Fast Downward Stone Soup 2023

Clemens Büchner,[1] Remo Christen,[1] Augusto B. Corrêa,[1] Salomé Eriksson,[1] Patrick Ferber,[1] Jendrik Seipp,[2] Silvan Sievers[1]

[1] University of Basel, Switzerland
[2] Linköping University, Sweden
{clemens.buechner, remo.christen, augusto.blaascorrea, salome.eriksson, patrick.ferber, silvan.sievers}@unibas.ch, jendrik.seipp@liu.se

Fast Downward Stone Soup (Helmert, Röger, and Karpas 2011) is a sequential portfolio planner, built on top of the Fast Downward planning system (Helmert 2006, 2009). It participated in three previous International Planning Competitions (IPC): 2011 (Helmert et al. 2011), 2014 (Röger, Pommerening, and Seipp 2014), and 2018 (Seipp and Röger 2018). In the last IPC, Fast Downward Stone Soup was the winner of the satisficing and the cost-bounded tracks.

In this planner abstract, we present the Fast Downward Stone Soup portfolio submitted to the sequential optimal and satisficing tracks of IPC 2023. After two IPCs (2014 and 2018) without participating in the optimal track, this is the first time after 12 years that an optimal version of Fast Downward Stone Soup enters the competition. The procedure used for building the portfolios is the same as in 2011, 2014, and 2018. Therefore, we only briefly explain this procedure here and refer the reader to the original paper for more details (Helmert, Röger, and Karpas 2011). We highlight also the configurations used for the optimal track portfolio, and the new additions for the satisficing track.

Building the Portfolios

The Stone Soup algorithm requires the following information as input:

- A set of *planning algorithms* $\mathcal{A}$. We use a different set of Fast Downward configurations depending on the track, which we describe below.
- A set of *training instances* $\mathcal{I}$, for which portfolio performance is optimized. We use a set of 7330 instances, described below.
- Complete *evaluation results* that include, for each algorithm $A \in \mathcal{A}$ and training instance $I \in \mathcal{I}$,
  - the *runtime* $t(A, I)$ of the given algorithm on the given training instance on our evaluation machines, in seconds (we did not consider anytime planners), and
  - the *plan cost* $c(A, I)$ of the plan that was found.

  Our time limits depend on the track while the memory limit is fixed to 8 GiB for generate this data. If algorithm A fails to solve instance I within these bounds, we set $t(A, I) = c(A, I) = \infty$.

The procedure computes a portfolio as a mapping $P : \mathcal{A} \to \mathbb{N}_0$ which assigns a time limit (possibly 0 if the algorithm is not used) to each component algorithm. It is a simple hill-climbing search in the space of portfolios, shown in Figure 1.

build-portfolio(*algorithms*, *results*, *granularity*, *timeout*):
  portfolio := $\{A \mapsto 0 \mid A \in$ *algorithms*$\}$
  repeat $\lfloor$*timeout*/*granularity*$\rfloor$ **times**:
    candidates := successors(*portfolio*, *granularity*)
    portfolio := $\arg\max_{C \in \textit{candidates}}$ score(C, *results*)
  portfolio := reduce(*portfolio*, *results*)
  return *portfolio*

Figure 1: Stone Soup algorithm for building a portfolio.

In addition to the algorithms and the evaluation results, the algorithm takes two parameters, *granularity* and *timeout*, both measured in seconds. The timeout is an upper bound on the total time for the generated portfolio, which is the sum of all component time limits. The granularity specifies the step size with which we add time slices to the current portfolio.

The search starts from a portfolio that assigns a time limit of 0 seconds to all algorithms. In each hill-climbing step, it generates all possible *successors* of the current portfolio. There is one successor per algorithm A, where the only difference between the current portfolio and the successor is that the time limit of A is increased by the given granularity.

We evaluate the quality of a portfolio P by computing its *portfolio score* $s(P)$. The portfolio score is the sum of *instance scores* $s(P, I)$ over all instances $I \in \mathcal{I}$. The function $s(P, I)$ is similar to the scoring function used for the International Planning Competitions since 2008. The only difference is that we use the best solution quality among our algorithms as reference quality (instead of taking solutions from other planners into account): if no algorithm in a portfolio P solves an instance I within its allotted runtime, we set $s(P, I) = 0$. Otherwise, $s(P, I) = c^*_I / c^P_I$, where c^*_I is the lowest solution cost for I of any input algorithm $A \in \mathcal{A}$ and c^P_I denotes the best solution cost among all algorithms $A \in \mathcal{A}$ that solve the instance within their allotted runtime $P(A)$.

In each hill-climbing step the search chooses the successor with the highest portfolio score. Ties are broken in favor of successors that increase the timeout of the component algorithm that occurs earliest in some arbitrary total order.

The hill-climbing phase ends when all successors would exceed the given time bound. A post-processing step reduces the time assigned to each algorithm by the portfolio. It considers the algorithms in the same arbitrary order used for breaking ties in the hill-climbing phase and sets their time limit to the lowest value that would still lead to the same portfolio score.

Training Benchmark Set

As benchmarks, we used all tasks and domains from previous IPCs, from Delfi (Katz et al. 2018), and from the 22.03 Autoscale collection (Torralba, Seipp, and Sievers 2021), leading to a set of 92 domains with 7330 tasks. We used Downward Lab (Seipp et al. 2017) to run all planners on all benchmarks on Intel Xeon Silver 4114 2.2 GHz processors, imposing a memory limit of 8 GiB and a time limit of 30 minutes for optimal planners and 5 minutes for satisficing and agile planners. For each run, we stored its outcome (plan found, out of memory, out of time, task not supported by planner, error), the execution time, the maximum resident memory, and if the run found a plan, the plan length and plan cost. This data set is available online (Büchner et al. 2023).

After this first stage, we selected a subset of these tasks for training. To even out the differences in domain sizes, we only use up to 30 tasks per domain which are solved by the fewest (but at least one) planners. For the satisficing/agile tracks, this results in 2405 remaining tasks. For optimal, there are 1446 tasks without conditional effects and 245 tasks with conditional effects left.

Planning Algorithms

Satisficing Track

For the satisficing track, we collect our input planning algorithms from several sources. First, we use the component algorithms of the previous Fast Downward Stone Soup portfolios that participated in the sequential satisficing track of other IPCs (Helmert et al. 2011; Röger, Pommerening, and Seipp 2014; Seipp and Röger 2018). In total, this gives us 87 different configurations.

Second, we add different combinations of h^{cea} (Helmert and Geffner 2008), h^{CG} (Helmert 2006), h^{FF} (Hoffmann and Nebel 2001), and h^{LM} (Richter, Westphal, and Helmert 2011) with greedy best-first search (GBFS) but using different open-lists:

- For each one heuristic $h \in \{h^{\text{cea}}, h^{\text{CG}}, h^{\text{FF}}\}$, we use one configuration of (eager) GBFS with the ϵ-greedy open-list (Röger and Helmert 2010) ordered by h. We also add the same configurations but using lazy GBFS (Richter and Helmert 2009).
- We use configurations using lazy GBFS alternating between three open-lists: $[h^{\text{FF}}, h, \text{PAR}(h^{\text{FF}}, h)]$ where $h \in \{h^{\text{cea}}, h^{\text{CG}}, h^{\text{LM}}\}$ and $\text{PAR}(h^{\text{FF}}, h)$ is a Pareto open-list using h^{FF} and h (Röger and Helmert 2010).
- We add two other configurations using lazy GBFS: one alternating between $[\epsilon\text{-greedy}(h^{\text{FF}}), h^{\text{LM}}]$ open-lists, and one alternating between $[h^{\text{FF}}, \epsilon\text{-greedy}(h^{\text{LM}})]$. For each of these two configurations, we also include configurations with additional open-lists only containing states generated by preferred operators.

All configurations using h^{LM} have one version using reasonable orders and one without them.

This makes for a total of 18 new configurations. Overall, this leaves us with 105 planner configurations as input for the hill-climbing procedure.

Optimal Track

For the optimal track, we distinguish whether planning algorithms support conditional effects or not. We use A* with the following heuristics without support for conditional effects:

- the blind heuristic
- BJOLP (Domshlak et al. 2011)
- Cartesian abstractions (Seipp and Helmert 2018):
  - for subtasks induced by goals and fact landmarks (one of four different limits: 10s, 60s, 300s generation time or 1 million state-changing transitions in all abstractions)
  - for subtasks only induced by goals (1 million transitions)
  - for subtasks only induced by fact landmarks (1 million transitions)
- h^2 (Haslum and Geffner 2000)
- h^{max} (Bonet and Geffner 2001)
- pattern databases (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001), all combined in the canonical PDB heuristic (Haslum et al. 2007), computed with the following pattern collections:
  - CEGAR with a maximum size of 1 million states in individual PDBs and 10 million states in all PDBs, computation time limit of 10s/60s/300s with enabling stagnation after 2s/12s/20s, enabling blacklisting after 75% of the computation time limit or on stagnation, computing wildcard plans (Rovner, Sievers, and Helmert 2019)
  - hill climbing (Haslum et al. 2007) (thus leading to iPDB) with a computation time limit for the main loop of the algorithm of 10s/60s/300s
  - interesting patterns of size 1/2/3 (Pommerening, Röger, and Helmert 2013)
- LM-cut (Helmert and Domshlak 2009)
- merge-and-shrink heuristics (Helmert et al. 2014; Sievers and Helmert 2021) with bisimulation-based shrinking (Nissim, Hoffmann, and Helmert 2011) and a size limit of 50000 states per abstraction, exact label reduction (Sievers, Wehrle, and Helmert 2014), one of two merge strategies: SCC-DFP or SCC-sbMIASM (Sievers, Wehrle, and Helmert 2016), and a computation time limit for the main loop of the algorithm of 10s/60s/300s
- operator-counting heuristics (Pommerening et al. 2014) with different types of constraints:

  - post-hoc optimization constraints over interesting patterns (size 1/2/3) (Pommerening, Röger, and Helmert 2013)
  - delete relaxation constraints, leading to h^+ (exact IP model and LP relaxation) (Imai and Fukunaga 2015)
  - state equation constraints (Pommerening et al. 2014)
  - state equation constraints and LM-cut constraints (Pommerening et al. 2014)
  - state equation constraints, LM-cut constraints, and delete relaxation constraints (LP relaxation) (Pommerening et al. 2014)
- diverse potential heuristics and potential heuristics optimized for the initial state or all states (Seipp, Pommerening, and Helmert 2015)

We consider all of the above with and without pruning operators during successor generation using atom-centric stubborn sets (Röger et al. 2020). If activated, this feature is turned off after the first 1000 expansions if less than 20% of operators are pruned. This makes for a total of 74 planning algorithms.

For tasks with conditional effects, we cannot use stubborn sets pruning and we only use those of the above algorithms that support conditional effects: blind, $h^{\max}$, and all merge-and-shrink variants. This is a total of 8 planning algorithms.

Resulting Portfolios

The resulting satisficing portfolio uses 26 of the 105 possible algorithms, running them for 18–383 seconds. On the training set of 2405 tasks, the portfolio achieves an overall score of 2083.03, which is much better than the best component algorithm with a score of 1490.94. For the agile track, we use the same portfolio as for satisficing.

For the optimal track, the planner uses different portfolios depending on the PDDL features present in a problem. If the problem has axioms, then the portfolio consists of only blind search for 1800 seconds. Otherwise, if the problem has conditional effects, the portfolio uses 3 of the 8 algorithms, running them between 229 and 1137 seconds. Finally, if the problem has neither axioms nor conditional effects, then the portfolio uses 9 of the 74 algorithms, running them for 83–542 seconds. This configuration achieves an overal coverage of 1538 from our training set of 1691 tasks. The best component algorithm solved 994 of these tasks.

Executing Sequential Portfolios

In the previous sections, we assumed that a portfolio simply assigns a runtime to each algorithm, leaving their sequential order unspecified. With the simplifying assumption that all planner runs use the full assigned time and do not communicate information, the order is indeed irrelevant. In reality the situation is more complex.

First, the Fast Downward planner uses a preprocessing phase that we need to run once before we start the portfolio, so we do not have the full 1800 seconds available.[1] Therefore, we treat per-algorithm time limits defined by the portfolio as relative, rather than absolute values: whenever we start an algorithm, we compute the total allotted time of this and all following algorithms and scale it to the actually remaining computation time. We then assign the respective scaled time to the run. As a result, the last algorithm is allowed to use all of the remaining time.

Second, in the satisficing setting we would like to use the cost of a plan found by one algorithm to prune the search of subsequent planner runs (in the optimal track we stop after finding the first plan). We therefore use the best solution found so far for pruning based on g values: only paths in the state space that are cheaper than the best solution found so far are pursued.

Third, planner runs often terminate early, e.g., because they run out of memory or find a plan. Since we would like to use the remaining time to continue the search for a plan or improve the solution quality, we sort the algorithms by their coverage scores in decreasing order, hence beginning with algorithms likely to succeed quickly.

[1]The preprocessing phase consists of converting the input PDDL task (Fox and Long 2003) into a SAS$^+$ task (Bäckström and Nebel 1995) with the Fast Downward translator component.

Competition Results

Fast Downward Stonesoup 2023 was successful in the IPC 2023. In the satisficing and agile tracks, it claimed the runner-up awards while it landed in 7th place (out of 22) in the optimal track. We show competition results from the satisficing, agile, and optimal track in Tables 1a, 1b, and 1c, respectively. In contrast to the competition, we use the coverage score, i.e., number of solved tasks, to simplify our analysis. The tables show the overall coverage as well as reasons for failure, split by individual domains. The competition used several domains with modern PDDL features (e.g., axioms) that are not supported by several competing planners. Therefore, the organizers also offered variants of these domains where these features are compiled away (marked with the suffix *-norm* in Tables 1a–1c). For the overall scores, as in the competition, we use the domain variant with the better score.

Satisficing Track

We observe that the domain variant does not make a significant difference for the satisficing portfolio, except for *slitherlink*. The feature that is compiled away there, are negative goal conditions. If used, our portfolio solves 0 tasks, while when compiled away on the PDDL level, our portfolio solves 6 of the 20 problems. This does not necessarily mean that the algorithms available in Fast Downward are not able to deal with this feature. Rather, we found that with the feature present, the translator component of Fast Downward runs out of memory for all available problems. This does not happen when using the normalized versions of these problems.

There is another domain where our portfolio solves 0 problems, namely *labyrinth*. The challenge in *labyrinth* seems to be grounding the planning tasks. And in fact, the portfolio only starts the search component for the first four (smallest) problems. In all other cases, again, the translator of Fast Downward runs out of memory.

	coverage	time-err	mem-err
folding	10	8	2
folding-norm	9	10	1
labyrinth	0	4	16
quantum-layout	20	0	0
recharging-robots	14	2	4
recharging-robots-norm	14	3	3
ricochet-robots	11	9	0
rubiks-cube	20	0	0
rubiks-cube-norm	20	0	0
slitherlink	0	0	20
slitherlink-norm	6	14	0
Sum of best (140)	81	37	22

(a) Satisficing

	coverage	time-err	mem-err
folding	7	10	3
folding-norm	7	10	3
labyrinth	3	12	5
quantum-layout	20	0	0
recharging-robots	11	5	4
recharging-robots-norm	12	4	4
ricochet-robots	4	16	0
rubiks-cube	20	0	0
rubiks-cube-norm	19	1	0
slitherlink	4	16	0
slitherlink-norm	4	16	0
Sum of best (140)	70	58	12

(b) Agile

	coverage	time-err	mem-err
folding	7	10	3
folding-norm	7	10	3
labyrinth	3	4	13
quantum-layout	13	7	0
recharging-robots	13	1	6
recharging-robots-norm	13	1	6
ricochet-robots	12	8	0
rubiks-cube	9	11	0
rubiks-cube-norm	9	11	0
slitherlink	0	0	20
slitherlink-norm	4	13	3
Sum of best (140)	61	54	25

(c) Optimal

Table 1: Number of solved tasks, out-of-time and out-of-memory errors per domain variant for the three tracks. The bottom row aggregates over the domain variants with highest coverage.

Other domains where Stone Soup solves only around half of the problems are *folding* and *ricochet-robots*. In *folding*, there is only one IPC competitor with a slightly higher score, suggesting that it is simply a hard set of problems. In contrast, the *ricochet-robots* domain is solved more successfully by several other planners and marks the main shortcoming of Stone Soup when compared to the winning planners, Maidu and Levitron. The fact that *ricochet-robots* heavily relies on 0-cost actions may hint at our portfolio dealing badly with them. We do not think that this feature is underrepresented in our training set though, as roughly a quarter of the domains used during training contain 0-cost actions.

Agile Track

Interestingly enough, we solve some of the *labyrinth* problems in the agile track, even though we run the same portfolio as in the satisficing track with a tighter overall time limit. Moreover, the translator does not run out of memory as often. This is because we turn off invariant generation in the agile case. It usually takes a lot of time and apparently also a lot of memory in the case of *labyrinth*.

For *slitherlink* we make a similar observation: The difference between the normalised and non-normalised variant that is present in the satisficing track disappears in the agile track, even though our approaches to the two tracks only differ in their usage of invariant generation during translation.

Optimal Track

We again observe almost no difference between the normalized and non-normalized variants of the domains. Once more, the only exception is *slitherlink*, where we know that the translator component alone already exhausts the given resource limits.

Looking at how many problems were solved by the best-performing competitor on a per-domain basis, Stone Soup has the largest deficits in *labyrinth* (3 vs. 8), *ricochet-robots* (12 vs. 17), and *slitherlink* (4 vs. 7). Unfortunately, we cannot really pin-point the underlying short-comings, but we have seen in the other tracks that these domains and their peculiarities in terms of PDDL features are somewhat troublesome for our portfolios. Stone Soup also fails to shine in the remaining domains, and does not reach top performance in any of them.

Acknowledgments

For a portfolio planner, not those who *combined* the components deserve the main credit but those who *contributed* them. We therefore wish to thank all Fast Downward contributors and the people who came up with the algorithms we use in our portfolio. Furthermore, we sincerely thank the organizers of the IPC 2023 classical tracks, Daniel Fišer and Florian Pommerening, for their hard work running the competition.

References

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

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129(1): 5–33.

Büchner, C.; Christen, R.; Corrêa, A. B.; Eriksson, S.; Seipp, J.; and Sievers, S. 2023. Code and experiment data for the IPC 2023 planner "Fast Downward Stone Soup" (deterministic track). https://doi.org/10.5281/zenodo.8340920.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Domshlak, C.; Helmert, M.; Karpas, E.; Keyder, E.; Richter, S.; Röger, G.; Seipp, J.; and Westphal, M. 2011. BJOLP: The Big Joint Optimal Landmarks Planner. In *IPC 2011 Planner Abstracts*, 91–95.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 140–149. AAAI Press.

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.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, 28–35.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 Planner Abstracts*, 38–45.

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

Imai, T.; and Fukunaga, A. 2015. On a Practical, Integer-Linear Programming Model for Delete-Free Tasks and its Use as a Heuristic for Cost-Optimal Planning. *Journal of Artificial Intelligence Research*, 54: 631–677.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 57–64.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1983–1990. AAAI Press.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the Most Out of Pattern Databases for Classical Planning. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2357–2364. AAAI Press.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based Heuristics for Cost-optimal Planning. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 226–234. AAAI Press.

Richter, S.; and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 Planner Abstracts*, 50–54.

Röger, G.; and Helmert, M. 2010. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 246–249. AAAI Press.

Röger, G.; Helmert, M.; Seipp, J.; and Sievers, S. 2020. An Atom-Centric Perspective on Stubborn Sets. In Harabor, D.; and Vallati, M., eds., *Proceedings of the 13th Annual Symposium on Combinatorial Search (SoCS 2020)*, 57–65. AAAI Press.

Röger, G.; Pommerening, F.; and Seipp, J. 2014. Fast Downward Stone Soup 2014. In *Eighth International Planning Competition (IPC-8): Planner Abstracts*, 28–31.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 362–367. AAAI Press.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research*, 62: 535–577.

Seipp, J.; Pommerening, F.; and Helmert, M. 2015. New Optimization Functions for Potential Heuristics. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 193–201. AAAI Press.

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

Seipp, J.; and Röger, G. 2018. Fast Downward Stone Soup 2018. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 80–82.

Sievers, S.; and Helmert, M. 2021. Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems. *Journal of Artificial Intelligence Research*, 71: 781–883.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In Brodley, C. E.; and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2358–2366. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An Analysis of Merge Strategies for Merge-and-Shrink Heuristics. In Coles, A.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 294–298. AAAI Press.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 376–384. AAAI Press.