Automatic Instance Generation for Classical Planning

Álvaro Torralba,[1] **Jendrik Seipp,**[2,3] **Silvan Sievers**[3]
[1]Aalborg University, Denmark
[2]Linköping University, Sweden
[3]University of Basel, Switzerland
alto@cs.aau.dk, jendrik.seipp@liu.se, silvan.sievers@unibas.ch

Abstract

The benchmarks from previous International Planning Competitions (IPCs) are the de-facto standard for evaluating planning algorithms. The IPC set is both a collection of planning domains and a selection of instances from these domains. Most of the domains come with a parameterized generator that generates new instances for a given set of parameter values. Due to the steady progress of planning research some of the instances that were generated for past IPCs are inadequate for evaluating current planners. To alleviate this problem, we introduce Autoscale, an automatic tool that selects instances for a given domain. Autoscale takes into account constraints from the domain designer as well as the performance of current planners to generate an instance set of appropriate difficulty, while avoiding too much bias with respect to the considered planners. We show that the resulting benchmark set is superior to the IPC set and has the potential of improving empirical evaluation of planning research.

Introduction

Automated planning aims to develop general solvers that find solutions to arbitrary sequential decision-making problems. This makes the evaluation of planners an essential part of planning research (Linares López, Celorrio, and Helmert 2013; Seipp et al. 2017). Evaluating planners on different benchmark sets may produce different results, leading to different conclusions. Not only is it important which domains we use, but also how we model them (Riddle, Holte, and Barley 2011), and which instances of a domain we select. Therefore, having a standardized benchmark set is important to increase the comparability of results across papers, and to avoid using benchmarks tailored for the proposed technique. The benchmarks from the International Planning Competition (IPC) are the current de-facto standard. This benchmark set has grown across the nine editions of the IPC so far, from 1998 to 2018 (e.g., Hoffmann and Edelkamp 2005; Hoffmann et al. 2006; Linares López, Celorrio, and Olaya 2015), and it features a diverse set of domains that pose interesting challenges for planning algorithms.

We deal with selecting a finite set of instances of a given domain to evaluate planning algorithms. So far, this was done by the IPC organizers by manually choosing suitable values for the parameters of an *instance generator*, to obtain instances of different sizes and difficulty (Vallati, Chrpa, and McCluskey 2018). However, there are several issues with the instance selection in the IPC set (Moraru and Edelkamp 2019). For example, different numbers of instances were selected per domain (from 5 to 150), which reduces the value of statistics aggregated over different domains. More importantly, the instances were selected to evaluate planners at the respective IPC, and they are not useful to evaluate current planners two decades later. In some of the domains all instances are trivially solved by modern planners, making it impossible to show significant advantages over a baseline. Furthermore, early IPC editions did not have a specialized track for optimal planning, and some of their instances are much too hard even for state-of-the-art optimal planners.

We identify desirable design principles for an instance set: (1) it is useful to evaluate current planners; (2) it avoids bias with respect to the considered planners; and (3) it keeps the spirit of the domain. Our main contribution is to frame the problem of finding a suitable instance selection that follows these principles as an optimization problem. Our tool, Autoscale, chooses one or more sequences of parameter configurations that induce a good instance set in order to evaluate current and near-future planners. Thus, given an *instance generator* for a planning domain and a set of *baseline* and *state-of-the-art* planners, Autoscale automatically generates an instance set that has the desired properties.

In other communities like SAT, there has been a lot of research on how to construct random instances (Selman, Mitchell, and Levesque 1996; Achlioptas et al. 2000; Giráldez-Cru and Levy 2015; Xu et al. 2005). In planning, some research has also explored how to generate new problems, e.g., around the phase transition (Rintanen 2004; Rieffel et al. 2014), with suitable initial states and goals for Sokoban (Bento, Pereira, and Lelis 2019), or via declarative instance generators (Fuentetaja and de la Rosa 2012). Our approach is complementary and it can be used to choose instance sets among the ones they can generate.

To evaluate Autoscale, we generate two separate sets of benchmarks for optimal and agile planning by selecting new instances for 26 domains from the standard IPC set. The results show clear advantages over the IPC set, illustrating the potential of the new benchmark set to improve the evaluation of future planning research.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

Informally, a classical *planning task* Π is defined by an initial state, a set of actions and a goal description. Given a task, a *planner* finds a *plan*, i.e., a sequence of actions that can be applied in the initial state to achieve the goal. A plan is optimal if it minimizes the summed-up cost of the actions among all plans. If the planner is guaranteed to find an optimal solution, it is an optimal planner, otherwise it is an agile planner. In both settings, we only consider solvable tasks. We denote by $t(p, \Pi)$ the runtime of a planner p to solve Π.

The IPC introduced numerous planning tasks from different problem settings, called *domains*. A planning task is typically divided into a domain and an instance file. The domain file defines the types of objects, their properties, and the action schemas. Each instance file can have a different number of objects, initial state and goals. Most domains have an instance generator,[1] which is a program that, given certain parameters and a random seed, generates a new instance of the domain. Formally, an instance generator is a function G that takes as input a tuple of parameter values ρ and a random seed $r \in \mathbb{N}^+$ and outputs a planning task $\Pi = G(\rho, r)$.

As an example, consider the Nomystery domain (Nakhost, Hoffmann, and Müller 2012), where a truck must deliver a set of packages to certain locations. To do that, there is a limited amount of fuel that is consumed by drive actions. Instances differ in the amount of fuel available, the number of locations and their connections, the number of packages, and their initial and final location. The instance generator for Nomystery accepts several parameters that allow the benchmark designer to control the difficulty of the generated instances: the number of locations, the number of packages, the number of edges between locations, the maximum fuel consumption between two locations, and the constrainedness $C \geq 1$, so that the amount of fuel in the initial state is set to C times the minimum fuel consumption required to solve the instance.

Instance Set Design Principles

We analyze desirable principles for the selection of instances from a given domain, similar to the ones considered by IPC organizers (e.g., Vallati, Chrpa, and McCluskey 2018).

Principle 1: Useful to Evaluate Current Planners The purpose of a benchmark set is to evaluate planners and compare their performance on a diverse class of problems. The selection of instances depends on our assumptions on the evaluation that will take place. Typically, there are two main goals for the evaluation of a novel algorithm: (1) understand its properties by comparing its performance against a baseline, and (2) compare it against the state of the art. Often, the main metric for comparison is coverage, i.e., the number of solved instances. Therefore, our goal is that, for any two planners A and B (possibly unknown at the time when the instance set is generated), if A is consistently faster than B on the instances of a domain, the probability that this is reflected on the coverage score should be as high as possible. For aggregated statistics to be meaningful, not only should all domains have the same number of instances, but their difficulty should also scale similarly. Otherwise, conclusions taken from the empirical evaluation may be misleading. To showcase this, we advance an excerpt of the results obtained with the IPC and our Autoscale benchmark set, as described in the experiments section. Table 1 compares three planners (for references see Table 3) in four domains with both benchmark sets. Evaluating these planners with IPC instances, we could conclude that Dual-BFWS is superior to the other two planners in these domains, both in total coverage and on a per-domain basis, since it has better or equal coverage in all domains. However, this conclusion is biased because instances are not well scaled. Instances in Grid, Driverlog, and Rovers are way too easy and therefore they do not show any differences between the planners. Using the Autoscale instances leads to a different conclusion: all three planners are complementary, with OLCFF being superior in total coverage. Of course, no strong conclusions can be taken out of only a few domains. However, using more domains will help to alleviate this issue only if the instances are well scaled.

	IPC				Autoscale			
	#	L	D	O	#	L	D	O
Grid	5	**5**	**5**	**5**	30	**17**	14	16
Driverlog	20	**20**	**20**	**20**	30	15	10	**25**
Rovers	40	**40**	**40**	**40**	30	**30**	23	28
Snake	20	5	**15**	12	30	6	**19**	16
Total	85	70	**80**	77	120	68	66	**85**

Table 1: Coverage of LAMA (L), and two IPC 2018 agile planners Dual-BFWS (D) and OLCFF (O).

Principle 2: Avoid Bias Given our first principle, a set of current planners is required in order to measure how useful the resulting instance set is. Thus, the instance selection necessarily depends on the considered set of planners, possibly introducing bias towards such a set of planners. While the bias cannot be entirely avoided, it should be minimized as much as possible, making sure that the resulting instance set is suitable to evaluate other future planners. Indeed, the Autoscale instance set featured in Table 1 was configured without using any planner after 2014, so it did not use any information regarding the two IPC 2018 planners, showing that our instance selection can generalize to future planners.

Principle 3: Keep the Spirit of the Domain The principles above aim to find an instance set that maximizes the amount of differences in performance that can be identified on a set of planners. We must not forget, however, that planning domains aim to model problems relevant in the real-world so instance sets that are pathological should be avoided. For example, in a Barman-like domain it may be more interesting to analyze planners' performance with respect to scaling the number of cocktails that must be prepared rather than arbitrarily scaling the number of ingredients of each cocktail over 100, even if that showcases more differences among the planners. This is a domain design decision, so the domain modeler should be allowed to establish constraints on the sets of instances that are acceptable.

[1] https://github.com/AI-Planning/pddl-generators

The Instance Selection Problem

Next, we model the problem of instance selection as an optimization process that complies with the design principles.

Smooth Scaling

Even though Principle 1 establishes that the objective is to observe differences in performance among the current planners, this should not be the direct optimization objective.

Rule 1 (Agnostic to Individual Planner Performance)**.** *The optimization process must not consider the individual results of all planners available for the optimization, but only consider the best planner per instance.*

The reason is that this would directly contradict Principle 2, as this metric heavily depends on the entire set of planners considered. For example, even if the set of planners is very diverse, if many of the planners belong to the same family of algorithms (e.g., heuristic search), this will bias the results towards finding instance sets adequate for them, ignoring the rest. This is related to achieving independence of irrelevant alternatives, i.e., including more planners in the optimization process should not affect the selection unless they change the state of the art (Seipp 2019).

Instead, our objective is to achieve a smooth scaling.

Rule 2 (Smooth Scaling)**.** *The optimization process aims to find a set that: (1) has easy instances solved by all planners, (2) has hard instances not solved by any current planner, and (3) instance difficulty grows smoothly.*

Condition (1) is necessary for experiments to be informative at all: if some planners do not solve any instance, no conclusions can be obtained about their relative performance. This happens in some domains of the IPC benchmark set for optimal planning. For example, Fišer, Torralba, and Shleyfman (2019) write that "In Childsnack, [they] measured about twice as many expanded states per second. However, no planner solved any instance in this domain.". Condition (2) is necessary for new algorithms to show that they can deal with instances that previous planners could not, as shown by our example in Table 1.

Condition (3) is necessary for differences in planner performance to be reflected in coverage. To see why, consider an idealized setting where a planner A whose runtime scales exponentially on an instance set $\{\Pi_1, \Pi_2, \dots\}$ ($t(A, \Pi_x) = C^x$ for some constant C) is compared to a planner B which is always faster than A by at least a factor of $K > 1$, i.e., $t(B, \Pi_x) \leq \frac{t(A,\Pi_x)}{K}$. Then there is a guaranteed difference in coverage if and only if (a) some instances are solved by B, (b) not all instances are solved by A, and (c) $K \geq C$. Otherwise, it is possible to choose instances with runtimes of A and B compatible with the exponential scalings but their coverage is equal and hence the performance difference of a factor of K is missed. For example, if $K = 2$ and $C = 3$, then (c) does not hold. For any time limit (e.g., 300 seconds), if the runtime of the last instance solved by A is close enough to the time limit (e.g., 250 seconds), the next instance cannot be solved by B within the time limit (e.g., $\frac{250 \cdot 3}{2} > 300$).

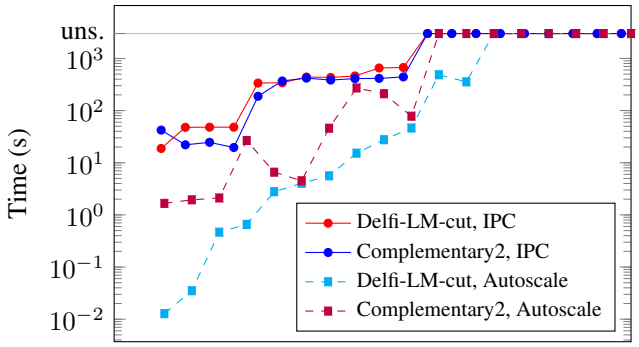

Figure 1: Runtimes of two IPC 2018 optimal planners in the Barman domain using the IPC and Autoscale instance sets. Instances unsolved by both planners are removed.

Real distributions of planner runtimes over instance sets differ from this idealized example: they usually involve constant factors, they may scale irregularly, and even for a single planner it may be impossible to obtain instances that scale according to the desired runtimes in some domains. But ideally, we aim for a collection where the easiest instance is quickly solved by most planners, has instances unsolved by current planners, and planner runtimes scale by approximately a factor of 1.5–2 between consecutive instances.

Figure 1 exemplifies why a smooth scaling is important to meet Principle 1 in practice. The plot shows the runtimes of two optimal planners on the IPC 2011 instance set and the Autoscale instance set for the Barman domain. In the IPC instances the difficulty does not grow smoothly. Instead, there are groups of four or seven instances of the same difficulty, and the runtime of all planners increases by about one order of magnitude from one group to the next. This is undesirable since we cannot observe differences in performance for some planners by inspecting their coverage. In contrast, the difficulty of the Autoscale instance set grows more smoothly: there are instances of more varied difficulty for all planners and fewer jumps in their runtime. Accordingly, now we can observe that Delfi-LM-cut is able to solve some instances that are not solved by Complementary2.

Baseline and State-of-the-Art Planners

The attentive reader may have noticed that Rules 1 and 2 contradict each other. If only the best planner per instance is considered, it is impossible to ensure that the resulting set has easy instances, solved by all planners. The latter requires to consider the worst time of any planner in each instance.

However, this would make the optimization process brittle, as including any single planner whose performance is off may alter the selection in an undesired way. For example, if we define easy instances as those solved within a minute by all planners and any of the considered planners includes a preprocessing phase of at least one minute, then no instance will be considered easy anymore. To address this, we consider two sets of planners: baseline and state of the art.

The set of state-of-the-art planners $\mathcal{A}$ should contain as many planners as possible. For each instance Π, we

characterize the performance of a set of planners $\mathcal{A}$ on Π as the minimum runtime of any planner, $t_{min}(\mathcal{A}, \Pi) = \min_{p \in \mathcal{A}} t(p, \Pi)$. We exclude a planner from consideration if it solves all instances that can be generated in under 30 seconds, as otherwise it would be impossible to find a smooth scaling. This happens only in domains solvable in polynomial time for which constructing hard instances of a reasonable size is not possible.

The set of baseline planners $\mathcal{B}$ represents the expected minimum performance of any planner that will later be evaluated to ensure that some instances are solved by those (or better) planners. We consider the worst runtime of any baseline planner for any instance, $t_{max}(\mathcal{B}, \Pi) = \max_{p \in \mathcal{B}} t(p, \Pi)$. Note that the choice of the baseline planners is a clear source of bias, since we ensure that some instances are solved by such planners. Thus, in our experiments we are very conservative and only choose a single planner as baseline.

Selection of Parameter Configurations

After having established the objectives of our optimization, the next step is to determine our decision variables. While it may be natural to directly select a set of instances $\{\Pi_1, \ldots, \Pi_n\}$, we avoid this and only select a set of suitable parameters $\{\rho_1, \ldots, \rho_n\}$ for the generator G.

Rule 3 (Parameter-based Selection)**.** *The optimization process does not select concrete instances, but rather only decides which parameters to use for the generator. The final benchmark set is then generated with a new random seed.*

This reduces the bias towards the set of considered planners (Principle 2), as we are not selecting concrete instances in which the planners behave in certain ways. Instead, planners are only used to find a suitable range of parameters. Moreover, choosing concrete instances would allow the optimization process to select instances with pathological distributions of initial states or goals, violating Principle 3. For example, in Nomystery one could choose instances where all packages have the same destination if this induces a smoother scaling. Ensuring that the final benchmark set is randomly sampled from the distribution of instances with the same parameters avoids this kind of unintended bias.

The main drawback of selecting parameters instead of instances is that in some domains there may be a huge variance on the runtime of instances with the same parameters (de la Rosa, Cenamor, and Fernández 2017; Cohen and Beck 2018). We model the runtime of a parameter configuration ρ as a distribution $T_{min}(\mathcal{A}, \rho)$, which corresponds to $t_{min}(\mathcal{A}, G(\rho, r))$ for all $r \in \mathbb{N}^+$. In practice, we estimate T_{min} by computing the runtime on a small sample of tasks. Admittedly, this reduces our ability to control that the instance set scales smoothly, but it is not a problem in practice for two reasons. On the one hand, by generating multiple instances across a sequence, the variance balances out. Even if one instance is particularly easy or hard, other instances of similar size in the sequence will make up for it. On the other hand, recall that the "desired/expected" runtimes are just a means to an end: even if the runtimes differ from the expected ones this is fine as long as the instances have a reasonable difficulty to be useful to evaluate current planners.

			locations		packages	
		constr.	b	m	b	m
OPT	$\Pi_1, \ldots, \Pi_{13}$	1.5	4.6	0.1	22	1
OPT	$\Pi_{14}, \ldots, \Pi_{30}$	2.0	9	0.1	3	1
AGL	$\Pi_1, \ldots, \Pi_{30}$	1.5	5	1	9	1.4

Table 2: Sequences chosen by Autoscale for the Nomystery domain in optimal (OPT) and agile (AGL) planning.

Instance Sequences

Finally, we also require that the parameter configurations are not chosen arbitrarily, but rather that they are organized in one or more sequences of instances.

Rule 4 (Sequence-based Selection)**.** *The parameter configurations of the instance set are not standalone, but rather they can be organized in one or more sequences.*

We distinguish between two types of parameters. *Linear parameters* can be assigned arbitrary non-negative numeric values, where larger values usually result in harder instances. They are typically used to specify the number of objects of a given type. Each generator should have at least one linear parameter that helps to control the difficulty of the generated instances. In our Nomystery example there are two linear parameters that control the number of locations and packages in the task. In contrast, *enumerated parameters* have a finite set of values, and we do not make any assumption about their impact on instance hardness. In our example, the constrainedness level that determines the amount of fuel available is an enumerated parameter. All other parameters are fixed to a predefined constant value.

We define a *sequence* of instances as a list of planning tasks $\Pi_1 = G(\rho_1, r_1), \Pi_2 = G(\rho_2, r_2), \ldots$ of increasing difficulty. To ensure that difficulty increases, all instances in the sequence have the same value for all enumerated parameters, whereas the value of linear parameters increases linearly across the sequence. We specify this via the base value b that the linear parameter takes for Π_1 and the slope m. For example, the selection made by Autoscale for Nomystery for agile planning (see Table 2) uses (b=5, m=1) for locations, and (b=9, m=1.4) for the packages. Then, the sequence consists of instances with the following numbers of locations and packages: $(5, 9), (6, 10), (7, 11), (8, 13), (9, 14)$, etc. Any duplicates are skipped so that all configurations in a sequence have different values.

This heavily restricts the combinations of parameters that are possible, as linear parameters are all increased at the same time. For example, a single sequence cannot contain both $(3, 2)$ and $(2, 3)$ because that would require to decrease one of the parameters, which is not allowed by our linear scaling. To allow more flexibility, our optimization may select multiple sequences for a single domain. This is the case for the OPT selection in Table 2, where two sequences with a different proportion of trucks and locations are chosen.

Considering sequences of instances has several advantages. It allows us to choose parameters that generate instances which current planners fail to solve within reason-

able time and for which there is no direct way of obtaining their runtime. More importantly, it makes the instance selection more interpretable (Principle 3). For example, our instance selection for OPT-Nomystery tests how planners scale with respect to the number of packages under two constrainedness levels, whereas AGL-Nomystery tests the scaling with respect to both the number of locations and packages. This allows us to describe the instance sets very compactly: e.g., Table 2 contains all information to recreate the corresponding sets of 30 instances for fixed random seeds.

Note that, in order to ensure that Principle 3 is fulfilled, it is the task of the domain designer to specify which scalings are desirable according to the spirit of the domain.

Rule 5 (User Constraints)**.** *The tool should be configurable via user constraints that specify which parameter configurations for sequences of instances are acceptable.*

An Automatic Tool for Instance Selection

Our tool, Autoscale, takes as input a tuple $(spec, G, \mathcal{A}, \mathcal{B})$, where $spec$ is a domain specification; G is an instance generator; $\mathcal{A}$ is a set of state-of-the-art planners; and $\mathcal{B}$ is a set of baseline planners. The output is a set of parameter configurations $\{\rho^1, \ldots, \rho^n\}$ that can be passed to the generator to generate a set of n instances. Autoscale works in two phases: the first phase designs a set of candidate sequences (Sequence Optimization), and the second phase performs a final selection of sub-sequences that adheres to our design principles as much as possible (Sequence Selection).

Domain Specification

To use Autoscale, the benchmark designer must specify how to call the instance generator, which parameters are available, and which value ranges are appropriate for each parameter. The snippet in Figure 2 shows the domain specification for Nomystery. For each linear parameter, lower and upper bounds for the base and slope values are provided. This allows the domain modeler to specify preferences on which parameters to scale (e.g., restricting the slope m for the number of locations to be between 0.1 and 1 indicates that scaling the number of packages is preferable).

Often, instance generators impose constraints on the range of parameter values or their combination. Those constraints must be enforced by adding a postprocessing function that updates the value of the parameters passed to the generator. This is an arbitrary function provided by the benchmark designer which receives the parameters that were automatically chosen and outputs the final parameters that will be provided to the generator. For example, if the number of packages has to be greater than the number of locations, instead of directly selecting the number of packages, our linear scaling will consider the number of locations and the number of additional packages. All of these adjustments must be done on a per-domain basis, since they depend on the specific characteristics of the domain and generator.

Note that, by assuming that linear parameters scale difficulty, we require the benchmark designer to identify cases where this is not the case and/or where there is a strong interaction between some of the parameters. Take as example

```
generator_command = "nomystery -l {locations}
   -p {packages} -n {edgefactor} -m {edgeweight}
   -c {constrainedness} -s {seed} -e 0"
parameters = [
LinearParam("locations", lower_b=3, upper_b=10,
                         lower_m=0.1, upper_m=1),
LinearParam("packages",lower_b=2,upper_b=20,lower_m=1),
ConstantParam("edgefactor", "1.5"),
ConstantParam("edgeweight", "25"),
EnumParam("constrainedness", [1.1, 1.5, 2.0])]
```

Figure 2: Nomystery domain specification with the generator command and its corresponding parameters.

the amount of fuel in Nomystery. Instead of a constrainedness value, the generator could have a parameter specifying the amount of fuel. However, this would not be well-suited to be a linear parameter because larger amounts of fuel can decrease the difficulty of solving the problem. Thus, Autoscale can deal with domains that exhibit a phase transition effect (Rieffel et al. 2014; Cohen and Beck 2017) by using the constrainedness level as an enumerated parameter, as in our example. Autoscale will automatically choose suitable levels of constrainedness (e.g., closer or further away from the phase transition) among the set of values that the domain designer considers to be relevant.

We emphasize that providing a domain specification is typically straightforward. For the 26 domains that we have configured with Autoscale so far, we were able to easily choose categories for the generator parameters. A notable exception are parameters that define the width and height of a grid, because they have a strong interaction, i.e., the number of cells is the product of both parameters. In that case, we had to consider them as a single parameter by defining a list of grid sizes sorted by the number of cells (e.g., $4{\times}5(20), 4{\times}6(24), 5{\times}5(25), 5{\times}6(30)$, etc.). The linear parameter just selects the position in this list, so the number of tiles in the grid scales linearly instead of quadratically.

Sequence Optimization

The first Autoscale phase generates sequences of 30 parameter configurations $\{\rho_1, \ldots, \rho_{30}\}$ by optimizing sequence parameters. To guide the search towards sequences where planner runtimes scale smoothly, we compute a penalty score for each sequence and search for the sequence that minimizes this score. Sequences are evaluated by running the set of state-of-the-art ($\mathcal{A}$) and baseline ($\mathcal{B}$) planners on the instances, using a time limit of 180 seconds per instance. A penalty is computed for each of them individually, and summed up.

Next, we describe the procedure for evaluating state-of-the-art planners. The procedure for baseline planners is equivalent, replacing t_{min} by t_{max}. First, we evaluate each parameter configuration ρ_i by sampling K tasks ($K = 3$ in our experiments). We estimate the average runtime of ρ_i, $\mathbb{E}[T_{min}(\mathcal{A}, \rho_i)]$ by computing $t_{min}(\mathcal{A}, G(\rho_i, r))$ for K different random seeds r. Since the sequences are generated with increasing values of the linear parameters, we assume that

the runtimes will always increase,[2] so we can stop our evaluation as soon as one instance is not solved under the time limit. In cases where this does not hold, we enforce it by sorting the instances by average runtime. Our assumption is that these anomalies stem from using different random seeds for the instance generator and the results could be reversed with other random seeds.

Let $T_1, \ldots, T_5$ be the set of runtimes for each of the first five instances with an average runtime above 5 seconds. We ignore those with lower runtime, considering that differences of ± 5 seconds are not meaningful enough. We only use five instances as harder instances will usually incur runtimes above the 180 seconds time limit. The penalty score for state-of-the-art planners is defined as $\sum_{i\in[2,5]} \sum_{t\in T_i, t'\in T_{i-1}} \frac{S(\max(t,t'),\min(t,t'))}{|T_i||T_{i-1}|}$ where

$$S(a,b) = \begin{cases} 3 - 2\frac{a}{b} & \text{if } a \leq 180 \text{ and } \quad 1 \leq \frac{a}{b} < 1.5 \\ 0 & \text{if } a \leq 180 \text{ and } 1.5 \leq \frac{a}{b} \leq 2 \\ 1 - 2\frac{b}{a} & \text{if } a \leq 180 \text{ and } \quad 2 < \frac{a}{b} \\ 2 & \text{if } a > 180 \end{cases}$$

This penalty is lower for sequences whose runtime scales smoothly, assigning a minimum score of 0 to any sequence where the runtimes of the considered planners scale exponentially with a factor between 1.5 and 2, e.g., if $K = 1$, $\langle 10, 15, 23, 35, 52, \ldots \rangle$, or $\langle 10, 20, 40, 80, 160, \ldots \rangle$. If not enough instances are solved in the $[10, 180]$ second interval, the sequence gets a penalty of 2 for each unsolved instance. The remaining two cases assign a penalty between 0 and 1, depending on how far they are from the 1.5–2 scaling. When $K > 1$, we compute the average penalty $S(a, b)$ for each pair of sampled runtimes for consecutive parameter configurations. This is representative of the scaling that one may encounter in the final instance set, and it tends to favor sequences with lower runtime variance.

On top of this, to avoid sequences where all instances are solved by the state-of-the-art planners, we add a penalty of 1 for each instance solved beyond 20 instances. Moreover, to guarantee that all sequences contain some instances solvable within the time limit and to speed up the evaluation we discard any sequence where the first three instances are not solved within 10, 60, and 180 seconds, respectively.

Of course, the concrete definition of this penalty function is arbitrary. What matters is that sequences that scale smoothly will minimize it, thereby guiding the parameter optimization towards good sequences.

Evaluating each sequence may be time consuming, as it may require to run all planners on up to $30 \cdot K$ instances. To speed-up the optimization, we use two important measures. First, we store the runtimes of any evaluated configuration in a database, to avoid repeating it more than once. Second, to avoid running all planners, we choose a subset of planners per domain. We select them by using data from previously known instances of the domain (e.g., from the IPC or previous Autoscale runs) to choose a subset of planners sufficient for obtaining the best runtime on 95% of the instances, while accepting an error of five seconds.

[2]Note that parameters with unpredictable influence on planner runtime should be considered enumerated parameters and remain constant for a given sequence.

Sequence Selection

After performing one or more optimization runs for a domain (using different random seeds) as described above, we collect all sequences seen during the optimization process. Since this set can be very large, we only keep the 100 sequences with the lowest penalty score per value of the enumerated parameters. For each group of sequences where the planners solve the same instances, we only keep one member of the group. This filtering ensures that we keep a set of diverse sequences with a good penalty score.

For each sequence, we collect the runtimes of all instances solved in 180 seconds from the sequence optimization phase. For the other instances, we estimate their runtime by assuming that runtimes increase according to the average increasing factor $\mathbb{E}[T(\mathcal{A}, \rho_i)]/\mathbb{E}[T(\mathcal{A}, \rho_{i-1})]$ observed on the instances solved between 5 and 180 seconds. This is a very rough estimate but it is accurate enough for the purpose of choosing where to end a sequence (see below).

We model the problem of selecting a suitable set of subsequences as a mixed-integer programming (MIP) problem, where constraints directly aim to model our instance set design principles. The decision variables model the start and end points of each sub-sequence of instances. The selection must satisfy the following *hard constraints* that model properties desirable for a good set of instances:

(H1) The number of selected instances must be exactly 30.

(H2) There must be at least one instance solvable by the baseline under 30 seconds.

(H3) All sequences must start with an instance that is solved under 180 seconds and end with an instance whose estimated runtime is higher than 2000 seconds.

(H4) Each parameter configuration must be used (with different random seeds) at most twice, and only once for domains whose generators are deterministic.

The objective is to minimize the summed-up penalty score of all sequences used, plus the penalty incurred for violating any of the following *soft constraints*:

(S1) The number of instances solved by the baseline under 30 seconds must be between 2 and 6 (with a penalty of $2x^2$ where x is the deviation wrt. the constraint).

(S2) The number of instances solved by state-of-the-art planners under 180 seconds must be between 8 and 15 (with a penalty of $2x^2$ where x is the deviation wrt. the constraint).

(S3) All sequences must end with an instance whose estimated runtime is between 18 000 and 180 000 seconds (that is, 1–2 orders of magnitude larger than the typical time limit of 1800 seconds). Larger times t incur a penalty of $100t/180\,000$ and smaller times incur a penalty of $100(18\,000/t)$.

	Optimal	Agile
Train	blind search (baseline, Helmert 2006), all four components of the FDSS 1 portfolio from IPC 2011 (Helmert et al. 2011) and SymBA$^*_1$ from IPC 2014 (Torralba et al. 2014)	greedy best-first search with FF heuristic (baseline, Hoffmann and Nebel 2001), LAMA (Richter and Westphal 2010), Madagascar (Rintanen 2012), Mercury (Katz and Hoffmann 2014), Jasper (Xie, Müller, and Holte 2014), and Probe (Lipovetzky et al. 2014)
Eval	five components of Delfi1 portfolio from IPC 2018 using symmetry pruning and partial order reduction (blind search, iPDB, LM-cut and two M&S variants, see Katz et al. 2018) and three vanilla IPC 2018 planners: Complementary2 (Franco, Lelis, and Barley 2018), DecStar (Gnad, Shleyfman, and Hoffmann 2018), Scorpion (Seipp 2018b)	eight vanilla IPC 2018 planners: Cerberus (Katz 2018), BFWS-PREF, DUAL-BFWS and POLY-BFWS (Francès et al. 2018), DecStar (Gnad, Shleyfman, and Hoffmann 2018), OLCFF (Fickert and Hoffmann 2018), Fast Downward Remix (Seipp 2018a) and Saarplan (Fickert et al. 2018)

Table 3: Choice of optimal and agile planners used during the optimization (Train) and evaluation (Eval).

(S4) If a parameter configuration is used more than once, there is a penalty of 100.

Constraints (H2), (S1) and (S2) ensure that the instance set contains some easy instances, so that any future planning algorithms are expected to solve at least some instances, allowing researchers to analyze the behaviour of their algorithms in the domain. Constraints (H3) and (S3) ensure that, whenever possible, at least some of the instances are expected to be out of reach for state-of-the-art planners. Together with minimizing the penalty score of the selected sequences, they aim to obtain a smooth scaling, since sequences must interpolate between easy and hard instances and sequences with smoother scaling are preferred. Finally, constraints (H4) and (S4) are needed to avoid duplicate instances and instances that are very similar to each other.

The penalties are set arbitrarily, scaling quadratically wrt. the deviation so that no constraint is completely ignored.

Experiments

We test Autoscale by running two completely separate experiments for optimal and agile planners. Both experiments consider 26 domains from previous IPCs with instance generators. We use planners available at the time of IPC 2014 for training (i.e., our optimization process) and planners from IPC 2018 for evaluation. This separation helps to evaluate whether our method can generate instances that are still adequate for empirical evaluations after several years. Table 3 lists the planners we used. We ran experiments on Intel Xeon Silver 4114 CPUs using Downward Lab (Seipp et al. 2017). All our code, planners, and benchmarks are publicly available (Torralba, Seipp, and Sievers 2021).

We implemented the first phase, i.e., sequence optimization, using the automatic configurator SMAC (Hutter, Hoos, and Leyton-Brown 2011). Since we limit each planner run during sequence optimization to three minutes, we adapt the preprocessing time limits for planners with preprocessing phases accordingly. We run five SMAC instances in parallel using different random seeds and let the SMAC runs share their discovered results with each other. Each SMAC run is limited to 50 hours of wall-clock time. After the sequence optimization phase finishes, we consider all sequences encountered during optimization for the sequence selection phase. We filter the instances as described in the previous section and solve the MIP for sequence selection using CPLEX 12.10, which finishes in under 30 seconds for each domain.

We evaluate the Autoscale (AS) benchmark set with both the training and evaluation planners, limiting each run to 30 minutes and 3.5 GiB. Table 4 shows the results, grouped by optimal and agile setting. For each setting, the first column (#s) shows the number of sequences used for the instance sets, and the remaining columns are divided into training and evaluation performance, which we evaluate according to two metrics. The first metric is the range of coverage scores per domain for both sets (cov range), which allows seeing how many instances are solved by all planners and how many remain unsolved by any of the planners. Second, we consider the number of pairwise comparisons in which one planner has higher coverage than another (comp), which quantifies how many differences in the performance of planners are reflected by the coverage score.

We first observe that the generated instance sets for optimal planning consist of more sequences (#s) than in agile planning. The reason is that in many domains increasing parameter values even slightly causes a big increase in the runtime of optimal planners on the resulting instances. Autoscale successfully compensates for this by selecting multiple sequences.

Next, we see that the AS set is preferable to the IPC set for many domains in the optimal setting, for both the training and the evaluation planners: its coverage ranges show that all planners solve some instances in all cases, and only in Gripper a training planner solves all instances. This does not hold for the IPC set, where at least one planner solves no instance in Childsnack nor Parking, and in three domains at least one training and evaluation planner solves all instances. The AS set increases the number of observed comparisons in 13 and 12 out of 26 domains using the training and evaluation planners, while the opposite is true in only 7 and 9 domains, respectively. In Openstacks, the large negative difference in observed comparisons is due to the two instance sets having different sizes. The IPC Openstacks set has a much larger set of 70 instances, and many planners solve around 42–46 of them. Moreover, a single training planner (SymBA$^*_1$) greatly outperforms the rest, including all evaluation planners from 2018. The AS set "reserves" 10 of its 30 instances to evaluate planners that outperform SymBA$^*_1$. Having such planners in our evaluation set would show the

		optimal									agile								
		training				evaluation					training					evaluation			
		cov range		comp (15)		cov range		comp (28)			cov range		comp (15)			cov range		comp (28)	
	#IPC	#s	IPC	AS	AS	diff	IPC	AS	AS	diff	#s	IPC	AS	AS	diff	IPC	AS	AS	diff
Barman	34/40	3	4–16	6–14	14	**+9**	4–11	10–16	24	**+12**	1	17–40	0–28	15	**+2**	39–40	4–20	26	**+19**
Blocksworld	35	1	18–34	6–11	14	**+2**	18–30	5–12	24	**+6**	1	35–35	5–27	15	**+15**	35–35	6–21	26	**+26**
Childsnack	20	3	0–4	8–15	5	0	0–6	8–19	21	**+8**	2	0–7	0–11	15	**+2**	1–20	2–30	28	**+1**
Data-Network	20	2	7–13	8–16	14	-1	6–14	8–17	25	-2	1	2–15	8–24	14	0	9–19	16–30	26	**+2**
Depots	22	3	4–9	8–14	13	**+4**	5–14	11–19	25	0	1	17–22	7–23	15	**+3**	22–22	14–21	25	**+25**
Driverlog	20	1	7–14	4–18	15	**+4**	7–15	4–30	27	**+5**	1	18–20	12–19	15	**+10**	20–20	8–25	25	**+25**
Elevators	50	3	7–44	5–15	13	-2	28–44	8–11	23	-3	1	11–50	2–30	15	**+6**	49–50	14–30	18	**+11**
Floortile	40	2	2–34	3–22	15	**+2**	16–34	8–16	18	-3	1	7–40	1–15	9	-5	4–40	2–12	24	**+7**
Grid	5	1	1–3	5–17	14	**+3**	1–3	5–14	26	**+7**	1	4–5	5–17	13	**+8**	5–5	12–16	21	**+21**
Gripper	20	1	7–20	8–30	11	0	8–20	9–30	7	0	1	20–20	30–30	0	0	20–20	26–30	7	**+7**
Hiking	20	2	9–19	1–18	12	-2	13–18	2–16	25	**+4**	2	1–20	3–15	14	**+5**	10–20	2–25	25	**+3**
Logistics	63	2	12–27	7–19	15	**+1**	13–36	8–30	25	-3	2	57–63	0–15	12	0	51–63	7–15	21	**+4**
Miconic	150	2	55–144	3–21	15	**+1**	56–143	3–21	27	0	1	150–150	30–30	0	0	150–150	30–30	0	0
Nomystery	20	2	8–20	3–18	14	0	8–20	3–30	28	**+10**	1	6–20	1–25	15	**+1**	12–20	6–30	27	**+4**
Openstacks	70	1	23–70	3–20	9	-3	42–64	4–6	7	-17	1	6–70	1–24	14	0	70–70	12–21	25	**+25**
Parking	40	4	0–11	11–19	14	-1	0–15	12–19	26	-2	1	23–40	0–24	15	**+1**	35–40	15–18	18	**+5**
Rovers	40	2	6–14	4–27	14	**+1**	6–13	5–27	22	-4	1	26–40	14–30	13	**+8**	38–40	15–30	27	**+20**
Satellite	36	2	6–10	7–23	15	**+3**	7–14	14–27	26	**+5**	1	28–36	7–21	15	**+1**	26–36	4–17	25	**+2**
Scanalyzer	50	3	15–29	9–20	5	-4	21–33	9–19	26	0	2	42–50	8–15	13	**+1**	48–50	11–13	20	**+8**
Snake	20	2	4–12	5–13	12	0	7–14	7–15	21	-1	2	5–12	5–23	14	**+2**	3–17	2–20	27	0
Storage	30	2	14–16	6–16	12	**+3**	15–18	8–23	27	**+6**	1	19–30	6–18	14	0	21–30	8–19	27	**+1**
TPP	30	2	6–8	2–16	12	**+1**	7–20	4–30	26	**+2**	1	23–30	8–24	15	**+6**	29–30	8–20	26	**+11**
Transport	70	1	23–33	4–24	9	-5	24–35	6–30	13	-8	1	13–70	4–18	15	**+1**	62–70	9–16	21	**+14**
Visitall	40	2	12–28	6–25	15	0	12–30	6–20	26	0	1	3–40	5–28	15	**+3**	36–40	19–30	24	**+17**
Woodworking	50	3	11–48	5–17	14	**+2**	38–50	15–25	26	**+5**	1	43–50	4–20	14	**+9**	28–50	3–30	27	**+14**
Zenotravel	20	2	8–13	3–16	14	0	7–13	3–29	27	**+4**	1	20–20	10–16	13	**+13**	20–20	6–14	22	**+22**

Table 4: Comparison of the IPC and Autoscale (AS) benchmark sets generated for optimal and agile planning, evaluated using the training and evaluation planners (cf. Table 3). The #IPC column shows the number of tasks per domain in the IPC set (equal in optimal and agile planning except for Barman), which is always 30 for the AS set. The #s columns show the number of sequences in the AS instance sets. The "cov range" columns show the minimum and maximum coverage of any planner. The "comp" columns report how many pairs of planners yield different coverage. We show the value for the AS set and the difference to the value for the IPC set, highlighting in bold the cases where the AS set is superior. The maximum possible number of pairwise comparisons is 15 for the 6 training planners and 28 for the 8 evaluation planners.

superiority of the AS set for comparing such planners. The remaining 20 instances, however, have to cover the same difficulty range compared to the original 70 instances, which causes most planners to solve exactly 6 instances.

For agile planning, the AS set is far superior to the IPC set: only in a single case (Floortile-training) the number of pairwise differences in coverage decreases, while in the vast majority of cases this metric increases for the AS set, often drastically. The reason is that the IPC set scales poorly in many domains and therefore exhibits a small coverage range, a problem that the AS set does not share. Using the AS set, we observe coverage differences for the evaluation planners in seven domains that are solved completely by all of these planners when using the IPC set.

The comparison of the training and evaluation results clearly shows that Autoscale is not too sensitive to the set of considered planners and the instance sets obtained with old planners are also useful to evaluate new planners. The reason is that the state of the art has not advanced enough in the four years to make the instances trained with 2014 planners outdated for evaluating 2018 planners. Consequently, we have reason to believe that a new benchmark set generated by Autoscale using modern IPC 2018 planners will be useful for many years.

Conclusions

Constructing a benchmark set to evaluate planning algorithms requires to select the parameters of an instance generator to obtain a balanced set of instances. We identified desirable principles for this selection and modeled the problem of generating such instance sets as an optimization problem. We introduced Autoscale, a new tool that is able to produce instance sets that follow our principles. As demonstrated by the experiments, the new instances make differences in planner performance more visible compared to the standard IPC set, even when the optimization is done on a different set of planners. We will release a new benchmark set optimized with current planners to replace the IPC set, improving empirical evaluations in future planning research (Torralba, Seipp, and Sievers 2021).

Acknowledgments

This research was supported by TAILOR, a project funded by the EU Horizon 2020 research and innovation programme under grant agreement no. 952215. Á. Torralba was employed by Saarland University and CISPA during part of the development of this paper.

References

Achlioptas, D.; Gomes, C. P.; Kautz, H. A.; and Selman, B. 2000. Generating Satisfiable Problem Instances. In *Proc. AAAI 2000*, 256–261.

Bento, D. S.; Pereira, A. G.; and Lelis, L. H. S. 2019. Procedural Generation of Initial States of Sokoban. In *Proc. IJCAI 2019*, 4651–4657.

Cohen, E.; and Beck, J. C. 2017. Problem Difficulty and the Phase Transition in Heuristic Search. In *Proc. AAAI 2017*, 780–786.

Cohen, E.; and Beck, J. C. 2018. Fat- and Heavy-Tailed Behavior in Satisficing Planning. In *Proc. AAAI 2018*, 6136–6143.

de la Rosa, T.; Cenamor, I.; and Fernández, F. 2017. Performance Modelling of Planners from Homogeneous Problem Sets. In *Proc. ICAPS 2017*, 425–433.

Fickert, M.; Gnad, D.; Speicher, P.; and Hoffmann, J. 2018. SaarPlan: Combining Saarland's Greatest Planning Techniques. IPC-9 abstracts.

Fickert, M.; and Hoffmann, J. 2018. OLCFF: Online-Learning h^{CFF}. IPC-9 abstracts.

Fišer, D.; Torralba, Á.; and Shleyfman, A. 2019. Operator Mutexes and Symmetries for Simplifying Planning Tasks. In *Proc. AAAI 2019*, 7586–7593.

Francès, G.; Geffner, H.; Lipovetzky, N.; and Ramiréz, M. 2018. Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants. IPC-9 abstracts.

Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The Complementary2 Planner in the IPC 2018. IPC-9 abstracts.

Fuentetaja, R.; and de la Rosa, T. 2012. A Planning-Based Approach for Generating Planning Problems. In *AAAI 2012 Workshop on Problem Solving Using Classical Planners*, 30–36.

Giráldez-Cru, J.; and Levy, J. 2015. A modularity-based random SAT instances generator. In *Proc. IJCAI 2015*, 1952–1958.

Gnad, D.; Shleyfman, A.; and Hoffmann, J. 2018. DecStar – STAR-topology DECoupled Search at its best. IPC-9 abstracts.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR* 26: 191–246.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. IPC 2011 abstracts.

Hoffmann, J.; and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *JAIR* 24: 519–579.

Hoffmann, J.; Edelkamp, S.; Thiébaux, S.; Englert, R.; dos Santos Liporace, F.; and Trüg, S. 2006. Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4. *JAIR* 26: 453–541.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR* 14: 253–302.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. LION 2011*, 507–523.

Katz, M. 2018. Cerberus: Red-Black Heuristic for Planning Tasks with Conditional Effects Meets Novelty Heuristic and Enchanced Mutex Detection. IPC-9 abstracts.

Katz, M.; and Hoffmann, J. 2014. Mercury Planner: Pushing the Limits of Partial Delete Relaxation. IPC-8 abstracts.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. IPC-9 abstracts.

Linares López, C.; Celorrio, S. J.; and Helmert, M. 2013. Automating the evaluation of planning systems. *AI Comm.* 26(4): 331–354.

Linares López, C.; Celorrio, S. J.; and Olaya, A. G. 2015. The deterministic part of the seventh International Planning Competition. *AIJ* 223: 82–119.

Lipovetzky, N.; Ramirez, M.; Muise, C.; and Geffner, H. 2014. Width and Inference Based Planners: SIW, BFS(f), and PROBE. IPC-8 abstracts.

Moraru, I.; and Edelkamp, S. 2019. Benchmarks Old and New: How to compare domain independence for cost-optimal classical planning? In *ICAPS Workshop on the IPC*, 36–39.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-Constrained Planning: A Monte-Carlo Random Walk Approach. In *Proc. ICAPS 2012*, 181–189.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR* 39: 127–177.

Riddle, P. J.; Holte, R. C.; and Barley, M. W. 2011. Does Representation Matter in the Planning Competition? In *Proc. SARA 2011*.

Rieffel, E. G.; Venturelli, D.; Do, M.; Hen, I.; and Frank, J. 2014. Parametrized Families of Hard Planning Problems from Phase Transitions. In *Proc. AAAI 2014*, 2337–2343.

Rintanen, J. 2004. Phase Transitions in Classical Planning: an Experimental Study. In *Proc. ICAPS 2004*, 101–110.

Rintanen, J. 2012. Planning as Satisfiability: Heuristics. *AIJ* 193: 45–86.

Seipp, J. 2018a. Fast Downward Remix. IPC-9 abstracts.

Seipp, J. 2018b. Fast Downward Scorpion. IPC-9 abstracts.

Seipp, J. 2019. Planner Metrics Should Satisfy Independence of Irrelevant Alternatives. In *ICAPS Workshop on the IPC*, 40–41.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Selman, B.; Mitchell, D. G.; and Levesque, H. J. 1996. Generating Hard Satisfiability Problems. *AIJ* 81(1–2): 17–29.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A Symbolic Bidirectional A* Planner. IPC-8 abstracts.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Code and Benchmarks from the paper "Automatic Instance Generation for Classical Planning". https://doi.org/10.5281/zenodo.4586397.

Vallati, M.; Chrpa, L.; and McCluskey, T. L. 2018. What you always wanted to know about the deterministic part of the International Planning Competition (IPC) 2014 (but were too afraid to ask). *Knowledge Engineering Review* 33.

Xie, F.; Müller, M.; and Holte, R. 2014. Jasper: the art of exploration in Greedy Best First Search. IPC-8 abstracts.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2005. A Simple Model to Generate Hard Satisfiable Instances. In *Proc. IJCAI 2005*, 337–342.