Optimal Strategy Schedules for Everyone

Hans-Jörg Schurr[1]

[1] *University of Lorraine, CNRS, Inria, and LORIA, Nancy, France*

Abstract

Parametrization of a theorem prover is critical for solving a problem. A specific parametrization is called a strategy and the best strategy usually differs from problem to problem. Strategy scheduling means trying multiple strategies within a time limit. veriT-schedgen is a toolbox to work with strategy schedules. In its core is a simple tool that uses integer programming to generate strong schedules automatically. Another tool translates the schedules into shell scripts. Hence, the toolbox can be used with any theorem prover. The generated schedules are optimal with regards to the number of solved benchmarks. While generating optimal schedules is NP hard, the generation time is short in practice.

Keywords

automated theorem proving, strategy scheduling, Python

1. Introduction

Most components of any theorem prover can be parametrized and fine-tuned. Selecting the right values for the parameters is usually not easy. Often there is no clear best choice, and even if there is one, overall non-optimal choices might work better for certain types of problems. A specific choice of values for all exposed parameters is a *strategy*. It is often crucial to use the correct strategy to solve a problem within a given timeout.

An approach to this problem is for the theorem prover to expose options to users that allow them to configure the used strategy. This is a fig leaf: defining the right strategy usually requires intimate knowledge of the inner workings of the solver. Furthermore, the developers of the prover have to set a sensible default. This is not easy either: usually prover developers do not know upfront the type of problems the prover will encounter. The default should normally also be somewhat generic. Overall, designing and using strategies is a subject that deserves some attention. Since for many problems there is a strategy that can solve the problem in a short time, it is natural to try multiple strategies on the problem. The easiest way to do this is to try strategies from a list one by one. Slightly more sophisticated is to prepare a list of strategies paired with a timeout: some strategies might be known to have diminishing returns if run for a longer time. We will call such a list a *schedule*.

In this paper we present a toolbox to generate and work with schedules. The core of the toolbox is a method based on integer programming to find a strong schedule for

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel
EMAIL: hans-jorg.schurr@inria.fr (H. Schurr)
ORCID: 0000-0002-0829-5056 (H. Schurr)

a given set of strategies and benchmarks that is optimal with regard to the number of benchmarks solved. The theorem prover developer has to define a list of strategies and allowed time slices and has to record how much time each strategy uses to solve benchmarks from a training set. The method encodes this problem of finding the schedule that solves the most benchmarks within an overall timeout as an integer programming problem (Section 2 to Section 4). Overall, the generation procedure is easy to understand and avoids surprises.

This encoding is used by our tool "veriT-schedgen" to generate schedules. It is implemented in Python (Section 5), has a simple user interface, and comes with auxiliary tools that can simulate and visualize schedules. The toolbox allows users to generate simple, static schedules and to integrate them with their solver of choice. In Section 6 we see an evaluation of the practical usefulness of veriT-schedgen. Of particular interest to us is how well the generated schedules perform on benchmarks that are not used to generate the schedules. That is: do the generated schedules *generalize* to unseen benchmarks?

Originally, the veriT-schedgen toolbox was developed for the SMT solver veriT [1]. While the toolbox is fully independent of the solver, this paper discusses it through the perspective of using it with veriT.

2. Integer Programming

Let us first direct our attention to integer programming. It is widely used to express optimization problems. Solving an integer programming problem means finding an assignment to variables, such that a linear combination of the variables is maximized and all linear inequalities from a given set are satisfied. If some variables must take integer values, while others allow real values, the problem is a *mixed* integer programming problem. Usually integer programming is presented in terms of linear arithmetic, but since we are interested in SMT solving we will use a notation similar to the many-sorted logic with theories commonly used in SMT solving [2].

Formally, a mixed integer programming problem is a tuple $(\mathcal{C}, \mathcal{O})$ where $\mathcal{C}$ is a set of formulas and $\mathcal{O}$ is a special term called the *objective function*. The formulas in $\mathcal{C}$ are formulas in the theory of linear real and integer arithmetic without boolean connectives and quantifiers. They are called the *linear constraints*. The objective function is a term of sort **Real** or **Int**.

A solution to a mixed integer programming problem is an interpretation Σ such that t evaluates to **true** for all $t \in \mathcal{C}$ and there is no $\Sigma' \neq \Sigma$ such that the evaluation of $\mathcal{O}$ under Σ' is greater than its evaluation under Σ. Hence, Σ maximizes $\mathcal{O}$.

Traditionally the objective function $\mathcal{O}$ is written as a linear combination

$$(c_1 \times x_1) + (c_2 \times x_2) + \cdots + (c_n \times x_n)$$

of some variables $x_i : \sigma$ and constants $c_i : \mathbf{Int}$ where the sort $\sigma \in \{\mathbf{Real}, \mathbf{Int}\}$. The linear constraints are traditionally written as equalities and inequalities in the form

$$(c_1 \times x_1) + (c_2 \times x_2) + \cdots + (c_n \times x_n) + c_{m+1} \bowtie (c_{n+1} \times x_{n+1}) + \cdots + (c_m \times x_m) + c_{m+2}$$

where $m \geq n$, the x_i and c_i are variables and constants just as in the goal function, and $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$. If the constant factors (c_{m+1} and c_{m+2}) are zero, they are usually omitted.

A special case of a variable of sort **Int** is a *binary* variable. An integer variable x is a binary variable if the problem contains the two constraints $x \geq 0$ and $x \leq 1$. Hence, a solution can only assign 0 or 1 to x. We will mark a binary variable with a dot: $\dot{x}$. Nevertheless, the sort of $\dot{x}$ is still **Int**. For simplicity we will also not explicitly show the two constraints for each binary variable.

Since integer programming is a well established field, there is a lot of material available. One example is "A Tutorial on Integer Programming" by Cornuéjols et al. [3]. Furthermore, the online documentation of the PuLP library that is used by veriT-schedgen contains good introductory examples.[1]

3. Schedule Generation as Integer Programming

In this section we will see how we can express the problem of finding an optimal schedule as a (mixed) integer programming problem. To do so, it is necessary to be a bit more precise about the notion of strategies, benchmarks, experiments, and schedules.

We have two finite, non-empty sets: the set S of strategies and the set B of benchmarks. Furthermore, we have a family of functions $(E_s \colon B \to \mathbb{R} \cup \{\infty\})_{s \in S}$ that represent the experiments. The value $E_s(b)$ is the time needed by the solver to solve the benchmark b when using the strategy s. If $E_s(b) = \infty$, the benchmark was not solved. To generate a schedule we also have to define the overall timeout $T > 0$ and a list $t_1, \ldots, t_n$ of allowed time slices. We assume that the list of time slices is in ascending order, all slices are positive, and no time slice appears twice (i.e., $0 < t_i < t_{i+1}$ for all $1 \leq i < n$). Furthermore, there is no value in having time slices that are longer than the overall timeout. Hence, $t_n \leq T$.

Within this environment, a *schedule* $\mathcal{S}$ is a finite set of pairs $(t, s) \in \mathcal{S}$ where $t \in \{t_1, \ldots, t_n\}$ and $s \in S$. The overall time used by the schedule must not exceed the total timeout:

$$\sum_{(t,s) \in \mathcal{S}} t \leq T.$$

The generated schedule should maximize the number of benchmarks solved within the overall timeout. Hence, we look for the schedule $\mathcal{S}$ such that

$$\left| \bigcup_{(t,s) \in \mathcal{S}} \{b \mid b \in B \text{ and } E_s(b) \leq t\} \right|$$

is maximal. If the optimal solution for this objective function is found, the generated schedule is *optimal* with regards to the number of benchmarks solved.

One aspect is missing here: the order in which the strategies in $\mathcal{S}$ should be tried. This order can be determined in a second phase after the optimal set of timeout, strategy

[1] The PuLP library is available at https://coin-or.github.io/pulp/

pairs has been calculated. This keeps the encoding simple. However, it is not obvious what measure should be optimized to find a good order. In Section 5 we will discuss approaches to this problem.

We start modelling the optimization problem by defining a binary variable $\dot{x}_{(t,s)}$ for every $(t, s) \in \{t_1, \ldots, t_n\} \times S$. If a $\dot{x}_{(t,s)}$ is 1, then the schedule runs the strategy s for the time t. Since every benchmark solved by a strategy in a short timeout will also be solved by a longer timeout, it is pointless to pick one strategy for more than one time slice. To model this we define the additional binary variables $\dot{x}_s$ expressing that the strategy s runs for any timeslice. We can now add our first set of constraints. For each $s \in S$ the constraint

$$\dot{x}_s = \sum_{1 \leq i \leq n} \dot{x}_{(t_i,s)}$$

expresses that each strategy can be picked at most once: since $\dot{x}_s$ is a binary variable, only one of the summands can be 1. This constraint eliminates corner cases. For example, if there is a schedule that solves all benchmarks well before the overall timeout, the integer programming solver can pick a strategy twice without changing the objective function.

With regards to solved benchmarks we will use two variables for each benchmark $b \in B$. The binary variable $\dot{x}_b$ is intended to be 1 iff the benchmark b is solved by at least one picked timeout, strategy pair. That is, there is a $(t, s) \in \{t_1, \ldots, t_n\} \times S$ such that $E_s(b) \leq t$ and $\dot{x}_{(t,s)}$ is 1. To model this we use an auxiliary *integer* variable x'_b that counts the number of times b is solved. This is implemented for each x'_b by the constraint

$$x'_b = \sum_{\dot{x} \in X_b} \dot{x} \text{ with } X_b := \left\{ \dot{x}_{(t,s)} \,|\, (t, s) \in \{t_1, \ldots, t_n\} \times S \text{ and } E_s(b) \leq t \right\}.$$

The following two constraints force $\dot{x}_b$ to 1 iff $x'_b \geq 1$.

$$\begin{aligned} \dot{x}_b |X_b| &\geq x'_b \\ \dot{x}_b &\leq x'_b + 0.5. \end{aligned}$$

If x'_b is not 0, then the first constraint ensures that $\dot{x}_b$ has to be 1, if x'_b is 0, then the second constraint ensures that $\dot{x}_b$ is 0 too. Since, x_b is a binary variable there is no other value smaller than 0.5.

Defining the objective function is now trivial. It is just the sum

$$\sum_{b \in B} \dot{x}_b.$$

To extract the schedule $\mathcal{S}$ from a solution for this encoding, we can collect the $\dot{x}_{(t,s)}$ that are set to 1. Due to the constraints, no two $\dot{x}_{(t_1,s)}$, $\dot{x}_{(t_2,s)}$ with $t_1 \neq t_2$ are ever 1.

A solution to this integer program maximizes the number of benchmarks solved within the overall timeout. Every strategy is assigned at most one of the allowed time slices. Since the time slices are predefined this solves a limited form of the general optimal schedule problem [4] which is NP hard. Our restricted version is also a variant of the knapsack problem: we want to maximize the value of the items (the number of solved benchmarks by the strategies) while respecting the weight limit (the timeout). Hence it is still NP hard.

4. Strategy Order and Combined Schedules

The solution of the integer program as discussed above does not give an order of the timeout, strategy pairs. The goal for selecting pairs is clear: solve as many problems as possible. There is also an intuitive goal for the order: to minimize the sum of solving times of the benchmarks solved by the schedule. Indeed, if we want to generate a schedule that performs as well as possible at the SMT competition [5], we would have to find this order. The single-query track of the competition ranks solver by the number of benchmarks solved within the overall timeout (20 min), but the sum of solving times serves as the tiebreaker [5, Section 7.1].

This is another optimization problem that is further complicated by the unpredictable behaviour of the solver on unsolved benchmarks. The solver might either run until terminated at the timeout, give up, or even crash. Since veriT uses quantifier instantiation, it often gives up when no new ground instances are generated because the input problem is satisfiable. In this case, no strategy can succeed, but all will be tried. Currently, the implementation of the optimization toolbox does not provide a procedure to compute this order. We tried an approach based on dynamic programming, but it was too slow in practice.

Furthermore, this order is not necessary the best for all application. For example, an application could interrupt schedules prematurely. To see why minimizing the sum of solving times is not the best in this situation consider two strategies a and b such that a solves three benchmarks after 2 s and b solves only one benchmark after 1 s. The sum of solving times of the schedule $[(2, a), (1, b)]$ is 9 s. The sum for the schedule $[(1, b), (2, a)]$ is 10 s. However, if the first schedule is interrupted after one second, no benchmark is solved. If the second benchmark is interrupted after one second, one benchmark is solved.

Alternatively, one might sort the timeout, strategy pairs by increasing timeout. The result is a schedule that tries short strategies in quick succession before using longer strategies. However, it might be that strategies that are only used with a short timeout are specialized to a small number of benchmarks and the more general strategies are used with longer timeouts. Furthermore, since the time needed to solve benchmarks is usually not evenly distributed, even a strategy used with a long timeout might solve many benchmarks very fast.

An alternative approach is to pick the pair that solves the most benchmarks not solved so far. This *best effort* order has the benefit that if the schedule is not used for the total timeout, the number of benchmarks not solved is minimized if the schedule is interrupted between pairs. The downside of this approach, however, is that the schedule would use a pair with a long timeout before using two pairs with short timeouts that cumulatively solve more benchmarks.

Hence, the order of a strong schedule should take the number of solved benchmarks, the time it takes to solve them, and the timeout of each pair into account. We can modify the best effort order to achieve this. Instead of picking the pair that solves the most benchmarks, we pick the pair that has the minimal estimated cost. To estimate the cost of a pair we calculate the sum of solving times. For benchmarks solved by the pair this is straightforward. For the other benchmarks an approximation is needed. Our

approximation is the solving time used by the virtual best solver formed by the other pairs in the schedule. The virtual best solver of a schedule $\mathcal{S}$ is the function $E_{\mathcal{S}} \colon B \to \mathbb{R} \cup \{\infty\}$ such that $E_{\mathcal{S}}(b) = \min(\{E_s(b) \mid (t,s) \in \mathcal{S} \wedge E_s(b) \leq t\} \cup \{\infty\})$. Based on this definition, the cost estimate for a pair $(t,s) \in \mathcal{S}$ formally is

$$c_{(t,s)} = \sum_{b \in \{b \mid E_s(b) < t\}} E_s(b) + \sum_{b \in \{b \mid E_s(b) > t \wedge E_{\mathcal{S}'}(b) < \infty\}} (t + E_{\mathcal{S}'}(b)) \,.$$

where $\mathcal{S}' = \mathcal{S} \setminus (s,t)$. We will also use the virtual best solver later as a baseline to evaluate schedules.

The ordered schedule is constructed by removing iteratively the pair (t,s) with the smallest cost estimate $c_{(t,s)}$ from $\mathcal{S}$ until no pair is left. The resulting schedule will usually first use pairs with short timeout. Longer timeouts will only be chosen if the benchmarks solved by this strategy offset the delay to solving the other benchmarks. Since a schedule typically contains less than a hundred pairs, ordering a schedule with this procedure is fast.

It is also possible to target multiple predefined timeouts with one schedule. For example, at the SMT competition solvers compete with both a timeout of 20 min and a timeout of 24 s. A simple approach to generate a schedule that work well in this setting is to perform multiple rounds of optimization: Let T_1 and T_2 be two timeouts such that $T_1 < T_2$ and let $\mathcal{S}_1$ be the optimal schedule for the timeout T_1. We can now define a new set of benchmarks B' that contains exactly the benchmarks from B that are not solved by the schedule $\mathcal{S}_1$ within the timeout T_1. Next, we can calculate another schedule $\mathcal{S}_2$ that solves the most benchmarks from B' within the timeout $T_2 - T_1$. Both rounds of optimization can use a different set of allowed time slices. To build the ordered joint schedule, we just search for the best order for both schedules independently and concatenate the resulting lists.

A small downside of this approach is that some strategies appear in both schedules and the solver has to perform some repeated work. This repeated work can be avoided for one strategy. Assume that there is a repeated strategy s. Hence, there is a pair $(s,t_1) \in \mathcal{S}_1$ such that $(s,t_2) \in \mathcal{S}_2$ for some t_2. In this case $t_2 > t_1$ since otherwise (s,t_2) would solve no benchmarks not already solved by $\mathcal{S}_1$. If we remove (s,t_1) form the the first schedule $\mathcal{S}_1$, the schedule solves fewer benchmarks, but only runs for the time $T_1 - t_1$. To get those benchmarks back, we move (s,t_2) to the beginning of the schedule $\mathcal{S}_2$. In the concatenated schedule s is used for the time t_1 before T_1 passes and is used for the time t_2 overall.

5. The veriT-schedgen Toolbox

We implemented the schedule generation procedures described above as part of a toolbox called veriT-schedgen. Beside optimization, the toolbox also contains some other tools that are useful when working with strategy schedules, such as a visualizer and simulator. This section provides an introduction – a user manual – to the toolbox. It also describes

important implementation details of the toolbox. The source code for the toolbox is available at https://gitlab.uliege.be/verit/schedgen/.

The veriT-schedgen toolbox is implemented in the Python programming language and uses the PuLP [6] package to express and solve the linear programming problems. The PuLP package can use multiple linear programming solvers as backend, but the default solver is good enough for the linear programming problems arising from our use case. The default solver is the open source solver CBC[2]. Both, the PuLP and CBC project, are developed by the COIN-OR foundation that develops open source software for operations research[3]. To store the experiment results veriT-schedgen uses the Pandas library[4]. This library provides a flexible data structure to work with tables.

All tools in the veriT-schedgen toolbox support the filtering of benchmarks by SMT-LIB logic. Since the SMT-LIB benchmark library is big (around 100 GiB), it is often not installed on the computer that is used for schedule generation. In this case the logic cannot determined by parsing the benchmark file. Instead, usually the first component of the benchmark path is used. For example, a benchmark `QF_UF/test.smt2` will have the logic `QF_UF` assigned. This matches the typical folder structure used by the SMT-LIB benchmark suite. In fact, it is not necessary to use the SMT-LIB logic for filtering. Instead, any string does the job.

The veriT-schedgen toolbox is a standard Python package. It has been tested with Python 3.10. To install the package it is enough to execute

```
$ python setup.py install
```

This will install the following tools that are part of the veriT-schedgen toolbox on the user systems.

schedgen-optimize.py is the main tool that performs the schedule generation. It takes the name of a folder that contains the experiments, a list of time slices, and the total timeout. After the generation is finished the result is written as a schedule into a CSV file. It is also possible to provide an existing scheduler to build a joint schedule. The benchmarks used to generate the schedule can be filtered as described above.

schedgen-finalize.py takes CSV files describing schedules, as generated by `schedgen-optimize.py`, and produces a shell script that runs a solver according to the schedules. The user can provide a template for the script and change various parameters used during generation. The default script can pick schedules by SMT-LIB logic.

schedgen-simulate.py can be used to predict the outcome of running a scheduler. To do so the user has to provide experiments for the strategies included in the schedule, but can use different benchmarks. Furthermore, it is possible to add noise to the simulation.

[2] Available on https://github.com/coin-or/Cbc.
[3] For more information see https://www.coin-or.org/
[4] See https://pandas.pydata.org/.

schedgen-query.py is a small tool that produces lists of benchmarks for debugging and analysis purposes. For example, it can list the union of the benchmarks solved by some strategies, or find the benchmarks from this list that a specific schedule can not solve.

schedgen-visualize.py can be used to visualize one or multiple schedules. It generates a bar plot that represents a schedule visually.

Each of those tools can generate an extensive documentation of the options it exposes via the standard `--help` option.

The tools accept experimental data in two formats: the GridTPT format and a simple CSV format. The GridTPT system [7] is a platform for testing SMT solvers and other theorem provers on grid computers. Its file format is based on CSV, but extended with a header that stores information about the experiment. For every strategy one file is used that contains the results for all benchmarks. The header also contains the command line options, i.e., the strategy used for the experiment. Furthermore, the header contains information such as the solver, when it was executed, and what timeout was used. Since the GridTPT format uses one file per strategy, the user has to declare a folder that contains all data files. The veriT-schedgen input parser searches recursively for compatible files in this folder.

The simple CSV format collects all results within one file. The data file contains five columns and header that stores the column names. Columns are seperated by a literal "`;`". The columns can be given in any order. The parser uses the header line to determine the order of the columns. The five columns are:

- `benchmark` contains the filename of the benchmark.
- `logic` stores the SMT-LIB logic of the benchmark or any other string that should be used to select subsets of benchmarks.
- `strategy` the strategy used.
- `solved` is `yes` if the benchmark was solved, "`no`" otherwise.
- `time` is the solving time in seconds as a positive float number. Any value that is not a valid floating point number is assumed to be ∞. It is possible to indicate a time even if the benchmark was not solved. Sometimes the solver might give up.

It is simple to add a custom parser. One has to implement a Python class whose constructor takes two arguments: the path to the folder or file containing the data and a list of logics used as filter. This list of logics might be `Null` if the data should not be filtered. If parsing succeeds, the resulting object should have a member `.frame` that contains the parsed data. This member is a Pandas data frame where the rows are the benchmarks and the columns are the strategies. Each cell contains the solving time in seconds stored as a standard float. If the benchmark was not solved by the strategy the cell stores the float '∞'.

Generated schedules are also stored in CSV files. The CSV dialect is as for the input data, but this time only two columns are used: `strategy` and `time`. The file is intended to be read from top to bottom and each line indicates a step in the schedule that executes

the strategy `strategy` for `time` seconds. Until schedules are transformed into a script by using `schedgen-finalize.py` the are called *pre-schedules*.

5.1. A veriT-schedgen Tutorial

This section shows how the veriT-schedgen toolbox can be used in practice. We will build and explore an optimal schedule for some artificial example benchmark data. This data can be found in the file `contrib/example_data.csv` and uses six made up strategies. The first strategy `base-strategy` solves 20 benchmarks within one second. The five other strategies `extra01` to `extra05` solve up to five benchmarks exclusively such that `extra01` solves five benchmarks not solved by any other strategy, `extra02` solves four, and so on. There is one benchmark `unsolved.smt2` that is not solved by any strategy and a strategy `bad-strategy` that solves only one benchmark in 1.5 seconds. Hence, if the total timeout is six seconds we expect the optimal schedule to run the strategies in the order just presented for one second each. The solving times have been sampled from a normal distribution to ensure the resulting graphs look interesting.

Since the example data is in the CSV format described above the first four lines of `base-strategy` are:

```
benchmark;logic;strategy;solved;time
base01.smt2;UF;base-strategy;yes;0.5189
base02.smt2;UF;base-strategy;yes;0.2164
base03.smt2;UF;base-strategy;yes;0.1754
```

Obviously, the first step is to generate a schedule. To do so we invoke the `schedgen-optimize.py` command:

```
$ schedgen-optimize.py -l UF -s 0.9 1.0 1.1 -t 6 \
    -c -d contrib/example_data.csv \
    contrib/example_schedule.csv
```

The option `-l UF` selects the UF logic, the option `-s 0.9 1.0 1.1` defines the time slices, and `-t 6` sets the total timeout. Finally, `-c` tells the tool to use the CSV parser, and the option `-d` gives the data location.

The tool then searches the optimal schedule and writes the result to `contrib/example_schedule.csv`. As enforced by the syntetic data, the optimal schedule uses the strategies in the order outlined above. The first four lines of the schedule are:

```
time;strategy
1.100;base-strategy
1.000;extra01
0.900;extra02
```

Furthermore, the output by the tool predicts how well the schedule will perform:

```
['UF']: This schedule solves 35/37 in 52.19s.
```

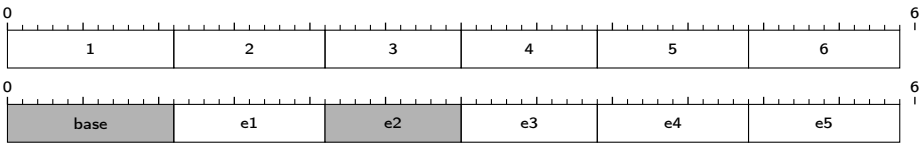

Figure 1: The visualization of the example schedule without and with highlights.

Two benchmarks are not solved by this strategy: the benchmark only solved by the "bad" strategy and the benchmark not solved by any strategy. Furthermore, we learn that solving all benchmarks solved by this schedule will take 52.19 seconds. The output also tells us the solving time before order optimization, but in this case nothing changes.

The simulator can give us a better prediction of the schedule performance, but let us first visualize the schedule:

```
$ schedgen-visualize.py -t 6 -p out.pgf \
    contrib/example_schedule.csv
```

Figure 1 shows the result of this command. The `-t` option works as before and defines the overall timeout. The options `-p out.pgf` tells the visualizer to generate a PGF/Ti*k*Z picture. If this option is omitted, the tool instead opens a window that shows the schedules. Finally, the tool expects a list of pre-schedules to visualize.

Since the different strategies are all used for roughly the same timeout, every strategy is shown as a block of similar size. Because some strategies are used for less than one second, there is a small gap of 0.1 seconds at the end. By default the visualizer simply assigns a number to each strategy when it is used the first time. Strategies often correspond to command line options and those can be long. A number ensures that there are no printing problems, but makes it possible to compare different schedules based on the same strategies.

It is possible to customize the names used for some strategies and to highlight them. This is done with the help of a *shorthand* file: a CSV file with three columns. As in all other CSV files, the fields are separated by a semicolon and there must be a header line. The `strategy` column lists the full strategy string, the `shorthand` column gives the name that should be printed, and the `highlight` can be set to `true` to indicate that the strategy should be highlighted (`false` otherwise). Strategies not listed in the shorthand files get assigned a number and stay unhighlighted. In Figure 1 the base strategy and the second additional strategy are highlighted and every strategy has a custom shorthand name. The command to produce this figure is:

```
$ schedgen-visualize.py -t 6 -c -p out.pgf \
    -a contrib/example_shorthand.csv \
    contrib/example_schedule.csv
```

Let us now simulate the generated schedule. To do so we can use the simulation tool:

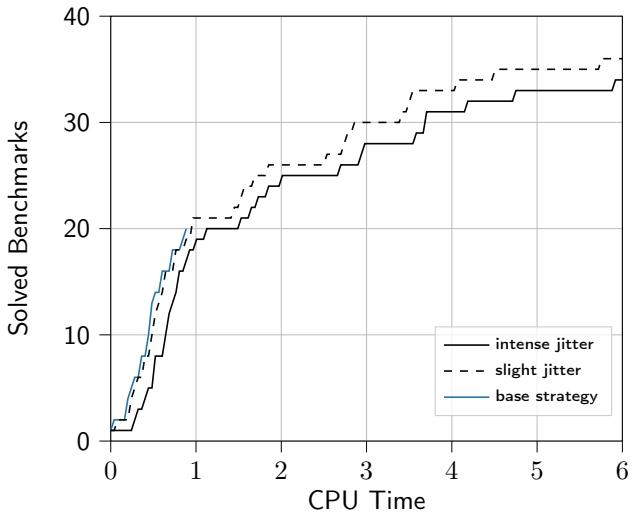

Figure 2: Some simulated density functions.

```
$ schedgen-simulate.py -l UF -t 6 \
    -c -d contrib/example_data.csv \
    --mu 0.05 --sigma 0.01 --seed 1 \
    contrib/example_schedule.csv simulation_1.txt
```

The options `-l`, `-t`, `-c`, and `-d` are as for the other tools. The tool expects two positional arguments. First, the schedule to simulate and, second, the file to write the results to. The output is always using the GridTPT [7] format, independent of the input format. The remaining options are `--mu`, `--sigma`, and `--seed`. Those control the random jitter that is applied to each solving time in the input dataset. The jitter is sampled from a normal distribution. The options `--mu` and `--sigma` control the mean and the standard deviation, respectively. To ensure simulations can be repeated it is possible to fix an integer seed for the random number generator with the option `--seed`. In this case we choose a slight jitter.

The cumulative density functions generated by the simulation are shown in Figure 2. As expected, the graph for the simulation with slight jitter follows almost exactly the base strategy for the first second. The plot also shows the result of using more jitter ($\mu = 0.2$, $\sigma = 0.05$). In this case two benchmarks are no longer solved.[5]

If we want to have a list of the benchmarks not solved by the schedule, we can use the `schedgen-query` tool:

```
$ schedgen-query.py -c -d contrib/example_data.csv \
    -q unsolved contrib/example_schedule.csv
special01.smt2
unsolved.smt2
```

[5]The script used to generate these graphs is the script `contrib/cdf.py` in the veriT-schedgen repository.

Category	Benchmarks
AUFLIA/20170829-Rodin	3272
AUFLIRA/nasa	18446
AUFLIRA/why	1271
UF/20170428-Barrett	2963
UF/sledgehammer	*4140*
UFLIA/boogie	1191
UFLIA/simplify2	2345
UFLIA/sledgehammer	3462
UFLIA/tokeneer	1872

Table 1
Categories with more than 1000 benchmarks.

The option `-q unsolved` asks the tool to print the list of all benchmarks not solved. An alternative is `-q compare` that shows benchmarks not solved by any strategy. To see the benchmarks that are solved by any strategy together with the minimal solving time the `-q best` option can be used. Finally, `-q schedule` lists all benchmarks solved by the given schedule together with the predicted solving time (without jitter).

Figure 2 is very artificial – it is based on synthetic example data. In the next section we see some real examples.

6. Evaluation

This section shows some empirical datapoints related to the real world performance of the schedules generated by veriT-schedgen. The evaluation of veriT-schedgen is simplified by the fact that the generated schedules do not depend on randomized heuristics.

Nevertheless, it is useful to know how many problems are solved by such schedules. Especially in comparison to simpler approaches, such as using only the best overall strategy, or constructing a schedule with a simple greedy approach. We generate the greedy schedule we start with an empty schedule and then iteratively add the timeout, strategy pair that solves the most benchmarks not yet solved. We stop when the timeout is reached.

In machine learning, it is common to use k-fold cross validation for such evaluations. This means that the data set is split into k subsets of equal size. Then the model (here: the schedule) is trained on the union of $k-1$ sets and evaluated of on the remaining set. This is repeated k times and the results are combined (for example by calculating the mean). In our evaluation we use veriT-schedgen to generate one schedule for each training set.

A major problem of such an evaluation is that benchmark libraries, such as the SMT-LIB library, are biased towards the types of problem submitted to the library. Furthermore, the SMT-LIB benchmark collection organizes the benchmarks in categories according to application and submitter. The benchmarks of each category likely share many characteristics. Furthermore, some sets contain thousands of benchmarks, while

Solved	Split 1	Split 2	Split 3	Split 4	Split 5	Mean (σ)
virtual best	1355	1318	1328	1293	1338	1326 (23.1)
generated	**1349**	**1306**	**1317**	**1283**	**1326**	**1316 (24.4)**
greedy	1340	1303	1314	1275	1326	1312 (24.7)
best strategy	1311	1267	1280	1243	1299	1280 (26.7)
PAR-2 score						Mean (σ)
virtual best	160 501	174 213	170 347	182 938	167 371	171 074 (8 316)
generated	**164 388**	**179 811**	**175 453**	**187 851**	**172 102**	**175 921 (8 736)**
greedy	169 183	183 040	178 817	192 482	173 655	179 435 (8 974)
best strategy	176 844	192 438	187 772	201 248	180 966	187 854 (9 606)

Table 2
Results of the first scenario.

other contain only tens. To address this issue to some degree the evaluation only uses categories with more than 1000 benchmarks, and uses three different scenarios. The first is a random sample of 1000 benchmarks from each category. The second is to hold out one category as the test set. This indicates how well a schedule performs on a category that was not seen during the schedule generation. Finally, to see how well such schedule generation can be used to adapt an SMT solver to one application, the last scenario uses only one category of benchmarks.

Our experiments here use the experimental data gathered for the evaluation of a recently published technique for SMT solvers [8].[6] The schedules and graphs used for this publication were generated by veriT-schedgen.

Instead of running the solver with the schedules, we can estimate the outcome of each schedule with the simulator. All generated schedules use a timeout of 180 s and the time slices 1, 2, 3, 4, 5, 8, 16, 32, 64. Since the benchmark solving rate decreases rapidly with longer timeouts, we can reduce the granularity of the time slices for bigger values. In the benchmarks used for the evaluation nine categories contain more than 1000 benchmarks. Table 1 lists them. The evaluation uses two metrics: the total number of solved benchmarks and the PAR-2 score. The PAR-2 score is the sum of all solving times plus twice the timeout for each benchmark not solved. Hence, a lower PAR-2 score is better. The PAR-2 score is used to score competitions such as the SAT competition [10].

Overall, the the first scenario uses 9000 benchmarks and 5-fold cross validation over all benchmarks. Therefore, the schedules are generated on 7200 benchmarks and evaluated on 1800 benchmarks. Table 2 shows the result of the evaluation. The first part shows the total number of benchmarks solved by the virtual best solver, the generated optimal schedule, the schedule generated by the greedy approach, and the single strategy that solves the most benchmarks. Here, the virtual best solver is constructed from all available strategies. The second part shows the corresponding PAR-2 scores. We see that the generated optimal schedule is much better in all split, but split 5. In this case both the greedy and the generated optimal schedule solve the same number of problems. Nevertheless, even in

[6] The data is available on Zenodo [9].

Solved	Split 1	Split 2	Split 3	Split 4	Split 5	Mean (σ)
virtual best	248	268	265	271	271	265 (10)
generated	**235**	**262**	**261**	**264**	**265**	**257 (13)**
greedy	237	255	259	261	262	255 (10)
best strategy	222	245	244	252	245	242 (11)
PAR-2 score						Mean (σ)
virtual best	209 094	202 099	202 821	201 265	200 737	203 203 (3 388)
generated	**214 328**	**204 673**	**204 829**	**203 835**	**203 863**	**206 306 (4 508)**
greedy	215 158	207 890	206 677	205 539	206 027	208 258 (3 957)
best strategy	218 768	210 590	210 727	208 061	210 516	211 732 (4 086)

Table 3
Results of generating schedules for UF/sledgehammer.

this case the generated schedule has a better PAR-2 score. Overall, the PAR-2 scores also show that the optimal schedule is better than the greedy schedule and much better then not using a schedule at all. Furthermore, the variance is also reduced by using an optimal schedule.

The second scenario uses the same 9000 benchmarks, but for each fold a different category is served as a test set. Since there are nine categories, this scenario is a 9-fold cross validation with training sets containing 8000 benchmarks, and test sets containing 1000. Note that the share of benchmarks that are unsolvable for veriT (e.g., satisfiable benchmarks) can vary widely from category to category. As a consequence the mean of the results is not very meaningful for this experiment. The results are in Table 4. To save space, the names of the categories are simplified. The category "Lsledgehammer" corresponds to the category UFLIA/sledgehammer. This scenario is, in a certain sense, a stress test for the schedules. Each category might have some characteristics not shared with any other category. The generated schedule seems to struggle with the AUFLIA/20170829-Rodin category. It only improves slightly over the greedy schedule, while the virtual best solver is significantly better. For three categories both schedules and the virtual best solver solve the same number of benchmarks. In one case (UFLIA/tokeneer) the optimal schedule has the worst PAR-2 score. Furthermore, in two cases (AUFLIA/20170829-Rodin and UFLIA/simplify2) the heuristic optimization of the order slightly increased the PAR-2 score. Overall, optimal schedules are clearly doing well in this scenario too.

The third, and final, scenario uses only the category UF/sledgehammer. This category contains problems generated by Sledgehammer [11]. Since veriT is also used by Sledgehammer, this is a sensible choice. The UF/sledgehammer category contains 4140 benchmarks. Again 5-fold cross validation is used: 3312 benchmarks for optimization and 828 for testing. Table 3 shows the results that mirror the other two scenarios. For two splits, all schedules solved the same number of benchmarks, and for the optimal schedule improves only slightly in the first split. Both effects are probably the result of the relative small number of benchmarks in the test set. In any case, the optimal schedule improves on the PAR-2 score over the greedy schedule.

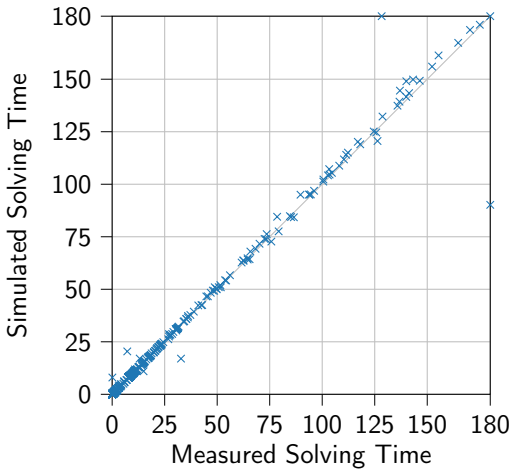

Figure 3: Simulation versus reality.

We also collected the predicted sum of solving times for the schedule before and after order optimization. The order before optimization is an implementation detail of the toolbox and not deterministic. The average ratio of the time before optimization and after is 0.71.

To finish this section, let us have a look at the real world behaviour of schedules in comparison with the simulation. This allows us to judge how reliable the simulated results presented so far are. The data gathered for Figure 3 in [8] serves as the base for this. Figure 3 shows a scatter plot of the solving times for the 180 s schedule with quantifier simplification by unification. The closer a point is to the diagonal, the more accurate the simulation was. This shows that the simulation is fairly accurate. In a few cases, benchmarks predicted to be solved are not solved in reality and vice versa. Those are benchmarks that are solved just before the time slice of a strategy ends. In this case a delay can make the benchmark unsolvable. The opposite is also possible: if a benchmark is solved during data gathering after the timeslice ends, the jitter can reduce the solving time below the timeout. Overall, this graph indicates that the simulation is a good tool to asses the efficacy of a schedule.

The evaluations has been performed on a laptop with an Intel i7-7600U CPU and 16 GB of memory. Generating the 19 schedules took 39 min 16 s overall. Hence, generation time is short. To generate the schedules for veriT at the SMT competition, GNU parallel [12] was used to generate the schedules for the different logics in parallel.

7. Conclusion

The usage of strategies with automatic theorem provers has attracted considerable attention. Many sophisticated systems exist [e.g., 13, 14, 15]. These systems often support the search for new strategies and can select strategies based on features of the input problem. They also often incorporate machine learning. Our approach is very

simple. It is oblivious to the properties of the problems involved and generates static schedules.

The veriT-schedgen toolbox itself is now mature. It has been used to generate schedules for veriTs participation in the 2021 and 2020 SMT competition. Furthermore, it helped in the evaluation of a new SMT solving technique [8]. Since it is independent of veriT, we hope that it can be used by developers of other theorem provers. We plan to extend the toolbox with two additional components. The first is a flexible schedule executor that can also execute strategies in parallel. The second is a strategy generation tool that can generate new strategies with good performance from a specification of the available parameters. A good starting point for this can be an algorithm based on random permutations such as the one used by MaLeS [13].

Acknowledgments

The usage of strategy scheduling in veriT was started by Haniel Barbosa. We thank Pascal Fontaine and the anonymous reviewers for many comments that improved the text. We thank Jasmin Blanchette for insightful discussions on the role of schedules. The author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka).

Table 4
Results of the second scenario.

Distance	Rodin	nasa	why	Barrett	sledgehammer	boogie	simplify2	Lsledgehammer	tokeneer	Mean (σ)
virtual best	764	984	982	535	337	689	995	424	922	720 (255.8)
generated	**746**	**984**	**982**	**527**	**332**	**677**	**992**	**406**	**922**	**714 (260.1)**
greedy	745	984	982	517	328	673	991	402	922	711 (262.4)
best strategy	716	981	979	491	296	648	893	374	922	693 (264.4)
PAR-2 score										Mean (σ)
virtual best	85 637	5 761	6 483	167 838	238 974	112 173	1 994	208 430	28 080	100 939 (92 338)
generated	**92 244**	**5 801**	**6 591**	**172 126**	**242 243**	**118 134**	**3 532**	**215 400**	**28 085**	**103 801 (94 333)**
greedy	95 380	6 146	6 697	176 215	245 105	120 403	9 799	218 310	28 081	105 909 (94 946)
best strategy	102 387	6 840	7 585	183 730	253 850	127 337	38 707	226 300	28 081	110 879 (95 467)

References

[1] T. Bouton, D. C. B. de Oliveira, D. Déharbe, P. Fontaine, veriT: An open, trustable and efficient SMT-solver, in: R. A. Schmidt (Ed.), CADE 22, volume 5663 of *LNCS*, Springer Berlin Heidelberg, 2009, pp. 151–156. doi:10.1007/978-3-642-02959-2_12.

[2] C. Barrett, P. Fontaine, C. Tinelli, The satisfiability modulo theories library (SMT-LIB), , 2016. Accessed: 2021-09-28.

[3] G. Cornuéjols, M. A. Trick, M. J. Saltzman, A tutorial on integer programming, , 1995. Accessed: 2022-04-07.

[4] A. Wolf, R. Letz, Strategy parallelism in automated theorem proving, in: Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, AAAI Press, 1998, pp. 142—146. doi:10.5555/646811.706867.

[5] H. Barbosa, J. Hoenicke, H. Antti, 16th international satisfiability modulo theories competition (smt-comp 2021): Rules and procedures, https://smt-comp.github.io/2021/rules.pdf, 2021. Accessed: 2021-08-08.

[6] S. Mitchell, M. O'Sullivan, I. Dunning, PuLP : A linear programming toolkit for Python, 2011.

[7] T. Bouton, D. Caminha B. De Oliveira, D. Déharbe, P. Fontaine, GridTPT: a distributed platform for theorem prover testing, in: 2nd Workshop on Practical Aspects of Automated Reasoning (PAAR), Edinburgh, United Kingdom, 2010, pp. 33–39.

[8] P. Fontaine, H.-J. Schurr, Quantifier simplification by unification in smt, in: B. Konev, G. Reger (Eds.), Frontiers of Combining Systems, Springer International Publishing, Cham, 2021, pp. 232–249. doi:10.1007/978-3-030-86205-3_13.

[9] Quantifier Simplification by Unification in SMT, Zenodo, 2021. doi:10.5281/zenodo.5088868.

[10] M. Heule, M. Järvisalo, M. Suda, SAT race 2019, , 2019. Accessed: 2022-02-28.

[11] J. C. Blanchette, S. Böhme, L. C. Paulson, Extending Sledgehammer with SMT solvers, Journal of Automated Reasoning 51 (2013) 109–128. doi:10.1007/s10817-013-9278-5.

[12] O. Tange, GNU parallel 20210722 ('blue unity'), 2021. URL: https://doi.org/10.5281/zenodo.5123056. doi:10.5281/zenodo.5123056, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.

[13] D. Kühlwein, J. Urban, MaLeS: A framework for automatic tuning of automated theorem provers, Journal of Automated Reasoning 55 (2015) 91–116. doi:10.1007/s10817-015-9329-1.

[14] E. K. Holden, K. Korovin, Heterogeneous heuristic optimisation and scheduling for first-order theorem proving, in: F. Kamareddine, C. Sacerdoti Coen (Eds.), Intelligent Computer Mathematics, Springer International Publishing, Cham, 2021, pp. 107–123. doi:10.1007/978-3-030-81097-9_8.

[15] J. Scott, A. Niemetz, M. Preiner, S. Nejati, V. Ganesh, Machsmt: A machine learning-based algorithm selector for smt solvers, in: J. F. Groote, K. G. Larsen (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, 2021, pp. 303–325. doi:10.1007/978-3-030-72013-1_16.