Journal of Global Optimization manuscript No.
(will be inserted by the editor)

PAVER 2.0: An Open Source Environment for Automated Performance Analysis of Benchmarking Data

Michael R. Bussieck · Steven P. Dirkse · Stefan Vigerske

Received: date / Accepted: date

Abstract In this paper we describe PAVER 2.0, an environment (i.e. a process and a suite of tools supporting that process) for the automated performance analysis of benchmarking data. This new environment improves on its predecessor by addressing some of the shortcomings of the original PAVER [6] and extending its capabilities. The changes serve to further the original goals of PAVER (automation of the visualization and summarization of benchmarking data) while making the environment more accessible for the use of and modification by the entire community of potential users. In particular, we have targeted the end-users of optimization software, as they are best able to make the many subjective choices necessary to produce impactful results when benchmarking optimization software. We illustrate with some sample analyses conducted via PAVER 2.0.

Keywords Benchmarking · Performance data analysis · Performance metrics

1 Introduction

Benchmarking is an important tool used in developing and evaluating solvers for mathematical programming problems. The process involves many steps or components, including the collection of suitable test problem instances, running solvers to get performance data, and reporting on and analyzing the data obtained. While there is a wealth of available content about the first steps (e.g. [5,10,17,20,21]), relatively little is available about the reporting and analysis of results [2,3,4,8,9, 24]. The original PAVER addressed that issue and has become a useful suite of tools [12], automating the process of analyzing and visualizing the performance data (e.g. objective values, solution times, and result codes) from multiple runs in order to summarize, gain insight, or detect inconsistencies.

As time has passed, limitations in the original PAVER have become apparent, as has the need for new features. Some of these issues have been addressed by making incremental changes to PAVER, but ultimately the decision was made

GAMS Development Corp.
Washington, DC 20007
E-mail: mbussieck@gams.com, sdirkse@gams.com, svigerske@gams.com

to undertake a complete rewrite of PAVER and at the same time to rewrite the Examiner tool for solution verification so that it can function as part of PAVER. We are also taking advantage of this opportunity and making the new PAVER and Examiner tools open-source as part of the COIN-OR initiative, with all of the advantages (in short, higher-quality, freely-available software) [7] that derive from such an arrangement.

A major aim of the new PAVER was to implement an infrastructure that allows simple, convenient access to a variety of different types of performance metrics without hardcoding all parameters of any specific metric. Currently supported metrics include those based on *counting* solver runs with certain properties, computing *mean values* and *quantiles* of solver run attributes, and *performance profiles*. The choice of which metric to include (e.g. those based on solving times, iteration numbers, or certain optimality gaps) is not part of the PAVER core, but instead can be based on the available data and is specified via a separate PAVER setup object. The hope is that such a design allows for easy adaptation and extension by end users as their needs grow and evolve. An additional aim was the possibility to check benchmarking data for failures, i.e., whether the results reported by a solver are consistent, optionally taking known data about an optimization instance into account, to report recognized inconsistencies back to the user, and to (optionally) exclude such failures from the actual benchmark.

While consistency checks based on the comparison of reported bounds on the objective value with known ones are simple to implement, checking whether a reported solution point is indeed feasible may be a more useful test from a practical point of view. These and related checks are implemented by the Examiner tool. This tool provides a flexible way to measure and report on the many different metrics for solution correctness/validity, chiefly feasibility and local optimality. The simple question, "Is the reported solution *really* a solution?" may not have a simple answer. The Examiner tool brings some structure and uniformity to the process of finding an answer.

In Section 2, we describe the original PAVER tool, while Section 3 describes the design and implementation of the completely rewritten PAVER. Where necessary we explicitly distinguish between the original and revamped PAVER. In most cases a reference to PAVER implies the revamped PAVER, aka PAVER 2.0. The rationale for, design, and function of the Examiner tool for solution verification is covered in Section 4. In Section 5 we discuss the reasons why people do benchmarking and how that has influenced our decision-making, especially our decision to focus on making it possible for end-users to conduct their own benchmarks. Section 6 presents two benchmark examples and supporting discussion, while the concluding Section 7 contains comments on future work.

PAVER 2.0 is available at `http://www.gamsworld.org/performance/paver2`. In addition to the Python source, it also includes the necessary data and scripts to reproduce the benchmark examples from Section 6.

2 PAVER 1.0 Review

PAVER was created as part of a quality assurance (QA) initiative started more than 10 years ago when GAMS introduced the first set of global optimization solvers into its system [6]. These QA efforts resulted in tools for *data collection*

and analysis and a comprehensive compilation of *test cases* made available at GAMS World's Performance World[1]. PAVER is the keystone of the data analysis instruments (also called *performance tools*) providing a variety of performance measurement tools with convenient and easy ways to represent results. The online *PAVER server*[2] [23] tool provides a performance analysis of benchmarking data collected in trace files (see Section 3.2) and uploaded by an anonymous web user. PAVER generates several results from the benchmarking data:

- The *solver square* allows for quick identification of models where two solvers disagree in the result code (e.g. local infeasible versus local optimal).
- *Resource time comparisons* between two solvers place each model into one of several buckets: cases that solved in nearly the same amount of time, where one solver was faster than the other, and one solver was much faster than the other. Within each of these buckets models are again partitioned, this time by objective value, into cases where the objectives are the same (within tolerance), A finds a better objective than B, and vice versa.
- *Performance profiles* [9] are cumulative distribution functions over a given performance metric and give perhaps the most complete information in terms of robustness, efficiency, and solution quality.

PAVER was implemented as a mix of AWK, GAMS (as a scripting language), and Gnuplot scripts that were easy to create but in the long run extremely difficult to maintain. Removal of design deficits (e.g., limitation to 8 trace files, need to manually edit trace files from runs with a single solver but with multiple options, limitation to 247 characters for the name of two trace files combined, requirement to have the trace files in the same directory as the performance tools, treating unbounded or infeasible instances as failure, etc.) and improvement suggestions (e.g., recognition of inconsistencies, availability of additional performance metrics, adaptivity and extensibility for purposes that go beyond comparing objective values and solution times, better performance, independence of GAMS, etc.) from PAVER users could not be implemented with a justifiable effort within the original design framework.

3 Updated infrastructure

Thus, to overcome the limitations of the first PAVER implementation, we have undertaken a complete rewrite. We have chosen to implement in Python, since it is a powerful scripting language that is freely and widely available, easy to learn, and offers extension packages for almost any purpose. One such extension is the *pandas* library[3], which provides optimized data structures and data analysis tools for the kind of calculations that we intent to do with PAVER.

PAVER currently produces output in the form of HTML and simple text. It can be used as command line tool or as a Python package. PAVER consists of the following components.

[1] `http://www.gamsworld.org/performance`
[2] `http://www.gamsworld.org/performance/paver`
[3] `http://pandas.pydata.org`

3.1 Data

All input data is stored in pandas data structures (DataFrame and Panel). On the highest level, we distinguish solver run identifiers. These identifier may contain the name of an option file, if one solver has been run with different settings, and a run identifier if a solver has been run several times on an instance. Several runs can be useful to guard against high sensitivity of a solver's performance on initial conditions like the order of variables and constraints or the choice of an initial seed of a random number generator [2,17]. PAVER then usually aggregates the various runs for a solver on an instance into a single record, e.g., by averaging solving times. This averaged record is then used for comparisons with other solvers.

For each solver identifier, we store a table that provides for each problem instance the values of certain solve attributes. So far, the only mandatory attribute for a solver run is the termination status, i.e., the reason why a solver stopped. This could be because the solver completed its algorithm, because a certain limit (e.g., time, iterations, memory) has been reached, because it rejected the instance due to capability problems, or others. A solver run may also contain the primal and dual bounds on the objective value, the solving time, the number of iterations or branch-and-bound nodes, etc. Since these attributes are optional, routines that evaluate the solver runs need to be prepared for missing data.

For each problem instance, we also store several statistics (e.g. the number of variables, discrete variables, constraints, and Jacobian nonzeros). If available, we further store information on the optimal value of a problem instance or bounds on this value. For a minimization (maximization) instance, we denote an upper (lower) bound on the optimal value by *primal bound* (i.e., the objective value of a best known feasible solution) and a lower (upper) bound on the optimal value by *dual bound*, see also [1].

After all input data has been assembled, we compute derived data. For example, from the final primal and dual bounds reported for a solver run, we compute the gap between these values and the gap to the optimal value of the corresponding instance, if known. The gap between primal (dual) bound and optimal value is denoted by *primal* (*dual*) *gap* [3].

If information on the progress of primal and dual bounds during the solver run is available, we also compute the *primal integral*, *dual integral*, and *gap integral* [3]. The primal integral is the integral of the relative gap between primal bound and optimal value for an instance over the solving time. A small primal integral indicates that a solver found a good feasible solution early in the search. Analogous statements can be made for the dual integral (measuring the distance of dual bound and optimal value) and the gap integral (measuring the distance of primal and dual bound).

3.2 File parser

File parsers are responsible for reading solving and benchmarking related data from a file. So far, PAVER can read comma separated value (csv) files that contain in each line the outcome of a solver run (instance name, solver name, options file name, instance statistics, solving statistics, ...). With GAMS, files in this format can be generated with the options `trace=file.trc` and `traceopt=3` (or `5`).

The Examiner tool (see Section 4) can generate csv files which contain the information above and additional information about the primal and dual infeasibility and complementarity gaps for the solution reported by a solver.

PAVER can also read csv files that report for a single solver run the progress of primal and dual bound over time or number of enumerated branch-and-bound nodes. Several of the GAMS solver links can write these files via the `miptrace` and `solvetrace` options.

Finally, PAVER can read information about the optimal value of a problem instance or bounds on that value from a *solution file*. For this input, we use the same syntax as in [24].

Additional parsers can easily be added to the tool. The next parser will be for result files from the Optimization Services project at COIN-OR.

3.3 Consistency Checks

Consistency checks check the solving data for obvious inconsistencies. As a result of such checks, either a single solver run or all solver runs for a specific instance can be marked as inconsistent. PAVER stores an explanatory string along with each inconsistency marker used when generating reports or displaying solver outcomes.

In situations where the primal and dual bounds for a solver run contradict known primal or dual bounds for that instance, only the particular solver run is marked as inconsistent. If the bounds reported by two solver runs contradict each other and known bounds are not available or do not serve to invalidate one of the solver results, the complete instance is marked as inconsistent, since it is not clear which solver reported a wrong result.

Additionally, PAVER can check that the infeasibilities reported by the solution checker Examiner do not exceed certain tolerances. See Section 4 for more details on this and Section 6.2 for an example. Finally, PAVER marks instances that terminated due to capability problems, an error, or an unknown reason as inconsistent, so they can be easily excluded from evaluations.

3.4 Statistics Generator

The statistics generator takes a set of *metrics* as input. The purpose of a metric is to specify the evaluation of a solve attribute (e.g., solving time, termination status, gap at termination) in a single object. Thus, a metric specifies a solve attribute, a number of parameters, and a set of filters on the list of instances for which statistical measures are computed and visualized. Filtering can be used, for example, to compute statistics only with respect to instances where no solver reported inconsistent results, which were solved to optimality by all solvers, or where all solvers found an optimal solution. The parameters of a metric specify which statistical measures are computed for the corresponding solve attribute and each filtered set of instances.

For the solve attribute and each filter of a metric, the following statistics can be computed (using standard pandas functionality): a count on the number of instances where a value for the attribute is available, arithmetic, geometric, and

shifted geometric means and standard deviations[4] and minimal and maximal values of the attribute and intermediate quantiles (e.g., 10%, 25%, median, 75%, and 90%). Further, visualizations of counts and means by bar charts and quantiles by boxplots are generated.

Next to the real solver runs, the *virtually best* and *virtually worst* solvers, given by taking, for each instance, the best and worst value of the attribute, respectively, are also included into the evaluation of statical measures. This allows, for example, to count how many instances were solved by at least one solver and how many were solved by all solvers and to compare this number to the number of solved instances by each single solver.

To further facilitate solver comparisons, the user can designate some or all solvers as reference solvers. By default, the virtually best solver is chosen for this role. For each reference solver and each attribute value, PAVER computes the ratios of each solver's attribute value to that of the reference solver. For these ratios, statistical measures like arithmetic means and quantiles are again computed. Additionally, PAVER calculates the number of instances for which a solver's attribute value is better than, similar to, or worse than the reference solver's value, where the classification into better/worse than or similar to is done w.r.t. user-specified absolute and relative tolerances. Again, bar charts and boxplots are generated for visualization. These comparisons to reference solvers mimic the pairwise solver comparisons of the *solver square* and *resource time* utility of PAVER 1.0.

Finally, the new PAVER can also compute and visualize performance profiles of the sort introduced in [9]. For a solve attribute and an instance filtering, PAVER can compute both relative and absolute profiles, i.e., those showing either the outcome of a solver in relation to the best outcome on that instance among all solvers, or those showing the absolute solver's outcome. Additionally, extended performance profiles as suggested by [18] can be generated. While the usual profiles only indicate for how many instances a solver was at most x times slower than the best solver, the extended profiles also show the number of instances for which a solver was at least x times faster than all other solvers. As before, the performance for the virtually best and worst solvers are also included.

3.5 Writer for Solving and Instance Data

The main purpose of the writer for solving data is to display the instance and solving data in a HTML or text table. In imitation of the display columns facility of SCIP [1], the writer routine takes an object for each column as input. The column object determines which data should be printed in its column and in which form by implementing methods to return the header of the column, the unit, the alignment, and a string, a color, and (optionally) a number for a certain row. In the HTML output, instances or solver runs that were found to be inconsistent are also marked by a light red background color, with a mouse-over text that provides additional explanation and detail. Optionally, the numbers in a column can be visualized in a bar chart.

[4] While arithmetic means are sensitive to variations of data with relatively large range and insensitive to variations of data with relatively small range, geometric means are more sensitive to variations close to zero. As a compromise, PAVER can also compute *shifted geometric means* [1], which reduce the effect of data points close to zero in the geometric mean by shifting.

3.6 Setup

A PAVER setup determines which consistency checks should be run, which metrics should be evaluated, and which columns should be included in the output of the solving and instance data. The default setup tries to determine useful metrics from the input data and the values of the command line parameters. These metrics evaluate the inconsistency markers and attributes like solving time, number of nodes, number of iterations, primal and dual bound / gap / integral, gap (between primal and dual bound), and gap integral. Additionally, a user can specify further solve attributes to be evaluated via the `--eval` option.

For filtering, the default setup considers an empty filter (i.e., do not exclude any instance) and filters that only include instances which did not fail, where the gap (between primal and dual bound) is below the gap tolerance (10^{-6} by default), below 1%, or below 10%, or instances with primal bound at most 10^{-6}, 1%, or 10% worse than a known optimal value. For means and quantiles, we usually require that such criteria need to hold for all instances in order to include them into the filter. However, a filter for performance profiles usually applies these criteria to each solver separately, thereby implicitly assuming an infinitely bad performance if a solver does not satisfy a criterion on an instance.

4 Solution verification

It's tempting to avoid the issue of solution verification entirely and simply accept all solver claims as valid. In fact, this is often what people do in practice. In some cases, where trusted and well-tested codes are being compared and where simplicity and ease of execution are the primary factors, such an approach is justified. However, one quickly discovers many uses for an independent solution verification tool.

Frequently, such tools are most useful when developing solvers and related code. Of course, errors in an algorithm or its coding can lead to false solutions. A perfect solver will stumble if the automatic differentiation techniques used deliver incorrect derivatives. And even when the solution is correct in one part of a code, it is very easy to make mistakes in solution reporting or transferring/translating solutions from one component to another. With the gross errors common in the development phase of optimization software, there is usually little doubt that the reported solution is in error, and those responsible for fixing the problem are usually grateful to have it pointed out to them.

In contrast, verification tools may be used to check if the solution satisfies a set of specified tolerances. Solvers typically do not report exact solutions, but rather a solution that satisfies tolerances for feasibility and optimality. The default tolerances vary from solver to solver, and more importantly, the exact definition of what metrics (measurements of feasibility, optimality, and complementarity) are compared to these tolerances vary as well. In practical work this is usually not an issue, although there are cases where a user will notice that a solver may return a solution that is "too loose". When running a benchmark, however, this is a more important topic, since stopping early in a systematic fashion by loosening tolerances will tend to make a solver look like a better performer. For this reason

it is useful to have an independent check that solutions are meeting a well-defined standard before they are accepted as feasible and optimal.

Others have previously recognized the need for such tools. Dolan, Moré and Munson [11] pointed out the bias that results when different convergence tests are used by competing solvers in a benchmark and proposed a specific convergence test to be applied a posteriori in order to remove this bias. They also present experimental evidence of the different performance profiles that result when solvers are required to satisfy a uniform convergence test. More recently, a solution checker has been added to the MIPLIB 2010 problem suite [17]. This checker is intended for use on MIP models so it avoids optimality checks altogether, checking only that the claimed solution point is feasible and consistent with the reported objective value. It performs this check in full precision using the GMP arbitrary precision arithmetic package [15]. We see in these references the necessity of adjusting the validation procedure based on the problem type and solution information returned by a solver. For continuous optimization we can check feasibility and local optimality, and, assuming convexity, global optimality as well. For problems with discrete variables we can check feasibility. If the solver returns dual information for the continuous subproblem defined by fixing discrete variables, local optimality can be checked as well. However, this check for local optimality is of lesser importance in discrete optimization, and checking global optimality can be as difficult as solving the problem.

The solution verification tool in the GAMS system, developed in part for the reasons given above, is the Examiner utility [14]. As part of our revamp of PAVER we have taken the core of the Examiner utility, the part not specific to GAMS, and broken it out as a standalone library, which we intend to make available as a COIN-OR project. We believe that making the code public in this way will have several advantages to the optimization community, including ourselves. It will be useful as an objective standard by which solutions can be verified when benchmarking solvers. Such use will be especially convenient when benchmarking solvers that, like Examiner, belong to COIN-OR. To illustrate this, we have created an interface between the Examiner library and the Optimization Services (OS) project in COIN-OR. This interface allows to pass model instances in the OSiL language and their solutions in the OSrL result to the Examiner library for verification. We also hope that solver developers find it as useful as we have when doing internal quality control. As Examiner developers, we have found that making Examiner open-source has been an effective driver for finding a clean separation of the Examiner library and the GAMS/Examiner link and for creating a project we can be pleased to have our name on. We also hope to benefit from an exchange of ideas and perhaps contributions of code that is only possible with an open-source project, so that Examiner is making the right checks in the best way possible.

The Examiner library is structured and used similarly to the solvers whose solutions it will be examining. This makes it easy to introduce solution validation via Examiner as a component of any benchmarking exercise. The model is loaded into Examiner as a group of arrays for linear models and additionally via a nonlinear function callback for nonlinear models. Solutions to be checked are passed in via subsequent calls. By default, the Examiner check for optimization models includes the following metrics: feasibility for primal and dual variables, feasibility for primal and dual constraints, and the primal and dual complementarity gaps. If desired, a particular subset of these checks can be selected. All tolerances to use for

these checks take reasonable if conservative defaults and can be set by the user. A number of different schemes for displaying and querying the results are supported. For example, when developing a model or debugging a solver, Examiner's log output showing the location and values for the largest errors exceeding the tolerance is quite useful, but when validating benchmarking results from many instances a simpler report indicating success or failure or simply logging the computed values of the various metrics for use in a later check is more appropriate.

When a solution fails a validity check due to a tolerance issue, it is not necessarily the case that an error has occurred. The right choice for the tolerances is always a subject of debate, and one must also consider that Examiner is operating on the original model formulation, while solvers perform their termination checks on a modified representation, typically one that has been presolved and scaled. There are both practical and philosophical reasons for Examiner to make its checks on the original model: many solvers don't make the scaled, presolved version of the model available, and even if they did, the quality of the solution in the original space is likely to be of more interest to the user. However, scaling is such an important factor that Examiner can optionally do a scaled check, after making a row scaling of the original model. In addition to checking a model that we expect is more similar to the solver's internal formulation, this check also identifies the row with maximum scale, which may be useful in adjusting the model during development or in explaining why a solver was unable to compute a sufficiently precise solution.

5 Subjectivity, reproducibility, and automation

It's useful now to take a step back and consider some different motivations for benchmarking optimization software:

- To find the "best" solver or to rank solvers. Historically, this has usually been done on the basis of speed but robustness and solution quality are also important considerations.
- To compare development versions of particular solvers with earlier, stable release versions. Such benchmarks are done by the solver developer to see if and by how much the new version improves over the old but also to discover cases where performance drops.
- To compare several different option settings. A solver developer might do this to choose good default values, while an end user might do this in an attempt to get better overall performance.
- To choose a solver or set of solvers to use for a particular application. In this case one may wish to rank solver combinations, especially if robustness and solution quality are primary concerns.

In each of the cases above, the outcome is determined not only by the attributes of the solvers being tested but by many subjective choices that go into the benchmark. Such choices include the set of problem instances to run, the computing environment to use, and the time limit to use. Once the raw data are obtained, there are still many choices to make: how to treat solver failure or time elapsed outcomes, what rules and tolerances to use in validating solutions, how to weight speed vs. solution quality, what other performance measures to consider,

etc. When those with an interest in the outcome (e.g. solver developers) do a benchmark it's only natural to expect some natural bias to influence the results, but even a neutral person is forced to make many choices, all of which can bias the results in one direction or another. The best solution for this issue of subjective bias is an environment that uses automation to make the benchmarking process easily reproducible, so that interested users of optimization software can conduct their own benchmarking experiments. This does not remove all subjectivity: the end user must make the same subjective choices as a solver developer or anyone else. But at least the results will be biased towards the set of model instances, performance measures, and validation rules that are most important to the user: her own. If we momentarily lump benchmarking results with statistics, a quote misattributed to Winston Churchill is relevant: "I only believe in statistics that I doctored myself" [25]. A user-generated benchmark will be most trusted and will have maximum impact.

As a practical matter, some users will be unwilling or unable to conduct their own benchmarks and will instead rely on the work of others. For example, Mittelmann's benchmarking talks at conferences are typically quite well attended and people follow his published [20] and online results [21,22] with interest. Even in such cases, it is beneficial to rely on work done using open-source, well-tested environments. The involvement of the larger community serves to identify and sometimes fix errors in tools like PAVER and Examiner, as well as prompting the inclusion of useful improvements and new features, so the analysis results are improved. In addition, anyone using the results of a benchmark done using an open, automated environment can be more confident that the results have not been intentionally biased or falsified, since doing so would be more difficult to hide in such a case. The likelihood of errors due to shortcuts or innocent human error is also reduced. This applies equally well to the process of running the solvers as to generating the analysis results, and while not the subject of this paper, it is important that an automated, reproducible process for running the models also be part of the testing environment.

6 Example benchmarks

6.1 MIP

First, we have run the 87 instances in the benchmark set of the MIPLIB 2010 [17] on 5 MIP solvers under GAMS with a timelimit of 1 hour and using the GAMS `trace` and `traceopt` options (see Section 3.2) to generate comma separated value files with statistics on the performance of each solver on each instance. Since, in the context of this paper, we are mainly interested in discussing PAVER 2 than in comparing MIP solvers, we disguise their real names here.

With the files at hand, we run PAVER with a command like

```
python src/paver/paver.py \
  balin.trc bifur.trc bombur.trc gimli.trc thorin.trc \
  solu/miplib2010.solu \
  --failtime 3600 --refsolver Gimli --writehtml mip
```

This lets PAVER read the 5 GAMS trace files `balin.trc`, `bifur.trc`, `bombur.trc`, `gimli.trc`, and `thorin.trc` and the solution file `miplib2010.solu`. The `failtime`

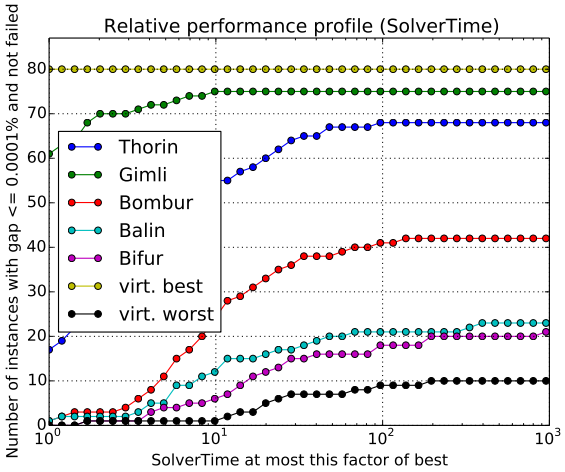

Fig. 1 Performance Profile for solving time of 5 MIP solvers and the corresponding virtually best and worst solvers.

option lets PAVER use a solving time value of 3600 seconds for mean value computations if a solver failed on an instance, i.e., an inconsistency check failed or a solver crashed. Additionally, we instruct PAVER to create comparisons of each solver's performance to the Gimli solver. The HTML output of PAVER is stored in a directory `mip`.

Figure 1 shows the performance profile generated by PAVER. It seems to show a clear ranking of the solvers in the order Gimli, Thorin, Bombur, Balin, and Bifur. Figure 2 shows bar charts for a number of performance metrics. It is seen that Balin failed on 23 instances, either because it terminated with an error or because it reported primal or dual bounds that were inconsistent with the known optimal value. This high number may indicate a bug in the Balin code. The mean solving time shows a slight edge for Balin before Bifur and Balin also solved two more instances to optimality than Bifur, however, Bifur found optimal solutions in four more cases than Balin[5]. This demonstrates that to evaluate a solver's performance, consideration of more than one metric (like mean solving time) is advised.

Note further, that for the mean solving time, a value of 3600 seconds was used for instances where a solver failed. If one considers only instances where no solver failed, Balin appears to have a larger distance to Bifur, i.e., instead of 2217s and 2277s for Balin and Bifur, resp., the mean times reduce to 1823s vs 2155s. Thus, while treating failed instances as if a timelimit was hit may yield large performance ratios for small but numerically unstable instances, eliminating them from the comparison may also falsify the benchmark, e.g., because the remaining test set may become too small (or imagine a case where a solver always crashes when it hits the timelimit), see also [2].

[5] PAVER uses a gap tolerance of 10^{-6} by default. However, we have run our solvers with a zero gap tolerance.

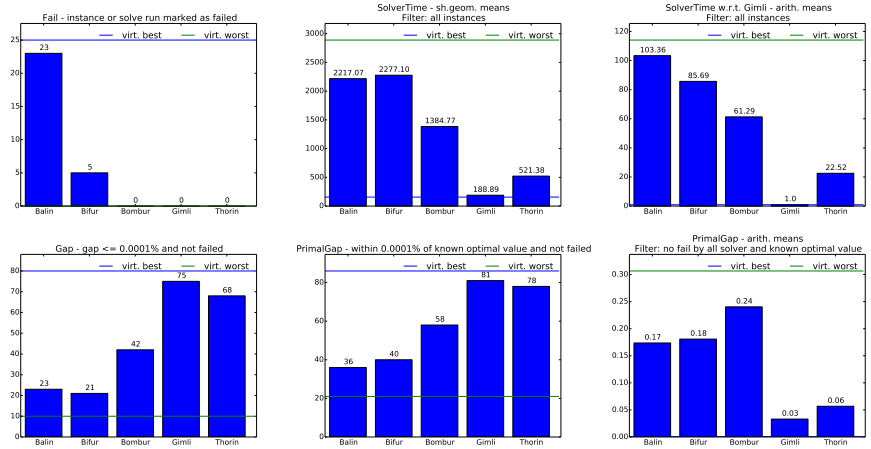

Fig. 2 Visualization of some performance metrics: Number of instances where a solver failed, shifted geometric mean of solving time, average ratio of a solver's solving time to Gimli's solving time, number of instances solved to optimality, number of instances where an optimal solution has been found, and average primal gap at termination.

6.2 NLP

As a second example we compare several GAMS-linked NLP solvers using the GLOBALLIB set of problem instances [19], running with a time limit of 15 minutes. We ran each NLP solver under GAMS/Examiner with options set to write trace files that include the primal/dual variable and constraint infeasibility and primal/dual complementarity gap, in addition to the usual trace output as described in Section 6.1. Note that we used only the NLP and DNLP models from GLOBALLIB, omitting the CNS models. We replace the solver names here with reindeer names to help in keeping the focus on PAVER and not the NLP solvers.

With the solver runs completed and the trace files in hand, we run PAVER with a command like

```
python src/paver/paver.py \
  comet.trc cupid.trc dasher.trc dancer.trc donner.trc \
  solu/globallib.solu \
  --failtime 900  --mintime 0.1 \
  --ccopttol inf  --ccfeastol inf \
  --writehtml localExamNo
```

This lets PAVER read the five GAMS solver trace files and the solution file `globallib.solu`. The `failtime` option lets PAVER use a solving time value of 900s for mean value computations if a solver failed on an instance, i.e., a consistency check failed or a solver crashed. Since all of the solvers in this set are local solvers, no dual bounds are available, so PAVER will skip any related reports or checks. A reduced minimum solver time is appropriate for this set of test instances, so we set `mintime` smaller than default. Setting `ccopttol` and `ccfeastol` to infinity turns off any checks on the Examiner-provided values: we are accepting the solver reports at face value in this case. The HTML output generated by PAVER is stored in the directory `localExamNo`.

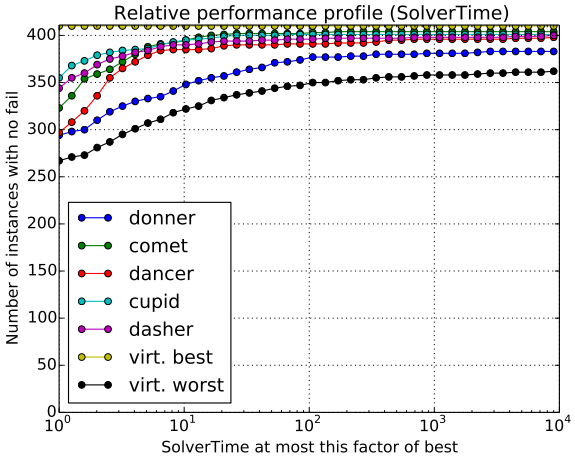

Fig. 3 Performance Profile for solving time of 5 NLP solvers and the corresponding virtually best and worst solvers.

Figure 3 shows the performance profile generated by PAVER. It suggests that all solvers are doing well, with Cupid, Dasher, and Comet leading the pack, Dancer a bit behind that, and Donner trailing behind on performance.

What about other performance metrics? Do they tell the same story? In this case, some of them do and some do not. As shown in Figure 4, the number of failed instances and mean solving time show the same sort of preference ranking as the performance profile. However, Dancer and Dasher seem to be finding a global optimal solution more frequently than the others, so the ranking can tilt if one weights this more heavily[6].

Recall that the results in Figures 3 and 4 essentially take the solver return values at face value, since the options `ccopttol` and `ccfeastol` were set to infinity. Our next PAVER run makes use of the Examiner-provided values by setting the feasibility and optimality tolerances to a modest value of 10^{-5}, but the parameters are otherwise unchanged from the previous run.

The difference is immediately apparent (Fig. 5): Cupid and to a lesser extent Donner suffer because they return solutions that are not precise enough. Comet and Dancer are affected somewhat but both only slightly. This is also illustrated by Figure 6, comparing the number of consistent solutions reported both without and with Examiner checks, and Figure 7, showing the increased solution time and decreased number of successes finding a global solution when results are required to pass a verification check. Note that results that fail to verify are assigned the `failtime` value of 900s.

This example has so far only involved local solvers, but what about global solvers? In this next benchmark we add data for the two solvers Blitzen and Prancer, both run in the same way as the other solvers. To keep the results from becoming too cluttered we also remove the solvers Cupid (it's really the Comet

[6] Note that our `globallib.solu` file contains known optimal values for only half of the GlobalLib instances, so the number of global optimal solutions found and the mean primal gap are computed w.r.t. these 214 instances only.

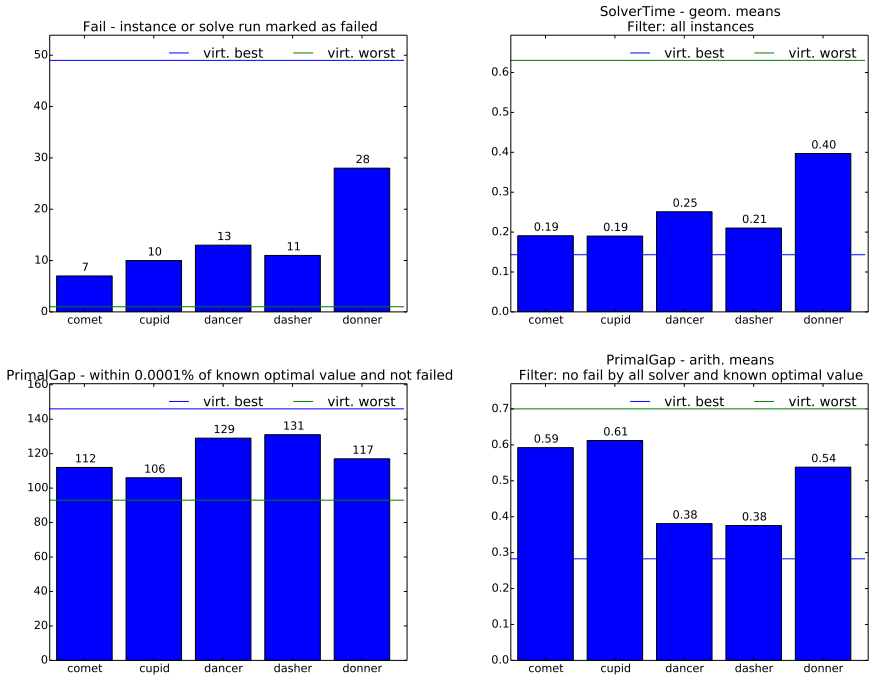

Fig. 4 Visualization of some performance metrics: Number of instances where a solver failed, geometric mean of solving time, number of instances where a global solution has been found, and average primal gap at termination.

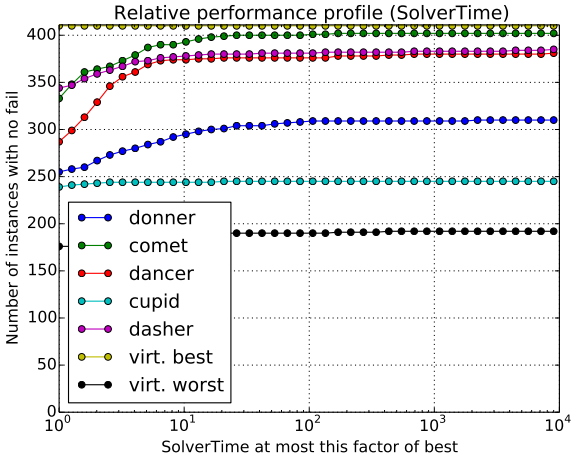

Fig. 5 Performance Profile for solving time of 5 NLP solvers and the corresponding virtually best and worst solvers, using Examiner.

solver run with over-wide tolerances) and Dancer (it's identical to Dasher with the exception of the linear algebra routines used). We chose to run PAVER with

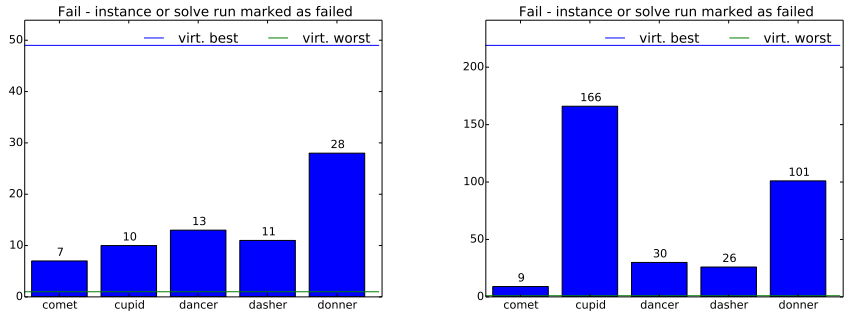

Fig. 6 Number of failed instances with no Examiner checks (left) and with Examiner checks on feasibility and optimality.

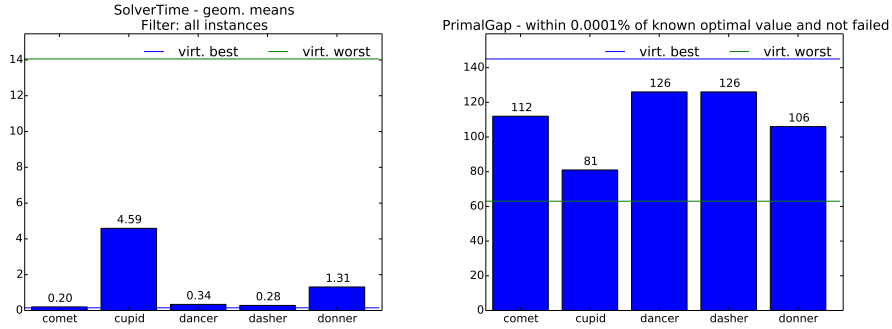

Fig. 7 Visualization of some performance metrics for Examiner-checked solutions: geometric mean of solving time and number of instances where a global solution has been found.

options similar to those used above. Since we are taking the global nature of these problem instances into account we ignore the Examiner-based optimality checks and check only feasibility. The PAVER command looks like

```
python src/paver/paver.py \
  blitzen.trc comet.trc dasher.trc donner.trc prancer.trc \
  solu/globallib.solu \
  --failtime 900  --mintime 0.1 \
  --ccopttol inf  --ccfeastol 1e-5 \
  --writehtml global
```

The PAVER output for this case contains several performance profiles. One subgroup shows solver time for those models where the gap is within a given tolerance. We show in Figure 8 the case where the gap is within 1%: the results are quite similar for 10% or 10^{-6}. The two global solvers are quite distinct in this case, with Prancer dominating Blitzen, but the local solvers all show as complete failures, since they do not report any gap at all. If we look at the solver time for instances with a small primal gap then the local solvers play a role, but we can only report on models with a known solution. Figure 9 shows such a performance profile. It's interesting to note that the ranking of the global solvers is reversed in

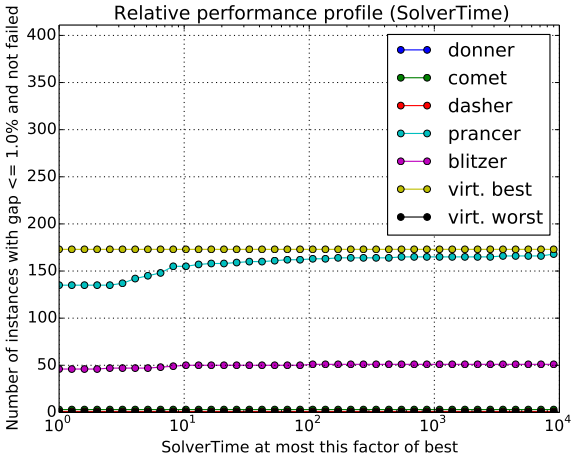

Fig. 8 Performance Profile for solving time for 5 NLP solvers, considering only runs that report a gap of at most 1% at termination, with feasibility validation.

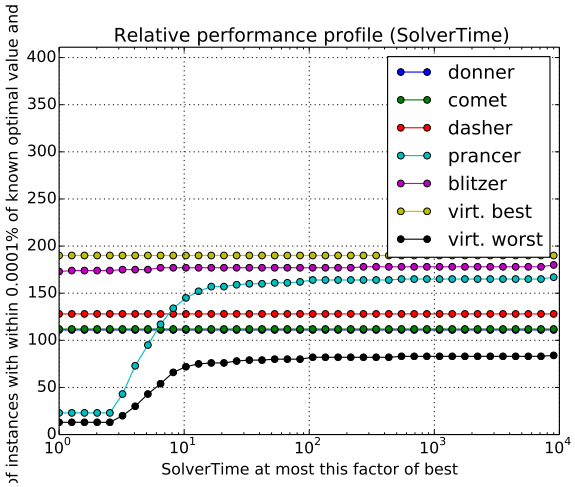

Fig. 9 Performance Profile for solving time for 5 NLP solvers on models with a near-zero primal gap, with feasibility validation.

this case, with Blitzen dominating Prancer. Evidently Blitzen is relatively strong in finding an optimal solution but Prancer is doing a better job bringing up the dual bound. This is another example of a case where a subjective choice (which metric to use and, implicitly, the set of models where that metric is defined) plays an important role in the process.

The conclusions above are consistent with what we see using other metrics. For example, in Figure 10, we see that both global solvers are more effective at finding the optimal solution than any of the local solvers, with Blitzen slightly more effective than Prancer. We also see that Prancer has the edge when the metric used is the dual gap.

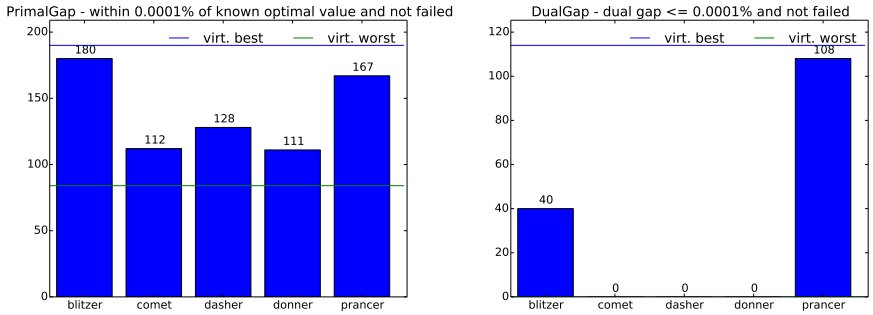

Fig. 10 Performance metrics for Examiner-checked solutions: number of instances where a global solution has been found and where the dual gap is nearly zero.

It is worth mentioning that the PAVER output also contains raw and processed results in an easily browsable form. This is quite useful when drilling down to see what's behind the various summary reports. For example, in the PAVER HTML output, the "global solution found" report from the left picture in Figure 10 is linked to a table showing the primal gap for all cases where it is below 10^{-6}.

7 Conclusions

In this paper we have described PAVER 2.0, our environment for verifying and analyzing benchmarking data. Like many software projects, it is never really finished but instead constantly evolves in the face of changing needs, expectations, and computing environments. Our future plans for PAVER include the addition of new performance metrics, e.g., the ones used in [13] and ones based on statistical tests for dominance of one set of data over another. For the Examiner component, we plan to include alternate metrics for convergence to a solution [11]. The time that PAVER spends to generate the various plots will likely be decreased, and additional output formats, especially LaTeX, are under consideration. Finally, we note that PAVER currently operates in a batch mode. The introduction of an interactive mode, allowing the user to filter instance data and calculate performance metrics on the fly (as is possible with [16]) could be useful.

Perhaps more important than our specific future plans for PAVER development are the possibilities for new directions and uses that we hope and expect will arise out of making PAVER an open-source environment. PAVER grew out of our own internal needs, but we believe it can be useful to others, and we look forward to the improvements to PAVER that can only come through the involvement and perspective of a larger group. If PAVER is used and adopted as a de-facto standard by the optimization community and serves to promote lively discussion and development in the area of performance analysis tools, we will consider it to be successful indeed.

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, TU Berlin (2007). urn:nbn:de:0297-zib-11129

2. Achterberg, T.: Benchmarking a MIP Solver. talk in CPAIOR master class (2010). URL http://cpaior2010.ing.unibo.it/?q=node/10
3. Berthold, T.: Measuring the impact of primal heuristics. Operations Research Letters **41**(6), 611–614 (2013). doi:10.1016/j.orl.2013.08.007
4. Billups, S.C., Dirkse, S.P., Ferris, M.C.: A comparison of large scale mixed complementarity problem solvers. Computational Optimization and Applications **7**(1), 3–25 (1997). doi:10.1023/A:1008632215341
5. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib - a collection of test models for mixed-integer nonlinear programming. INFORMS Journal on Computing **15**(1), 114–119 (2003). doi:10.1287/ijoc.15.1.114.15159
6. Bussieck, M.R., Drud, A.S., Meeraus, A., Pruessner, A.: Quality assurance and global optimization. In: C. Bliek, C. Jermann, A. Neumaier (eds.) Global Optimization and Constraint Satisfaction, *Lecture Notes in Computer Science*, vol. 2861, pp. 223–238. Springer (2003). doi:10.1007/978-3-540-39901-8_17
7. Why open source? (2013). URL http://www.coin-or.org. [Online; accessed 15-May-2013]
8. Crowder, H., Dembo, R.S., Mulvey, J.M.: On reporting computational experiments with mathematical software. ACM Transactions on Mathematical Software **5**(2), 193–203 (1979). doi:10.1145/355826.355833
9. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical Programming **91**(2), 201–213 (2002). doi:10.1007/s101070100263
10. Dolan, E.D., Moré, J.J., Munson, T.S.: Benchmarking optimization software with COPS 3.0. Tech. Rep. ANL/MCS-273, Mathematics and Computer Science Division, Argonne National Laboratory (2004). URL http://www.mcs.anl.gov/~more/cops
11. Dolan, E.D., Moré, J.J., Munson, T.S.: Optimality measures for performance profiles. SIAM Journal on Optimization **16**(3), 891–909 (2006). doi:10.1137/040608015
12. Drud, A.S.: Testing and tuning a new solver version using performance tests. INFORMS 2002, San Jose, session on 'Benchmarking and performance testing of optimization software'. URL http://www.gams.com/presentations/present_performance.pdf. [Online; accessed 15-May-2013]
13. Exler, O., Lehmann, T., Schittkowski, K.: A comparative study of SQP-type algorithms for nonlinear and nonconvex mixed-integer optimization. Mathematical Programming Computation **4**(4), 383–412 (2012). doi:10.1007/s12532-012-0045-0
14. GAMS Development: GAMS/Examiner, *User's Manual* (2013). URL http://www.gams.com/dd/docs/solvers/examiner.pdf. [Online; accessed 8-May-2013]
15. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library (2012). URL http://gmplib.org
16. Hendel, G.: PyEvalGui - GUI components to facilitate evaluation of SCIP and other solving software (2013, in development)
17. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010 – mixed integer programming library version 5. Mathematical Programming Computation **3**(2), 103–163 (2011). doi:10.1007/s12532-011-0025-9
18. Mahajan, A., Leyffer, S., Kirches, C.: Solving mixed-integer nonlinear programs by QP-diving. Preprint ANL/MCS-P2071-0312, Argonne National Laboratory (2012). URL http://www.optimization-online.org/DB_HTML/2012/03/3409.html
19. Meeraus, A.: Globallib (2013). URL http://www.gamsworld.org/global/globallib.htm. [Online; accessed 8-May-2013]
20. Mittelmann, H.D.: An independent benchmarking of SDP and SOCP solvers. Mathematical Programming **95**(2), 407–430 (2003). doi:10.1007/s10107-002-0355-5
21. Mittelmann, H.D.: DTOS – A service for the optimization community. SIAG/OPT Views-and-News **18**, 17–20 (2007)
22. Mittelmann, H.D.: Decision tree for optimization software (2013). URL http://plato.asu.edu/guide.html. [Online; accessed 8-May-2013]
23. Mittelmann, H.D., Pruessner, A.: A server for automated performance analysis of benchmarking data. Optimization Methods & Software **21**(1), 105–120 (2006). doi:10.1080/10556780500065366
24. SCIP development team: How to run automated tests with SCIP. URL http://scip.zib.de/doc/html/TEST.shtml
25. Wikiquote: Winston churchill — wikiquote, (2013). URL http://en.wikiquote.org/w/index.php?title=Winston_Churchill&oldid=1552921. [Online; accessed 8-May-2013]