Fast Downward Stone Soup

Malte Helmert and **Gabriele Röger** and **Jendrik Seipp**
University of Freiburg, Germany
{helmert,roeger,seipp}@informatik.uni-freiburg.de
core team members

Erez Karpas
Technion, Israel
karpase@technion.ac.il
core team member

Jörg Hoffmann and **Emil Keyder**
INRIA, France
contributors

Raz Nissim
Ben-Gurion University, Israel
contributor

Silvia Richter
NICTA, Australia
contributor

Matthias Westphal
University of Freiburg, Germany
contributor

Abstract

Fast Downward Stone Soup is a sequential portfolio planner that uses various heuristics and search algorithms that have been implemented in the Fast Downward planning system.

We present a simple general method for concocting "planner soups", sequential portfolios of planning algorithms, and describe the actual recipes used for Fast Downward Stone Soup in the sequential optimization and sequential satisficing tracks of IPC 2011.

Before We Can Eat

Since the original implementation of the Fast Downward planner (Helmert 2006; 2009) for the 4th International Planning Competition (IPC 2004), various researchers have used it as a starting point and testbed for a large number of additional search algorithms, heuristics, and other capabilities (e. g., Helmert, Haslum, and Hoffmann 2007; Helmert and Geffner 2008; Richter, Helmert, and Westphal 2008; Helmert and Domshlak 2009; Richter and Helmert 2009; Röger and Helmert 2010; Keyder, Richter, and Helmert 2010).

Experiments with these different planning techniques have convinced us of two facts:

1. There is no single common search algorithm and heuristic that dominates all others for classical planning.
2. The coverage of a planning algorithm is often not diminished significantly when giving it less runtime, or put differently: if a planner does not solve a planning task quickly, it is likely not to solve it at all.

Fast Downward Stone Soup is a planning system that builds on these two observations by combining several components of Fast Downward into a *sequential portfolio*. In a sequential portfolio, several algorithms are run in sequence with short (compared to the 30 minutes allowed at the IPC) timeouts, in the hope that at least one of the component algorithms will find a solution in the time allotted to it.

There are two main versions of Fast Downward Stone Soup entered into the IPC: one for optimal planning, and one for satisficing planning. (Each version in turn has two variants, which differ from each other in smaller ways than the optimal planner differs from the satisficing one.)

The optimal portfolio planner exchanges no information at all between the component solvers that are run in sequence. The overall search ends as soon as one of the solvers finds a solution, since there would be no point in continuing after this.

The satisficing portfolio planner is an anytime system that can improve the quality of its generated solution over time. Here, the only information communicated between the component solvers is the quality of the best solution found so far, so that later solvers in the sequence can prune states whose "cost so far" (*g value*) is already as large as or larger than the cost of the best solution that was previously generated.

What is Stone Soup?

The name "Fast Downward Stone Soup" draws from a folk tale (for example told in Hunt and Thomas 2000, p. 7), in which hungry soldiers who are left without food take camp near a small village. They boil a pot of water over their campfire, and into the water they put three stones. This strange behaviour incites the curiosity of the villagers, to whom the soldiers explain that their "stone soup" is known as a true delicacy in the land where they come from, and that it would taste even better after adding some carrots. If the villagers could provide some carrots, they might participate in the feast. Hearing this, one of the villagers fetches the required ingredient, after which the soldiers explain that the recipe could be improved even further by adding potatoes, which another villager readily provides. Ingredient after ingredient is added in this fashion, until the soldiers are happy with the soup and finish its preparation with the final step in the recipe: removing the stones.

The stone soup tale is a story of collaboration. The final result, which benefits from the ingredients provided by a large number of villagers as well as the initiative of the soldiers, is more tasty and more satisfying than what any of the involved parties could have produced by themselves.

We consider the story a nice metaphor for the bits-and-pieces additions by many different parties that Fast Downward has seen in the last four or so years, which is part of the reason for calling the planner "Fast Downward Stone Soup". The second reason is that sequential portfolio algorithms in general can be seen as a "soup" of different algorithms that are stirred together to achieve a taste that hopefully exceeds that of the individual ingredients.

The idea to name a piece of software after the stone soup story is inspired by a similar case, the open-source computer game "Dungeon Crawl Stone Soup[1]", which incidentally would make for an excellent challenge of AI planning technology, similar to but much more complex than the venerable Rog-O-Matic (Mauldin et al. 1984).

Culinary Basics

Fast Downward Stone Soup is not a very sophisticated portfolio planner. Due to deadline pressures, our portfolio was chosen by a very simple selection algorithm, which had to be devised and implemented within a matter of a few hours, without any experimental evaluation, and based on limited and noisy training data. The algorithm does not aim to minimize the training data needed, does not use a separate training and validation set, and completely ignores the intricate time/cost trade-off in satisficing planning. Therefore, we do not recommend our approach as state of the art or even particularly good; rather, we describe it here to document what we did, and as a baseline for future, more sophisticated portfolio approaches.

In order to build a portfolio, we assume that the following information is available:

- A set of *planning algorithms* $\mathcal{A}$ to serve as component algorithms ("ingredients") of the portfolio. Our implementation assumes that this set is not too large; we used 11 ingredients for optimal planning and 38 ingredients for satisficing planning.
- A set of *training instances* $\mathcal{I}$, for which portfolio performance is optimized. We used the subset of IPC 1998–2008 instances that were supported by all planning algorithms we used as ingredients, a total of 1116 instances.[2]
- 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, and
  - the *plan cost* $c(A, I)$ of the plan that was found. (For training instances from IPC 1998–2006, this is simply the plan length.)

  We used a timeout of 30 minutes and memory limit of 2 GB to generate this data. In cases where an instance could not be solved within these bounds, we set $t(A, I) = c(A, I) = \infty$.

  The plan cost is of course only relevant for the satisficing track, since in the optimization track, all component algorithms produce optimal plans. We did not consider anytime planners as possible ingredients. If we had, a single runtime value and plan cost value would of course not have been sufficient to describe algorithm performance on a given instance.

In the following, we represent a (sequential) *portfolio* as a mapping $P : \mathcal{A} \to \mathbb{R}_0^+$ which assigns a time limit to each component algorithm. Time limits can be 0, indicating that a given algorithm is not used in the portfolio. The *total time limit* of portfolio P is the sum of all component time limits, $\sum_{A \in \mathcal{A}} P(A)$.

Judging the Taste of a Soup

We say that portfolio P *solves* a given instance I if any of the component algorithms solves it within its assigned runtime, i. e., if there exists an algorithm A such that $t(A, I) \leq P(A)$. The *solution cost* achieved by portfolio P on instance I is the minimal cost over all component algorithms that solve the task in their allotted time, $c(P, I) := \min \{ c(A, I) \mid A \in \mathcal{A}, t(A, I) \leq P(A) \}$. (If the portfolio does not solve I, we define the achieved solution cost as infinite.)

To evaluate the quality of a portfolio, we compute an instance score in the range 0–1 for each training instance and sum this quantity over all training instances to form a portfolio score. Higher scores correspond to better portfolios for the given benchmark set, either because they solve more instances, or because they find better plans.

In detail, training instances not solved by the portfolio are assigned a score of 0. The score of a solved instance I is computed as the lowest solution cost of any algorithm in algorithm set $\mathcal{A}$ on I, $\min_{A \in \mathcal{A}} c(A, I)$, divided by the cost achieved by the portfolio, $c(P, I)$. Note that this ratio always falls into the range 0–1 since the cost achieved by the portfolio cannot be lower than the cost achieved by the best component algorithm. (We assume that optimal costs are never 0, so that division by 0 is avoided.)

This scoring function is almost identical to the one used for IPC 2008 and IPC 2011 except that we use the best solution quality *among our algorithms* as the reference quality, rather than an objective "best known" solution as mandated by the actual IPC scoring functions. This difference is simply due to lack of time in preparing the portfolios; we did not have a set of readily usable reference results.

In the case of optimal planning, only optimal planning algorithms can be used as ingredients. In this case, the scoring function simplifies to 0 for unsolved and 1 for solved tasks, since all solutions for a given instance have the same cost.

Preparing a Planner Soup

We now describe the generic algorithm for building a planner portfolio, and then detail the specific ingredients used for IPC 2011.

[1] `http://crawl.develz.org`

[2] Fine print: we included IPC 2008 instances which require action cost support, even though three of our ingredients for optimal planning did not support costs. These planners automatically failed on all IPC 2008 instances. IPC 2008 used different instance sets for satisficing and optimal planning, and we followed this separation in our training. For technical reasons to do with hard disk space usage on our experimentation platform, we omitted the cyber security domain from IPC 2008 from the satisficing benchmark suite.

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

Figure 1: Algorithm for building a portfolio.

We use a simple hill-climbing search in the space of portfolios, shown in Figure 1. In addition to the set of ingredients (*algorithms*) and evaluation results (*results*) as described above, it takes two further arguments: the step size with which we add time slices to the current portfolio (*granularity*) and an upper bound on the total time limit for the portfolio to be generated (*timeout*). Both parameters are measured in seconds. In all cases, we set the total time limit to 1800, the time limit of the IPC.

Portfolio generation starts from an initial portfolio which assigns a runtime of 0 to each ingredient (i. e., does nothing and solves nothing). We then perform hill-climbing: in each step, we generate a set of possible *successors* to the current portfolio, which are like the current portfolio except that each successor increases the time limit of one particular algorithm by *granularity*. (Hence, the number of successors equals the number of algorithms.) We then commit to the best successor among these candidates and continue, for a total of $\lfloor \mathit{timeout}/\mathit{granularity} \rfloor$ iterations. (If we continued further after this point, the total time limit of the generated portfolio would exceed the given timeout.)

Of course there may be ties in determining the best successor, for example if none of the successors improves the current portfolio. Such ties are broken in favour of successors that increase the timeout of the component algorithm that occurs earliest in some arbitrary total order that we fix initially. We did not experiment with more sophisticated tie-breaking strategies or other search neighbourhoods.

After hill-climbing, a post-processing step reduces the time limit applied to each ingredient by considering the different ingredients in order (the same arbitrary order used for breaking ties between successors in the hill-climbing phase) and setting the time limit of each ingredient to the lowest (whole) number that would still lead to the same portfolio score. For example, if algorithm A is assigned a time limit of 720 seconds after hill-climbing but reducing this time limit to 681 seconds would not affect the portfolio score, its time limit is reduced to 681 (or less, if that still does not affect the score).

Optimizing IPC 2011 Soups

For the sequential optimization track of IPC 2011, we used the following ingredients in the portfolio building algorithm:

- *blind:* A* with a "blind" heuristic that assigns 0 to goal states and the lowest action cost among all actions of the given instance to all non-goal states. Apart from bug fixes and other minor changes, this is the baseline planner used in the sequential optimization track of IPC 2008. This algorithm was contributed by Silvia Richter.
- $h^{\max}$: A* with the $h^{\max}$ heuristic introduced by Bonet and Geffner (2001). This was implemented by Malte Helmert with contributions by Silvia Richter.
- *LM-cut:* A* with the landmark-cut heuristic (Helmert and Domshlak 2009). This was implemented by Malte Helmert. The LM-cut planner was also entered into IPC 2011 as a separate competitor.
- *RHW landmarks*, h^1 *landmarks* and *BJOLP:* LM-A* with the admissible landmark heuristic (Karpas and Domshlak 2009) using "RHW landmarks" (Richter, Helmert, and Westphal 2008), h^1-based landmarks (Keyder, Richter, and Helmert 2010) and, in the case of the "big joint optimal landmarks planner (BJOLP)", the combination of both, respectively.

  The landmark synthesis algorithms were implemented by Silvia Richter and Matthias Westphal (RHW landmarks) and Emil Keyder (h^1-based landmarks), the admissible landmark heuristic by Erez Karpas with some improvements by Malte Helmert based on earlier code by Silvia Richter and Matthias Westphal, and the LM-A* algorithm by Erez Karpas.

  BJOLP was also entered into IPC 2011 as a separate competitor.
- *M&S-LFPA*: A* with a merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007), using the original abstraction strategies suggested by Helmert et al. ("linear f-preserving abstractions"). We use three different abstraction size limits: 10000, 50000, and 100000. This was implemented by Malte Helmert.
- *M&S-bisim 1* and *M&S-bisim 2*: A* with two different merge-and-shrink heuristics, using the original merging strategies of Helmert et al. and two novel shrinking strategies based on the notion of bisimulation. The new shrinking strategies were implemented by Raz Nissim.

  A sequential portfolio of these two planners was entered into IPC 2011 as a separate competitor called "Merge-and-Shrink".

After some unprincipled initial experimentation, we set the granularity parameter for the portfolio building algorithm to 120 seconds. The resulting portfolio is shown in Table 1, which also shows the score (number of solved tasks) of the portfolio and of its ingredients on the training set.[3]

We see that the portfolio makes use of four of the eleven possible ingredients: LM-cut, BJOLP, and the two new merge-and-shrink variants.

With 654 solved instances, the portfolio significantly outperforms BJOLP, the best individual configuration, which solves 605 instances. Moreover, the portfolio does not fall far short of the holy grail of portfolio algorithms (sequential

[3]The performance of the M&S-LFPA algorithms appears to be very bad because we did not manage to implement action-cost support for these algorithms in time, so that they failed on all IPC 2008 tasks. Hence, the numbers reported are not indicative of the true potential of these heuristics.

Algorithm	Score	Time	Marginal
BJOLP	605	455	46
RHW landmarks	597	0	—
LM-cut	593	569	26
h^1 landmarks	588	0	—
M&S-bisim 1	447	175	8
$h^{\max}$	427	0	—
M&S-bisim 2	426	432	20
blind	393	0	—
M&S-LFPA 10000	316	0	—
M&S-LFPA 50000	299	0	—
M&S-LFPA 100000	286	0	—
Portfolio	654	1631	
"Holy Grail"	673		

Table 1: Variant 1 of Fast Downward Stone Soup (sequential optimization). For each algorithm A, the table shows the score (number of solved instances) achieved by A on the training set when given the full 1800 seconds, next to the time that A is assigned by the portfolio. The last column shows the marginal contribution of A, i. e., the number of instances that are no longer solved when removing A from the portfolio.

or otherwise), which is to solve the union of all instances solved by any of the possible ingredients. In our training set, there are 673 instances solved by any of the component algorithms, only 19 more than solved by the portfolio.

The portfolio in Table 1 is not globally optimal in the sense that no other fixed sequential portfolio could achieve a higher score. Indeed, after the planner submission deadline, and with substantial manual effort, we managed to find a slightly better portfolio that solves one more training instance while respecting the 1800 second limit. However, while our portfolio is not optimal on this training set, it is certainly close. We conclude that for this data set, a more sophisticated algorithm for searching the space of portfolios would not increase the number of solved instances substantially. However, a more sophisticated algorithm might guard against overfitting, and hence achieve better performance on unseen instances.

We entered the portfolio shown in Figure 1 into the sequential optimization track of IPC 2011 as variant 1 of Fast Downward Stone Soup. To partially guard against the dangers of overfitting to our training set, we also entered a second portfolio as variant 2, which included equal portions of blind search, LM-cut, BJOLP, and the two M&S-bisim variants.

Satisficing IPC 2011 Soups

Computing a good portfolio for satisficing planning is more difficult than in the case of optimal planning for various reasons. One major difficulty in the case of Fast Downward is that there is a vastly larger range of candidate algorithms to consider.

Initial experiments showed that in some cases greedy best-first search was preferable to weighted A*; in other cases the opposite was true, with no weight uniformly better than others. Sometimes, deferred evaluation is the algorithm of choice, sometimes eager evaluation is better (Richter and Helmert 2009). And last not least, combining different heuristics is very often, but far from always, beneficial (Röger and Helmert 2010).

Since generating experimental data on all training instances takes a significant amount of time, we had to limit our set of ingredients to a subset of all promising candidates. Specifically, we only considered planning algorithms with the following ingredients:

- *search algorithm:* Of the various search algorithms implemented in Fast Downward, we only experimented with greedy best-first search and with weighted A* with a weight of 3. (This weight was chosen very arbitrarily with no experimental justification at all.)
- *eager vs. lazy:* We considered both "eager" (textbook) and "lazy" (deferred evaluation) variants of both search algorithms. This is backed by the study of Richter and Helmert (2009), in which these two variants appear to be roughly equally strong, with somewhat different strengths and weaknesses.
- *preferred operators:* We only considered search algorithms that made use of preferred operators. For eager search, we only used the "dual-queue" method of exploiting preferred operators, for lazy search only the "boosted dual-queue" method, using the default (and rather arbitrary) boost value of 1000. These choices are backed by the results of Richter and Helmert (2009).
- *heuristics:* Somewhat arbitrarily, we restricted attention to four heuristics: additive heuristic h^{add} (Bonet and Geffner 2001), FF/additive heuristic h^{FF} (Hoffmann and Nebel 2001; Keyder and Geffner 2008), causal graph heuristic h^{CG} (Helmert 2004), and context-enhanced additive heuristic h^{cea} (Helmert and Geffner 2008).

  These are the four heuristics that in past experiments have produced best performance when used in isolation. We did not include the landmark heuristic used in LAMA (Richter and Westphal 2010), even though it has been shown to produce very good performance when combined with some of the other heuristics (see, e. g., Richter, Helmert, and Westphal 2008).

  Since Fast Downward supports combinations of multiple heuristics and these are very often beneficial to performance (Röger and Helmert 2010), we considered planner configurations for each of the 15 non-empty subsets of the four heuristics. Backed by the results of Röger and Helmert (2010), we only considered the "alternation" method of combining multiple heuristics.
- *action costs:* We only considered configurations of the planner that treat all actions as if they were unit-cost in the computation of heuristic values and (for weighted A*) g values. This was more due to a mistake in setting up the experiments to generate the training data than due to a conscious decision, but as Richter and Westphal (2010) have shown, this is not necessarily a bad way of handling

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	h^{FF}	926.13 / 1021	88	1.82 / 0
Weighted A* ($w = 3$)	Lazy	h^{FF}	921.71 / 1023	340	10.02 / 5
Greedy best-first	Eager	h^{FF}, h^{CG}	919.24 / 1023	76	1.15 / 0
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}	909.75 / 1021	0	—
Greedy best-first	Eager	h^{FF}, h^{CG}, h^{cea}	907.52 / 1010	73	1.25 / 0
Greedy best-first	Eager	h^{FF}, h^{cea}	906.92 / 1008	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}, h^{cea}	903.57 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}	900.52 / 1015	90	1.51 / 1
Greedy best-first	Eager	h^{add}, h^{CG}, h^{cea}	892.08 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{cea}	890.96 / 1002	0	—
Greedy best-first	Eager	h^{CG}, h^{cea}	889.93 / 1009	0	—
Greedy best-first	Eager	h^{add}, h^{CG}	888.64 / 1014	0	—
Greedy best-first	Lazy	h^{FF}	880.12 / 1042	171	7.24 / 9
Greedy best-first	Eager	h^{cea}	878.58 / 990	84	3.45 / 2
Greedy best-first	Eager	h^{add}, h^{cea}	877.41 / 999	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}, h^{cea}	874.64 / 1035	0	—
Weighted A* ($w = 3$)	Eager	h^{FF}	874.18 / 920	87	2.75 / 0
Greedy best-first	Eager	h^{add}	872.74 / 1006	0	—
Greedy best-first	Lazy	h^{FF}, h^{cea}	872.48 / 1037	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}	871.77 / 1045	49	1.93 / 2
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}, h^{cea}	861.06 / 1032	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{cea}	860.64 / 1031	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}	860.04 / 1042	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}	859.72 / 1046	0	—
Weighted A* ($w = 3$)	Lazy	h^{cea}	849.66 / 1001	0	—
Weighted A* ($w = 3$)	Eager	h^{cea}	844.67 / 938	0	—
Greedy best-first	Lazy	h^{CG}, h^{cea}	841.78 / 1026	27	1.25 / 0
Greedy best-first	Lazy	h^{add}, h^{cea}	839.60 / 1020	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}, h^{cea}	835.33 / 1019	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}	831.28 / 1030	0	—
Weighted A* ($w = 3$)	Lazy	h^{add}	830.39 / 1006	50	0.90 / 0
Weighted A* ($w = 3$)	Eager	h^{add}	828.76 / 936	166	3.35 / 3
Greedy best-first	Lazy	h^{cea}	827.57 / 1014	56	2.04 / 2
Weighted A* ($w = 3$)	Eager	h^{CG}	822.46 / 906	89	2.30 / 1
Greedy best-first	Lazy	h^{add}	808.80 / 1019	0	—
Greedy best-first	Eager	h^{CG}	802.47 / 920	0	—
Weighted A* ($w = 3$)	Lazy	h^{CG}	782.14 / 908	73	2.57 / 1
Greedy best-first	Lazy	h^{CG}	755.43 / 924	0	—
Portfolio			1057.57 / 1071	1519	
"Holy Grail"			1078.00 / 1078		

Table 2: Variant 1 of Fast Downward Stone Soup (sequential satisficing). The performance column shows the score/coverage of the configuration over all training instances. The portfolio uses 15 of the 38 possible configurations, running them between 27 and 340 seconds. The last column shows the decrease of score and number of solved instances when removing only this configuration from the portfolio.

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	h^{FF}	960.77 / 1021	330	26.12 / 4
Greedy best-first	Lazy	h^{FF}	914.58 / 1042	411	22.32 / 14
Greedy best-first	Eager	h^{cea}	909.07 / 990	213	9.93 / 5
Greedy best-first	Eager	h^{add}	904.49 / 1006	204	4.56 / 3
Greedy best-first	Lazy	h^{cea}	856.91 / 1014	57	6.17 / 4
Greedy best-first	Lazy	h^{add}	840.94 / 1019	63	1.64 / 0
Greedy best-first	Eager	h^{CG}	829.34 / 920	208	3.48 / 0
Greedy best-first	Lazy	h^{CG}	781.27 / 924	109	3.17 / 1
Portfolio			1064.23 / 1069	1595	
"Holy Grail"			1073.00 / 1073		

Table 3: Variant 2 of Fast Downward Stone Soup (sequential satisficing). Columns as in Table 2.

action costs in the IPC 2008 benchmark suite, and all previous IPC benchmarks are unit-cost anyway.

The implementations of these various planner components are due to Malte Helmert (original implementation of lazy greedy best-first search; implementation of all heuristics except FF/additive), Silvia Richter (implementation of all other search algorithms and of FF/additive heuristic), with further contributions by Gabriele Röger (search algorithms, preferred operator handling mechanisms, heuristic combination handling mechanisms) and by Erez Karpas (search algorithms).

We should emphasize that many potentially good search algorithms were not included in our portfolio, such as the combination of FF/additive heuristic and landmark heuristic used by LAMA (Richter and Westphal 2010). Also, the evaluation data we used for our analysis was partially noisy since some runs were performed before and others after major bug fixes, and machines with different hardware configurations were used for different experiments, introducing additional noise. Finally, there is good reason to believe that our simple hill-climbing algorithm for building portfolios is not good enough to find the strongest possible portfolios according to our scoring criterion.

For variant 1 of Fast Downward Stone Soup in the sequential satisficing track, we considered all possible ingredient combinations for greedy best-first search but due to limited time only included results for weighted A* using single-heuristic algorithms.

With all the caveats mentioned above, the portfolio found by the hill-climbing procedure, shown in Table 2, does indeed achieve a substantially better score than any of the ingredient algorithms. (After significant experimentation, we set the granularity parameter of the algorithm to 90 seconds.) The total score for the best ingredient, eager greedy search with the FF/additive heuristic, is 926.13, while the portfolio scores 1057.57, which is a very substantial gap. The difference between the portfolio and the "holy grail" score of 1078 (achieved by a portfolio which runs each candidate algorithm for 1800 seconds, which of course hugely exceeds the IPC time limit) is much smaller, but nevertheless substantial, so we suspect that better sequential portfolios than the one we generated exist.

For variant 2 we used only greedy best-first search with a single heuristic. The hill-climbing procedure (this time using a granularity of 110 seconds) found the portfolio shown in Table 3. Note that the performance scores are not comparable to the ones of variant 1 because they are computed for a different algorithm set $\mathcal{A}$. The best single algorithm is again eager greedy search with the FF/additive heuristic with a score of 960.77. The total score of the portfolio is 1064.23 which likewise is a huge improvement over the best single algorithm. The gap to the "holy grail" score of 1073 is narrower than for variant 1.

Serving the Soup

We have finished our description of how we computed the portfolio that entered the IPC. We now describe how exactly a run of the portfolio planner proceeds. The simplified view of a portfolio run is that the different ingredients are run in turn, each with their specified time limit, on the input planning task. However, there are some subtleties that make the picture more complicated:

- The Fast Downward planner that underlies all our ingredients consists of three components: translation, knowledge compilation, and search (Helmert 2006). The translation and knowledge compilation steps are identical for all ingredients, so we only run them once, rather than once for each ingredient. (To reflect that this computation is common to all algorithms, the training data we use for selecting portfolios is also based on search time only, not total planning time.)

  While translation and knowledge compilation are usually fast, there are cases where they can take substantial amounts of time, which means that by the time the actual portfolio run begins, we are no longer left with the complete 1800 second IPC time limit.
- The overall time budget can also change in unexpected ways during execution of the portfolio when an ingredient finishes prematurely. In addition to planner bugs, there are three reasons why an algorithm might finish before reaching its time limit: running out of memory, terminating cleanly without solving the instance[4], or finding a

[4]Most of our ingredients are complete algorithms which will

plan. In cases where the full allotted time is not used up by a portfolio ingredient, we would like to do something useful with the time that is saved.

- If a solution is found, we need to consider how to proceed. For optimal planning, the only sensible behaviour is of course to stop and return the optimal solution, but for satisficing search it is advisable to use the remaining time to search for cheaper solutions.

The first and second points imply that we need to adapt to changing time limits in some way. The second and third points imply that the *order* in which algorithms are run can be important. For example, we might want to first run algorithms that tend to fail or succeed quickly. For the first optimization portfolio, we addressed this ordering issue by beginning with those algorithms that use up memory especially quickly. For the first satisficing portfolio, we sorted algorithms by decreasing order of coverage, hence beginning with algorithms likely to *succeed* quickly. For the other portfolios, we used more arbitrary orderings.

To address changing time budgets, we treat per-algorithm time limits defined by the portfolio as *relative*, rather than absolute numbers. For example, consider a situation where after translation, knowledge compilation and running some algorithms in the portfolio, there are still 930 seconds of computation time left. Further, assume that the *remaining* algorithms in the portfolio have a total assigned runtime of 900 seconds, of which 300 seconds belong to the *next* algorithm to run. Then we assign 310 seconds, which is $300/900 = 1/3$ of the remaining time, to the next algorithm. Note that this implies that once the last algorithm in the portfolio is reached, it automatically receives all remaining computation time.[5]

The final point we need to discuss is how to take care of the anytime aspect of satisficing planning. We do this in a rather ad-hoc fashion, by modifying the portfolio behaviour after the first solution is found. First of all, the best solution found so far is always used for pruning based on g values: only paths in the state space that are cheaper than the best solution found so far are pursued.[6]

In both satisficing portfolios, all search algorithms initially ignore action costs (as in our training), since this can be expected to lead to the best coverage (Richter and Westphal 2010). However, unless all actions of task to solve are unit-cost, once a solution has been found we re-run the successful ingredient in a way that takes action costs into account, since this can be expected to produce solutions of higher quality (again, see Richter and Westphal 2010). This is done in the same way as in the LAMA planner, by treating all actions of cost c with cost $c + 1$ in the heuristics, to avoid the issues with zero-cost actions noted by Richter and Westphal (2010). All remaining ingredients of the portfolio are modified in the same way for the current portfolio run.

In the second sequential portfolio, for which we specifically limited consideration to greedy best-first search (which tends to have good coverage, but poor solution quality), we make an additional, more drastic modification once a solution has been found. Namely, we *discard* all further ingredients mentioned in the portfolio, based on the intuition that the current ingredient managed to solve the instance and therefore appears to be a good algorithm for the given instance. Hence, we use the remaining time to perform an anytime search based on the same heuristic and search type (lazy vs. greedy) as the successful algorithm, using the RWA* algorithm (Richter, Thayer, and Ruml 2010) with the weight schedule $\langle 5, 3, 2, 1 \rangle$.

Towards Better Recipes

We close our planner description by briefly mentioning a number of shortcomings of the approach we pursued for Fast Downward Stone Soup, as well as some steps towards improvements.

First, we used a very naive local search procedure. The need to tune the granularity parameter in the portfolio building algorithm highlights a significant problem with our local search neighbourhood. With a low granularity, it can easily happen that no single step in the search neighbourhood improves the current portfolio, causing the local search to act blindly. On the other hand, with a high granularity, we must always increase the algorithm time limits by large amounts even though a much smaller increase might be sufficient to achieve the same effect. A more adaptive neighbourhood would be preferable, for example along the lines of greedy algorithms for the knapsack problem that prefer packing items that maximize the value/weight ratio.

Second, our approach needed complete experimental data for each ingredient of the portfolio. This is a huge limitation because it means that we cannot experiment with nearly as many different algorithm variations as we would like to (as hinted in the description of the satisficing case, where we omitted many promising possibilities). A more sophisticated approach that generates additional experimental data (only) when needed and aims at making decisions with limited experimental data, as in the FocusedILS parameter tuning algorithm (Hutter et al. 2009) could mitigate this problem.

Third, we had to choose all possible ingredients for the portfolio a priori. We believe that there is significant potential in growing a portfolio piecemeal, adding one ingredient at a time, and then specifically *searching* for a new ingredient that complements what is already there, similar to the Hydra algorithm that has been very successfully applied to SAT solving (Xu, Hoos, and Leyton-Brown 2010).

Fourth, unlike systems like Hydra or ISAC (Kadioglu et al. 2010) that learn a classifier to determine on-line which algorithm from a given portfolio to apply to a given instance, we only use *sequential* portfolios, i. e., apply each selected ingredient to each input instance when running the portfolio

---

not terminate without finding a solution on solvable inputs, but a few exceptions exist. Namely, those algorithms that are based on h^{CG} and/or h^{cea} are not complete because these heuristics can assign infinite heuristic estimates to solvable states, hence unsafely pruning the search space.

[5] If the *last* algorithm in the sequence terminates prematurely, we have leftover time with nothing left to do. Our portfolio runner contains special-purpose code for this situation. We omit details as this seems to be an uncommon corner case.

[6] We do not prune based on h values since the heuristics we use are not admissible.

planner. We believe that this is actually not such a serious problem in planning due to the "solve quickly or not at all" property of many current planning algorithms. Indeed, it may be prudent not to commit to a single algorithm selected by an imperfect classifier.

Finally, the largest challenge we see is in building a portfolio that addresses the anytime nature of satisficing planning in a principled fashion, ideally exploiting information from previous successful searches to bias the selection of the next algorithm to run in order to find an improved solution. As far as we know, this is a wide open research area, and we believe that it holds many interesting theoretical questions as well as potential for significant practical performance gains.

References

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

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.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. 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.

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

Hunt, A., and Thomas, D. 2000. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – instance-specific algorithm configuration. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 751–756. IOS Press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 335–340. IOS Press.

Mauldin, M. L.; Jacobson, G.; Appel, A.; and Hamey, L. 1984. ROG-O-MATIC: A belligerent expert system. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*.

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., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 975–982. AAAI Press.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 137–144. AAAI Press.

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.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, 210–216. AAAI Press.