests

We performed tests with three MILP solvers, using a subset of parameters available in each. For some parameters, our interfaces did not consider every possible parameter value. This is because the performance of the algorithm seems to be more sensitive to the numbers of parameter values than to the number of parameters. We do not explore this issue here, however, since the purpose of this paper is to demonstrate the usefulness of the algorithm and not to explore in detail the relationship of its effectiveness to the number of parameters and parameter values.

The exact parameters and parameter values are given below.

- CPLEX 9.0 with six parameters

  - mip emphasis (CPX_PARAM_MIPEMPHASIS): automatic, feasibility, optimality, moving best bound
  - node selection (CPX_PARAM_NODESEL): best-bound, best-estimate, alternative best-estimate
  - branching variable (CPX_PARAM_VARSEL): automatic, pseudo costs, strong branching
  - dive type (CPX_PARAM_DIVETYPE): automatic, traditional dive, probing dive, guided dive
  - fractional cuts (CPX_PARAM_FRACCUTS): off, automatic, on
  - MIR cuts (CPX_PARAM_MIRCUTS): off, automatic, on

- CPLEX 9.0 with ten parameters
  - the six listed above
  - disjunctive cuts (CPX_PARAM_DISJCUTS): off, automatic, on
  - clique cuts (CPX_PARAM_CLIQUES): off, automatic, on
  - node presolve selector (CPX_PARAM_PRESLVND): off, automatic, force
  - probing (CPX_PARAM_PROBE): off, automatic, on
- CBC 1.01 with seven parameters
  - strong branching (strong): 0, 5, 9
  - cost strategy (cost): off, priorities
  - gomory cuts (gomory): off, on, root, ifmove
  - MIR cuts (mixed): off, on, root, ifmove
  - probing (probing): off, on, root, ifmove
  - clique cuts (clique): off, on, root, ifmove
  - feasibility (feas): off, on
- GLPK 4.11 with six parameters
  - scaling (LPX_K_SCALE): none, equilibration, geometric mean, geometric then equilibration
  - pricing (LPX_K_PRICE): largest coefficient, steepest edge
  - branching (LPX_K_BRANCH): first variable, last variable, drieback-tomlin, most fractional
  - backtrack (LPX_K_BTRACK): depth first, breadth first, best projection, best bound
  - lp presolve (LPX_K_PRESOL): off, on

  - method: old b&b (lpx_integer), new b&b (lpx_intopt), new b&b with cuts (lpx_intopt and LPX_K_USECUTS)

We used 4 sets of instances: p and misc instances from MIPLIB 3.0 (Bixby et al., 1998), aircraft landing instances from OR-Library (Beasley, 1990), and circuit instances from research that authors Baz and Hunsaker are currently involved with. The circuit instances map an acyclic graph onto a particular implementation of an electronic fabric architecture.

- p instances: p0033, p0201, p0282
- misc instances: misc03, misc06, misc07
- aircraft landing instances:
  - easy: airland2-R2, airland4-R4, airland7-R1
  - medium: airland4-R2, airland5-R1, airland5-R2
- circuit instances: sobel, laplace, gsm

The primary purposes of this article are to demonstrate that our methods can find settings better than the default settings for a variety of solvers and instance classes and to demonstrate that the resulting settings are also good on other instances from the same instance classes. To this end, Section 5.3 compares the results from STOP with those of the default settings on the tested instances. Section 5.4 looks at the question of whether good settings for the test instances are good for related instances. Prior to those results, we briefly consider two internal issues. Section 5.1 considers the effect of internal randomness in STOP, while Section 5.2 compares some of the many options within STOP.

5.1. Randomness within STOP

For all methods of selecting settings, the first setting is selected randomly, and this selection affects the remaining selections (except in the case that all settings are selected randomly). To investigate whether this randomness has a strong effect on the results, we ran STOP 30 times for the misc instances with CBC and CPLEX with a variety of configurations of STOP. This resulted in 630 final solution times for each solver.

For CBC, the settings found by STOP had total solution times ranging from 21–124 seconds, compared to the default of 129 seconds. However, in fact the times fell into two intervals: 21–26 seconds and 115–124 seconds. Clearly, it is preferable to have a solution in the "good" interval of 21–26 seconds. Of the 630 trials, 82% of them were in this "good" interval. Although the percentage differs depending on the configuration of STOP, even the worst configurations had 60% of their trials in this interval, while many configurations had 90–100% of their trials in the the good interval. With CPLEX the results are not bimodal as they are

Table 3: Best run time found by 21 different configurations of STOP for CPLEX on misc instances.

	Random	Pairwise	Heuristic
No learning 64	41.36	41.37	39.95
Regression tree $48 + 16$	37.94	38.27	36.16
Regression tree $32 + 32$	37.02	38.02	36.05
Regression tree $16 + 48$	39.75	37.62	36.51
Neural net $48 + 16$	36.57	38.32	36.79
Neural net $32 + 32$	37.57	35.26	35.86
Neural net $16 + 48$	44.99	40.09	39.59

with CBC, but 64% of the times are in the range 34–35 seconds while the remaining 36% are in the range 42–69 seconds (compared to the default of 82 seconds).

Based on this preliminary analysis, we note that although randomness may have a significant impact on the settings found, STOP generally has a good chance of finding "better" settings and even the "worse" settings it finds are better than the default. For the remainder of our tests we consider only a single trial.

5.2. Comparing options within STOP

Based on the options for setting selection, machine learning, and number of settings to try, there are an infinite number of ways to run STOP. In this section we informally compare 21 options for running STOP, all of them based on testing a total of 64 settings. We have 3 options for selection of settings: random, pairwise, heuristic. We tested 7 different options for machine learning: no machine learning with $64 + 0$ (the first number in the addition represents the number of initial settings, and the second one is the number of additional settings using machine learning), regression trees with $48 + 16$, with $32 + 32$, with $16 + 48$, neural networks with $48 + 16$, with $32 + 32$, with $16 + 48$.

Table 3 shows the solution times of the tests on misc instances with CPLEX with six parameters. This is a single example for illustration and is not meant to be a conclusive comparison. The default setting takes 81.84 secs. On the other hand, the solution time for the true best setting is 34.00 secs. As can be seen from the Table 3, the results of all 21 options are similar. The longest one takes 44.99 secs and the fastest one 35.26 secs. Results of random selection are generally worse than the other two selection options. We believe the main reason for this is that random selection misses some parameter interaction. The best settings are often found when heuristic option is used.

In general, we found similar behavior in our computational tests. There is not a large difference among the options for selection of settings. However, the best settings are generally observed when pairwise coverage is used. Table 4 shows how often each configuration of STOP found the best setting (out of the 21 configurations for STOP). Out of 11 tests in which all 21 configurations were considered, pairwise coverage found the best

Table 4: Number of times that each configuration was the best out of the 21 configurations.

	Random	Pairwise	Heuristic
No learning 64		1	1
Regression tree $48 + 16$			
Regression tree $32 + 32$			
Regression tree $16 + 48$	2		
Neural net $48 + 16$		2	
Neural net $32 + 32$		2	1
Neural net $16 + 48$	1	1	

setting in 6 of them. The random option found the best setting in 3 tests, whereas the heuristic option found the best in 2 tests.

Machine learning helps to find the best setting. In 9 out of the 11 tests the best settings were observed when one of the machine learning options is used. Neural networks were used when finding the best setting in 7 tests, whereas regression trees were used when finding the best setting in 2 tests. For the remaining 2 tests the best setting is observed when machine learning is not used.

Based on this preliminary analysis, we focus our tests on two of the configurations, both using pairwise coverage and a neural network. We refer to these as $p32nn32$ (for the $32 + 32$ option) and $p48nn16$ (for the $48 + 16$ option).

5.3. Comparison with default settings

In this section we compare the solution times of the settings obtained from STOP with the times of the default settings of the MIP solvers. Based on the analysis from the previous section, we focus our attention on two possible options for STOP: $p32nn32$ and $p48nn16$. The p refers to the choice of pairwise coverage, the nn refers to the use of a neural network, while the first and second numbers indicate how many settings are tested before and after machine learning is performed, respectively.

In Table 5, Table 6 and Table 7 we show the solution times for the default settings and percentage improvement of STOP settings found with $p32nn32$ and $p48nn16$. In cases where we tested all 21 configurations for STOP, we also report the percentage improvement for the worst of the 21 configurations, and the percentage improvement for the best of the 21 configurations. To demonstrate the potential usefulness of STOP, however, we believe that it makes more sense to look at the performance of a single option, such as $p32nn32$ or $p48nn16$, across all solvers and instance classes.

For p and misc instances all 1296 ($= 3^4 * 4^2$) settings were tested for CPLEX with six parameters to find the true best solution time. For p instances, we also tested all 104976 settings for CPLEX with ten parameters. We did these exhaustive tests in order to see how STOP compares to the true best.

Table 5: Solution times comparison on CPLEX. Note that “Worst” and “Best” refer to the setting reported by STOP with the worst and best of the 21 configurations.

			STOP Methods (% improvement)				
CPLEX		Default (s)	Worst	p32nn32	p48nn16	Best	True best
p	6 par.	0.6059	32.9%	34.3%	35.2%	35.2%	36.0%
	10 par.	0.6109	33.0%	35.2%	36.2%	36.6%	41.6%
misc	6 par.	81.84	45.0%	56.9%	53.2%	56.9%	58.5%
	10 par.	81.41	36.4%	40.7%	47.3%	58.5%	unknown
circuit	6 par.	98.84	2.4%	35.4%	33.5%	35.9%	unknown
aircraft-med	6 par.	505.79	60.7%	69.7%	66.2%	69.8%	unknown
aircraft-easy	6 par.	1.62	35.2%	54.9%	54.3%	55.6%	unknown

Table 6: Solution times comparison on CBC.

		STOP Methods (% improvement)			
CBC	Default (s)	Worst	p32nn32	p48nn16	Best
p	22.57	74.1%	74.5%	74.3%	74.5%
misc	129.37	52.6%	77.9%	82.7%	82.7%
circuit	58.76	1.0%	1.0%	3.0%	3.3%
aircraft-easy	137.28	87.7%	88.2%	87.7%	88.2%

Table 7: Solution times comparison on GLPK. Because of long running times, only two configurations of STOP were tested.

		STOP Methods (% improvement)	
GLPK	Default (s)	p32nn32	p48nn16
p	3.08	45.8%	30.8%
misc	533.71	87.2%	87.2%
circuit	593.26	63.0%	63.1%
aircraft-easy	2024.35	99.5%	99.4%

Table 8: CPLEX default settings and the suggested settings of STOP.

Parameter	Default	Circuit	Aircraft-med
MIP emphasis	automatic	feasibility	feasibility
Node selection	best-bound	best-bound	best-bound
Branching var. sel.	automatic	automatic	pseudo costs
Dive type	automatic	traditional dive	traditional dive
Fractional cuts	automatic	off	on
MIR cuts	automatic	off	automatic

Table 9: CBC default settings and the suggested settings of STOP.

Parameter	Default	Circuit	Aircraft-easy
Strong branching	5	0	0
Cost strategy	off	priorities	priorities
Gomory cuts	ifmove	off	root
MIR cuts	ifmove	ifmove	root
Probing	ifmove	off	on
Clique cuts	ifmove	on	root
Feasibility pump	on	on	off

As can be seen from Table 5, Table 6 and Table 7, STOP's worst method's settings are always better than the default settings. Moreover, the reference configurations of $p32nn32$ and $p48nn16$ find settings that are often much faster than the default settings. In only one case, circuit instances on CBC (Table 6), does STOP fail to find a significant improvement over the default setting. However, the solution time for the default setting is faster than the best method of CPLEX, 63.31 (Table 5). This indicates that the default CBC setting is good for the circuit instances, which is why STOP was not able to find a much better setting.

5.4. Tests on Similar Instances

In Section 2 we stated that an assumption for our approach is that good settings for the test instances will be good settings for similar type of instances. To test this assumption and to see how well STOP behaves, we used the settings reported from STOP to test similar instances.

We considered three instances (sobel, laplace, adpcm-decoder) to find better parameter settings for the circuit project and three instances (airland4-R2, airland5-R1, airland5-R2) for aircraft landing. The default setting and the suggested setting of STOP can be seen in the Tables 8, 9, and 10. We did not perform additional tests with p or misc instances since there are not many additional instances from those classes.

Tables 11, 12, and 13 show the solution times of the default settings, STOP's suggested settings and the percentage change in time over the default setting for the circuit instances. We performed these tests with a one hour time limit. In every case for which an optimal solution was found, STOP's settings lead to a

Table 10: GLPK default settings and the suggested settings of STOP.

Parameter	Default	Circuit	Aircraft-easy
Scaling	equilibration	equilibration	equilibration
Pricing	steepest edge	steepest edge	largest coeff
Branching	driebeck-tomlin	most fractional	last var
Backtrack	best projection	best bound	breadth first
LP presolve	off	on	on
Method	new b&b	old b& b	new b& b with cuts

Table 11: (CPLEX) The solution times comparison between the default settings and STOP's settings for the circuit project.

Instance	Default	STOP	% Change
gsm	168.17	81.64	51.45
idctcol	578.57	358.24	38.08
idctrow	238.41	152.70	35.95
adpcm-coder	386.90	243.62	37.03

shorter solution time.

Tables 14, 15, and 16 show the solution times comparison between the default settings and STOP's settings for the aircraft landing instances. In only two cases did instances have longer solution times with the STOP settings, and in both these cases the difference is less than two seconds. In many cases, however, the improvement by STOP is also insignificant. However, when the difference is significant STOP does significantly better, such as with airland8-R2 with CPLEX and airland6-R2 with GLPK. The results for CBC are the least impressive, though even in this case the setting recommended by STOP is slightly better than the default.

6. Conclusion

We have presented a method to tune software parameters using ideas from software testing and machine learning. The method is based on the key observation that for many classes of instances, results will be good

Table 12: (CBC) The solution times comparison between the default settings and STOP's settings for the circuit project.

Instance	Default	STOP	% Change
gsm	33.09	31.34	5.28
idctcol	3600	3600	0.00
idctrow	3456.58	2466.63	28.63
adpcm-coder	277.67	16.61	94.01

Table 13: (GLPK) The solution times comparison between the default settings and STOP's settings for the circuit project.

Instance	Default	STOP	% Change
gsm	1568	661.61	57.80
idctcol	3600	3600	0.00
idctrow	891.33	696.81	21.82
adpcm-coder	3600	3600	0.00

Table 14: (CPLEX) The solution times comparison between the default settings and STOP's settings for the aircraft landing.

Instance	Default	STOP	% Change
airland1-R1	0.05	0.04	20.00
airland2-R3	0.21	0.17	19.05
airland3-R2	0.61	1.44	-136.07
airland4-R3	64.38	59.34	7.83
airland5-R3	93.52	76.63	18.06
airland6-R2	4.51	2.39	47.01
airland7-R1	0.58	0.56	3.45
airland8-R1	2.83	2.83	0.00
airland8-R2	14268.83	3372.61	76.36
airland8-R3	11.56	8.55	26.04

Table 15: (CBC) The solution times comparison between the default settings and STOP's settings for the aircraft landing.

Instance	Default	STOP	% Change
airland1-R1	0.30	0.26	13.33
airland2-R3	9.96	4.34	56.42
airland3-R2	5.27	4.88	7.40
airland4-R3	3600	3600	0.00
airland5-R3	3600	3600	0.00
airland6-R2	11.16	12.70	-13.79
airland7-R1	0.77	0.74	3.89
airland8-R1	453.44	393.38	13.24
airland8-R2	3600	3600	0.00
airland8-R3	3600	3600	0.00

Table 16: (GLPK) The solution times comparison between the default settings and STOP's settings for the aircraft landing.

Instance	Default	STOP	% Change
airland1-R1	0.43	0.18	58.13
airland2-R3	1.48	0.86	41.89
airland3-R2	3600	4.23	$\geq$99.88
airland4-R3	3600	3600	0.00
airland5-R3	3600	3600	0.00
airland6-R2	3600	551.11	$\geq$ 84.69
airland7-R1	69.48	5.46	92.14
airland8-R1	3600	3600	0.00
airland8-R2	3600	3600	0.00
airland8-R3	620.05	505.99	18.39

if a few critical parameters have good values, though the set of critical parameters depends on the class of instances.

In our computational tests of MILP solvers, the worst result our implementation reported out of 21 configurations is always better than the solution time of the default setting. Moreover, the solution times achieved by two reference configurations are often much faster than the solution time of the default setting. Our tests of related instances supports the assumption that good settings for the test instances will also be good for similar instances.

We leave to future work a careful study of configuring STOP itself, including the possibility of a more sophisticated use of machine learning. Based on the tests for this paper, we felt that none of the three options for selecting settings appears to dominate the others, but the best settings are often observed when pairwise coverage is used. Machine learning certainly helps to find the best setting. The two reference configurations of $p32nn32$ and $p48nn16$ appear to be good choices.

Although developed for MILP solvers, our method is flexible and may be applicable to other algorithms for which the key observation is valid. In addition, the method can be easily extended and may be useful with different metrics, such as the maximum time for several instances, the percent gap after a fixed amount of time, or the solution quality for a heuristic algorithm.

Our implementation of the method, STOP, has been released under a free and open-source license.

Acknowledgments

Work by authors Hunsaker and Baz was supported in part by an IBM Faculty Partnership Award.

References

Achard, P., E. De Schutter. 2006. Complex parameter landscape for a complex neuron model. *PLoS Comput Biol* **2**.

Audet, Charles, Dominique Orban. 2006. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM J. on Optimization* **17** 642–664.

Beasley, J. E. 1990. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society* **41** 1069–1072.

Bixby, R. E., S. Ceria, C. M. McZeal, M. W. P. Savelsbergh. 1998. An updated mixed integer programming library: MIPLIB 3.0. *Optima* **58** 12–15.

Breiman, L., J.H. Friedman, R.A. Olshen, C.J. Stone. 1984. *Classification and Regression Trees*. Chapman & Hall (Wadsworth, Inc.), New York.

Cohen, David M., Siddhartha R. Dalal, Michael L. Fredman, Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.* **23** 437–444.

Haas, Joseph, Maxim Peysakhov, Spiros Mancoridis. 2005. GA-based parameter tuning for multi-agent systems. Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, Eckart Zitzler, eds., *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, vol. 1. ACM Press, Washington DC, USA, 1085–1086.

Kohavi, Ron, George John. 1995. Automatic parameter selection by minimizing estimated error. Armand Prieditis, Stuart Russell, eds., *Machine Learning: Proceedings of the Twelfth International Conference*. Morgan Kaufmann, 304–312. URL `citeseer.ist.psu.edu/kohavi95automatic.html`.

Mitchell, T.M. 1997. *Machine Learning*. McGraw Hill, Boston.

Nissen, S. 2003. Implementation of a fast artificial neural network library (fann). Tech. rep., Department of Computer Science University of Copenhagen (DIKU). Http://fann.sf.net.

R Development Core Team. 2007. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Rumelhart, D., G. Hinton, R. Williams. 1986. Learning internal representations by error propagation. D. Rumelhart, J.L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 318–362.

Werbos, P.J. 1974. Beyond regression. Ph.D. thesis, Harvard University.

Wu, C. F. Jeff, Michael Hamada. 2000. *Experiments; Planning, Analysis, and Parameter Design Optimization*. John Wiley & Sons, New York.