Efficient Solver Scheduling and Selection for Satisfiability Modulo Theories (SMT) Problems

David Mojžíšek and Jan Hůla
University of Ostrava, 30. dubna 22, 702 00 Ostrava, Czech Republic

Keywords: Satisfiability Modulo Theories (SMT), Solver Scheduling, Algorithm Selection, Dynamic Scheduling.

Abstract: This paper introduces innovative concepts for improving the process of selecting solvers from a portfolio to tackle Satisfiability Modulo Theories (SMT) problems. We propose a novel solver scheduling approach that significantly enhances solving performance, measured by the PAR-2 metric, on selected benchmarks. Our investigation reveals that, in certain cases, scheduling based on a crude statistical analysis of training data can perform just as well, if not better, than a machine learning predictor. Additionally, we present a dynamic scheduling approach that adapts in real-time, taking into account the changing likelihood of solver success. These findings shed light on the nuanced nature of solver selection and scheduling, providing insights into situations where data-driven methods may not offer clear advantages.

1 INTRODUCTION

We are introducing a series of innovative concepts and refinements to the process of selecting solvers from a portfolio to solve problems of Satisfiability Modulo Theories (SMT). Our methodology exhibits similarities to MachSMT (Scott et al., 2021), albeit with the potential for superior performance in various operational scenarios. Furthermore, we show that in many cases trained Machine Learning (ML) model can be omitted and one can even ignore features of problem at hand and simply run the same schedule, designed on training set, for all testing examples. As an illustrative case, we have chosen the scheduling of SMT solvers, but it is essential to underscore that our proposed techniques are universally applicable to situations where algorithm selection from a diverse portfolio is a requisite.

The central emphasis of our approach revolves around comparing the trained machine learning algorithm and the scheduling approach without any training. Importantly, the role of this algorithm is not merely to identify a single best solver for a given problem but rather to construct an efficient schedule similar to (Pimpalkhare, 2020). When creating this schedule, we consider the interaction between the solving capabilities of different algorithms. This consideration is crucial, as it helps us steer clear of scenarios where two algorithms, capable of solving the same class of problems, also share a propensity for failure on similar examples, when run consecutively.

Our contribution comprises two key parts. First, we propose a novel solver scheduling approach that significantly improves the PAR-2 metric in selected benchmarks. Second, we explore the necessity of employing a ML predictor. Our investigation reveals that, in certain cases, scheduling based on crude statistical analysis of training data performs just as well, if not better, than an ML predictor. The superiority of this scheduling is clearly visible especially in case when it competes against a single selected solving algorithm. These findings highlight the nuanced nature of solver selection and scheduling, shedding light on instances where machine learning methods do not offer clear advantages.

2 PRELIMINARIES

2.1 Domain Description - Satisfiability Modulo Theories

The SMT-LIB standard (Barrett et al., 2010) defines the language and standardized theories. A repository of benchmark problems is maintained within the SMT-LIB standard.
2.2 SMT Solvers

Satisfiability Modulo Theories (SMT) solvers are indispensable tools in formal methods, effectively automating logical reasoning across a multitude of theories and their combinations. They find applications in critical domains, including software and hardware verification, symbolic execution, and constraint solving (de Moura and Björner, 2012), (Godefroid et al., 2012), (Björner and de Moura, 2014). It is worth noting that the performance of different SMT solvers can vary significantly when applied to the same problem instance. One solver may outperform others in a specific instance, while the situation might reverse when faced with a different problem. This variability is often linked to the unique features and characteristics of each problem instance.

This interplay between problem instances and solver performance underscores the complexity of selecting the most appropriate solver for the problem at hand. To tackle this challenge, practitioners often rely on extensive experimentation, benchmarking, and profiling of solvers to gain insight into their strengths and weaknesses. Such insights enable the development of strategies for effectively matching problem instances with the most suitable solvers, contributing to more efficient and reliable automated reasoning. Understanding these dynamics between problem features and solver performance is crucial in harnessing the full potential of SMT solvers across various application areas.

2.2.1 Notable SMT Solvers

It is not a goal to describe all existing solvers, but for reference we give a few examples of solvers often recurring in our predictions and schedules.

- Z3. Developed at Microsoft Research (De Moura and Björner, 2008).
- CVC4. Developed jointly by Stanford University and the University of Iowa (Barrett et al., 2011).
- MathSAT. A joint project of the Fondazione Bruno Kessler (FBK-irst) and the University of Trento (Bruttomesso et al., 2008).

2.3 Problem Description

Let $S = \{s_1, \ldots, s_l\}$ be a set of $l$ SMT solvers that we have at our disposal. Our goal is to produce an effective algorithm which on a per-instance basis creates a schedule from the set $S$. This means that we want to find a function $f_\theta$ (parameterized by $\theta$) that takes a representation of an SMT formula $q$ and the set $S$ as input and outputs an ordered tuple $f_\theta(q,S) = ((i_1,t_1), \ldots, (i_n,t_n))$ where $i_j$’s are indices of selected solvers and $t_j$’s are times assigned to these solvers. The sum $\sum_i t_i \leq t_{\text{max}}$ is restricted by $t_{\text{max}}$ which is the maximum time we are willing to spend solving the problem.

Given a formula $q$ and its schedule $f_\theta(q,S)$, we measure how long it takes to solve the formula using this schedule. We denote this measurement by $M(q,f_\theta(q,S))$ and set it to a constant number $t_{\text{pen}}$.

Note that this definition of problem wraps all cases discussed in our experiments. The schedule with one solver can correspond to selecting the single best one, and the greedy schedule treats the selection of the solver as a constant function with respect to the varying input.
(with $t_{pen} > t_{max}$) if the formula was not solved within the time limit $t_{max}$.

We assume that the problems/formulas we want to solve come from an unknown distribution $P$ and that we have a finite set of independent and identically distributed samples $Q = \{q_1, \ldots, q_m\}$ where $q_i \sim P$.

Because the distribution $P$ is unknown, we can only try to minimize an approximation to the objective function samples in $Q$:

$$\hat{\theta} = \arg \min_{\theta} \sum_{q_i \in Q} M(q_i, f_\theta(q_i, S)).$$

The parameters of this function cannot be directly optimized with respect to the objective function by gradient-based methods because it involves discrete choices.

### 3 SCHEDULING OF SMT SOLVER

In this paper, we compare a machine learning model that selects the best solver for a given problem instance, a dynamic schedule based on this selection, and a trivial way to schedule solvers without even looking at the problem. For that reason, we first need to clarify how solvers are ranked with a machine learning model.

#### 3.1 Interval Prediction

Our predictor is inspired by the Empirical Hardness Model (EHM) (Leyton-Brown et al., 2009) used in MachSMT. For each logic and solver combination, we train a dedicated model to predict the runtime required for that solver to solve an SMT problem instance based on its features. The key change here is that we don’t directly train the EHM. Instead, we divide the runtime into multiple indexed intervals represented as $I = \{i_1, i_2, \ldots, i_n\}$, defined by their endpoint values $t(i_k)$. Our goal is to predict in which of these intervals the solver will complete the instance. This transformation turns the regression task into a classification problem.

We explored various methods to divide the runtime into intervals. It is crucial to consider that most instances are either solved very quickly or remain unsolved. A uniform split would result in unbalanced classes. We found that using a power or exponential function for splitting worked better, but also left us with empty classes. Ultimately, we opted for creating perfectly balanced classes by using quantiles derived from the solving times of the specific solver on instances from the training set. This resulted in four classes. For unsolved instances, we introduced a fifth class, making the last interval to correspond to a timeout of $t(i_k) = 2 \times timeout$ (corresponding to the penalty for not solving the instance that is used to compute the PAR-2 score). As a result, the endpoints of the intervals vary for each predictor, and each class has a different meaning.

Regarding the predictor’s output, our interest extends beyond merely predicting the interval. We seek scores for each interval that can be normalized and interpreted as probabilities representing the solver’s likelihood of solving the instance within that specific interval. Once we have these probabilities for a given solver, we can compute an upper bound for the expected runtime by calculating the expected value of the endpoints in each interval: $E_{i \in I} [t(i) \times p_s(i)]$. Here, $p_s(i)$ is the probability that solver $s$ will solve the given formula in interval $i$. This expected runtime serves the same purpose as EHM, helping to rank the solvers.

The predictor predicts the runtime from the features of the input problem. **Problem features** are syntactical properties of the input. They include information like frequencies of problem grammatical constructs or some meta-information like file size. Since we used a feature extracted from the publicly available MachSMT repository, we refer to it for further information. It is important to note that features are extracted in the same way for each logic, thus there appear useless features or features that are always 0. MachSMT leverages it by dimensionality reduction (PCA). We used a framework named AutoGluon (Erickson et al., 2020) that automatically selects only a relevant subset of the available features. The reason for that is that we wanted a fast and simple way to reproduce MachSMT results, but with interval predictions. The feature engineering is outside the scope of this study. However, we note here that in the past, we (Hula et al., 2021) tried to use Graph Neural Networks (GNNs, (Hamilton, 2020)) to automatically extract features from Directed Acyclic Graph (DAG) formula representation and train predictor end-to-end. This is inspired by other deep learning tasks, where expert features are replaced by learned ones (by projecting the input into low-dimensional continuous space), for example, in image processing, where filters in trained Convolutional Neural Network (CNNs) extract features in the input image (Albawi et al., 2017) and can project them into an embedding vector in the penultimate layers. The method was shown to be suitable for feature extraction without expert feature en-

---

2In the sense of pessimistic estimate because endpoints of intervals are used.

3https://github.com/MachSMT/MachSMT
gineering, but too expansive for computation com-
pared to simple syntactical MachSMT features. The
attempts to extract features from logical/mathemati-
cal expressions with (Graph) Neural Networks have
been recently and extensively studied, for example by
(Crouse et al., 2019), (Glorot et al., 2019) or (Wang
et al., 2017). One drawback of these features in the
context of logical formulae embedding is their inter-
pretability; for CNNs there are at least some methods
that can visualize what a model is focusing on (Sel-
varaju et al., 2017).

3.2 Greedy Selection

Here we describe a greedy method that can outper-
form intricate algorithm selections and is based on the
specifics of the benchmarks used and its crude statis-
tics. The significant feature of the benchmarks we are
dealing with is that the runtime is in many cases so
long that some solvers solve most instances within a
small fraction of this runtime. If a predictor from the
previous section selects only one best solver, without
schedule, we assign it the whole runtime.

We argue that in some cases where we do not care
about edge instances that need much more runtime
than average to be solved, it is more important to ask
how to use runtime resource in the most efficient way.
Exploiting the whole portfolio by creating a schedule
rather than having the ability to select a single solver
that might fail is a natural way to do it. The experi-
mental part confirms that those edge instances form a
minority and schedule-based methods solve more in-
stances overall.

Our goal is to solve as many instances as possi-
ble within the time limit (and achieve a good score),
thus wasting runtime when the probability of success
greatly diminishes after a few seconds is in direct con-
tradiction.

There are many ways to build an effective or a
baseline schedule. One can, for example, take mul-
tiple solvers that were ranked as best by the predictor
and split the runtime between them. We decided to go
a different route by not looking at the given example
at all.

We select $n$ solvers for our schedule and assign
them runtime in the following way. First, we run
through the whole portfolio of solvers and look at how
many instances they can solve on training set: 1) we
select the solver that solves the most instances 2) we
remove all instances which the selected solver solved
from further consideration. 1) and 2) are repeated un-
til $n$ solvers are selected. The second parameter is the
threshold $q$ that can be between 0 and 1. To every
solver in the schedule we assign the time $T_i$ in which
it can solve $q \cdot K_i$, where $K_i$ is the number of instances
that solver $i$ solves, as its base runtime. We try to se-
lect $n$ and $q$ to balance the number of solved instances
and give each solver enough runtime to solve as much
as it is capable of. The sum of times assigned to se-
lected solvers based on parameters $n$ and $q$ should not
exceed maximal runtime. The actual timeout is di-
vided between the selected solvers proportionally by
$T_i$ (so they get a bit more time). The order in which
solvers are used is also based on time $T_i$ (ascending).

This method does not need any learning or knowl-
edge about the problem at hand. There can be found
some similarities to the method proposed by (Ama-
dini et al., 2014), who, however, used the distance be-
tween the example and examples in the training set to
select the best sub-portfolio of solvers.

3.3 Dynamic Schedule

This schedule is constructed greedily, relying on the
predictions of the model. The process starts by schedul-
ing the solver with the smallest expected value and
running it for the duration of it is the first interval
(intervals are different for each solver). If the prob-
lem remains unsolved during this interval, we recom-
pute the expected value for the chosen solver by ig-
noring the first interval. Concretely, we just normal-
ize the probabilities for the remaining intervals and
then compute the expected value over these intervals
again. We also need to subtract the time for which
the solver has already run to obtain the upper bound
of the expected remaining runtime.

If this new expected value is smaller than the
second-smallest expected value computed previously,
then we run the same solver for the next interval; oth-
erwise, we switch to the solver with the next-smallest
expected value.

We continue in this fashion, always deciding
which solver to run after every interval, until the time-
out is reached. For this reason, the schedule is dy-
namic as it is calculated on the fly. It is also a preemp-
tive schedule (Zhao et al., 1987), which means that
individual solvers can be stopped and resumed later.
This means that the result is theoretical since in prac-
tice we might not be able to freeze solvers and resume
the computation later easily. Nevertheless, it can lead
to good improvement, and we propose it as a univer-
sal method rather than as a tool that only schedules
SMT solvers. The process is also shown in the figure
2. This way wallclock time is not reduced at the ex-
 pense of the CPU, as stopped solvers’ state could be
stored in the memory.
4 EXPERIMENTS

4.1 Dataset

This experimental part of the paper is intended as a case study comparing approaches with and without scheduling, as well as with and without prediction in the context of SMT solver selection. For this purpose, we selected five SMT-LIB logics (Barrett et al., 2016) and data from the 2019 SMT competition\(^4\), single-query track. It includes solving times for different solvers on different logics. The timeout is always set to 2400 s. In Table 1, we show the summary of the benchmarks used.

For the used solvers, we performed a little pruning. First, we removed solvers if they missed solving time data on the majority of examples. For the rest, we removed examples on which some solvers miss data. To have clearer data and not confuse models, we also removed solvers with very little or no solved instances at all. To make competition fair, we removed the "Par4" solver, which runs multiple SMT solvers in parallel (uses more CPU time to minimize wall-clock time) and in case of QF_LIA the SPASS-SAT solver designed specifically to work on that and similar logics while outperforming all other solvers by large margin. Lastly, if there were two versions of the same solver with almost the same results (Z3), we removed one. We keep the full names of the solvers used as they are in the SMT-COMP data table in graph legends, so our choice is transparent.

Table 1: Number of problems and solvers per benchmark and number of solved examples by at least one solver.

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th># of problems</th>
<th># of solvers</th>
<th># of solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_NRA</td>
<td>2482</td>
<td>9</td>
<td>2659</td>
</tr>
<tr>
<td>UFNIA</td>
<td>6253</td>
<td>5</td>
<td>4875</td>
</tr>
<tr>
<td>AUFLIA</td>
<td>1638</td>
<td>7</td>
<td>1464</td>
</tr>
<tr>
<td>QF_LIA</td>
<td>3136</td>
<td>8</td>
<td>3084</td>
</tr>
</tbody>
</table>

\(^4\)https://smt-comp.github.io/2019/

4.2 Evaluation

For the evaluation we use K-fold cross-validation protocol, it allows us to infer results for all problems in the benchmarks. The data set is uniquely divided into \(k\) folds (in our case \(k = 5\)) with the training and test portion. All five test datasets are mutually exclusive, but together cover whole original dataset. All models are trained from random state on each of those folds. In the same way, our greedy schedule is always constructed again for a new fold. Solvers and schedules are not only compared by the number of instances but also by the standard PAR-2 score.

PAR-2 score is the sum of runtimes on all problems within the dataset. Instances on which the solver fails to decide are penalized by two times maximal runtime. The objective is to minimize this score.

The PAR-2 improvement is always related to the best-solver performance. For this paper, we decided to compare the following:

- Virtual Best Solver (VBS). This is a hypothetical optimal algorithm that selects the best possible solver for each problem. It represents an upper bound for a possible improvement.
- Single best approach chooses the best solver according to the predictions of the model and runs it until the timeout. In this work LightGBM is the gradient boosting framework, that we always used as our predictor (Ke et al., 2017).
- Greedy schedule selects always \(n\) solvers that together solve the most instances in the training set and run them consecutively for the assigned time (based on parameter \(q\), computed from runtimes on solved problems in training set). Stops when reaching the maximum runtime or solve the example. For this work, we fixed \(n = 3\) and \(q = 0.8\).
- Dynamic schedule is a preemptive schedule that runs multiple solvers and their order is based on evolving expected solving time as described in 3.3.
Table 2: The result table comparing 3 approaches to the Best solver and to best possible selector called VBS.

<table>
<thead>
<tr>
<th>Benchmark:</th>
<th>UFNIA</th>
<th>AUFLIA</th>
<th>QF.NRA</th>
<th>QF.LIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Solver</td>
<td>solver</td>
<td>Vampire-4.3-smt</td>
<td>Vampire-4.4</td>
<td>Yices 2.6.2</td>
</tr>
<tr>
<td></td>
<td>solved</td>
<td>4034</td>
<td>1382</td>
<td>2165</td>
</tr>
<tr>
<td></td>
<td>PAR-2</td>
<td>13 189 767</td>
<td>1 242 230</td>
<td>3 307 748</td>
</tr>
<tr>
<td>VBS</td>
<td>solved</td>
<td>4875</td>
<td>1464</td>
<td>2659</td>
</tr>
<tr>
<td></td>
<td>PAR-2</td>
<td>7 929 162</td>
<td>851 466</td>
<td>918 030</td>
</tr>
<tr>
<td></td>
<td>PAR-2 impr. ceiling</td>
<td>66.35 %</td>
<td>45.89 %</td>
<td>260.3 %</td>
</tr>
<tr>
<td>Single Best</td>
<td>solved</td>
<td>4510</td>
<td>1403</td>
<td>2498</td>
</tr>
<tr>
<td></td>
<td>PAR-2</td>
<td>10 126 778</td>
<td>1 148 163</td>
<td>1 712 536</td>
</tr>
<tr>
<td></td>
<td>PAR-2 impr.</td>
<td>30.25 %</td>
<td>8.19 %</td>
<td>93.15 %</td>
</tr>
<tr>
<td>Greedy Schedule</td>
<td>solved</td>
<td>4655</td>
<td>1485</td>
<td>2582</td>
</tr>
<tr>
<td></td>
<td>PAR-2</td>
<td>9 174 250</td>
<td>970 656</td>
<td>1302598</td>
</tr>
<tr>
<td></td>
<td>PAR-2 impr.</td>
<td>43.77 %</td>
<td>27.98 %</td>
<td>153.93 %</td>
</tr>
<tr>
<td>Dynamic Schedule</td>
<td>solved</td>
<td>4672</td>
<td>1431</td>
<td>2579</td>
</tr>
<tr>
<td></td>
<td>PAR-2</td>
<td>9 501 057</td>
<td>1 020 999</td>
<td>1 320 703</td>
</tr>
<tr>
<td></td>
<td>PAR-2 impr.</td>
<td>38.82 %</td>
<td>21.67 %</td>
<td>150.45 %</td>
</tr>
</tbody>
</table>

Figure 3: Example result plot for UFNIA logic. It shows how many problems (y-axis) a given method solves up to a certain time (x-axis). The greedy schedule performed best at the beginning. Schedules are different for each fold, but to get a better idea here is an example of a greedy schedule for a single fold: ((z3-4.8.4, 1.83 s), (CVC4-2019-06-03, 125.5 s), (vampire-4.3, 2272.67 s)). In the end, the three methods achieved a similar number of solved examples.

### 4.3 Results

The results of three selected methods for four selected benchmarks are summarized in table 2 and for UFNIA logic also shown in figure 3. The remaining figures with graphs for other logics can be found in Appendix.

Our single best predictor based on interval prediction performs very close to the EHM predictor in the original MachSMT paper (we compare it to the PAR-2 score for SolverLogic found on their Github⁵, although direct comparison might be inaccurate since the solver set could be different. Concretely, for

⁵https://github.com/MachSMT/MachSMT/tree/main/data/results/2019/SQ

365
selected logics, they achieved: UFNIA: 10.58 %, QF NRA: 92.27 %, AUFLIA: 12.44 % improvement over best solver, for QF LIA we effectively removed the best-performing solver SPASS-SAT in 2019 SQ track so comparison does not make much sense. We note that there is no significant change in results with different prediction regimes; one can predict directly solving time, intervals as we do or even just classify a problem in binary fashion to solved/unsolved (and for an instance pick single best solver with the highest probability for solving) while using the same prediction model.

Overall, the greedy approach, which creates the same schedule for all problems only by looking at solving times and number of solved instances in the training data, performed comparably or better than other approaches. It is clear that this is caused by a combination of long maximal runtime and fast solving times for solved instances. Selecting a single best solver is easiest and would probably be the best option if the solving time was close to maximal runtime on average. For example, in QF NRA some solvers solved 75% of instances under 5s, and rarely did the time needed to solve 75% of instances exceed 20s. However, the greedy method is sensitive to the correct time assignment for selected solvers. The dynamic schedule worked pretty reliably on all benchmarks.

5 RELATED WORK

Algorithm selection and scheduling (Kadioglu et al., 2011) is recognized as an important topic as a consequence of the need for fast and reliable problem handling in practical applications. Portfolio-based algorithm selection with a machine learning model was popularized by Leyton-Brown et al. (Leyton-Brown et al., 2003).

The proposed work comes from MachSMT (Scott et al., 2021), their main approach is to try to select only one solver from the whole portfolio. Except EHM they also incorporate pairwise predictor for solver selection. In the context of the selection of logic solvers with EHM their predecessor is SATZilla (Xu et al., 2008).

Our work is also related to various approaches that use ML for solver scheduling, especially in the domain of SMT (Balunovic et al., 2018) use imitation learning techniques to schedule strategies within the Z3 solver. Similarly, (Ramírez et al., 2016) uses an evolutionary algorithm to generate strategies for the Z3 solver.

For an overview of various use cases of ML methods for combinatorial problems and algorithm selection, see the following survey papers: (Bengio et al., 2020; Kerschke et al., 2019; Talbi, 2020). For a more specific overview focused on GNNs, see (Cappart et al., 2021).

6 CONCLUSION

In conclusion, our research has introduced a novel approach to solving selection and scheduling that takes advantage of dynamic scheduling policies and machine learning techniques. While our initial focus was on the scheduling of Satisfiability Modulo Theories (SMT) solvers, it is important to highlight that our methodology extends beyond this specific application. The selection and scheduling of algorithms is widespread in various domains. Our work was not intended solely to solve SMT solver scheduling, but rather used scheduling of SMT solvers as a practical example.

Our investigation revealed that the greedy schedule, which does not rely on machine learning predictions but instead selects solvers based on training data statistics, often performed as well as or better than the machine learning-based approach.

The results of this case study showed that solver selection and scheduling are nuanced tasks and that machine learning methods may not always offer clear advantages over simpler approaches based on statistics.

In future work, it would be valuable to further explore the dynamic scheduling approach, considering its potential in scenarios where preemption and resumption of solvers may be practically feasible.

ACKNOWLEDGEMENTS

The results were supported by the Ministry of Education, Youth, and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902.

REFERENCES


APPENDIX

Additional Graphs and Dynamic Schedule Algorithm

Figure 4: On AUFLIA greedy method solved almost the same number instances as VBS. The drawback is that at the beginning of runtime it did not perform as well.

Figure 5: In QF_NRA logic the dynamic schedule performed very similarly to the greedy one, while both outperformed single solver selection by considerable amount.
Figure 6: In QF_LIA the greedy schedule did not perform as well as others. The reason behind it is that the average solving times for selected solvers in training data were very small, and at the same time very similar, thus applying the greedy method in the same way as to other logics, split the runtime between the selected 3 solvers evenly (two visible jumps in number of solved instances). This shows that our greedy method cannot be applied blindly and one should consider a different way of solving time assignment. Greedy schedule example from a single fold: ((Ctrl-Ergo-2019, 713.77 s), (CVC4-2019-06-03, 808.76 s), (z3-4.8.4, 877.45 s)), although 75% of the solved examples in the entire dataset are solved by 12.9 s, 18.88 s and 18.42 s by these solvers, respectively.

Algorithm 1: Dynamic scheduling with predictions.

```
Data: D: array of lists. List D[i] = [(t_i^1, d_i^1), ..., (t_i^n, d_i^n)] corresponds to solver i with t_i^j being the length of the j-th interval and d_i^j its score.

Result: Total runtime spent on the formula

1 runtime ← 0;
2 while runtime < timeout do
3   expectedTimes ← getExpectedTimes(D);
4   currBest ← arg min(expectedTimes);
5   nextIntLen, score ← getFirst(D[currBest]);
6   solved, time ← runSolver(currBest, nextIntLen);
7   runtime ← runtime + time;
8   if solved then
9     return runtime;
10   else
11     D[currBest] ← removeFirst(D[currBest]);
12 return 2 × timeout; // unsolved instance
```