Machetli: Simplifying Input Files for Debugging

Lucas Galery Käser, Clemens Büchner, Augusto B. Corrêa, Florian Pommerening, Gabriele Röger

University of Basel, Switzerland
lucas.galerykaeser@tuta.io
{clemens.buechner, augusto.blaascorrea, florian.pommerening, gabriele.roeger}@unibas.ch

Abstract

Debugging can be a painful task, especially when bugs only occur for large input files. We present Machetli, a tool to help with debugging in such situations. It takes a large input file and cuts away parts of it, while still provoking the bug. The resulting file is much smaller than the original, making the bug easier to find and fix. In our experience, Machetli was able to reduce planning tasks with thousands of actions to trivial tasks that could even be solved by hand. Machetli is an open-source project and it can be extended to other use cases such as debugging SAT solvers or LaTeX compilation bugs.

Introduction

Most planning researchers have been through the following scenario: they implement an algorithm, run local tests in small tasks obtaining the expected results, and as soon as they try their algorithm in the largest tasks available, something goes wrong. It can be either some bug in the source code, some unexpected output, or something completely unknown. Sometimes, it almost feels to be a rule of thumb that if an unexpected behavior has to happen, it will happen with the largest possible task.

To make the debugging process less frustrating, we introduce a new tool called *Machetli*. Roughly speaking, Machetli receives as input a planner P, a task Π, and a description of the unexpected behavior triggered when P tries to solve Π. Machetli then does a *local search* on the *space of planning tasks*, aiming at finding the *minimal task* where the unexpected behavior still occurs. This process is done by pruning actions, predicates, and/or objects of the task and then checking if the unexpected behavior is still present.

In our demo, we show how to use Machetli and which use cases are already implemented in the tool. Machetli is a set of Python libraries that can be installed through `pip` (Python Software Foundation 2022). The open-source code is available online, together with its documentation and a Jupyter notebook tutorial.[1] Our demo video is also publicly available.[2]

[1]The available material can be found on the following page: https://github.com/aibasel/machetli. Machetli is distributed under GNU General Public License v3.0.

[2]https://ai.dmi.unibas.ch/videos/machetli-icaps2022demo.mp4

Overview of the Tool

Machetli was originally designed for debugging planners by simplifying tasks that trigger some unexpected behavior. Currently, Machetli can handle task descriptions in PDDL (Haslum et al. 2019) and SAS^+ (Bäckström and Nebel 1995; Helmert 2009). It transforms the input by removing actions, predicates, and/or objects. Then, a local search transforms the original task, keeps instances where the behavior persists, and repeats this process until no more transformations are found to reproduce the behavior. At this point, Machetli returns the last task where the bug still occurred.

The straightforward way to use Machetli is to run the given algorithm with the input and check for a specific behavior, such as a particular output message or system call. Although these might be the most frequent use cases, Machetli allows the user to specify the behavior using Python code, which makes the tool more powerful. For example, we successfully used Machetli with two different planners where one had a correct behavior and the other an unexpected behavior. Machetli then used the difference in their behavior to find a smaller task where this still happened, so we could debug the problematic planner.

In principle, Machetli works with arbitrary algorithms and inputs. Its procedure consists of three main ingredients: a *local search*, a set of transformations called *successor generators*, and an *evaluator* to determine whether the (un-) expected behavior occurs for a given input.

Local Search

Machetli's search procedure is already outlined above. In its heart, it is inspired by the idea of hill-climbing search. However, our implementation uses no quantitative metric of quality between neighboring instances. Instead, we use two concepts to drive our search towards the objectives that (i) the input shrinks over time and (ii) the intended behavior persists. The first objective is achieved by the design of the successor generators. The second objective is achieved by discarding those successors that do not reproduce the expected behavior. We discuss these objectives in the next two subsections. Figure 1 illustrates the overall procedure. It is important to note that although it is not guaranteed that the runtime decreases for smaller inputs, this tends to happen in practice.

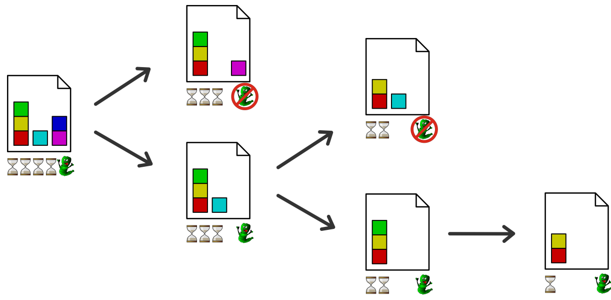

Figure 1: Sketch of a Machetli search: a BLOCKSWORLD task triggers some bug in a planner and Machetli removes blocks to generate similar, smaller instances where the bug persists.

Successor Generators

An important part of the search is how to explore the neighborhood of the current search state. Each state corresponds to an input file for the evaluator procedure (see next subsection). Our idea of a successor of this state is a smaller input file. To achieve this, successor generators usually remove parts of the input such that the syntax of the input language is not corrupted.

So far, we have implemented successor generators for PDDL and SAS$^+$ planning tasks, as described above. The three generators currently implemented for PDDL are: (i) removal of an action schema, where an action schema is simply removed from the PDDL task; (ii) removal of objects, where an object is removed from the task. In case this object is a domain constant occurring in a (partially) ground action, the atom where it occurs is also removed from the action; (iii) removal of predicates, where a predicate symbol and all its occurrences are removed. For this specific generator, the user can choose to replace the removed atoms with $\top$ or $\bot$. Since operators are grounded in SAS$^+$, this allows for more complex transformations. Our implementation includes merging two operators, and removing operator preconditions or effects. It is also possible to use multiple successor generators at once. In our experience, combining different successor generators produces better results than using them in isolation.

The user can extend the set of available successor generators even beyond planning applications. The choice of successor generators must of course fit the format of the input.

Evaluator

An evaluator is a procedure that takes any input (in our case a planning task formalized in PDDL or SAS$^+$) and returns true or false. The evaluator can run any subroutines, call external programs (e.g., a planner), read and process their output, etc. It can be tailored to whatever use case comes to mind. Its return value reflects whether the given input triggers the behavior of interest to the user. We share some examples in the next section. During the search, the evaluator is called for every successor. The first successor for which the evaluation returns true is chosen as starting point for the next iteration.

Use Cases

We conclude with some success stories of Machetli. They are all based on the Fast Downward planner (Helmert 2006). However, Machetli is not restricted to Fast Downward and can trivially handle other classical planners.

Although Machetli does not guarantee a minimal result, our experiments show that it reduces the size of the tasks significantly.

Segmentation Fault

One of our first use cases of Machetli was to debug a segmentation fault happening during the constraint generation for the state-equation heuristic (Bonet 2013). Starting from an SAS$^+$ SOKOBAN task with 1536 actions and 184 variables where the segmentation fault bug was occurring, Machetli found a smaller task with only 2 actions and 2 variables in around 2 minutes where the same bug occurred. To check that the cause of the segmentation fault in the new task was the same as in the original one, we ran both tasks with GDB (GNU Project Debugger), where we could see that the bug was still happening at the same part of the code.

Inadmissible h^+

While implementing h^+ as an operator-counting heuristic (Imai and Fukunaga 2014), we found that said implementation is off by 1 for a single task out of over 1000 planning tasks from the international planning competitions 1998–2018. The investigation using Machetli reduced the number of linear program (LP) variables from 68364 to 108 and the number of LP constraints from 3772 to 46. As it turns out, the source of the issue was an error tolerance in the LP solver which only showed up in the presence of large action costs, in our case the PARCPRINTER domain. Fixing this issue was straightforward once it was understood.

This result was achieved by first transforming the PDDL task and afterwards transforming the SAS$^+$ task starting from the PDDL result. As Machetli is not limited to debugging planner behavior, taking one step further could include implementing successor generators for the LP description directly.

Wrongly Reported Unsolvability

A long-standing bug in the landmark code was only recently understood thanks to Machetli. A task was reported unsolvable by LAMA (Richter and Westphal 2010) even though solutions exist. This is due to incorrect landmark orderings introduced based on axioms. This observation was only possible because Machetli's output resulted in a landmark graph with 8 nodes and 10 edges in contrast to 330 nodes and 12724 edges in the original problem. Machetli's search finished in less than 10 minutes.

Acknowledgments

This research was supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant agreement no. 952215. Machetli was initially developed as part of Lucas Galery Käser's Bachelor's thesis.

References

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence*, 11(4): 625–655.

Bonet, B. 2013. An Admissible Heuristic for SAS$^+$ Planning Obtained from the State Equation. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2268–2274. AAAI Press.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Imai, T.; and Fukunaga, A. 2014. A Practical, Integer-Linear Programming Model for the Delete-Relaxation in Cost-Optimal Planning. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 459–464. IOS Press.

Python Software Foundation. 2022. Python Package Index - PyPI. https://pypi.org/.

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