TamerLite: a Heuristic Search Temporal and Numeric Planner

Alessandro Valentini, Davide Lusuardi, Andrea Micheli

Fondazione Bruno Kessler, Trento, Italy
alvalentini@fbk.eu, lusuardi@fbk.eu, amicheli@fbk.eu

Abstract

We present TamerLite, a heuristic search planner for temporal and numeric planning problems. TamerLite is a complete rewrite of the search techniques of the TAMER planner, designed around a modular collection of search schemata and heuristics that can be freely combined into different planner configurations. It is built on top of the Unified Planning framework, from which it inherits parsing, simplification and grounding, and offers a dual implementation: a Python codebase intended for fast prototyping and experimentation, mirrored by a Rust reimplementation aimed at performance. While TamerLite natively supports temporal planning, in this paper we focus on the numeric configurations submitted to the Agile and Satisficing tracks of the competition.

Code — https://github.com/fbk-pso/tamerlite

Introduction

TamerLite is a heuristic search planner that supports both temporal and numeric planning: it natively handles temporal features such as durative actions, and the numeric setting is just one of the problem classes it can address.

TamerLite is the result of a complete rewrite of the heuristic search techniques implemented in TAMER[1] (Valentini, Micheli, and Cimatti 2020), a temporal planner developed at Fondazione Bruno Kessler. The rewrite was driven by the goal of obtaining a cleaner and more modular architecture, in which the core ingredients of heuristic search planning—search algorithms and heuristics—are decoupled and can be recombined freely. As a result, TamerLite exposes a space of configurations rather than a single fixed planning strategy.

TamerLite implements a range of forward heuristic search strategies, spanning both single-queue algorithms (such as A*, Weighted A*, Greedy Best-First Search and Enforced Hill Climbing) and multi-queue schemes that interleave several open lists guided by different heuristics. This flexibility makes it possible to assemble configurations tailored to different objectives, for instance favouring fast plan generation or plan quality.

In the remainder of this paper we describe the planning fragments supported by TamerLite, its implementation and its integration with the Unified Planning framework, and finally the specific configurations submitted to the Agile and Satisficing tracks of the competition. TamerLite is publicly available as open source under the GPL license.

[1] https://tamer.fbk.eu

Planning Fragments Supported

TamerLite addresses a rich subset of the features expressible in the Unified Planning framework. Beyond conjunctive STRIPS-like conditions and effects, it supports the following fragments.

- **Negative and disjunctive conditions** – preconditions and goals are not restricted to conjunctions of positive literals, but may also contain negations and disjunctions. Generic propositional formulae are allowed.
- **Existential and universal conditions** – conditions may be quantified over the (finite) objects of the problem, which are compiled away during grounding.
- **Numeric planning (SNP and LNP)** – TamerLite handles both Simple Numeric Planning, where numeric effects are restricted to constant increments and decrements, and the more general Linear Numeric Planning, which allows linear expressions over the numeric state variables.
- **Temporal planning** – TamerLite natively supports durative actions with conditions and effects at the start, at the end and over the whole duration, duration inequalities, and intermediate conditions and effects (ICE) occurring at arbitrary time points within an action.

The numeric track of the IPC 2026 exercised the numeric fragments above, while the temporal features are available but not evaluated in this setting.

Implementation Details

TamerLite follows a deliberately narrow design philosophy: rather than being a self-contained planning system, it implements only the components that are specific to heuristic search, namely the search algorithms and the heuristics, and delegates everything else to a supporting framework. Concretely, TamerLite is built on top of the Unified Planning (UP) framework (Micheli et al. 2025), an open-source library that provides a common modelling API and a collection of planning utilities. Parsing of the problem description, simplification of conditions and expressions, and grounding

of the lifted problem into a propositional and numeric representation are all carried out by UP. TamerLite consumes the grounded problem produced by UP and is therefore free to concentrate on the search itself, while automatically benefiting from the modeling features and the broader ecosystem of the framework.

The internal architecture reflects this separation. A grounded problem is translated into a compact representation suited for state expansion; on top of it, the search component maintains the open and closed structures and drives node expansion, querying the heuristic component to estimate goal distances. Search algorithms and heuristics are independent modules, so that any heuristic can be paired with any search strategy to instantiate a concrete planner configuration.

A distinctive feature of TamerLite is its dual implementation. All the algorithms are implemented in pure Python, which makes the codebase easy to read, modify and extend, and is convenient for rapid prototyping and for experimenting with new search and heuristic ideas. The same algorithms are then reimplemented in Rust, trading the flexibility of the Python version for the runtime efficiency required to be competitive on hard instances. The two implementations expose the same configuration space and are interchangeable behind the UP integration, so that the Python version can be used during development and the Rust version when performance matters.

The Rust implementation is exposed to the rest of the system through the PyO3 library (The PyO3 Project Developers 2025), which generates native Python bindings for the Rust code, so that TamerLite reuses the UP frontend unchanged and the choice between the two implementations is a single configuration switch. All the configurations submitted to IPC 2026 use the Rust implementation, which delivers the performance needed to be competitive within the time and memory budgets of the competition.

While relying on UP greatly simplifies and standardizes the frontend of the planner — parsing, simplification and grounding are inherited for free and kept consistent with the rest of the ecosystem — it also has a cost. These preprocessing steps are carried out in Python, and are therefore slower than a dedicated, hand-tuned implementation could be. As a consequence, TamerLite is not specifically tailored to the demanding conditions of the Agile track, where the time spent in parsing and grounding is not negligible with respect to the overall planning budget.

Configurations

Because search algorithms and heuristics are decoupled, TamerLite is best described not as a single planner but as a space of configurations obtained by selecting a search mode, a search algorithm, a heuristic, and a set of optional enhancements. The main dimensions of this space are the following.

- **Modes**: oneshot and anytime. In oneshot mode the planner stops at the first solution found, whereas in anytime mode it keeps searching for solutions of improving quality until the resources are exhausted.
- **Search**: *A**, *Weighted A**, *Greedy Best-First Search* (GBFS), *Breadth-First Search* (BFS), *Enforced Hill Climbing* (EHC), and a *multi-queue search* strategy combining several open lists driven by different heuristics.
- **Heuristics**: the well-known relaxation-based heuristics h_{FF} (Hoffmann and Nebel 2001), numeric h_{add} and h_{max} (Scala et al. 2020), together with custom heuristics (which are user-defined, arbitrary Python functions that can drive the search). At the moment, we do not have specialized heuristics for the temporal setting, instead, we relax the temporal aspects of the problems and use classical/numeric heuristics for guidance.
- **Enhancements**: On top of the basic search algorithms we implemented several enhancements that can be freely enabled or disabled. Reachability and relevance analysis can prune irrelevant actions (complete), weak equality in the temporal case collapses states that are propositionally identical but temporally different (incomplete), symmetry breaking recognizes symmetric objects (complete), compression-safe actions (Coles et al. 2009) (complete), and memory-bounded search based on a bounded priority-queue together with a Bloom filter as a compact representation of the closed set (incomplete).

Competition Configurations

Two configurations of TamerLite have been submitted for evaluation in each of the two numeric tracks. All submitted configurations employ the memory-bounded search mechanism described above.

Agile Track

1. A *Greedy Best-First Search* configuration guided by the h_{add} heuristic.
2. A *Weighted A** configuration guided by the h_{add} heuristic and using the evaluation function:

$$f(n) = 0.2 \cdot g(n) + 0.8 \cdot h(n).$$

Satisficing Track

1. A *Weighted A** configuration identical to the one submitted to the Agile Track.
2. A portfolio configuration combining *Weighted A** and *Greedy Best-First Search*, both guided by the h_{add} heuristic. The available planning time is divided equally between the two search algorithms.

Conclusion

We presented TamerLite, a heuristic search planner for temporal and numeric planning and a modern rewrite of the TAMER planner. By implementing only the search schemata and the heuristics and relying on the Unified Planning framework for parsing, simplification and grounding, TamerLite stays compact and modular, exposing a broad space of configurations built from interchangeable search algorithms and heuristics. Its dual Python and Rust implementation combines ease of prototyping with the efficiency needed to tackle hard instances.

References

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2009. Extending the Use of Inference in Temporal Planning as Forwards Search. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, manipulating and solving AI planning problems in Python. *SoftwareX*, 29: 102012.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramírez, M. 2020. Subgoaling Techniques for Satisficing and Optimal Numeric Planning. *J. Artif. Intell. Res.*, 68: 691–752.

The PyO3 Project Developers. 2025. PyO3: Rust bindings for Python. https://github.com/PyO3/pyo3.

Valentini, A.; Micheli, A.; and Cimatti, A. 2020. Temporal Planning with Intermediate Conditions and Effects. In *AAAI 2020*.