# Adapting $h^{++}$ for Proving Plan Non-Existence
## (IPC 2014 Planner Abstract)

**P@trik Haslum**
Australian National University & NICTA Optimisation Research Group
firstname.lastname@anu.edu.au

$h^{++}$ is an incremental lowerbounding procedure, based on repeatedly computing minimum-cost plans for a relaxation of the planning problem and strengthening the relaxation (Haslum 2012). If the relaxed plan is valid also for the real (unrelaxed) problem, it is an optimal plan.

If the relaxed problem is unsolvable, so is the original problem. If the original planning problem is unsolvable, the successive strengthening is guaranteed to eventually produce a relaxation that is also unsolvable.

The only change made to the $h^{++}$ procedure for the unsolvability competition is that it computes a non-optimal relaxed plan in each iteration. The relaxed problem considered in each iteration is a delete relaxation (in later iterations, of a modification of the original problem). Thus, the existence of a plan for the relaxation, and extracting such a plan if one exists, can be done in polynomial time. Finding a cost-optimal relaxed plan, in contrast, is NP-hard.

The relaxation strengthening problem transformation is potentially exponential in size. It is possible to define a different strengthening scheme, which does not preserve optimal cost, but which does preserve unsolvability and restricts growth to polynomial (Haslum 2009). However, this was not implemented for the competition.

## References

Haslum, P. 2009. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to $h^m$. In *Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09).*

Haslum, P. 2012. Incremental lower bounds for additive cost planning problems. In *Proc. 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, 74–82.

# Fast Downward Dead-End Pattern Database

**Florian Pommerening** and **Jendrik Seipp**
University of Basel
Basel, Switzerland
{florian.pommerening,jendrik.seipp}@unibas.ch

This paper describes our submission to the Unsolvability International Planning Competition 2016. It uses a *dead-end pattern database* to prune states in a breadth-first search.

Pattern databases (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001) are usually computed by projecting a planning task onto a subset of its variables (the *pattern*). For every abstract state (i.e., partial state defined on the variables in the pattern) the perfect goal distance in the projection is computed and stored. If an abstract state has no path to an abstract goal in the projection, any concrete state consistent with it cannot have a path to a goal state either. If we reach such a state during the search, it can be pruned.

One simple way to use PDBs for detecting unsolvability is to compute PDBs for a collection of patterns and then use these PDBs for pruning states during a search in the transition system of the original planning task: for every encountered state, retrieve the heuristic value of all PDBs; if any of them is $\infty$, prune the state.

However, all entries other than $\infty$ in the PDB can never be used for pruning. Likewise, abstract states that are unreachable in the abstraction are unreachable in the original task and can also never be used for pruning. *Dead-end pattern databases* thus consider only abstract states from a PDB that are reachable in the abstraction and have an infinite goal distance. Viewing each such abstract state as a partial state, we end up with a set of partial states. Any concrete state that is consistent with any partial state in the set can be pruned.

During a preprocessing step, we compute a collection of patterns and generate the PDB for each pattern. After constructing each PDB, we add the partial states that can potentially lead to pruning to our collection of dead ends and destroy the PDB again, so we only have one complete PDB and our growing collection of dead ends in memory at all times. We limit time and memory spent in the preprocessing phase and start searching once the limits are reached or all patterns in our collection have been handled. If any of the partial states is consistent with the initial state, we can stop the preprocessing early and immediately report the task as unsolvable.

The pattern collection we used for the IPC systematically computes all patterns of a certain size. We restrict our attention to *interesting* patterns as defined by Pommerening, Röger, and Helmert (2013). Once all patterns of one size are handled, we continue with the next larger size and repeat this process until either

- the time limit of 900 seconds is reached, or
- the memory limit of 10 million partial states stored in the dead-end PDB is reached, or
- a partial state consistent with the initial state is found, or
- no larger interesting pattern exists.

We implemented dead-end PDBs as a heuristic in the Fast Downward planning system (Helmert 2006) and use it to prune a simple breadth-first search. To efficiently store the set of partial states, we use a match tree data structure, similar to the way the successor generator is stored in Fast Downward. Each inner node of the match tree corresponds to one variable and has a child for each value of the variable and one additional child for a "don't care" value. Leaves determine whether the path leading to them represents a dead-end. A new partial state $p$ can be added to the match tree by following the correct value successor for every variable on which $p$ is defined and the "don't care" successor for other variables until a leaf is reached. If that leaf denotes a dead end, a more general partial state already is contained in the match tree. Otherwise, the leaf is replaced with a sequence of nodes for all remaining variables in the domain of $p$ followed by a leaf denoting a dead-end. A concrete state can be tested against all partial states in the match tree by always following both the matching value successor and the "don't care" successor. If a leaf denoting a dead end is found, the state can be pruned.

## Acknowledgments

We would like to thank all Fast Downward contributors, and Malte Helmert in particular.

## References

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

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 84–90.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *Proc. IJCAI 2013*, 2357–2364.

# Reachlunch Entering The Unsolvability IPC 2016

**Tomáš Balyo**[*]
Karlsruhe Institute of Technology
Karlsruhe, Germany
biotomas@gmail.com

**Martin Suda**[†]
Vienna University of Technology
Vienna, Austria
msuda@forsyte.tuwien.ac.at

**Abstract**

Reachlunch is a sequential portfolio planner designed to recognize unsatisfiable planning instances. In the first stage it runs blind depth-first search. If the problem is not solved by DFS then it is encoded into propositional satisfiability using the Compact Reinforced encoding. The encoded problem is handed to our (non)reachability solver based on the PDR/IC3 algorithm and implemented on top of the SAT solver Minisat.

## Introduction

Reachlunch is a planner designed to recognize unsatisfiable planning instances. Although the main focus of the planning community has traditionally been on satisfiable (solvable) problems only, more recently the importance of detecting unsatisfiable (unsolvable) instances is getting recognized and addressed (see, e.g., Bäckström, Jonsson, and Ståhlberg, 2013; Hoffmann, Kissmann, and Álvaro Torralba, 2014). This is also testified by the emergence of the Unsolvability planning competition (Muise and Lipovetzky, 2016).

The main motivation behind Reachlunch is to explore the potential for the detection of unsatisfiable planning instances of Property Directed Reachability (PDR), also called IC3, a very successful algorithm developed in the model checking community (Bradley, 2011; Eén, Mishchenko, and Brayton, 2011). As explained by Suda (2014), PDR is designed for deciding reachability in symbolically represented transition systems, which is a representation to which a PDDL planning benchmark can be translated in a straightforward way by using most of the standard encoding schemes of the planning as satisfiability paradigm (Kautz and Selman, 1996). Reachlunch uses Reinforced Encoding (Balyo, Barták, and Trunda, 2015) for that purpose.

Reachlunch is a portfolio system and complements the power of PDR with another engine (actually executed first, for a limited amount of time) based on blind depth first search. In the following sections we describe the individual ingredients behind the design of Reachlunch.

[*]Supported by DFG project SA 933/11-1.

[†]Supported by the ERC Starting Grant 2014 SYMCAR 639270 and the Austrian research project FWF RiSE S11409-N23.

## Symbolic Transition Systems

By a Symbolic Transition System (STS) we mean a finite state transition system described using the language of propositional logic, namely conjunctive normal form (CNF). A transition system is a graph having states as vertices and transitions between states as edges. There is a distinguished subset of vertices for initial states and another for goal states. We are interested in answering whether there exists a path from an initial state leading to a goal state. As a symbolic description, an STS can be exponentially more succinct than the explicit enumeration of the transition system's states. However, basic questions concerning the existence of an initial (or goal) state or the task of enumerating state's successors need to be in general delegated to a SAT solver.

Formally, an STS is a tuple $S = (\Sigma, I, U, G, T)$, where $\Sigma = \{x, y, \ldots\}$ is a finite signature, i.e. a finite set of propositional variables, $I, G, U$ are sets of clauses over $\Sigma$, and $T$ is a set of clauses over $\Sigma \cup \Sigma'$, where $\Sigma' = \{x', y', \ldots\}$ is the set of variables for the next state, a distinct copy of $\Sigma$. The set of states of $S$ is formed by all the Boolean valuations over $\Sigma$ which satisfy the $U$-clauses. Of these, those that also satisfy $I$ are the initial states and those that also satisfy $G$ are the goal states. There is a transition between states $s$ and $t$ iff $(s, t') \models T$, where $t'$ is the valuation that works on the variables of $\Sigma'$ in the same way as $t$ works on those of $\Sigma$, i.e. $t'(x') = t(x)$ for any $x \in \Sigma$.

It is easy to observe that a planning problem can be translated into an STS. In fact, most of the standard encoding schemes of the planning as satisfiability paradigm (Kautz and Selman, 1996) can be used for this purpose. At the same time, STS can be used as an input of many reachability checking algorithms developed by the hardware model checking community, in particular PDR.

## The SAT Encoding used by Reachlunch

To obtain an STS, Reachlunch uses the Reinforced Encoding (Balyo, Barták, and Trunda, 2015), which is a combination of the traditional Direct Encoding (Kautz and Selman, 1992), which encodes state variable values, and SASE (Huang, Chen, and Zhang, 2010), which encodes the transitions between the values of state variables in the planning problem. The Reinforced Encoding encodes both, which is redundant in the sense of Boolean variables, but on other hand it reduces the number of clauses in the formula and enhances unit propagation. This helps SAT solvers to solve the formulas faster.

## Property Directed Reachability

PDR is best understood as a hybrid between an explicit and a symbolic search of the given STS $S$. It explicitly constructs a path consisting of concrete states, starting from a goal state and regressing it step by step towards an initial state. (The opposite direction of traversal is also possible.) At the same time, it maintains symbolic reachability information, which is locally refined whenever the current path cannot be extended further. The reachability information guides the path construction and is bound to eventually converge to a certificate of non-reachability, if no path of arbitrary length exists.

In more detail, the reachability information takes the form of a sequence $F_i$ of sets of clauses. The first set in the sequence, $F_0$, is fixed to be equal to $I$. Each of the following sets $F_i$ *over-approximates* the image of $F_{i-1}$ with respect to the transition relation. These clause sets play a role similar to an admissible heuristic. They represent a lower bound estimate for the distance of a state to the initial state and thus provide a means to guide the search towards it. However, while a heuristic value of a particular state is normally computed only once and it remains constant during the search for a plan, the clause sets in PDR are refined continually. The refinement happens on demand, driven by the states encountered during the search.

Since the path construction happens in the context of a concrete encoding, PDR can be likened to an instance of the planning as satisfiability approach in which the construction of the assignment is controlled to grow only in one direction. PDR also proceeds iteratively, gradually disproving existence of plans of length $k = 0, 1, 2, \ldots$. After each iteration, however, a special *clause propagation* phase tries to bring as many clauses as possible from $F_i$ to $F_{i+1}$ while preserving their logical relation. If it is achieved during this phase that $F_i = F_{i+1}$ for some $i < k$, the algorithm terminates having shown that there is no path connecting the goal and initial state. In a nutshell, the proof is inductive and consists of the following three claims:

$$I \models F_i, \quad F_i \wedge T \models F_i', \quad F_i \wedge G \models \bot.$$

The individual single-step reachability queries within the STS are typically implemented by a call to a SAT solver. However, for a simple encoding of STRIPS problems an explicit polynomial time procedure can be devised. More details on this interesting algorithm presented from the planning perspective can be found in (Suda, 2014).

## The Other Engine: Blind Depth First Search

Our depth first search algorithm traverses the search space of the planning instance in a depth-first fashion by systematically trying each applicable action in each reachable state while avoiding revisiting already explored states. The ordering of the actions is guided by a trivial heuristic that prefers actions leading to states that are similar to the goal state. The heuristic simply counts the number of state variables that have the same values as the goal values of the particular variables.

## The overall architecture

Reachlunch is a portfolio system running in two stages. The first stage executes the just described simple brute-force depth-first-search (DFS) for a limited amount of time. If the DFS cannot traverse all the reachable states within its time limit and prove that no solution exists then the second stage is executed.

The second stage is relies on Property Directed Reachability. The input problem is encoded into an STS and then handed over to an implementation of PDR. We designed a new file format for this exchange, which we call DIMSPEC (Suda, 2016). It is a simple modification of the well-known DIMACS CNF format used by most SAT solvers extended to define the four individual clause sets of an STS – $I, U, G, T$.

### Implementation Details

Our planner takes input in the SAS+ format (Bäckström and Nebel, 1995). For benchmarks provided in the PDDL format we use Fast Downward (Helmert, 2006) to translate them into the SAS+ format.

The actual implementation of PDR is called minireachIC3 and is freely available (Suda, 2013). It relies on Minisat version 2.2 (Eén and Sörensson, 2003) as the backend SAT solver and is implemented in the C++ language.

The depth first search and the translation of SAS+ to SAT is implemented withing the Freelunch planning library (Balyo, 2016) which is written in Java. Hence the name of our planner – Reachlunch – Freelunch combined with reachability reasoning.

## References

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11:625–656.

Bäckström, C.; Jonsson, P.; and Ståhlberg, S. 2013. Fast detection of unsolvable planning instances using local consistency. In Helmert, M., and Röger, G., eds., *SOCS*. AAAI Press.

Balyo, T.; Barták, R.; and Trunda, O. 2015. Reinforced encoding for planning as sat. In *Acta Polytechnica CTU Proceedings*.

Balyo, T. 2016. Freelunch, an open-source java planner and planning library. Web site, `http://freelunch.eu`.

Bradley, A. R. 2011. SAT-based model checking without unrolling. In Jhala, R., and Schmidt, D. A., eds., *VMCAI*, volume 6538 of *LNCS*, 70–87. Springer.

Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In Giunchiglia, E., and Tacchella, A., eds., *SAT*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Eén, N.; Mishchenko, A.; and Brayton, R. K. 2011. Efficient implementation of property directed reachability. In Bjesse, P., and Slobodová, A., eds., *FMCAD*, 125–134. FMCAD Inc.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hoffmann, J.; Kissmann, P.; and Álvaro Torralba. 2014. “Distance”? Who cares? Tailoring Merge-and-Shrink heuristics to detect unsolvability. In *ICAPS 2014 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*. To appear.

Huang, R.; Chen, Y.; and Zhang, W. 2010. A novel transition based encoding scheme for planning as satisfiability. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of ECAI*, 359–363.

Kautz, H. A., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In Clancey, W. J., and Weld, D. S., eds., *AAAI/IAAI, Vol. 2*, 1194–1201. AAAI Press / The MIT Press.

Muise, C., and Lipovetzky, N. 2016. Unsolvability international planning competition. http://unsolve-ipc.eng.unimelb.edu.au/.

Suda, M. 2013. minireachIC3, a Minisat-based implementation of PDR. Web site, `https://github.com/quickbeam123/minireachIC3`.

Suda, M. 2014. Property directed reachability for automated planning. *J. Artif. Intell. Res. (JAIR)* 50:265–319.

Suda, M. 2016. DIMSPEC, a format for specifying symbolic transition systems. Web site, `http://forsyte.at/dimspec/`.

# IPROVERPLAN: a system description

**Konstantin Korovin**
University of Manchester,
Manchester, UK

**Martin Suda***
Vienna University of Technology,
Vienna, Austria

## Introduction

IPROVERPLAN is an automated planning system that combines searching for plans and proving non-existence of solutions. In the Unsolvability International Planning Competition (Muise and Lipovetzky, 2016) only non-existence of solutions is reported.

The idea behind IPROVERPLAN is to lift a traditional encoding of planning into SAT (Kautz and Selman, 1992) to first-order level and to use an extension of the first-order theorem prover IPROVER (Korovin, 2008) for solving (non-)reachability questions in thus obtained first-order transition system. Thus the main feature of IPROVERPLAN is that it does not start the solving process by grounding the PDDL input. The planner also uses the lifted (i.e. first order) invariants produced by the algorithm adapted from Helmert (2009, Sect. 5) for pruning the search space.

## First-Order Transition Systems

We encode planning domains as transition systems represented in (many-sorted) first-order logic. One of the main motivations behind this encoding is that first-order logic provides a higher level representation compared to propositional logic and in particular, avoids upfront grounding of the problem and at the same time reasoning can still be done efficiently by first-order theorem provers.

Our encoding falls into the effectively propositional (EPR) fragment of first-order logic which in the clausal form consists of sets of first-order clauses that do not contain function symbols other than constants. The EPR fragment is decidable (NEXPTIME-complete) and there are efficient calculi and systems for reasoning within this fragment.

We use EPR-based bounded model checking (BMC) for solving reachability problems (Emmer et al., 2012; Pérez and Voronkov, 2007; Emmer et al., 2010) and an extension of BMC with k-induction for solving non-reachability problems (Khasidashvili et al., 2015). In a nutshell, bounded model checking solves the reachability problem by symbolically unrolling the transition relation upto some bound $n$, and checking satisfiability of the resulting formula. If the BMC-unrolling of the system is satisfiable at a bound $n$ then a goal state is reachable in $n$ steps from an initial state and in this case we are done, otherwise we repeat unrolling of the system with an increased bound $n + 1$. One of the advantages of using a first-order encoding is that the system representation is not copied during the unrolling which can be the case with propositional translations.

## The encoding

IPROVERPLAN lifts traditional encodings of planning into SAT (Kautz and Selman, 1992) to first-order level. This is straightforward in the sense that predicates which are usually introduced for a SAT encoding have naturally positions for arguments and our encoding supplies universal first-order variables for these arguments instead of exhaustively grounding the predicates. However, there are various subtleties connected with the lifting which need to be addressed.

**Multiple types/sorts.** PDDL benchmarks can declare and make use of a hierarchy of types, including disjunctive types. This hierarchy needs to be flattened in order to be mapped to many-sorted first order logic.

**Finite domain.** It is in general necessary to express that the domain of discourse contains only the declared objects and that these objects are distinct. This can be expressed with the help of the equality predicate, but, in particular, the distinctness criterion leads to quadratically many axioms in the number of objects.

**Negative information about the initial state.** Although the initial state is in PDDL described using only positive information about the facts that hold, a first order encoding needs to also negatively express all the facts which do not hold. We avoid generating all the “non-mentioned”, potentially exponentially many ground facts, by using the equality predicate.

**First-delete-then-add semantics.** Even for problems officially declared as STRIPS, we sometimes need to resort to techniques for expressing conditionality of effects. That is because two first-order effects of an action may contradict each other and the semantics of PDDL dictates that in such a case the positive effect should have priority and be reflected in the successor state.[1]

*Supported by the ERC Starting Grant 2014 SYMCAR 639270 and the Austrian research project FWF RiSE S11409-N23.

[1]Without this extra measure the action would erroneously be-

**Skolemization.** At each time step of the modelled plan at least one action is applied. In the first-order lifting, we do not know which specific arguments will an action take. Thus these arguments are modelled as existential variables in the encoding and need to be Skolemized during translation to clause normal form. In order to stay within the EPR fragment, Skolem functions are translated into Skolem predicates (Baumgartner et al. (2009); Khasidashvili et al. (2015)).

There are two encodings we lifted into first order and experimented with. A serial encoding with an at-least-one axiom and classical frame axioms (McCarthy and Hayes, 1969) and a parallel encoding with mutual exclusions and explanatory frame axioms (Haas, 1987). We refer to (Ghallab et al., 2004, ch. 7.4) for further details. The competition version of iPROVERPLAN uses the serial encoding.

## iProver

iProver is an automated theorem prover for many-sorted first-order logic, based on an instantiation calculus Inst-Gen (Korovin, 2013, 2008). The basic idea behind Inst-Gen is to interleave model-guided on demand instantiations of first-order formulae with propositional reasoning in an abstraction-refinement scheme. The calculus behind iProver is a decision procedure for the EPR fragment and iProver is particularly efficient in this fragment (Sutcliffe, 2014). iProver incorporates first-order bounded model checking and k-induction which we utilised for solving planning (un)-reachability problems.

iProver is implemented in OCaml and incorporates a wide range of simplification and preprocessing techniques (Korovin, 2008; Khasidashvili and Korovin, 2016). iProver uses MiniSAT (Eén and Sörensson, 2004) for reasoning with ground abstractions and Vampire for clausification (Kovács and Voronkov, 2013; Hoder et al., 2012).

## The architecture

As a computer program, iPROVERPLAN consists of three main parts. The first part is a PDDL parser and encoder written in python. Given a PDDL input, it generates two outputs: 1) the encoded first-order transition system and 2) a Prolog representation of the input used by the invariant generator.

The second part is an SWI-Prolog implementation of the invariant generating algorithm described by Helmert (2009, Sect. 5). The invariants produced by this part enrich the transition system as universally quantified clauses referring to every time moment. Although logically redundant they enable early pruning of obviously unreachable states.

Finally, the transition system is translated into first-order conjunctive normal form by Vampire and passed to iProver.

## References

P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *J. Applied Logic*, 7(1):58–74, 2009.

N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT'03*, pages 502–518. Springer, 2004.

M. Emmer, Z. Khasidashvili, K. Korovin, C. Sticksel, and A. Voronkov. EPR-based bounded model checking at word level. In *IJCAR*, pages 210–224, 2012.

Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD*, pages 137–144, 2010.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning – theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Andrew R. Haas. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop on Reasoning about Action*. Morgan Kaufmann, 1987.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

Krystof Hoder, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Preprocessing techniques for first-order clausification. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 44–51, 2012.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

Zurab Khasidashvili and Konstantin Korovin. Predicate elimination for preprocessing in first-order theorem proving. In *SAT'16*, page to appear, 2016.

Zurab Khasidashvili, Konstantin Korovin, and Dmitry Tsarkov. EPR-based k-induction with counterexample guided abstraction refinement. In *GCAI 2015*, pages 137–150. EasyChair, 2015.

Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In *IJCAR 2008*, volume 5195 of *LNCS*, pages 292–298. Springer, 2008.

Konstantin Korovin. Inst-Gen - a modular approach to instantiation-based automated reasoning. In *Programming Logics*, pages 239–270. Springer, 2013.

Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, pages 1–35, 2013.

John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

Christian Muise and Nir Lipovetzky. Unsolvability international planning competition. http://unsolve-ipc.eng.unimelb.edu.au/, February 2016.

Juan Antonio Navarro Pérez and Andrei Voronkov. Encodings of bounded LTL model checking in effectively propositional logic. In *CADE-21*, pages 346–361, 2007.

Geoff Sutcliffe. The CADE-24 automated theorem proving system competition - CASC-24. *AI Com.*, 27(4):405–416, 2014.

come unaplicable in the encoding due to the contradicting effects.

# MS-Unsat and SimulationDominance: Merge-and-Shrink and Dominance Pruning for Proving Unsolvability

**Álvaro Torralba, Jörg Hoffmann, Peter Kissmann**
Saarland University
Saarbrücken, Germany
{torralba,hoffmann}@cs.uni-saarland.de, kissmann@googlemail.com

**Abstract**

This paper describes three different planners that participated in the 2016 unsolvability International Planning Competition (IPC). They use the Merge-and-Shrink (M&S) framework in different ways. MS-unsat tailors M&S to derive perfect unsolvability abstractions, proving unsolvability without any search. MS-unsat-irr uses the same approach with irrelevance pruning techniques to eliminate transitions and operators from the planning task. SimulationDominance performs a search using simulation-based dominance and irrelevance pruning, making use of M&S heuristics and $h^{max}$ as dead-end detectors.

## Introduction

Abstractions map the state space of the problem into a smaller abstract state space. They are commonly used to derive admissible heuristics for cost-optimal planning, by using the optimal distance in the abstract state space as an admissible estimation for the original problem. Abstraction techniques are very promising for proving unsolvability since proving that any abstraction is unsolvable is a sufficient condition for proving unsolvability (Bäckström *et al.* 2013). The question is how to design suitable abstractions for the problem at hand.

Merge-and-shrink (M&S) is a framework for deriving abstractions in a flexible way. It was originally devised for model-checking (Dräger *et al.* 2006; 2009) and later adapted to planning (Helmert *et al.* 2007; 2014; Sievers *et al.* 2014). The behavior of M&S is determined by the *shrinking* and *merging* strategies. Some shrinking strategies are *safe*, meaning that they preserve plan-existence so that the resulting abstraction is solvable if and only if the original problem is (Hoffmann *et al.* 2014). If non-safe shrinking is used, the resulting abstractions can be used as dead-end detector heuristics in a A$^*$ search.

Another further use of M&S was to derive a set of transition systems in order to compute a dominance relation (Torralba and Hoffmann 2015). This dominance relation can be used for dominance pruning during search, eliminating states such that another “at least as good” state is known. Also, this dominance relation can be used for irrelevance pruning, removing transitions during the M&S process or even planning actions while preserving at least one optimal plan (Torralba and Kissmann 2015).

In this paper we present three different planners. MS-unsat employs M&S with safe shrinking to prove unsolvability without any search on the original state space. MS-unsat-irr uses the same strategy as MS-unsat, plus irrelevance pruning. The SimulationDominance planner uses search with simulation-based dominance and irrelevance pruning, $h^{max}$, and a set of M&S heuristics. The core ideas of these planners were introduced in previous work (Hoffmann *et al.* 2014; Torralba and Hoffmann 2015; Torralba and Kissmann 2015). This paper provides a general overview of the related literature and describes the configuration we chose for the planners.

## Merge-and-Shrink

Merge-and-shrink is a framework to construct abstraction functions (Helmert *et al.* 2007; 2014). M&S works with a set of transition systems, initialized with the *atomic abstractions*, i.e. projections onto single state variables. Then, it interleaves *merging steps*, in which two transition systems are replaced by their synchronized product, with *shrinking steps*, which apply abstraction to keep the size of the transitions systems at bay. The algorithm stops when only one transition system remains and this is guaranteed to be an abstraction of the original problem. The algorithm depends on two strategies. The shrinking strategy selects how to apply abstraction to reduce the size of the transition systems. The merging strategy selects which two transition systems to merge at every step.

### Shrinking strategies

Shrinking strategies decide which states to aggregate in order to reduce the size of the transition systems. The most popular shrinking strategy is bisimulation (Nissim *et al.* 2011), which computes the coarsest goal-preserving bisimulation relation and aggregates states that are bisimilar. An important property of bisimulation is that, if only bisimulation shrinking is applied at every step, the resulting transition system is a bisimulation of the original planning task. Since bisimulation preserves goal-distance, the resulting heuristic will be perfect and cost-optimal planning can be decided without any search. Exact label reduction aggregates some labels while preserving the structure of the state space, increasing the shrinking achieved by bisimulation while preserving its useful properties.

However, when only plan existence matters, one can further shrink the transition systems while keeping a perfect heuristic such that the abstraction is solvable if and only if the original problem is. Hoffmann *et al.* (2014) introduced safe shrinking strategies based on the concept of *own-labels*, i.e. labels that only affect a single transition system and have no preconditions or effects on the rest. *Own-path* shrinking aggregates all abstract states in a cycle of own-labeled transitions. Intuitively, since those transitions can be performed with no preconditions or effects on the rest of the problem those abstract states are interchangeable and can be aggregated. Moreover, if all goal variables have been merged in a transition system, states with an own-labeled path to a goal state can be aggregated since they are always solvable. Own-path and bisimulation shrinking are *safe* shrinking strategies, so if no other shrinking is used, the resulting heuristic is the unsolvability-perfect heuristic so that it can decide whether the problem is solvable without any search.

If the size of the abstraction is still too large, other approximations can be used, such as greedy bisimulation (Nissim *et al.* 2011) or K-catching bisimulation (Katz *et al.* 2012). We use the approximate bisimulation strategy introduced by Nissim *et al.*, in which they set a maximum limit for the abstraction size.

### Merge strategies

Merge strategies can be classified into *linear* and *non-linear* merge strategies. Linear merge strategies are characterized by a variable ordering, merging an atomic abstraction at every iteration of the algorithm. The first merge strategies were linear merge strategies based on causal graph (Knoblock 1994; Helmert *et al.* 2007). Hoffmann *et al.* (2014) made an empirical study of 81 different linear merge strategies for proving unsolvability, based on the following criteria:

- **Tr**, **TrOwn**, **TrGoal**, **TrOwnGoal**: Maximize number of transitions whose labels are relevant for both transition systems. If *own* is activated, ignore transitions that are not own-labeled. If *goal* is activated considers only transitions going into a goal state.
- **CG**, **CGRoot**, and **CGLeaf:** Prefer variables with an outgoing causal graph arc to an already selected variable. If there are several such variables prefer the one ordered before (CGRoot) or behind (CGLeaf) in the strongly connected components of the causal graph. It may use the complete causal graph (Com) or only pre-eff edges.
- **LevelRoot** and **LevelLeaf:** Derived from FD's full linear order (Helmert 2006). LevelRoot prefers variables "closest to be causal graph roots", and LevelLeaf prefers variables "closest to be causal graph leaves".
- **Goal:** Prefer goal variables over non-goal variables.

Sievers *et al.* (2014) reformulated the M&S framework and generalized label reduction to work with non-linear merge strategies. They also introduced in planning the DFP non-linear merge strategy, originally used in the context of model-checking (Dräger *et al.* 2006). Other relevant non-linear merge strategy is MIASM (Fan *et al.* 2014). A recent analysis of linear and non-linear merging strategies was made by Sievers *et al.* (2016).

## Simulation-Based Dominance Pruning

Dominance pruning techniques aim to avoid the exploration of some parts of the state space, if they are proven to be worse than others (Hall *et al.* 2013). This is formalized in terms of a relation on the state space of the planning task, $\preceq$, such that $s \preceq t$ implies that $t$ is "at least as close to the goal" as $s$. Our approach is based on the well-known notion of simulation relations (Milner 1971; Gentilini *et al.* 2003). A relation $\preceq$ is a simulation if for any two states $s, t$ such that $s \preceq t$ and any transition $s \xrightarrow{l} s'$, exists another transition $t \xrightarrow{l} t'$ such that $s' \preceq t'$. The coarsest goal-respecting simulation relation can be computed in polynomial time on the size of the state space, though this is still exponential in the size of the planning task.

In order to compute a relation in polynomial time we follow a compositional approach in which the dominance relation is derived from simulation relations computed on a partition of the planning task (Torralba and Hoffmann 2015). A partition of the planning task is a set of transition systems, $\Theta_1, \dots, \Theta_k$ such that their synchronized product equals the state space of the planning task. In order to derive such partition, we use the M&S algorithm with bisimulation shrinking, changing the stopping condition by forbidding any merge that would exceed a maximum limit on the number of transitions. The coarsest goal-respecting simulations for each $\Theta_i$, $\preceq_i$, can then be combined to define a dominance relation on the state space of the planning task, $s \preceq t$ iff $\alpha_i(s) \preceq_i \alpha_i(t)$ for all $i \in 1, \dots, k$. However, the number of problems in which non-trivial simulation relations exist are limited because the transition $t \xrightarrow{l} t'$ has to use exactly the same label as $s \xrightarrow{l} s'$.

To overcome this limitation, we introduce a relation between the labels of the transition systems. A label $l'$ dominates $l$ in a transition system $\Theta_i$ iff for any transition $s \xrightarrow{l} s'$ exists another $s \xrightarrow{l'} s''$ such that $s' \preceq s''$. Then, a label-dominance simulation computes the simulation relation of all transition systems $\preceq_1, \dots, \preceq_k$ simultaneously, allowing $t \xrightarrow{l'} t'$ to simulate $s \xrightarrow{l} s'$ if $s' \preceq t'$ and $l'$ dominates $l$ on all other transition systems. Moreover, a *noop* action with no preconditions and effects is introduced in order to capture the notion of "doing nothing". Label-dominance simulation with *noop* actions finds coarser relations that are able to achieve pruning in many different benchmark domains.

Once a dominance relation has been computed, in order to perform dominance pruning during search, we keep a Binary Decision Diagram (Bryant 1986) that represents the set of all states dominated by any expanded state. Anytime a state is generated, it is pruned if it is contained in such set. To avoid unnecessary overhead, we disable dominance pruning if no state has been pruned after *1000* expansions.

## Irrelevance Pruning

Irrelevance pruning removes actions from the planning task while preserving at least one (optimal) solution. Label-dominance simulation relations can be used to detect such irrelevant transitions. Subsumed transition pruning (Torralba and Kissmann 2015) eliminates transitions $s \xrightarrow{l} t$ from the M&S transition systems if there exists another transition from $s$, $s \xrightarrow{l'} t'$ that simulates it, i.e. $t \preceq t'$ and $l'$ dominates $l$ in all other transition systems. Removing such transitions might cause some parts of the abstract state space to become unreachable, leading to additional pruning and simplification of the M&S transition systems. If all transitions corresponding to a planning action are removed, the action can be completely removed from the planning task while still preserving plan existence.

Subsumed transition pruning can be interleaved with label reduction and bisimulation shrinking but not with other shrinking strategies such as own-path shrinking. Even though both subsumed transition pruning and own-path shrinking preserve solvability (so their combination does as well) the resulting abstraction cannot safely be used to detect dead-ends on the original state space. Also, applying label reduction is not always beneficial for subsumed transition pruning so we follow three different steps, where $M$ is a parameter that controls how large the transition systems are:

1. M&S with subsumed transition pruning and a limit of $M$ transitions. Without label-reduction or any shrinking.
2. M&S with subsumed transition pruning, label-reduction and bisimulation shrinking. Limit of $M$ transitions.
3. M&S with label-reduction, and own-path + bisimulation shrinking.

If dominance pruning is used, the label-dominance simulation relation is computed after the second step.

## IPC Configuration

We implemented the new merge and shrinking strategies on top of the Fast Downward Planning System (Helmert 2006) (version from July 16th, 2014). All our planners use $h^2$ forward and backward relevance analysis in order to eliminate operators and simplify the planning task prior to the search (Alcázar and Torralba 2015).

All runs of M&S use the exact label reduction by Sievers *et al.* (Sievers *et al.* 2014), interrupting it if it takes more than 60 seconds. To avoid overhead, if there are more than 200 labels, label-dominance is computed only with respect to the *noop* action.

### MS-unsat and MS-unsat-irr

We submit two different configurations MS-unsat and MS-unsat-irr. MS-unsat uses the best configuration reported by Hoffmann *et al.* (2014), using CGRoot-Goal-LevelLeaf merge and own-label shrinking.

MS-unsat-irr uses two runs of M&S with irrelevance pruning. In the first one, it uses the DFP non-linear merge strategy with irrelevance pruning with a limit of $M = 50\,000$ transitions and 300 seconds. If the task has not been proven unsolvable by the first run, irrelevant operators are removed from the problem. Afterwards, it performs another M&S run using CGRoot-Goal-LevelLeaf, and subsumed transition pruning up to a limit of 50 000 transitions.

### SimulationDominance

The SimulationDominance planner performs a search using dominance and irrelevance pruning, and the $h^{max}$ heuristic (Bonet and Geffner 2001) and M&S abstractions as dead-end detectors.

The dominance pruning relation is derived using the DFP-merge strategy with a limit of 100 000 transitions. Then, it uses M&S to generate a list of M&S abstractions, that are used during the search to detect dead-ends. All M&S runs use subsumption pruning up to $M = 100\,000$ transitions and set a limit of 500000 abstract states for bisimulation on the third step. Multiple linear merge strategies are used in a sequential fashion: TrOwnGoal-CGComLeaf-Goal, Tr, TrOwnGoal, Tr, TrOwn, CG-Goal, CGLeaf-Goal, CGRoot-Goal, CGComLeaf-Goal, TrOwnGoal-CGComLeaf-Goal. All these strategies are run twice, using the LevelLeaf and random tie-breaking, respectively. Each run of M&S may take up to 300 seconds and the overall abstraction generation may take up to 1400 seconds, after which the search starts.

## Conclusions

In this paper, we have introduced three different papers that participated in the 2016 edition of the unsolvability IPC: MS-unsat, MS-unsat-irr, and SimulationDominance. MS-unsat and MS-unsat-irr make use of M&S with a safe shrinking strategy that allows to prove unsolvability without searching the original state space. SimulationDominance uses M&S to construct a set dead-end detection heuristics as well as a label-dominance simulation relation used for dominance and irrelevance pruning.

**Acknowledgments** We'd like to thank the Fast Downward development team for sharing the latest version of their Fast Downward Planning System and, in particular, to Silvan Sievers, Martin Wehrle, and Malte Helmert for their work on M&S (Sievers *et al.* 2014). This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation".

## References

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press, 2015.

Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. In Malte Helmert and Gabriele Röger, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, pages 29–37. AAAI Press, 2013.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2006.

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, 11(1):27–37, 2009.

Gaojian Fan, Martin Müller, and Robert Holte. Non-linear merging strategies for merge-and-shrink based on variable interactions. In Stefan Edelkamp and Roman Bartak, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press, 2014.

Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

David Hall, Alon Cohen, David Burkett, and Dan Klein. Faster optimal planning with partial-order pruning. In Daniel Borrajo, Simone Fratini, Subbarao Kambhampati, and Angelo Oddi, editors, *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, Rome, Italy, 2013. AAAI Press.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiebaux, editors, *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pages 176–183, Providence, Rhode Island, USA, 2007. Morgan Kaufmann.

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*, 61(3), 2014.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Thorsten Schaub, editor, *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, Prague, Czech Republic, August 2014. IOS Press.

Michael Katz, Jörg Hoffmann, and Malte Helmert. How to relax a bisimulation? In Blai Bonet, Lee McCluskey, José Reinaldo Silva, and Brian Williams, editors, *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 101–109. AAAI Press, 2012.

Craig Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*, pages 481–489, London, UK, September 1971. William Kaufmann.

Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1983–1990. AAAI Press/IJCAI, 2011.

Silvan Sievers, Martin Wehrle, and Malte Helmert. Generalized label reduction for merge-and-shrink heuristics. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pages 2358–2366, Austin, Texas, USA, January 2014. AAAI Press.

Silvan Sievers, Martin Wehrle, and Malte Helmert. An analysis of merge strategies for merge-and-shrink heuristics. In Amanda Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS'16)*. AAAI Press, 2016.

Álvaro Torralba and Jörg Hoffmann. Simulation-based admissible dominance pruning. In Qiang Yang, editor, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 1689–1695. AAAI Press/IJCAI, 2015.

Álvaro Torralba and Peter Kissmann. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In Levi Lelis and Roni Stern, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*, pages 122–130. AAAI Press, 2015.

# MS-Unsat and SimulationDominance: Merge-and-Shrink and Dominance Pruning for Proving Unsolvability

**Álvaro Torralba, Jörg Hoffmann, Peter Kissmann**
Saarland University
Saarbrücken, Germany
{torralba,hoffmann}@cs.uni-saarland.de, kissmann@googlemail.com

**Abstract**

This paper describes three different planners that participated in the 2016 unsolvability International Planning Competition (IPC). They use the Merge-and-Shrink (M&S) framework in different ways. MS-unsat tailors M&S to derive perfect unsolvability abstractions, proving unsolvability without any search. MS-unsat-irr uses the same approach with irrelevance pruning techniques to eliminate transitions and operators from the planning task. SimulationDominance performs a search using simulation-based dominance and irrelevance pruning, making use of M&S heuristics and $h^{max}$ as dead-end detectors.

## Introduction

Abstractions map the state space of the problem into a smaller abstract state space. They are commonly used to derive admissible heuristics for cost-optimal planning, by using the optimal distance in the abstract state space as an admissible estimation for the original problem. Abstraction techniques are very promising for proving unsolvability since proving that any abstraction is unsolvable is a sufficient condition for proving unsolvability (Bäckström *et al.* 2013). The question is how to design suitable abstractions for the problem at hand.

Merge-and-shrink (M&S) is a framework for deriving abstractions in a flexible way. It was originally devised for model-checking (Dräger *et al.* 2006; 2009) and later adapted to planning (Helmert *et al.* 2007; 2014; Sievers *et al.* 2014). The behavior of M&S is determined by the *shrinking* and *merging* strategies. Some shrinking strategies are *safe*, meaning that they preserve plan-existence so that the resulting abstraction is solvable if and only if the original problem is (Hoffmann *et al.* 2014). If non-safe shrinking is used, the resulting abstractions can be used as dead-end detector heuristics in a A$^*$ search.

Another further use of M&S was to derive a set of transition systems in order to compute a dominance relation (Torralba and Hoffmann 2015). This dominance relation can be used for dominance pruning during search, eliminating states such that another "at least as good" state is known. Also, this dominance relation can be used for irrelevance pruning, removing transitions during the M&S process or even planning actions while preserving at least one optimal plan (Torralba and Kissmann 2015).

In this paper we present three different planners. MS-unsat employs M&S with safe shrinking to prove unsolvability without any search on the original state space. MS-unsat-irr uses the same strategy as MS-unsat, plus irrelevance pruning. The SimulationDominance planner uses search with simulation-based dominance and irrelevance pruning, $h^{max}$, and a set of M&S heuristics. The core ideas of these planners were introduced in previous work (Hoffmann *et al.* 2014; Torralba and Hoffmann 2015; Torralba and Kissmann 2015). This paper provides a general overview of the related literature and describes the configuration we chose for the planners.

## Merge-and-Shrink

Merge-and-shrink is a framework to construct abstraction functions (Helmert *et al.* 2007; 2014). M&S works with a set of transition systems, initialized with the *atomic abstractions*, i.e. projections onto single state variables. Then, it interleaves *merging steps*, in which two transition systems are replaced by their synchronized product, with *shrinking steps*, which apply abstraction to keep the size of the transitions systems at bay. The algorithm stops when only one transition system remains and this is guaranteed to be an abstraction of the original problem. The algorithm depends on two strategies. The shrinking strategy selects how to apply abstraction to reduce the size of the transition systems. The merging strategy selects which two transition systems to merge at every step.

### Shrinking strategies

Shrinking strategies decide which states to aggregate in order to reduce the size of the transition systems. The most popular shrinking strategy is bisimulation (Nissim *et al.* 2011), which computes the coarsest goal-preserving bisimulation relation and aggregates states that are bisimilar. An important property of bisimulation is that, if only bisimulation shrinking is applied at every step, the resulting transition system is a bisimulation of the original planning task. Since bisimulation preserves goal-distance, the resulting heuristic will be perfect and cost-optimal planning can be decided without any search. Exact label reduction aggregates some labels while preserving the structure of the state space, increasing the shrinking achieved by bisimulation while preserving its useful properties.

However, when only plan existence matters, one can further shrink the transition systems while keeping a perfect heuristic such that the abstraction is solvable if and only if the original problem is. Hoffmann *et al.* (2014) introduced safe shrinking strategies based on the concept of *own-labels*, i.e. labels that only affect a single transition system and have no preconditions or effects on the rest. *Own-path* shrinking aggregates all abstract states in a cycle of own-labeled transitions. Intuitively, since those transitions can be performed with no preconditions or effects on the rest of the problem those abstract states are interchangeable and can be aggregated. Moreover, if all goal variables have been merged in a transition system, states with an own-labeled path to a goal state can be aggregated since they are always solvable. Own-path and bisimulation shrinking are *safe* shrinking strategies, so if no other shrinking is used, the resulting heuristic is the unsolvability-perfect heuristic so that it can decide whether the problem is solvable without any search.

If the size of the abstraction is still too large, other approximations can be used, such as greedy bisimulation (Nissim *et al.* 2011) or K-catching bisimulation (Katz *et al.* 2012). We use the approximate bisimulation strategy introduced by Nissim *et al.*, in which they set a maximum limit for the abstraction size.

### Merge strategies

Merge strategies can be classified into *linear* and *non-linear* merge strategies. Linear merge strategies are characterized by a variable ordering, merging an atomic abstraction at every iteration of the algorithm. The first merge strategies were linear merge strategies based on causal graph (Knoblock 1994; Helmert *et al.* 2007). Hoffmann *et al.* (2014) made an empirical study of 81 different linear merge strategies for proving unsolvability, based on the following criteria:

- **Tr**, **TrOwn**, **TrGoal**, **TrOwnGoal**: Maximize number of transitions whose labels are relevant for both transition systems. If *own* is activated, ignore transitions that are not own-labeled. If *goal* is activated considers only transitions going into a goal state.
- **CG**, **CGRoot**, and **CGLeaf:** Prefer variables with an outgoing causal graph arc to an already selected variable. If there are several such variables prefer the one ordered before (CGRoot) or behind (CGLeaf) in the strongly connected components of the causal graph. It may use the complete causal graph (Com) or only pre-eff edges.
- **LevelRoot** and **LevelLeaf:** Derived from FD's full linear order (Helmert 2006). LevelRoot prefers variables "closest to be causal graph roots", and LevelLeaf prefers variables "closest to be causal graph leaves".
- **Goal:** Prefer goal variables over non-goal variables.

Sievers *et al.* (2014) reformulated the M&S framework and generalized label reduction to work with non-linear merge strategies. They also introduced in planning the DFP non-linear merge strategy, originally used in the context of model-checking (Dräger *et al.* 2006). Other relevant non-linear merge strategy is MIASM (Fan *et al.* 2014). A recent analysis of linear and non-linear merging strategies was made by Sievers *et al.* (2016).

## Simulation-Based Dominance Pruning

Dominance pruning techniques aim to avoid the exploration of some parts of the state space, if they are proven to be worse than others (Hall *et al.* 2013). This is formalized in terms of a relation on the state space of the planning task, $\preceq$, such that $s \preceq t$ implies that $t$ is "at least as close to the goal" as $s$. Our approach is based on the well-known notion of simulation relations (Milner 1971; Gentilini *et al.* 2003). A relation $\preceq$ is a simulation if for any two states $s, t$ such that $s \preceq t$ and any transition $s \xrightarrow{l} s'$, exists another transition $t \xrightarrow{l} t'$ such that $s' \preceq t'$. The coarsest goal-respecting simulation relation can be computed in polynomial time on the size of the state space, though this is still exponential in the size of the planning task.

In order to compute a relation in polynomial time we follow a compositional approach in which the dominance relation is derived from simulation relations computed on a partition of the planning task (Torralba and Hoffmann 2015). A partition of the planning task is a set of transition systems, $\Theta_1, \dots, \Theta_k$ such that their synchronized product equals the state space of the planning task. In order to derive such partition, we use the M&S algorithm with bisimulation shrinking, changing the stopping condition by forbidding any merge that would exceed a maximum limit on the number of transitions. The coarsest goal-respecting simulations for each $\Theta_i$, $\preceq_i$, can then be combined to define a dominance relation on the state space of the planning task, $s \preceq t$ iff $\alpha_i(s) \preceq_i \alpha_i(t)$ for all $i \in 1, \dots, k$. However, the number of problems in which non-trivial simulation relations exist are limited because the transition $t \xrightarrow{l} t'$ has to use exactly the same label as $s \xrightarrow{l} s'$.

To overcome this limitation, we introduce a relation between the labels of the transition systems. A label $l'$ dominates $l$ in a transition system $\Theta_i$ iff for any transition $s \xrightarrow{l} s'$ exists another $s \xrightarrow{l'} s''$ such that $s' \preceq s''$. Then, a label-dominance simulation computes the simulation relation of all transition systems $\preceq_1, \dots, \preceq_k$ simultaneously, allowing $t \xrightarrow{l'} t'$ to simulate $s \xrightarrow{l} s'$ if $s' \preceq t'$ and $l'$ dominates $l$ on all other transition systems. Moreover, a *noop* action with no preconditions and effects is introduced in order to capture the notion of "doing nothing". Label-dominance simulation with *noop* actions finds coarser relations that are able to achieve pruning in many different benchmark domains.

Once a dominance relation has been computed, in order to perform dominance pruning during search, we keep a Binary Decision Diagram (Bryant 1986) that represents the set of all states dominated by any expanded state. Anytime a state is generated, it is pruned if it is contained in such set. To avoid unnecessary overhead, we disable dominance pruning if no state has been pruned after *1000* expansions.

## Irrelevance Pruning

Irrelevance pruning removes actions from the planning task while preserving at least one (optimal) solution. Label-dominance simulation relations can be used to detect such irrelevant transitions. Subsumed transition pruning (Torralba and Kissmann 2015) eliminates transitions $s \xrightarrow{l} t$ from the M&S transition systems if there exists another transition from $s$, $s \xrightarrow{l'} t'$ that simulates it, i.e. $t \preceq t'$ and $l'$ dominates $l$ in all other transition systems. Removing such transitions might cause some parts of the abstract state space to become unreachable, leading to additional pruning and simplification of the M&S transition systems. If all transitions corresponding to a planning action are removed, the action can be completely removed from the planning task while still preserving plan existence.

Subsumed transition pruning can be interleaved with label reduction and bisimulation shrinking but not with other shrinking strategies such as own-path shrinking. Even though both subsumed transition pruning and own-path shrinking preserve solvability (so their combination does as well) the resulting abstraction cannot safely be used to detect dead-ends on the original state space. Also, applying label reduction is not always beneficial for subsumed transition pruning so we follow three different steps, where $M$ is a parameter that controls how large the transition systems are:

1. M&S with subsumed transition pruning and a limit of $M$ transitions. Without label-reduction or any shrinking.
2. M&S with subsumed transition pruning, label-reduction and bisimulation shrinking. Limit of $M$ transitions.
3. M&S with label-reduction, and own-path + bisimulation shrinking.

If dominance pruning is used, the label-dominance simulation relation is computed after the second step.

## IPC Configuration

We implemented the new merge and shrinking strategies on top of the Fast Downward Planning System (Helmert 2006) (version from July 16th, 2014). All our planners use $h^2$ forward and backward relevance analysis in order to eliminate operators and simplify the planning task prior to the search (Alcázar and Torralba 2015).

All runs of M&S use the exact label reduction by Sievers *et al.* (Sievers *et al.* 2014), interrupting it if it takes more than 60 seconds. To avoid overhead, if there are more than 200 labels, label-dominance is computed only with respect to the *noop* action.

### MS-unsat and MS-unsat-irr

We submit two different configurations MS-unsat and MS-unsat-irr. MS-unsat uses the best configuration reported by Hoffmann *et al.* (2014), using CGRoot-Goal-LevelLeaf merge and own-label shrinking.

MS-unsat-irr uses two runs of M&S with irrelevance pruning. In the first one, it uses the DFP non-linear merge strategy with irrelevance pruning with a limit of $M = 50\,000$ transitions and 300 seconds. If the task has not been proven unsolvable by the first run, irrelevant operators are removed from the problem. Afterwards, it performs another M&S run using CGRoot-Goal-LevelLeaf, and subsumed transition pruning up to a limit of 50 000 transitions.

### SimulationDominance

The SimulationDominance planner performs a search using dominance and irrelevance pruning, and the $h^{max}$ heuristic (Bonet and Geffner 2001) and M&S abstractions as dead-end detectors.

The dominance pruning relation is derived using the DFP-merge strategy with a limit of 100 000 transitions. Then, it uses M&S to generate a list of M&S abstractions, that are used during the search to detect dead-ends. All M&S runs use subsumption pruning up to $M = 100\,000$ transitions and set a limit of 500000 abstract states for bisimulation on the third step. Multiple linear merge strategies are used in a sequential fashion: TrOwnGoal-CGComLeaf-Goal, Tr, TrOwnGoal, Tr, TrOwn, CG-Goal, CGLeaf-Goal, CGRoot-Goal, CGComLeaf-Goal, TrOwnGoal-CGComLeaf-Goal. All these strategies are run twice, using the LevelLeaf and random tie-breaking, respectively. Each run of M&S may take up to 300 seconds and the overall abstraction generation may take up to 1400 seconds, after which the search starts.

## Conclusions

In this paper, we have introduced three different papers that participated in the 2016 edition of the unsolvability IPC: MS-unsat, MS-unsat-irr, and SimulationDominance. MS-unsat and MS-unsat-irr make use of M&S with a safe shrinking strategy that allows to prove unsolvability without searching the original state space. SimulationDominance uses M&S to construct a set dead-end detection heuristics as well as a label-dominance simulation relation used for dominance and irrelevance pruning.

**Acknowledgments** We'd like to thank the Fast Downward development team for sharing the latest version of their Fast Downward Planning System and, in particular, to Silvan Sievers, Martin Wehrle, and Malte Helmert for their work on M&S (Sievers *et al.* 2014). This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation".

## References

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press, 2015.

Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. In Malte Helmert and Gabriele Röger, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, pages 29–37. AAAI Press, 2013.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 2006.

Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer*, 11(1):27–37, 2009.

Gaojian Fan, Martin Müller, and Robert Holte. Non-linear merging strategies for merge-and-shrink based on variable interactions. In Stefan Edelkamp and Roman Bartak, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press, 2014.

Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

David Hall, Alon Cohen, David Burkett, and Dan Klein. Faster optimal planning with partial-order pruning. In Daniel Borrajo, Simone Fratini, Subbarao Kambhampati, and Angelo Oddi, editors, *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, Rome, Italy, 2013. AAAI Press.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiebaux, editors, *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pages 176–183, Providence, Rhode Island, USA, 2007. Morgan Kaufmann.

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*, 61(3), 2014.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Thorsten Schaub, editor, *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, Prague, Czech Republic, August 2014. IOS Press.

Michael Katz, Jörg Hoffmann, and Malte Helmert. How to relax a bisimulation? In Blai Bonet, Lee McCluskey, José Reinaldo Silva, and Brian Williams, editors, *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 101–109. AAAI Press, 2012.

Craig Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*, pages 481–489, London, UK, September 1971. William Kaufmann.

Raz Nissim, Jörg Hoffmann, and Malte Helmert. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1983–1990. AAAI Press/IJCAI, 2011.

Silvan Sievers, Martin Wehrle, and Malte Helmert. Generalized label reduction for merge-and-shrink heuristics. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pages 2358–2366, Austin, Texas, USA, January 2014. AAAI Press.

Silvan Sievers, Martin Wehrle, and Malte Helmert. An analysis of merge strategies for merge-and-shrink heuristics. In Amanda Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS'16)*. AAAI Press, 2016.

Álvaro Torralba and Jörg Hoffmann. Simulation-based admissible dominance pruning. In Qiang Yang, editor, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 1689–1695. AAAI Press/IJCAI, 2015.

Álvaro Torralba and Peter Kissmann. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In Levi Lelis and Roni Stern, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*, pages 122–130. AAAI Press, 2015.

# Decoupled Search for Proving Unsolvability

**Daniel Gnad** and **Álvaro Torralba** and **Jörg Hoffmann**
Saarland University
Saarbrücken, Germany
{gnad, torralba, hoffmann}@cs.uni-saarland.de

**Martin Wehrle**
University of Basel
Basel, Switzerland
martin.wehrle@unibas.ch

## Introduction

Decoupled State Space Search is a recently introduced method to handle the well-known state space explosion problem (Gnad and Hoffmann 2015; Gnad, Hoffmann, and Domshlak 2015). By exploiting the structure of the problem within the search – as opposed to doing that within a heuristic function guiding the search – the size of the *decoupled state space* can be exponentially smaller than that of the standard state space. Decoupled search achieves that by partitioning the task into several components, called *factors*, trying to identify a *star topology*, with a single *center factor* that interacts with multiple *leaf factors*. By enforcing such a star structure, and thereby simplifying the dependencies between the components, decoupled search has proved to be very efficient and able to compete with other state-of-the-art planners in both satisficing and optimal search. We have also seen good performance of decoupled search in the limited unsolvable benchmarks, available prior to this competition. Whether these results translate to the competition is mainly dependent on the structure of the used domains. Since the currently implemented method to identify factorings is only capable of detecting so-called X-shape profiles, we cannot perform decoupled search in absence of such structure. In such a case, we simply run standard search, instead. As a side remark, this limitation is merely due to the preliminary methods to identify suitable factorings. In general, *every* task has a star topology and can be tackled by decoupled search.

Depending on the particular factoring profile that has been identified, we also enable extensions of decoupled search that have recently been developed, namely partial-order reduction (POR) (Gnad, Wehrle, and Hoffmann 2016) and dominance pruning (Torralba et al. 2016). POR via strong stubborn sets is a technique that is well-known in standard search and originates from the model checking community (Valmari 1989; Alkhazraji et al. 2012; Wehrle and Helmert 2012; 2014). Dominance pruning identifies states that can be safely discarded, without affecting completeness (and optimality). Both of these techniques can only be used if the identified factoring has the form of a fork. We also enable POR, whenever our factoring method does not find a suitable profile.

## Preliminaries

We use a finite-domain state variable formalization of planning (e. g. (Bäckström and Nebel 1995; Helmert 2006)). A *finite-domain representation* planning task, short FDR task, is a quadruple $\Pi = \langle V, A, I, G \rangle$. $V$ is a set of *state variables*, where each $v \in V$ is associated with a finite domain $\mathcal{D}(v)$. We identify (partial) variable assignments with sets of variable/value pairs. A complete assignment to $V$ is a *state*. $I$ is the *initial state*, and the *goal* $G$ is a partial assignment to $V$. $A$ is a finite set of *actions*. Each action $a \in A$ is a tuple $\langle \mathsf{pre}(a), \mathsf{eff}(a) \rangle$ where the *precondition* $\mathsf{pre}(a)$ and *effect* $\mathsf{eff}(a)$ are partial assignments to $V$.

For a partial assignment $p$, $\mathcal{V}(p) \subseteq V$ denotes the subset of state variables instantiated by $p$. For any $V' \subseteq \mathcal{V}(p)$, by $p[V']$ we denote the assignment to $V'$ made by $p$. An action $a$ is *applicable* in a state $s$ if $\mathsf{pre}(a) \subseteq s$, i. e., if $s[v] = \mathsf{pre}(a)[v]$ for all $v \in \mathcal{V}(\mathsf{pre}(a))$. Applying $a$ in $s$ changes the value of each $v \in \mathcal{V}(\mathsf{eff}(a))$ to $\mathsf{eff}(a)[v]$, and leaves $s$ unchanged elsewhere; the outcome state is denoted $s[\![a]\!]$. We also use this notation for partial states $p$: by $p[\![a]\!]$ we denote the assignment over-writing $p$ with $\mathsf{eff}(a)$ where both $p$ and $\mathsf{eff}(a)$ are defined. The outcome state of applying a sequence of (respectively applicable) actions is denoted $s[\![\langle a_1, \dots, a_n \rangle]\!]$. A *plan* for $\Pi$ is an action sequence s.t. $G \subseteq I[\![\langle a_1, \dots, a_n \rangle]\!]$. For the task of proving a task unsolvable, we are only interested in the existence of a plan that transforms the initial state $I$ to a goal state $s_G$, with $s_G[v] = G[v]$ for all $v \in \mathcal{V}(G)$. We consequently ignore action costs in the following.

To identify the required structure for factoring the variables, we need the notion of the *causal graph* (e. g. (Knoblock 1994; Jonsson and Bäckström 1995; Brafman and Domshlak 2003; Helmert 2006)). The causal graph of a planning task captures state variable dependencies. We use the commonly employed definition in the FDR context, where the causal graph *CG* is a directed graph over vertices $V$, with an arc from $v$ to $v'$, which we denote $(v \rightarrow v')$, if $v \neq v'$ and there exists an action $a \in A$ such that $(v, v') \in [\mathcal{V}(\mathsf{eff}(a)) \cup \mathcal{V}(\mathsf{pre}(a))] \times \mathcal{V}(\mathsf{eff}(a))$. In words, the causal graph captures precondition-effect as well as effect-effect dependencies, as result from the action descriptions. A simple intuition is that, whenever $(v \rightarrow v')$ is an arc in *CG*, changing the value of $v'$ may involve changing that of $v$ as well. We assume for simplicity that *CG* is weakly connected (this is wlog: else, the task can be equivalently split into several independent tasks).

## Decoupled Search

We run decoupled search like introduced by Gnad, Hoffmann, and Domshlak (2015), with the same factoring strategy and search settings, i. e., optimized for satisficing search. Since there is no difference in the main algorithm, we only give a brief summary, here.

Prior to search, the factoring of the input task $\Pi$ is performed, by analyzing its causal graph. Denote by $\mathcal{F}^{\text{SCC}}$ the factoring whose factors are the strongly connected components (SCC) of *CG*. The *interaction graph* $IG(\mathcal{F})$ of a factoring $\mathcal{F}$ is the directed graph whose vertices are the factors, with an arc $(F \rightarrow F')$ if $F \neq F'$ and there exist $v \in F$ and $v' \in F'$ such that $(v \rightarrow v')$ is an arc in *CG*. The actual factoring works as follows: In a first step, each leaf in $\mathcal{F}^{\text{SCC}}$ will be assigned to a single leaf factor $F^L$. If the causal graph is not strongly connected, i. e., at least one such leaf exists, this results in a fork factoring, where – denoting by $F^{C'}$ the remaining components – all transitions in $IG(\mathcal{F}^{\text{SCC}})$ are of the form $(F^{C'} \rightarrow F^L)$. In a second step, each root from the sub-graph of $IG(\mathcal{F}^{\text{SCC}})$ that only contains the components in $F^{C'}$ is also assigned to a new leaf factor. By $F^C$ we denote the remaining components that have not been assigned to a leaf. Finally, all leaves $F^{L_2}$ detected in the second step that introduce transitions in $IG(\mathcal{F}^{\text{SCC}})$ of the form $(F^{L_2} \rightarrow F^{L_1})$ will be put back into $F^C$, to prevent dependencies across leaf factors. If leaves have been detected in both steps and at least one of those from step 2 has not been removed, this results in X-shape factoring with "inverted-fork" leaves that provide preconditions for the center, and "fork" leaves, that only have preconditions on the center and themselves. If only in the second step leaves have been added, this results in a pure inverted-fork factoring.

Given a factoring $\mathcal{F}$ with center factor $F^C$ and leaves $F^L \in \mathcal{F}^L$, decoupled search is performed as follows:

The search will only branch over center actions, i. e., those actions affecting a variable in $F^C$. Along such a path of center actions $\pi^C$, for each leaf factor $F^L$, the search maintains a set of leaf paths, i. e., actions only affecting variables of $F^L$, that *comply* with $\pi^C$. Intuitively, for a leaf path $\pi^L$ to comply with a center path, it must be possible to embed $\pi^L$ into $\pi^C$ such that the $F^L$-preconditions of all center actions are provided by $\pi^L$ at the respective points in $\pi^C$, and the $F^C$ preconditions of all leaf actions are provided by $\pi^C$.

A decoupled state corresponds to an end state of such a center action sequence. The main advantage over standard search originates from a decoupled state being able to represent exponentially many explicit states, thereby getting rid of having to enumerate all of them. A decoupled state can "contain" many explicit states, because by instantiating the center with a center action sequence, the leaf factors are mutually independent. Thus, the more leaves in the factoring, the more explicit states can potentially be represented by a single decoupled state.

### Decoupled strong stubborn sets

In addition to the plain decoupled search variant outlined above, we enable decoupled strong stubborn sets (DSSS), when the factoring method results in a fork topology. The usage of this technique is identical to what has been introduced in Gnad, Wehrle, and Hoffmann (2016), so we don't give the formal details, here. DSSS are a straightforward extension of POR to the decoupled search setting, where some care must be taken due to the specific structure of the decoupled state space, especially the distinction between center and leaf actions. Like in the standard state space, it removes transitions that will lead to different permutations of action sequences leading to the same outcome state. The only minor difference to the original implementation is a "safety belt", that disables DSSS if after the first 1000 expansions, not a single transition has been removed.

### Decoupled dominance pruning

Another decoupled search extension that has only recently been introduced is dominance pruning (Torralba et al. 2016), where decoupled states that are dominated by other – already visited – states can be safely discarded. We only deploy a very lightweight pruning method, namely *frontier* pruning. The plain decoupled search variant performs a duplicate checking that can already detect certain forms of dominance, in particular if two decoupled states have the same center state and all leaf states reachable in one state are (at most as costly) also reachable in the other. Frontier pruning improves this by only comparing a subset of the reached leaf states, those that can possibly make so far unreached leaf states available. It has originally been developed for optimal planning, but can be easily adapted to become more efficient, when optimal solutions do not matter, by replacing the real cost of reaching a leaf state by 0, if a state has been reached at any cost.

Additionally, we also employ a leaf simulation, originally proposed by Torralba and Kissmann (2015), to remove superfluous leaf states and leaf actions, discovering transitions that can be replaced by other transitions, then running a reachability check on the leaf state space. In some domains, this can tremendously reduce the size of the leaf state spaces.

## Implementation

Decoupled Search has been implemented as an extension of the Fast Downward (FD) planning system (Helmert 2006). By changing the low-level state representation, many of FD's built-in algorithms and functionality can be used with only minor adaptations. Of particular interest for the task of proving unsolvability are the A$^*$ algorithm, the $h^{max}$ heuristic (Bonet and Geffner 2001) for dead-end pruning, and partial-order reduction via strong stubborn sets. On top of the standard FD preprocessor, we perform a relevance analysis based on $h^2$, in order to eliminate actions and simplify the planning task prior to the search (Alcázar and Torralba 2015). In some domains, this relevance analysis is even powerful enough to detect a task unsolvable without actually having to start the search.

All our search variants run A$^*$ using $h^{max}$. The actual search configuration depends on the identified factoring $\mathcal{F}$ as follows:

(i) $|\mathcal{F}| \leq 2$ (at most 1 leaf factor): Run standard search using strong stubborn sets.

(ii) $|\mathcal{F}| > 2$ (at least 2 leaf factor) and $\mathcal{F}$ is a fork factoring: Run decoupled search using decoupled strong stubborn sets, frontier dominance pruning, and leaf simulation.

(iii) $|\mathcal{F}| > 2$ (at least 2 leaf factor) and $\mathcal{F}$ is not a fork factoring: Run decoupled search without extensions.

The combination of decoupled strong stubborn sets *and* dominance pruning has not been formally described, before. We are not going into this, either, but rather give the intuition of why this still results in a complete search algorithm. Given a set of actions applicable in a state, decoupled strong stubborn sets prune the subset of these actions, that start different permutations of actions leading to the same outcome state. By guaranteeing that one of these permutations applicable in the current state will not be pruned, search using strong stubborn sets remains complete (and optimal).

In contrast to that, dominance pruning removes a state $s$ that is dominated by another decoupled state $t$. By analyzing the structure of the leaf factors and comparing only the relevant leaf states of $s$ to those of $t$, it can detect that all states that are reachable from $s$ are also (at most as costly) reachable from $t$.

Putting things together, decoupled strong stubborn sets and dominance pruning are orthogonal methods to reduce the size of the state space – one removes transitions that correspond to redundant permutations of action sequences, the other removes states that cannot reach anything that could not be reached before. Consequently, it is safe to combine both, resulting in a state space that can be significantly smaller than when only using one of the techniques.

The case distinction outlined above allows a flexible adaptation to the given input problem, and since computing the factoring in most tasks finishes within split seconds, there is (almost) no computational overhead to determine which search variant to use.

**Acknowledgments.** This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/6-1, “Star-Topology Decoupled State Space Search”.

## References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15).* AAAI Press.

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A stubborn set algorithm for optimal planning. In Raedt, L. D., ed., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, 891–892. Montpellier, France: IOS Press.

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(1–2):5–33.

Brafman, R., and Domshlak, C. 2003. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research* 18:315–349.

Gnad, D., and Hoffmann, J. 2015. Beating LM-cut with $h^{max}$ (sometimes): Fork-decoupled state space search. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15).* AAAI Press.

Gnad, D.; Hoffmann, J.; and Domshlak, C. 2015. From fork decoupling to star-topology decoupling. In Lelis, L., and Stern, R., eds., *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15).* AAAI Press.

Gnad, D.; Wehrle, M.; and Hoffmann, J. 2016. Decoupled strong stubborn sets. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16).* AAAI Press/IJCAI.

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

Jonsson, P., and Bäckström, C. 1995. Incremental planning. In *European Workshop on Planning*.

Knoblock, C. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68(2):243–302.

Torralba, Á., and Kissmann, P. 2015. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In Lelis, L., and Stern, R., eds., *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*, 122–130. AAAI Press.

Torralba, Á.; Gnad, D.; Dubbert, P.; and Hoffmann, J. 2016. On state-dominance criteria in fork-decoupled search. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16).* AAAI Press/IJCAI.

Valmari, A. 1989. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, 491–515.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12).* AAAI Press.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In Chien, S.; Do, M.; Fern, A.; and Ruml, W., eds., *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14).* AAAI Press.

# Django: Unchaining the Power of Red-Black Planning

**Daniel Gnad** and **Marcel Steinmetz** and **Jörg Hoffmann**
Saarland University
Saarbrücken, Germany
{gnad, steinmetz, hoffmann}@cs.uni-saarland.de

**Abstract**

Red-black planning is a powerful method, allowing in principle to interpolate smoothly between fully delete relaxed planning, and real (completely unrelaxed) planning. Alas, the method has been chained to the use as a heuristic function, necessitating to compute a red-black plan on every search state, entailing an exclusive focus on tractable fragments. The Django system unleashes red-black planning on the problem of proving unsolvability *within the red-black relaxation*. We introduce red-black state space search that can solve arbitary red-black planning problems, and we prove unsolvability by iteratively painting more and more red variables black.

## Introduction

Red-black planning (Katz, Hoffmann, and Domshlak 2013b) interpolates between fully delete relaxed planning, and real (completely unrelaxed) planning, by selecting a subset of state variables – the "red" ones – which take the delete-relaxed semantics, accumulating their values; while the remaining state variables – the "black" ones – retain the original value-switching semantics. If all variables are red, we have a delete relaxation, if all variables are black, we have the original planning task. In between we have a hybrid *red-black relaxation* more informed than the delete relaxation.

The method has so far been used for the design of heuristic functions (Katz, Hoffmann, and Domshlak 2013b; 2013a; Katz and Hoffmann 2013; Gnad and Hoffmann 2015b; Domshlak, Hoffmann, and Katz 2015), computing a red-black plan on every search state akin to the wide-spread relaxed plan heuristic (Hoffmann and Nebel 2001). Naturally, this entails an exclusive focus on tractable fragments of red-black plan generation. Our key observation in the Django system is that red-black relaxation can be useful also for proving unsolvability *within the relaxation*. This is promising because the red variables still carry the information "what needs to be done", while avoiding full enumeration across these variables. Consider, for example, a truck with restricted fuel having to transport some packages. If we delete-relax ("paint red") the packages, they still need to be transported, to the effect that, if there is insufficient fuel, then the red-black relaxation is unsolvable. Contrast the latter with projections, recently suggested for proving unsolvability (Bäckström, Jonsson, and Ståhlberg 2013): projecting away the packages, the task becomes trivially solvable as there is no goal anymore.

Django therefore unleashes the power of red-black planning, through *red-black state space search*, which mixes standard forward state space search with standard delete-relaxed planning methods (Hoffmann and Nebel 2001), essentially by searching over black-variable states and augmenting each state transition with a delete-relaxed planning step over the red variables. If all variables are black, this defaults to forward search. If all variables are red, it defaults to delete-relaxed planning. In between, we have a hybrid. Given this hybrid, we can prove unsolvability by fixing a variable order, and then, starting with all variables being red, painting more and more variables black until the red-black relaxation is unsolvable.

On the unsolvable benchmarks introduced by Hoffmann et al. (2014), this method excels in 3 domains and thus, overall, substantially improves the state of the art, at least when using our new better variable ordering strategy, not the old one that we had designed at IPC planner submission time. The authors are curious to see how much luck Django will have with whatever benchmarks will be used in the Unsolvability IPC 2016. But whatever happens, Django, remember: After the showers, the sun will be shining . . . [1]

## Django

### Framework

Django is implemented on top of FD (Helmert 2006) (who would have guessed!). It uses the mutex-optimized preprocessor by Alcazar and Torralba (2015) to get an optimized finite-domain variable encoding.

[1]For the reader looking for an algorithm description fitting "Django" as an acronym: there is none. We just like the movie.

Obviously we're not going to go into tremendous detail here, but let it be said that we use the *finite-domain representation (FDR)* framework, notating *planning tasks* as $\Pi = (V, A, I, G)$. $V$ is a set of finite-domain *state variables* $v$, each associated with a finite domain $D_v$. A complete assignment to $V$ is a *state*. $I$ is the *initial state*, and the *goal* $G$ is a partial assignment to $V$. $A$ is a finite set of *actions*, each $a \in A$ being a pair $(\mathsf{pre}_a, \mathsf{eff}_a)$ of the action's *precondition* $\mathsf{pre}_a$ and *effect* $\mathsf{eff}_a$, each a partial assignment to $V$.

The semantics of a planning task $\Pi$ is defined in terms of its *state space*, which is a (labeled) *transition system* $\Theta_\Pi = (S, T, s_0, S_G)$ defined in the usual manner, $S$ being the set of all states, $T$ being the transitions given by the actions $A$, $s_0$ being the initial state, and $S_G$ being the goal states. A plan is a path from $s_0$ to some state in $S_G$. We want to prove that no plan exists.

## Red-Black Planning

The delete relaxation can be captured in FDR in terms of state variables that accumulate, rather than switch between, their values. Red-black planning is the partial delete relaxation resulting from doing so only for a subset of the state variables (the "red" ones), keeping the original value-switching semantics for the others (the "black" ones) (Katz, Hoffmann, and Domshlak 2013b; Domshlak, Hoffmann, and Katz 2015).

Formally, a *red-black planning task* is a tuple $\Pi = (V^{\mathsf{B}}, V^{\mathsf{R}}, A, I, G)$. Here, $V^{\mathsf{B}}$ are the black variables, and $V^{\mathsf{R}}$ are the red ones. We require that $V^{\mathsf{B}} \cap V^{\mathsf{R}} = \emptyset$, and given the overall set of variables $V := V^{\mathsf{B}} \cup V^{\mathsf{R}}$, the remainder of the task syntax is defined exactly as before. The major change lies in the semantics. *Red-black states* $s^{\mathsf{RB}}$ assign each variable $v$ a subset $s^{\mathsf{RB}}(v) \subseteq D_v$ of its possible values. Initially, in the *red-black initial state*, the value subset contains the single value $I(v)$. If $v$ is a black variable, then action effects on $v$ overwrite $v$'s previous value, so that $s^{\mathsf{RB}}(v)$ always contains exactly one element; if $v$ is a red variable, then action effects on $v$ are accumulated into the previous value subset. A *red-black goal state* is one where, for every goal variable $v$, $G(v) \in s^{\mathsf{RB}}(v)$.

Given an FDR task $\Pi = (V, A, I, G)$, a *painting* is a partition of the variables $V$ into two subsets, $V^{\mathsf{B}}$ and $V^{\mathsf{R}}$. Given a painting, a plan for the red-black planning task $(V^{\mathsf{B}}, V^{\mathsf{R}}, A, I, G)$ is called a red-black plan for $\Pi$.

## Red-Black State Space Search

Red-black planning generalizes both, delete-relaxed planning and real planning, so in particular deciding red-black plan existence is, in general, **PSPACE**-hard. To solve arbitrary red-black planning problems, we need a search algorithm: red-black state space search.

Essentially, the search branches only over those actions affecting black variables, while handling the other actions through red forward fixed points associated with individual state transitions. To keep this paper crisp, we give an outline only, and we refer the masochistic and/or interested reader to our SOCS'16 paper for the details (Gnad et al. 2015).

Like typical relaxed planning algorithms, red-black state space search consists of a forward phase, followed by a backward phase. The forward phase chains forward until reaching the goal ("state space search with a relaxed planning graph at each transition"), and the backward phase extracts a red-black plan ("extracting the solution path with a relaxed plan extraction step at each transition").

It is cumbersome to spell this out formally. But it should be possible to get an intuition across. Without actually introducing the notations, consider this (slightly simplified) definition from our SOCS'16 paper:

**Definition 1 (RB State Space)** *Let* $\Pi = (V^{\mathsf{B}}, V^{\mathsf{R}}, A, I, G)$ *be an RB planning task. The* red-black state space *of* $\Pi$*, denoted* $\Theta^{\mathsf{RB}}_{\Pi}$*, is the transition system* $\Theta^{\mathsf{RB}} = (S^{\mathsf{RB}}, T^{\mathsf{RB}}, s^{\mathsf{RB}}_0, S^{\mathsf{RB}}_G)$ *where:*

*(i)* $S^{\mathsf{RB}}$ *is the set of all red-black states.*

*(ii)* $s^{\mathsf{RB}}_0$ *is the red-black initial state.*

*(iii)* $S^{\mathsf{RB}}_G$ *contains the red-black states* $s^{\mathsf{RB}}$ *where* $\mathcal{F}^+(s^{\mathsf{RB}})$ *is a red-black goal state.*

*(iv)* $T^{\mathsf{RB}}$ *is the set of transitions* $s^{\mathsf{RB}} \xrightarrow{a} t^{\mathsf{RB}}$ *where a affects at least one black variable, a is applicable to* $\mathcal{F}^+(s^{\mathsf{RB}})$*, and* $t^{\mathsf{RB}} = \mathsf{outcomeState}(\mathcal{F}^+(s^{\mathsf{RB}}), a)$*.*

The notation "$\mathcal{F}^+(s^{\mathsf{RB}})$" denotes the extension of the red-black state $s^{\mathsf{RB}}$ with all those values of red variables that can be reached from $s^{\mathsf{RB}}$ by applying actions with red effects only. In other words, $\mathcal{F}^+(s^{\mathsf{RB}})$ adds, into the value subsets $s^{\mathsf{RB}}(v)$ of the red variables $v$, the red-planning fixed point ("delete-relaxed fixed-point layer in a relaxed planning graph"), when considering only those red-effect actions whose black preconditions are satisfied in $s^{\mathsf{RB}}$.

Given this, item (iii) just says that we can stop at $s^{\mathsf{RB}}$ if its red fixed point contains the goal. Item (iv) says that, to transition from one red-black state $s^{\mathsf{RB}}$ to another $t^{\mathsf{RB}}$ via action $a$, we first execute the red fixed point on $s^{\mathsf{RB}}$, to obtain $\mathcal{F}^+(s^{\mathsf{RB}})$; then we check whether $a$ is applicable to $\mathcal{F}^+(s^{\mathsf{RB}})$; and if so, we simply apply $a$ to that fixed point, treating $\mathcal{F}^+(s^{\mathsf{RB}})$ like any other red-black state.

Say now that the forward phase has found a path to the goal, i. e., a path $\pi = \langle s^{\mathsf{RB}}_0, a_0, s^{\mathsf{RB}}_1, \dots, a_{n-1}, s^{\mathsf{RB}}_n \rangle$ in $\Theta^{\mathsf{RB}}_{\Pi}$, where $s^{\mathsf{RB}}_n \in S^{\mathsf{RB}}_G$. In standard state space search, we would simply return the actions $a_0, \dots, a_{n-1}$ labeling the path. But in our case here, that would account only for the black-affecting actions. To collect the red-affecting actions, at each transition $s^{\mathsf{RB}}_i \xrightarrow{a} s^{\mathsf{RB}}_{i+1}$ along $\pi$, we need to extract a red plan supporting the subgoals needed at time $i+1$, propagating new needed subgoals to time $i$. The subgoals needed at time $n$ are simply the red goals; each red-plan extraction step is a standard relaxed plan extraction step on the red fixed point leading from $i$ to $i+1$; once we reach time 0, we can schedule all the red plans along $a_0, \dots, a_{n-1}$ and have a red-black plan.

The reader might have noticed that the author just got carried away – this paper, competition, and planning system being exclusively about proving unsolvability, we will never actually get to the backward red-black plan extraction phase, or if we do, then we know that the relaxation is not informed enough and we need to paint more variables black. Apologies for the inconvenience; then again, the backward phase is part of red-black state space search, and that search also has other possible uses (cf. our SOCS'16 paper), so the author is right now choosing to just leave this in.

In any case, coming back to what does matter for our purpose here: it is easy to see that, if the goal cannot be reached in $\Theta_\Pi^{\mathsf{RB}}$, then $\Pi$ is unsolvable. This is simply because red-black relaxation preserves plans, and goal reachability in $\Theta_\Pi^{\mathsf{RB}}$ is equivalent to red-black plan existence.

## Wrapping it Up with a Variable Ordering Strategy

To turn the above into an actual automatic planner, we need to decide how to actually paint the variables – which ones are to be red, which ones are to be black?

Previous work designed such *painting strategies* for the purpose of heuristic functions. For the purpose of proving unsolvability, matters are different in that it makes a lot of sense to merely *try* a painting, and, if it does not succeed, try another one. The simplest possible way to do this – or at least these authors could not think of a simpler one – is to start with all variables being red, then iteratively check whether there is a red-black plan; if no, stop (unsolvability proved); else, pick a red variable $v$, paint it black, and iterate. The question then just remains how to pick the next variable.

At the time of planner submission, the authors simplified even this simple question, fixing a variable order a priori, not taking into account any new information found during the process. Specifically, we used a variable ordering strategy that we denote as *RBb*, the "b" standing for breadth-first (we leave it to the reader's imagination what the "RB" may be for). The strategy builds the DAG of strongly connected components (SCC) of the input task's causal graph, and processes these (i. e., orders the variables) in a breadth-first manner, from root SCCs to leaf SCCs.

We later on realized that it is actually a good idea to take information found during the process into account, specifically *conflicts* in the red-black plan found in the previous iteration. The notion of conflicts is inspired by painting strategies underlying heuristic functions (Domshlak, Hoffmann, and Katz 2015). Given a red variable $v$, a conflict on $v$ is an action in the red-black plan whose precondition on $v$ would not be satisfied when painting $v$ black. The idea is to select, as the next red $v$ to be painted black, one with a maximal number of conflicts. We denote this by *RBc*, and we denote by *RBbc* the strategy that applies RBb and breaks ties, for inclusion of the next variable within an SCC, by the maximal number of conflicts.

And this is all there is to say about Django ...

... except, catering for the unlikely case where Django does not work on the benchmarks wisely chosen by the IPC organizers, let us show off a little bit with our results on the previous benchmarks by Hoffmann et al. (2014):

## Own Experiments

Table 1 shows coverage data, i. e., the number of instances proved unsolvable. We compare against a selection of approaches from Hoffmann et al.'s (2014) extensive experiments, namely blind search ("Bli") and search with $h^{\max}$ as canonical simple methods; exhaustive testing of small projections ("SP") as per Bäckström et al. (2013) to compare against this recently proposed method; constrained BDDs (Torralba and Alcázar 2013) ("BDD") as a competitive symbolic method (named "BDD $H^2$" in (Hoffmann, Kissmann, and Torralba 2014)); as well as the two most competitive variants of merge-and-shrink by Hoffmann et al., namely their "Own+A $H^2$" (here: "MS1") and their "Own+K N100k M100k $h^{\max}$" (here: "MS2"). This selection of planners represents the state of the art – we should really say: represent*ed* the state of the art at planner submission time – in proving unsolvability in planning.

| Domain | # | Bli | $h^{\max}$ | SP | BDD | MS1 | MS2 | RBb | RBc | RBbc | BP | DS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bottleneck | 25 | 10 | 21 | 10 | 15 | 10 | 21 | 12 | **25** | **25** | 5 | 0 |
| 3UNSAT | 30 | **15** | **15** | 0 | **15** | **15** | **15** | **15** | **15** | **15** | 5 | 0 |
| Mystery | 9 | 2 | 2 | 6 | **9** | **9** | 6 | 7 | 2 | 2 | 0 | 0 |
| NoMystery | 25 | 0 | 0 | 8 | 14 | **25** | **25** | 24 | 24 | 24 | 14 | 24 |
| PegSol | 24 | **24** | **24** | 0 | **24** | **24** | **24** | 12 | 22 | 22 | 8 | 0 |
| Rovers | 25 | 0 | 1 | 3 | 10 | 17 | 9 | **25** | 11 | **25** | 0 | 0 |
| Tiles | 20 | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | **10** | 0 |
| TPP | 25 | 5 | 5 | 2 | 1 | **9** | **9** | 2 | 1 | 1 | 0 | 0 |
| $\sum$ | 183 | 66 | 78 | 39 | 98 | 119 | 119 | 107 | 110 | **124** | 42 | 24 |

Table 1: Number of instances proved unsolvable. Best values highlighted in **boldface**. Left part: state of the art as per Hoffmann et al. (2014). Middle part: red-black state space search. Right part: particular comparisons. Explanations and abbreviations see text.

Our best configuration, RBbc, beats the state of the art in overall coverage. It excels in Bottleneck and Rovers, where red-black state space search is the only method able to solve all instances. In NoMystery, together with merge-and-shrink and DS (regarding which: see below), it performs way better than all other planners. In the remaining domains, the performance of red-black state space search is not as remarkable, about in the mid-range in Mystery, PegSol, and TPP, and on par with other planners in 3UNSAT and Tiles where no planner seems to manage to do something interesting.

The "BP" and "DS" columns stand for *black-projection*, respectively *decoupled search* (Gnad and Hoffmann 2015a; 2015b). BP is like our incremental RBbc method but considering the black variables only. It follows RBbc's variable ordering, until RBbc terminates; if the projection onto the black variables is at this point still solvable, then BP continues with the RBb variable ordering. BP has not been previously explored, and is included here to show the benefit of considering red variables in addition to the black ones. The data clearly attests to that benefit.

DS identifies a partition of the variables inducing a "star topology", then searches only over the "center" component of the star, enumerating the possible moves for each "leaf" component separately. We include it here because, like red-black state space search, it can avoid the enumeration across packages in NoMystery (each package is a leaf component). DS is, however, limited to tasks with a useful star topology that can be identified with the current variable partitioning methods. The latter is rare on this benchmark set, and the data clearly shows the benefit of not having that limitation.[2]

Consider finally Figure 1, a direct comparison between red-black state space search and black-projection, as the set

[2]Remark by the author: Isn't it great how one can bash one's own work in one's own papers?

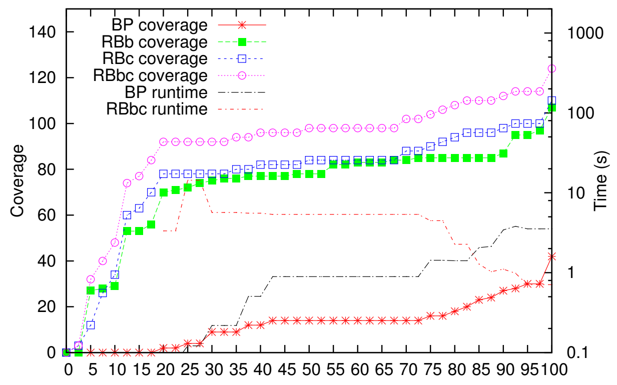

Figure 1: Coverage and average runtime of red-black state space search, compared to black-projection, as a function of the fraction of black variables. Explanations see text.

of black variables $V^B$ grows. This provides an in-depth view of the advantages of taking into account the remaining variables $V \setminus V^B$ as red ones, rather than ignoring them completely. The coverage advantage is dramatic, as red-black state space search can make do with *much* smaller sets $V^B$. The runtime averages are taken over the commonly solved instances for each value of $x$. We see that, as expected, red-black state space search incurs a substantial overhead for those tasks tackled also by projection with small $V^B$. Yet as the $V^B$ required in projection grows larger, that disadvantage becomes smaller and finally disappears completely.

## Conclusion

Django is unchained! Red-black planning has finally escaped the cage of computational tractability! What more is there to say?

Well, let us say that this is the beginning, not the end, of the story (oops I almost said "movie" here). Django can still be improved in a gazillion ways, including but not limited to: better variables ordering strategies; re-using state space information (e. g. dead-end regions) from previous iterations; adaptive paintings choosing red/black variables depending on state; etc. It should also be noted that Django is not doomed to just prove unsolvability – if the red-black plan in some iteration happens to be a real plan, then we can also stop. The question then is how to fruitfully interleave both purposes, choosing the next black variable, perhaps, based on the current hypothesis whether the task will turn out to be solvable or unsolvable.

Figure 2: THE END.

**Acknowledgments.** This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation".

## References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press.

Bäckström, C.; Jonsson, P.; and Ståhlberg, S. 2013. Fast detection of unsolvable planning instances using local consistency. In Helmert, M., and Röger, G., eds., *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, 29–37. AAAI Press.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence* 221:73–114.

Gnad, D., and Hoffmann, J. 2015a. Beating LM-cut with $h^{max}$ (sometimes): Fork-decoupled state space search. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press.

Gnad, D., and Hoffmann, J. 2015b. Red-black planning: A new tractability analysis and heuristic function. In Lelis, L., and Stern, R., eds., *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*. AAAI Press.

Gnad, D.; Steinmetz, M.; Jany, M.; Hoffmann, J.; Serina, I.; and Gerevini, A. 2015. Partial delete relaxation, unchained: On intractable red-black planning and its applications. In Baier, J., and Botea, A., eds., *Proceedings of the 9th Annual Symposium on Combinatorial Search (SOCS'16)*. AAAI Press.

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

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

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Schaub, T., ed., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*. Prague, Czech Republic: IOS Press.

Katz, M., and Hoffmann, J. 2013. Red-black relaxed plan heuristics reloaded. In Helmert, M., and Röger, G., eds., *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, 105–113. AAAI Press.

Katz, M.; Hoffmann, J.; and Domshlak, C. 2013a. Red-black relaxed plan heuristics. In desJardins, M., and Littman, M., eds., *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI'13)*, 489–495. Bellevue, WA, USA: AAAI Press.

Katz, M.; Hoffmann, J.; and Domshlak, C. 2013b. Who said we need to relax *all* variables? In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 126–134. Rome, Italy: AAAI Press.

Torralba, A., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In Helmert, M., and Röger, G., eds., *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, 175–183. AAAI Press.

# CLone: A Critical-Path Driven Clause Learner

**Marcel Steinmetz** and **Jörg Hoffmann**
Saarland University
Saarbrücken, Germany
{steinmetz,hoffmann}@cs.uni-saarland.de

**Abstract**

In this paper, we introduce a planner that, similar to CDCL (conflict-driven clause learning) SAT solvers, learns from making wrong decisions in a way guaranteeing to preclude these mistakes in the future. The planner we describe, named CLone, conducts a depth-first forward search in the state space of the problem. To identify dead ends early on, and thus to reduce search effort, we make use of the critical path heuristic $h^C$. $h^C$ is defined relative to a given set $C$ of fact conjunctions. During search, we identify unrecognized dead ends, i. e., dead ends $s$ with $h^C(s) < \infty$, and we use them to update $C$ in order to recognize $s$, and possibly also dead ends similar to $s$. As the evaluation of $h^C$ is getting computationally more expensive the more conjunctions are added to $C$, we further maintain and continuously update a set of clauses $\Delta$ with the property that if $s \not\models \phi$ for some $\phi \in \Delta$ then $h^C(s) = \infty$. Although these clauses are subsumed by $h^C$ by definition, they have a tremendous impact in practice. For a more detailed explanation of this approach, we refer the reader to (Steinmetz and Hoffmann 2016).

## Introduction

CLone determines the (un-)solvability of a problem by running a depth-first like search in the state space. When only running search in the state space (i. e. without using additional state reduction techniques), one has to construct the entire state space to show the unsolvability of a problem. To reduce the number of states that actually have to be touched by the search, and thus to avoid building the whole state space, we rely on heuristic functions $h$, i. e., estimations of the distance from a given state to the goal. Contrary to finding solutions to solvable problems, however, we are not interested in the actual goal distance estimations given by the heuristics: if a problem is unsolvable, then the search has to explore all states $s$ with $h(s) < \infty$ – where $h(s) = \infty$ means that $s$ is even unsolvable in the relaxation underlying $h$. This led to the definition of *unsolvability heuristics*, heuristics that either return $\infty$ ("dead-end") or 0 ("don't know") (Bäckström *et al.* 2013; Hoffmann *et al.* 2014). Concrete unsolvability heuristics have been designed based on state-space abstractions, specifically projections (pattern databases (Edelkamp 2001)) and merge-and-shrink abstractions (Helmert *et al.* 2014). The empirical results are impressive, especially for merge-and-shrink which convincingly beats state-of-the-art BDD-based planning techniques (Torralba and Alcázar 2013) on a suite of unsolvable benchmark tasks.

Critical-path heuristics lower-bound goal distance through the relaxing assumption that, to achieve a conjunctive subgoal $G$, it suffices to achieve the most costly *atomic* conjunction contained in $G$. In the original critical-path heuristics $h^m$ (Haslum and Geffner 2000), the atomic conjunctions are all conjunctions of size $\leq m$, where $m$ is a parameter. As part of recent works (Haslum 2009; 2012; Keyder *et al.* 2014), this was extended to arbitrary sets $C$ of atomic conjunctions. Following Hoffmann and Fickert (2015), we denote the generalized heuristic with $h^C$. A well-known and simple result is that, for sufficiently large $m$, $h^m$ delivers perfect goal distance estimates. As a corollary, *for appropriately chosen $C$, $h^C$ recognizes all dead-ends*. Our idea thus is to refine $C$ during search, based on the dead-ends encountered.

We start with a simple initialization of $C$, to the set of singleton conjunctions. During search, components $\hat{S}$ of unrecognized dead-ends, where $h^C(s) < \infty$ for all $s \in \hat{S}$, are identified (become *known*) when all their descendants have been explored. We *refine* $h^C$ on such components $\hat{S}$, adding new conjunctions into $C$ in a manner guaranteeing that, after the refinement, $h^C(s) = \infty$ for all $s \in \hat{S}$. The refined $h^C$ has the power to *generalize* to other dead-ends search may encounter in the future, i. e., refining $h^C$ on $\hat{S}$ may lead to recognizing also other dead-end states $s' \notin \hat{S}$. It is known that computing critical-path heuristics over large sets $C$ is (polynomial-time yet) computationally expensive. Computing $h^C$ on all search states often results in prohibitive runtime overhead. We tackle this with a form of *clause learning*. For a dead-end state $s$ where $h^C$ evaluates to $\infty$, we extract a clause $\phi$ that guarantees for all states $s'$ with $s' \not\models \phi$ that $h^C(s') = \infty$. When testing whether a new state $s'$ is a dead-end, we first evaluate the clauses $\phi$, and invoke the computation of $h^C(s')$ only in case $s'$ satisfies all clauses $\phi \in \Delta$.

The resulting algorithm approaches the elegance of clause learning in SAT (e. g. (Marques-Silva and Sakallah 1999; Moskewicz *et al.* 2001; Eén and Sörensson 2003)): When a subtree is fully explored, the $h^C$-refinement and clause learning (1) learns to refute that subtree, (2) enables backjumping to the shallowest non-refuted ancestor, and (3) generalizes to other similar search branches in the future.

For full details on the techniques used in this planner, we refer the reader to (Steinmetz and Hoffmann 2016).

## Background

We consider planning tasks $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ in STRIPS encoding. $\mathcal{F}$ gives a set of **facts**; $\mathcal{A}$ a set of **actions**; $\mathcal{I} \subseteq \mathcal{F}$ is the **initial state**; and $\mathcal{G} \subseteq \mathcal{F}$ the **goal**. Each $a \in \mathcal{A}$ has a **precondition** $pre(a) \subseteq \mathcal{F}$, an **add list** $add(a) \subseteq \mathcal{F}$, and a **delete list** $del(a) \subseteq \mathcal{F}$. Action costs are irrelevant with respect to the solvability of planning tasks, so we assume unit cost throughout. In action preconditions and the goal, the fact set is interpreted as a conjunction; we will use the same convention for the conjunctions in the set $C$, i. e., the $c \in C$ are fact sets $c \subseteq \mathcal{F}$. A **state** $s$, in particular the initial state $I$, is a set of facts, namely those true in $s$ (the other facts are assumed to be false). There is a transition from state $s$ to $s[[a]]$ via action $a$ if $a$ is **applicable** to $s$, i. e., $pre(a) \subseteq s$, and $s[[a]] := (s \setminus del(a)) \cup add(a)$. **Goal states** are all states $s$ where $G \subseteq s$. A **dead-end** is a state for which no path to a goal state exists. Viewing the state space of $\Pi$, denoted $\Theta^{\Pi}$, as a directed graph over states, given a subset $\mathcal{S}'$ of states, by $\Theta^{\Pi}|_{\mathcal{S}'}$ we denote the subgraph induced by $\mathcal{S}'$. If there is a path in $\Theta^{\Pi}|_{\mathcal{S}'}$ from $s$ to $t$, then we say that $t$ is **reachable** from $s$ in $\Theta^{\Pi}|_{\mathcal{S}'}$.

A **heuristic** is a function $h$ mapping states to natural numbers or $\infty$. The family of **critical-path heuristics**, which underly Graphplan (Blum and Furst 1997) and were formally introduced by Haslum and Geffner (2000), estimate goal distance through the relaxation assuming that, from any goal set of facts, it suffices to achieve the most costly subgoal (sub-conjunction). The family is parameterized by the set of atomic subgoals considered. Formally, for a fact set $G$ and action $a$, define the **regression** of $G$ over $a$ as $R(G, a) := (G \setminus add(a)) \cup pre(a)$ in case that $add(a) \cap G \neq \emptyset$ and $del(a) \cap G = \emptyset$; otherwise, the regression is undefined and we write $R(G, a) = \bot$. By $\mathcal{A}[G]$ we denote the set of actions where $R(G, a) \neq \bot$. Let $C$ be any set of conjunctions. The generalized critical-path heuristic (Hoffmann and Fickert 2015) $h^C(s)$ is defined through $h^C(s) := h^C(s, \mathcal{G})$ where

$$h^C(s, G) = \begin{cases} 0 & G \subseteq s \\ 1 + \min_{a \in \mathcal{A}[G]} h^C(s, R(G, a)) & G \in C \\ \max_{G' \subseteq G, G' \in C} h^C(s, G') & \text{else} \end{cases} \quad (1)$$

A well known property of critical path heuristics is that they are *admissible*, i. e., that they always underestimate the real goal distance. In other words, if $h^C(s) = \infty$, then, as desired, $s$ is indeed a dead end, and $s$ can be disregarded in search without loosing completeness. Note that $h^C(s) = \infty$ occurs (only) due to empty minimization in the middle case of Equation 1, i. e., if every possibility to achieve the global goal $\mathcal{G}$ incurs at least one atomic subgoal not supported by any action.

Similarly as for $h^m$, $h^C$ can be computed in time polynomial in $|C|$ and the size of $\Pi$. It is known that, in practice, $h^m$ is reasonably fast to compute for $m = 1$, consumes substantial runtime for $m = 2$, and is mostly infeasible for $m = 3$. The behavior is similar when using arbitrary conjunction sets $C$, in the sense that large $C$ causes similar issues as $h^m$ for $m > 1$. As hinted, we will use a clause-learning technique to alleviate this.

**Algorithm 1:** CLone

```
1  Procedure DFS(Π)
2      $Open$ := empty stack; push $I$ to $Open$;
3      $Closed := \emptyset$;
4      while $Open$ is not empty do
5          $s \leftarrow Open.top()$;
6          $Open.pop()$;
7          if $s \in Closed$ then
8              continue;
9          if $h^C(s) = \infty$ then
10             Backtrack($s$);
11             continue;
12         if $\mathcal{G} \subseteq s$ then
13             return solvable;
14         $Closed := Closed \cup \{s\}$;
15         for all $a \in \mathcal{A}$ applicable to $s$ do
16             push $s[[a]]$ to $Open$;
17         CheckAndLearn($s$);
18     return unsolvable;
19 Procedure CheckAndLearn($s$)
20     $\mathcal{R}[s] := \{t \mid t$ reachable from $s$ in $\Theta^{\Pi}|_{Open \cup Closed}\}$;
21     if $\mathcal{R}[s] \subseteq Closed$ then
22         refine $C$ s.t. $h^C(t) = \infty$ for every $t \in \mathcal{R}[s]$;
23         Backtrack($s$);
24 Procedure Backtrack($s$)
25     label $s$;
26     for every unlabeled parent $t$ of $s$ do
27         CheckAndLearn($t$);
```

## Search, Fail, Refine & Repeat

CLone runs search in the state space of the problem using $h^C$ as a dead end identifier. During search, CLone keeps track of expanded states to identify yet unrecognized dead ends. Whenever an unrecognized dead end $s$ is found, the set $C$ is extended by new atomic conjunction, guaranteeing that $h^C(s) = \infty$ after the refinement. In order to avoid as many of the rather expensive computations of $h^C$ as possible, CLone learns clauses as sufficient conditions to $h^C(s) = \infty$ each time a state is found where $h^C(s)$ has been evaluated to $\infty$. The clauses are used to filter states before $h^C$ is evaluated.

### Identifying Failures in Search

Consider Algorithm 1. At the heart of CLone, it performs a depth-first forward search in the state space of the problem, while maintaining a closed list for duplicate checking. $h^C$ is used as an *efficient* method to identify dead ends, eliminating necessity of exploring any of the state's successors.

To identify also dead ends not (yet) recognized by $h^C$, CLone analyzes the search space after each state expansion. The corresponding code, function CheckAndLearn in Algorithm 1, performs a full lookahead search in the current search space ($\Theta^{\Pi}|_{Open \cup Closed}$), looking for states that have not been expanded so far. Intuitively, a state $s$ is a **known dead-end** if the search has already proved that $s$ is a dead end, meaning that all states $t$ reachable from $s$ have already been explored and no such state $t$ is a goal state, i. e., $\mathcal{R}[s] \subseteq Closed$. It is easy to see that the concept of "known dead-end" does capture exactly our intentions:

**Proposition 1.** *Let $s$ be a known dead-end during the execution of Algorithm 1. Then $s$ is a dead-end.*

Vice versa, if $\mathcal{R}[s] \not\subseteq Closed$, then some descendants of $s$ have not yet been explored, so the search does not know whether or not $s$ is a dead-end.

Once a known dead-end $s$ is found, $C$ is refined in a way guaranteeing that $h^C(s) = \infty$ afterwards. As we know already that $s$ is a dead end, forcing that $h^C(s) = \infty$ seems to be redundant. However, the reason of the refinement of $C$ is not actually to have $h^C$ recognize $s$ as dead end, but rather the hope that dead ends similar to $s$ will be recognized as well due to this very refinement.

To guarantee that all known dead-ends are found, and thus to learn as much as possible, $\mathcal{R}[t] \subseteq Closed$ has to be checked for each state $t \in Closed$ after every state expansion. Naively checking this property for each state $t \in Closed$ after every expansions is clearly infeasible. Instead, CLone uses the observation that the property $\mathcal{R}[t] \subseteq Closed$ can only change for ancestors $t$ of the state $s$ that was expanded last. To find all these states, CLone checks this condition on the parents of $s$, and recursively continues on those parents satisfying the condition.

Although CLone could in principle also run any other $Closed$-list based search algorithm, the key advantage of DFS in our setting is that it focuses on completely exploring subtrees, and hence it is able to identify unrecognized dead-ends quickly.

## Failure Analysis & $h^C$ Refinement

Once identified a known though unrecognized dead end $s$, we have to find conjunctions $X$ that, when added to $C$, guarantee that $h^{C \cup X}(s) = \infty$. To find $X$, CLone makes use of the specific context in which $C$ is going to be refined. Observe that whenever `CheckAndLearn`$(s)$ calls the refinement of $C$, it holds: *(*) For every transition $t \rightarrow t'$ where $t \in \mathcal{R}[s]$, either $t' \in \mathcal{R}[s]$ or $u^C(t') = \infty$.* We will refer to this by the **recognized neighbors** property. This is because $\mathcal{R}[s]$ contains only closed states, so it contains all states $t$ reachable from $s$ except for those where $h^C(t) = \infty$.

Algorithm 2 shows the overall refinement process. We use $\hat{S} := \mathcal{R}[s]$ to denote the component on which $C$ is refined, and $\hat{T} := \{t' \mid t \xrightarrow{a} t', t \in \hat{S}, t' \notin \hat{S}\}$ to denote recognized neighbors. As shown at the top of Algorithm 2, CLone computes $X$ by recursively adding an unreachable subgoal $x \subseteq R(G, a)$ for each $a \in \mathcal{A}[G]$ to $X$, corresponding to the middle case of Equation 1, and then continuing on $G = x$ until either $h^C(s, G)$ is already $\infty$ for every $s \in \hat{S}$, or $a[G] = \emptyset$. Note that determining whether some $x \subseteq R(G, a)$ is unreachable from a state $s$ is PSPACE-complete in general. To still find such an $x$ efficiently, CLone uses the recognized neighbors property:

**Algorithm 2:** Refining $C$ for $\hat{S}$ with recognized neighbors $\hat{T}$. $C$ and $X$ are global variables.

```
Procedure Refine($G$)
    $x :=$ ExtractX($G$);
    $X := X \cup \{x\}$;
    for $a \in \mathcal{A}[x]$ where ex. $s \in \hat{S}$ s.t. $h^C(s, R(x, a)) < \infty$ do
        if there is no $x' \in X$ s.t. $x' \subseteq R(x, a)$ then
            Refine($R(x, a)$);
Procedure ExtractX($G$)
    $x := \emptyset$;
    /* Lemma 2 (ii)                                  */
    while $\exists t \in \hat{T}$ so that $h^C(t, x) < \infty$ do
        $c_0 := \emptyset$; $n_0 := 0$;
        for each $c \in C$ where $c \subseteq G$ do
            $n := |\{t \in \hat{T} \mid h^C(t, x) < \infty, h^C(t, c) = \infty\}|$;
            if $n \geq n_0$ or ($n = n_0$ and $|c \setminus x| < |c_0 \setminus x|$) then
                $c_0 := c$; $n_0 := n$;
        $x := x \cup c_0$;
    /* Lemma 2 (i)                                   */
    for every $s \in \hat{S}$ do
        if $x \subseteq s$ then
            select $p \in G \setminus s$; $x := x \cup \{p\}$;
    return $x$;
```

**Lemma 2** (Steinmetz and Hoffmann 2016)**.** *If $x \subseteq G$ satisfies*

*(i) for every $s \in \hat{S}$, $x \not\subseteq s$; and*

*(ii) for every $t \in \hat{T}$, there exists $c \in C$ such that $c \subseteq x$ and $h^C(t, c) = \infty$;*

*then $x$ is unreachable from every $s \in \hat{S}$.*

To ensure (ii) of Lemma 2 in the computation of $x$, CLone's procedure `ExtractX`$(G)$ tries to greedily construct a minimal conjunction that covers all recognized neighbors. It does so by merging an atomic conjunction $c_0 \in C, c_0 \subseteq G$ into $x$ that covers as many recognized neighbors as possible, while being minimal in size, as long as the condition (ii) is not satisfied. If the resulting $x$ is not contained in any $s \in \hat{S}$ then we are done, otherwise for each affected $s$ we add a fact $p \in G \setminus s$ into $x$, to ensure Lemma 2 (i). Putting things together, we get the desired result:

**Theorem 2** (Steinmetz and Hoffmann 2016)**.** *Algorithm 2 is correct:*

*(i) The execution is well defined, i. e., it is always possible to extract a conflict $x$ as specified.*

*(ii) The algorithm terminates.*

*(iii) Upon termination, $h^{C \cup X}(s) = \infty$ for every $s \in \hat{S}$.*

## Clause Learning

As pointed out, the clauses we learn do not have the same pruning power as $h^C$. Yet they have a dramatic runtime advantage, which is key to applying learning and pruning liberally. We always evaluate the clauses prior to evaluating $h^C$, and we learn a new clause every time $h^C$ is evaluated and returns $\infty$.

Different from the clause learning approach presented in our prior work (Steinmetz and Hoffmann 2016), CLone computes the clauses directly from the structure underlying $h^C$. Say $h^C$ has been evaluated on $s$ to $\infty$. CLone constructs a clause $\phi$ so that $s' \not\models \phi$ implies $h^C(s') = \infty$ following Equation 1. In detail, $\phi$ is set to a disjunction of atomic conjunctions so that (1) for each atomic conjunction $c$ taking part in $\phi$: $h^C(s, c) = \infty$; (2) for each $c \in \phi$ and for each $a \in \mathcal{A}[c]$, there must be some $c' \in \phi$ with $c' \subseteq R(c, a)$ (cf. middle case of Equation 1); and (3) there must be an atomic conjunction $c \in \phi$ so that $c \subseteq \mathcal{G}$ (cf. last case of Equation 1). In this way, CLone ensures that $\phi$ is self contained, i. e., that for each atomic conjunction $c \in \phi$, $\phi$ covers all possible ways of achieving $c$. In other words, if a state $s'$ does not satisfy $\phi$, i. e., $c \not\subseteq s'$ for every $c \in \phi$, then we obtain $h^C(s') = \infty$ as a direct consequence.

## Implementation

CLone is implemented on top of Fast Downward (Helmert 2006), extended by the $h^2$ preprocessor (Alcázar and Torralba 2015). For $h^C$, following Hoffmann and Fickert (2015), we use counters over pairs $(c, a)$ where $c \in C$, $a \in \mathcal{A}[c]$, and $R(c, a)$ does not contain a fact mutex. The depth-first search of CLone breaks ties (order of children) randomly .

Given a problem in form of a PDDL domain and a PDDL problem file, Fast Downward first compiles these files into an FDR planning task. All methods (and in particular the search), except of the computation and refinement of $h^C$, operate directly on this FDR encoding. Only the computation and refinement of $h^C$ pretend to have a STRIPS encoding of the problem by considering variable value pairs as facts, and threating the actions accordingly.

CLone initializes $C$ to the set of all unit conjunctions, i. e., $C = \{\{p\} \mid p \in \mathcal{F}\}$. Additionally, if the causal graph of the FDR task contains more than one maximal SCC, CLone adds the conjunctions of facts to $C$ corresponding to the variable value assignments of all pairs of variables that are part of the root SCC. The intuition behind this is that the root component of a problem's causal graph is usually a central part of the problem structure, and having the pairs of the values of the corresponding variables often helps the refinement algorithm to find smaller sets $X$.

**Acknowledgments.** This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/5-1 "Critically Constrained Planning via Partial Delete Relaxation".

## References

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. *ICAPS'15*.

Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. *SOCS'13*, 29–37.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.

Stefan Edelkamp. Planning with pattern databases. *ECP'01*, 13–24.

Niklas Eén and Niklas Sörensson. An extensible sat-solver. *SAT'03*, 502–518.

Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. *AIPS'00*, 140–149.

Patrik Haslum. $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to $h^m$. *ICAPS'09*, 354–357.

Patrik Haslum. Incremental lower bounds for additive cost planning problems. *ICAPS'12*, 74–82.

Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*, 61(3), 2014.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Jörg Hoffmann and Maximilian Fickert. Explicit conjunctions w/o compilation: Computing $h^{FF}(\pi^c)$ in polynomial time. *ICAPS'15*.

Jörg Hoffmann, Peter Kissmann, and Álvaro Torralba. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. *ECAI'14*.

Emil Keyder, Jörg Hoffmann, and Patrik Haslum. Improving delete relaxation heuristics through explicitly represented conjunctions. *Journal of Artificial Intelligence Research*, 50:487–533, 2014.

Joao Marques-Silva and Karem Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC-01*.

Marcel Steinmetz and Jörg Hoffmann. Towards clause-learning state space search: Learning to recognize dead-ends. *AAAI'16*.

Álvaro Torralba and Vidal Alcázar. Constrained symbolic search: On mutexes, BDD minimization and more. *SOCS'13*, 175–183.

# Fast Downward Aidos

**Jendrik Seipp** and **Florian Pommerening** and **Silvan Sievers** and **Martin Wehrle**
University of Basel
Basel, Switzerland
{jendrik.seipp,florian.pommerening,silvan.sievers,martin.wehrle}@unibas.ch

**Chris Fawcett**
University of British Columbia
Vancouver, Canada
fawcettc@cs.ubc.ca

**Yusra Alkhazraji**
University of Freiburg
Freiburg, Germany
alkhazry@informatik.uni-freiburg.de

This paper describes the three Fast Downward Aidos portfolios we submitted to the Unsolvability International Planning Competition 2016. All three Aidos variants are implemented in the Fast Downward planning system (Helmert 2006). We use a pool of techniques as a basis for our portfolios, including various techniques already implemented Fast Downward, as well as three newly developed techniques to prove unsolvability.

We used automatic algorithm configuration to find a good Fast Downward configuration for each of a set of test domains and used the resulting data to select the components, their order and their time slices for our three portfolios.

For Aidos 1 and 2 we made this selection manually, resulting in two portfolios comprised mostly of the three new techniques. Aidos 1 distributes the 30 minutes based on our experiments, while Aidos 2 distributes the time uniformly.

Aidos 3 contains unmodified configurations from the tuning process with time slices automatically optimized for the number of solved instances per time. It is based both on the new and existing Fast Downward components.

The remainder of this planner abstract is organized as follows. First, we describe the three newly developed techniques. Second, we list the previously existing components of Fast Downward that we have used for configuration. Third, we describe the benchmarks used for training and test sets. Fourth, we describe the algorithm configuration process in more detail. Finally, we briefly describe the resulting portfolios.

## Dead-End Pattern Database

A dead-end pattern database (PDB) stores a set of partial states that are reachable in some abstraction, and for which no plan exists in the abstraction. Every state $s$ encountered during the search can be checked against the dead-end PDB: if $s$ is consistent with any of the stored partial states, then $s$ can be pruned.

Since we also submitted a stand-alone planner using only a dead-end PDB to the IPC, we refer to its planner abstract (Pommerening and Seipp 2016) for details on this technique.

## Dead-end Potentials

*Dead-end potentials* can prove that there is no plan for a state $s$ by finding an invariant that must be satisfied by all states reachable from $s$ but that is unsatisfied in every goal state. The invariants we consider are based on *potentials*, i.e., numerical values assigned to each state. If potentials exist such that

(1) no operator application decreases a state's potential, and

(2) the potential of $s$ is higher than the potential of all goal states,

then there cannot be a plan for $s$.

In order to describe the form of potentials used in our implementation, we first introduce more terminology. A *feature* is a conjunction of facts. We say that feature $F$ is true in state $s$ if all facts of $F$ are true in $s$. We define a numerical *weight* for each feature. The potential of a state $s$ is defined as the sum of all weights for the features that are true in $s$.

If the planning task is in transition normal form (Pommerening and Helmert 2015), the conditions (1) and (2) can be expressed as linear constraints over the feature weights. We can use an LP solver to check if there is a solution for these constraints. A solution of the LP forms a certificate for the unsolvability of $s$.

Dead-end potentials can show unsolvability using any set of features. The default feature set we use in most configurations contains all features of up to two facts.

We note that the dual of the resulting LP produces an operator counting heuristic (Pommerening et al. 2014). In fact, this is the implementation strategy we used for this method.

We use dead-end potentials to prune dead ends in every encountered state. Since only the bounds of the LP differ between states, the LP can be reused by adapting the bounds instead of having to be recreated for every state.

## Resource Detection

For a given planning task $\Pi$ with operator cost function *cost*, we check for *depletable resource variables* (shortly called resource variables in the following). We call a variable $v$ a resource variable if the atomic projection $\Pi^v$ of $\Pi$ onto $v$ yields, apart from self-loops, a directed acyclic graph (DAG). Intuitively, if this is the case, the number of operator applications that change the value of $v$ is bounded. We use this knowledge for pruning an optimal search in the projection of $\Pi$ onto all variables except $v$, called $\Pi^{\bar{v}}$.

Currently, our approach handles only a *single* resource variable. This resource variable is computed as follows. For

$\Pi$'s variable set $V$, we check for each variable $v$ in $V$ if the above DAG property *and* an additional quality criterion hold for $v$. The additional quality criterion requires i) the domain size of $v$ to be $\geq 5$, and ii) the number of operators in $\Pi^{\bar{v}}$ to be at most $85\%$ of the number of operators in $\Pi$. If no such resource variable is found, we abort immediately (and switch to the other configurations in our portfolios). If there are several such resource variables, we choose the one with the largest domain size among them. Overall, we either end up with no resource variable found (abstaining from the following steps), or with exactly one variable with the above properties on acyclicity, domain size, and operator reduction in the corresponding abstractions.

In case a resource variable $v$ has been found, we exploit this variable for detecting unsolvability as follows. Consider any cost function $cost'$ that maps operators inducing self-loops in $\Pi^v$ to 0. Let $L$ be the cost of the most expensive path in $\Pi^v$ using $cost'$ ($L$ is finite because the state space of $\Pi^v$ is a DAG except for edges where $cost'$ is 0). *Every* operator sequence $\pi = \langle o_1, \dots, o_n \rangle$ with $cost'(\pi) > L$ cannot be applicable in $\Pi$ because its cost exceeds the highest possible cost in the projection $\Pi^v$. Thus *every* plan $\pi$ of $\Pi$ must have $cost'(\pi) \leq L$. The projection of these plans to $V \setminus \{v\}$ must be a plan in $\Pi^{\bar{v}}$. We hence obtain a sufficient criterion for checking unsolvability of $\Pi$: Perform an optimal search for $\Pi^{\bar{v}}$ with an $f$-bound equal to $L$; if no plan is found in $\Pi^{\bar{v}}$ this way, then $\Pi$ is unsolvable.

Any cost function $cost'$ which maps self-loops in $\Pi^v$ to 0 works for this technique, but some lead to more pruning in $\Pi^{\bar{v}}$'s search space than others. A node is pruned in the search for $\Pi^{\bar{v}}$ if its $f$-value exceeds $L$, so a good cost function maximizes the number of operator sequences with maximal cost in $\Pi^v$. We compute $cost'$ by solving a linear program. Let $O$ be the operator set in $\Pi$ with corresponding abstract operator set $O^{\bar{v}}$ in $\Pi^{\bar{v}}$. We maximize the weighted sum

$$\sum_{o^{\bar{v}} \in O^{\bar{v}}} cost'(o^{\bar{v}}) \cdot |\{o \in O \mid o^{\bar{v}} \text{ is the projection of } o\}|,$$

using the constraints that the summed $cost'$ values are $L$ on every path in $\Pi^v$ from the source of the DAG (the initial value of $v$) to an artificial sink connecting all sinks of the DAG. In our implementation, we fix $L$ to 1000. Every other value of $L$ would have correspondingly scaled solutions of $cost'$ but since we round costs to integers, we have to set $L$ sufficiently high to avoid rounding too many different costs to the same value.

## Other Fast Downward Components

In addition to the three techniques described above, we used the following Fast Downward components for detecting unsolvability.

**Search** We implemented a simple breadth-first search that we used for most configurations. Compared to Fast Downward's general-purpose eager best-first search, it has a considerably smaller overhead. This search method is called `unsolvable_search` in the configurations listed in the appendix.

Configurations using resource detection must find optimal plans in the projection where the resource variable is projected out of the task. For those configurations, we used A$^*$ search.

**Heuristics** In addition to our new techniques, we made the following heuristics available for configuration.

- Blind heuristic
- CEGAR (Seipp and Helmert 2013; 2014): additive and non-additive variants
- $h^m$ (Haslum and Geffner 2000): naive implementation
- $h^{\max}$ (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999)
- LM-cut (Helmert and Domshlak 2009)
- Merge-and-shrink (Helmert et al. 2014; Sievers, Wehrle, and Helmert 2014)
- Operator counting heuristics (Pommerening et al. 2015).
- The canonical PDBs heuristic either combining PDBs from systematically generated patterns (Pommerening, Röger, and Helmert 2013) or PDBs from iPDB hill climbing (Haslum et al. 2007), and the zero-one PDBs heuristic combining PDBs from a genetic algorithm (Edelkamp 2006). Sievers, Ortlieb, and Helmert (2012) describe implementation details.
- Potential heuristics (Pommerening et al. 2015) with different objective functions as described by Seipp, Pommerening, and Helmert (2015). We also added a variant of the potential heuristic that maximizes the average potential of all syntactic states (called *unsolvable-all-states-potential heuristic*). This variant sets all operator costs to zero, allowing to prune all states with a positive potential.

**Pruning** We used the following two pruning methods:

- Strong stubborn sets: the first variant instantiates strong stubborn sets for classical planning in a straight-forward way (Alkhazraji et al. 2012; Wehrle and Helmert 2014). The second variant (Wehrle et al. 2013) provably dominates the Expansion Core method (Chen and Yao 2009) in terms of pruning power.

  While the standard implementation of strong stubborn sets in Fast Downward entirely precomputes the interference relation, we enhanced the implementation by computing the interference relation "on demand" during the search, and by switching off pruning completely in case the amount of pruned states falls below a given threshold.
- $h^2$-mutexes (Alcázar and Torralba 2015): an operator pruning method for Fast Downward's preprocessor. We use this method for all three portfolios.

## Benchmarks

In this section we describe the benchmark domains we used for evaluating our heuristics and for automatic algorithm configuration.

We used the collection of unsolvable tasks from Hoffmann, Kissmann, and Torralba (2014) comprised of the domains 3unsat, Bottleneck, Mystery, Pegsol, RCP-NoMystery, RCP-Rovers, RCP-TPP and Tiles. Futhermore, we used the unsolvable Maintenance (converted to STRIPS) and Tetris instances from the IPC 2014 optimal track.

Finally, we created two new domains and modified some existing IPC domains to contain unsolvable instances. The following list describes these domains.

**Cavediving** (IPC 2014). We generated unsolvable instances by limiting the maximal capacity the divers can carry.

**Childsnack** (IPC 2014). We generated unsolvable instances by setting the ratio of available ingredients to required servings to values less than 1.

**NoMystery** (IPC 2011). We generated unsolvable instances by reducing the amounts of fuel available at each location.

**Parking** (IPC 2011). We generated unsolvable instances by setting the number of cars to $2l-1$, where $l$ is the number of parking curb locations.

**Sokoban** (IPC 2008). We used the twelve methods described by Zerr (2014) for generating unsolvable instances.

**Spanner** (IPC 2011). We generated unsolvable instances by making the number of nuts exceed the number of spanners.

**Pebbling** (New). Consider a square $n \times n$ grid. We call the three fields in the upper left corner (i.e., coordinates $\langle 0,0 \rangle$, $\langle 0,1 \rangle$ and $\langle 1,0 \rangle$) the *prison*. The prison is initially filled with pebbles, all other fields are empty. A pebble on position $\langle x,y \rangle$ can be moved if the fields $\langle x+1,y \rangle$ and $\langle x,y+1 \rangle$ are empty. Moving the pebble "clones" it to the free fields, i.e., the pebble is removed from $\langle x,y \rangle$ and new pebbles are added to $\langle x+1,y \rangle$ and $\langle x,y+1 \rangle$. The goal is to free all pebbles from the prison, i.e., have no pebble on a field in the prison. This problem is unsolvable for all values of $n$.

**PegsolInvasion** (New). This domain is related to the well-known peg solitaire board game. Instead of peg solitaire's "cross" layout, PegsolInvasion tasks have a rectangular $n \times m$ grid, where $m = n + x > n$. Initially, the $n \times n$ square at the bottom of the grid is filled with pegs. The goal is to move one peg to the middle of the top row using peg solitaire movement rules. This problem is unsolvable for all values of $n \geq 1$ and $x \geq 5$.

## Algorithm Configuration

In the spirit of previous work (Vallati et al. 2011; Fawcett et al. 2011; Seipp et al. 2012; 2015), we used algorithm configuration to find configurations for unsolvable planning tasks. Here, we employed SMAC v2.10.04, a state-of-the-art model-based configuration tool (Hutter, Hoos, and Leyton-Brown 2011).

Some of the heuristics listed above are not useful for proving unsolvability. On the other hand, all of the mentioned heuristics are useful for our resource detection method, since we try to *solve* the modified tasks. We therefore considered two algorithm configuration scenarios for Fast Downward, one tailored towards unsolvability detection, the other towards resource detection.

### Configuring for Unsolvability

Our configuration space for detecting unsolvability only includes one search algorithm, our new breadth-first search. We include all new techniques, existing heuristics and pruning methods described above, except for the following heuristics:

- All potential heuristics other than the unsolvable-all-states-potential heuristic. Since the other variants use bounds on each weight, they always compute finite heuristic values and will never prune any state.
- The canonical PDBs heuristic and the zero-one PDBs heuristic. Both techniques can increase the heuristic value, but will not lead to more pruning than taking the maximum over the PDBs.
- LM-cut, because it can only detect states as unsolvable that are also detected as unsolvable by $h^{\max}$, which is faster to compute.
- Additive variant of CEGAR.

Using several hand-crafted Fast Downward configurations, we identified domains from our benchmark set containing *easy-non-trivial* instances, i.e., instances that are not trivially unsolvable and for which one or more of the configurations could prove unsolvability within 300 CPU seconds. These domains were 3unsat, Cavediving, Mystery, NoMystery, Parking, Pegsol, Tiles, RCP-NoMystery, RCP-Rovers, RCP-TPP, and Sokoban. The three RCP domains were further subdivided by instance difficulty into two sets each, allowing algorithm configuration to find separate configurations for easy and hard tasks. We used the easy-non-trivial instances as the training sets for each problem domain, while keeping any remaining instances from each domain for use in a held-out test set not used during configuration.

We then performed 10 independent SMAC runs for each of the 14 domain-specific training sets. Each SMAC run was allocated 12 CPU hours of runtime, and each individual run of Fast Downward was given 300 CPU seconds of runtime and 8 GB of memory. The starting configuration was a combination of the dead-end pattern database and operator counting heuristics. The 10 best configurations selected by SMAC for each considered domain were evaluated on the corresponding test set. We selected the configuration with the best penalized average runtime (PAR-10) as the incumbent configuration for that domain.

We then extended the training set for each domain by including any instances for which unsolvability was proven in under 300 CPU seconds by the incumbent configuration for that domain. Then we performed an additional 10 independent runs of SMAC on the new training sets for each domain, using the incumbent configuration for that domain as the starting configuration. We again evaluated the 10 best configurations for each domain on the corresponding test set, and selected the configuration with the highest PAR-10 score as the representative for this domain.

### Configuring for Resource Detection

Our configuration space for resource detection allows only A$^*$ search, but includes all other components described above (new techniques, all listed heuristics and pruning methods).

We chose the easy-non-trivial instances from the three RCP domains as our benchmark set. Similar to the procedure above we subdivided the tasks from the three domains into three sets by difficulty, yielding 9 benchmark sets in total.

We employed the same procedure as above for finding representative configurations from the resource detection configuration space for these 9 sets. In this scenario we used LM-cut as the starting configuration.

## Portfolios

Using the representative configurations from the two configuration scenarios described above, we obtained a total of 23 separate Fast Downward configurations. We evaluated the performance of each on our entire 928-instance benchmark set with a 1800 CPU second runtime cutoff. We used the resulting data for constructing Aidos 1 and 2 manually, and for computing Aidos 3 automatically.

### Manual portfolios: Aidos 1 and 2

Analyzing the results, we distilled three configurations that together solve all tasks solved by any of the 23 representative configurations. The three configurations use $h^2$-mutexes during preprocessing and stubborn sets to prune applicable operators during search. In particular, they use the stubborn sets variant that provably dominates EC (called `stubborn_sets_ec` in the appendix). We adjusted the minimum pruning threshold individually for the three techniques. Techniques that can be evaluated fast on a given state got a higher minimum pruning threshold. The three configurations differ in the following aspects:

**C1** Breadth-first search using a dead-end pattern database.

**C2** Breadth-first search using dead-end potentials with features of up to two facts.

**C3** Resource detection using an A$^*$ search. The search uses the CEGAR heuristic and operator counting with LM-cut and state equation constraints.

Adding other heuristics did not increase the number of solved tasks on our benchmark set. The three configurations did not dominate each other, so it made sense to include all of them in our portfolio. The only question was how to order them and how to assign the time slices.

Both C1 and C2 prove many of our benchmark tasks unsolvable in the initial state. On such instances the configurations usually take less than a second. Since the unsolvability IPC uses time scores to break ties we start with two short runs of C1 and C2. This avoids spending a lot of time using one configuration, when another solves the task very quickly.

Next, we run the resource detection method (C3). It will be inactive on tasks where no resources are found and therefore not consume any time. Experiments showed that the dead-end potentials use much less memory than the dead-end PDB. To avoid a portfolio that runs out of memory while executing the last component and therefore does not use the full amount of time, we put the dead-end potentials (C2) last.

Results on our benchmarks showed that C3 did not solve any additional tasks after 420 seconds. Similarly, C2 did not solve any additional tasks after 100 seconds. Since C1 tends to solve more tasks if given more time, we limited the times for the other two configurations to 420 and 100 seconds and alotted the remaining time (1275 seconds) to C1.

Aidos 2 is almost identical to Aidos 1, the only difference being that it equally distributes the time among the three main portfolio components.

### Automatic portfolio: Aidos 3

In order to automatically select configurations and assign both order and allocated runtime for Aidos 3, we used the greedy schedule construction technique of Streeter and Smith (2008). Briefly, given a set of configurations and corresponding runtimes for each on a benchmark set, this technique iteratively adds the configuration which maximizes $\frac{n}{t}$, where $n$ is the number of additional instances solved with a runtime cutoff of $t$. This can be efficiently solved for a given benchmark set, as the runtime required for each configuration on each instance is known and thus a finite set of possible $t$ need to be considered. Usually, this results in a schedule beginning with many configurations and short runtime cutoffs in order to quickly capture as much coverage as possible. In order to avoid schedule components with extremely short runtime cutoffs, we set a minimum of 1 CPU second for each component.

Using the performance of the 23 configurations obtained from our two configuration scenarios configurations evaluated on our entire benchmark set (i.e., all domains without distinction of training or test set), this process resulted in the Aidos 3 portfolio with 11 schedule components and runtime cutoffs ranging from 2 to 1549 CPU seconds. All configurations use $h^2$-mutexes during preprocessing.

## Acknowledgments

We would like to thank all Fast Downward contributors. We are especially grateful to Malte Helmert, not only for his work on Fast Downward, but also for many fruitful discussions about the unsolvability IPC. Special thanks also go to Álvaro Torralba and Vidal Alcázar for their $h^2$-mutexes code.

## References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. 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)*, 2–6. AAAI Press.

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A stubborn set algorithm for optimal planning. In De Raedt, L.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P., eds., *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, 891–892. IOS Press.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In Biundo, S., and Fox, M., eds., *Recent Advances in AI Planning. 5th European Conference on Planning (ECP 1999)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 360–372. Heidelberg: Springer-Verlag.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, 714–719. AAAI Press.

Chen, Y., and Yao, G. 2009. Completeness and optimality preserving reduction for planning. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1659–1664.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Domain-specific configuration using Fast Downward. In *ICAPS 2011 Workshop on Planning and Learning*, 13–17.

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.

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.

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.; 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. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 441–446. IOS Press.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In Coello, C. A. C., ed., *Proceedings of the Fifth Conference on Learning and Intelligent OptimizatioN (LION 2011)*, 507–523. Springer.

Pommerening, F., and Helmert, M. 2015. A normal form for classical planning tasks. 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)*, 188–192. AAAI Press.

Pommerening, F., and Seipp, J. 2016. Fast downward dead-end pattern database. In *Unsolvability International Planning Competition: planner abstracts*.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 226–234. AAAI Press.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3335–3341. 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.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 347–351. AAAI Press.

Seipp, J., and Helmert, M. 2014. Diverse and additive Cartesian abstraction heuristics. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 289–297. AAAI Press.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning portfolios of automatically tuned planners. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 368–372. AAAI Press.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3364–3370. AAAI Press.

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., *Pro-*

*ceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 193–201. AAAI Press.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 105–111. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized label reduction for merge-and-shrink heuristics. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2358–2366. AAAI Press.

Streeter, M. J., and Smith, S. F. 2008. New techniques for algorithm portfolio design. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI 2008)*, 519–527.

Vallati, M.; Fawcett, C.; Gerevini, A.; Holger, H.; and Saetti, A. 2011. ParLPG: Generating domain-specific planners through automatic parameter configuration in LPG. In *IPC 2011 planner abstracts, Planning and Learning Part*.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 323–331. AAAI Press.

Wehrle, M.; Helmert, M.; Alkhazraji, Y.; and Mattmüller, R. 2013. The relative pruning power of strong stubborn sets and expansion core. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 251–259. AAAI Press.

Zerr, D. 2014. Generating and evaluating unsolvable strips planning instances for classical planning. Bachelor's thesis, University of Basel.

## Appendix – Fast Downward Aidos Portfolios

We list the configurations forming our three portfolios. Our portfolio components have the form of pairs (time slice, configuration), with the first entry reflecting the time slice allowed for the configuration, which is in turn shown below the time slice.

### Aidos 1

```
1,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    feature_constraints(max_size=2)], cost_type=zero)
--search unsolvable_search([h_seq], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.20))

4,
--search unsolvable_search([deadpdbs(max_time=1)], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.80))

420,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    lmcut_constraints()])
--heuristic h_cegar=cegar(subtasks=[original()], pick=max_hadd, max_time=relative
    time 75, f_bound=compute)
--search astar(f_bound=compute, eval=max([h_cegar, h_seq]),
    pruning=stubborn_sets_ec(min_pruning_ratio=0.50))

1275,
--search unsolvable_search([deadpdbs(max_time=relative time 50)],
    pruning=stubborn_sets_ec(min_pruning_ratio=0.80))

100,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    feature_constraints(max_size=2)], cost_type=zero)
--search unsolvable_search([h_seq], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.20))
```

### Aidos 2

```
1,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    feature_constraints(max_size=2)], cost_type=zero)
--search unsolvable_search([h_seq], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.20))

4,
--search unsolvable_search([deadpdbs(max_time=1)], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.80))

598,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    lmcut_constraints()])
--heuristic h_cegar=cegar(subtasks=[original()], pick=max_hadd, max_time=relative
    time 75, f_bound=compute)
--search astar(f_bound=compute, eval=max([h_cegar, h_seq]),
    pruning=stubborn_sets_ec(min_pruning_ratio=0.50))

598,
--search unsolvable_search([deadpdbs(max_time=relative time 50)],
    pruning=stubborn_sets_ec(min_pruning_ratio=0.80))

599,
--heuristic h_seq=operatorcounting([state_equation_constraints(),
    feature_constraints(max_size=2)], cost_type=zero)
--search unsolvable_search([h_seq], pruning=stubborn_sets_ec(
    min_pruning_ratio=0.20))
```

**Aidos 3**

```
8,
--heuristic h_blind=blind(cache_estimates=false, cost_type=one)
--heuristic h_cegar=cegar(subtasks=[original(copies=1)], max_states=10,
    use_general_costs=true, cost_type=one, max_time=relative time 50,
    pick=min_unwanted, cache_estimates=false)
--heuristic h_deadpdbs=deadpdbs(patterns=combo(max_states=1), cost_type=one,
    max_dead_ends=290355, max_time=relative time 99, cache_estimates=false)
--heuristic h_deadpdbs_simple=deadpdbs_simple(patterns=combo(max_states=1),
    cost_type=one, cache_estimates=false)
--heuristic h_hm=hm(cache_estimates=false, cost_type=one, m=1)
--heuristic h_hmax=hmax(cache_estimates=false, cost_type=one)
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    constraint_generators=[feature_constraints(max_size=3), lmcut_constraints(),
    pho_constraints(patterns=combo(max_states=1)), state_equation_constraints()],
    cost_type=one)
--heuristic h_unsolvable_all_states_potential=unsolvable_all_states_potential(
    cache_estimates=false, cost_type=one)
--search unsolvable_search(heuristics=[h_blind, h_cegar, h_deadpdbs,
    h_deadpdbs_simple, h_hm, h_hmax, h_operatorcounting,
    h_unsolvable_all_states_potential], cost_type=one, pruning=stubborn_sets_ec(
    min_pruning_ratio=0.9887183754249436))

6,
--heuristic h_deadpdbs=deadpdbs(patterns=genetic(disjoint=false,
    mutation_probability=0.2794745683909153, pdb_max_size=1, num_collections=40,
    num_episodes=2), cost_type=normal, max_dead_ends=36389913, max_time=relative
    time 52, cache_estimates=false)
--heuristic h_deadpdbs_simple=deadpdbs_simple(patterns=genetic(disjoint=false,
    mutation_probability=0.2794745683909153, pdb_max_size=1, num_collections=40,
    num_episodes=2), cost_type=normal, cache_estimates=false)
--heuristic h_lmcut=lmcut(cache_estimates=true, cost_type=normal)
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    constraint_generators=[feature_constraints(max_size=2), lmcut_constraints(),
    pho_constraints(patterns=genetic(disjoint=false,
    mutation_probability=0.2794745683909153, pdb_max_size=1, num_collections=40,
    num_episodes=2)), state_equation_constraints()], cost_type=normal)
--heuristic h_zopdbs=zopdbs(patterns=genetic(disjoint=false,
    mutation_probability=0.2794745683909153, pdb_max_size=1, num_collections=40,
    num_episodes=2), cost_type=normal, cache_estimates=true)
--search astar(f_bound=compute, mpd=false, pruning=stubborn_sets_ec(
    min_pruning_ratio=0.2444996579070121), eval=max([h_deadpdbs,
    h_deadpdbs_simple, h_lmcut, h_operatorcounting, h_zopdbs]))

2,
--heuristic h_deadpdbs_simple=deadpdbs_simple(patterns=systematic(
    only_interesting_patterns=true, pattern_max_size=3), cost_type=one,
    cache_estimates=false)
--search unsolvable_search(heuristics=[h_deadpdbs_simple], cost_type=one,
    pruning=null())

2,
--heuristic h_deadpdbs_simple=deadpdbs_simple(patterns=genetic(disjoint=true,
    mutation_probability=0.32087500872172836, num_collections=30, num_episodes=7,
    pdb_max_size=1908896), cost_type=one, cache_estimates=false)
--heuristic h_hm=hm(cache_estimates=false, cost_type=one, m=3)
--heuristic h_pdb=pdb(pattern=greedy(max_states=18052), cost_type=one,
    cache_estimates=false)
--search unsolvable_search(heuristics=[h_deadpdbs_simple, h_hm, h_pdb],
    cost_type=one, pruning=null())

2,
--heuristic h_blind=blind(cache_estimates=false, cost_type=one)
--heuristic h_deadpdbs=deadpdbs(cache_estimates=false, cost_type=one,
    max_dead_ends=4, max_time=relative time 84, patterns=systematic(
    only_interesting_patterns=false, pattern_max_size=15))
--heuristic h_deadpdbs_simple=deadpdbs_simple(patterns=systematic(
    only_interesting_patterns=false, pattern_max_size=15), cost_type=one,
    cache_estimates=false)
--heuristic h_merge_and_shrink=merge_and_shrink(cache_estimates=false,
    label_reduction=exact(before_shrinking=true, system_order=random,
    method=all_transition_systems, before_merging=false), cost_type=one,
    shrink_strategy=shrink_bisimulation(threshold=115,
    max_states_before_merge=56521, max_states=228893, greedy=true,
    at_limit=use_up), merge_strategy=merge_dfp(atomic_before_product=false,
    atomic_ts_order=regular, product_ts_order=random, randomized_order=true))
--search unsolvable_search(heuristics=[h_blind, h_deadpdbs, h_deadpdbs_simple,
    h_merge_and_shrink], cost_type=one, pruning=null())

4,
--heuristic h_cegar=cegar(subtasks=[original(copies=1)], max_states=114,
    use_general_costs=false, cost_type=normal, max_time=relative time 1,
    pick=max_hadd, cache_estimates=false)
--heuristic h_cpdbs=cpdbs(patterns=genetic(disjoint=true,
    mutation_probability=0.7174375735405052, num_collections=4, num_episodes=170,
    pdb_max_size=1), cost_type=normal, dominance_pruning=true,
    cache_estimates=false)
--heuristic h_deadpdbs=deadpdbs(cache_estimates=true, cost_type=normal,
    max_dead_ends=12006, max_time=relative time 21, patterns=genetic(
    disjoint=true, mutation_probability=0.7174375735405052, num_collections=4,
    num_episodes=170, pdb_max_size=1))
--heuristic h_deadpdbs_simple=deadpdbs_simple(cache_estimates=false,
    cost_type=normal, patterns=genetic(disjoint=true,
    mutation_probability=0.7174375735405052, num_collections=4, num_episodes=170,
    pdb_max_size=1))
--heuristic h_lmcut=lmcut(cache_estimates=true, cost_type=normal)
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    cost_type=normal, constraint_generators=[feature_constraints(max_size=2),
    lmcut_constraints(), pho_constraints(patterns=genetic(disjoint=true,
    mutation_probability=0.7174375735405052, num_collections=4, num_episodes=170,
    pdb_max_size=1)), state_equation_constraints()])
--heuristic h_pdb=pdb(pattern=greedy(max_states=250), cost_type=normal,
    cache_estimates=false)
--search astar(f_bound=compute, mpd=true, pruning=null(), eval=max([h_cegar,
    h_cpdbs, h_deadpdbs, h_deadpdbs_simple, h_lmcut, h_operatorcounting,
    h_pdb]))

7,
--heuristic h_blind=blind(cache_estimates=false, cost_type=one)
--heuristic h_cegar=cegar(subtasks=[original(copies=1)], max_states=5151,
    use_general_costs=false, cost_type=one, max_time=relative time 44,
    pick=max_hadd, cache_estimates=false)
--heuristic h_hmax=hmax(cache_estimates=false, cost_type=one)
```

```
--heuristic h_merge_and_shrink=merge_and_shrink(cache_estimates=false,
    label_reduction=exact(before_shrinking=true, system_order=random,
    method=all_transition_systems_with_fixpoint, before_merging=false),
    cost_type=one, shrink_strategy=shrink_bisimulation(threshold=1,
    max_states_before_merge=12088, max_states=100000, greedy=false,
    at_limit=return), merge_strategy=merge_linear(variable_order=cg_goal_random))
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    constraint_generators=[feature_constraints(max_size=2), lmcut_constraints(),
    state_equation_constraints()], cost_type=one)
--heuristic h_unsolvable_all_states_potential=unsolvable_all_states_potential(
    cache_estimates=false, cost_type=one)
--search unsolvable_search(heuristics=[h_blind, h_cegar, h_hmax,
    h_merge_and_shrink, h_operatorcounting, h_unsolvable_all_states_potential],
    cost_type=one, pruning=null())

37,
--heuristic h_hmax=hmax(cache_estimates=false, cost_type=one)
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    constraint_generators=[feature_constraints(max_size=10),
    state_equation_constraints()], cost_type=zero)
--search unsolvable_search(heuristics=[h_hmax, h_operatorcounting],
    cost_type=one, pruning=stubborn_sets_ec(min_pruning_ratio=0.4567602354825518))

33,
--heuristic h_all_states_potential=all_states_potential(max_potential=1e8,
    cache_estimates=true, cost_type=normal)
--heuristic h_blind=blind(cache_estimates=false, cost_type=normal)
--heuristic h_cegar=cegar(subtasks=[goals(order=hadd_down), landmarks(
    order=original, combine_facts=true), original(copies=1)], max_states=601,
    use_general_costs=false, cost_type=normal, max_time=relative time 88,
    pick=min_unwanted, cache_estimates=true)
--heuristic h_deadpdbs_simple=deadpdbs_simple(cache_estimates=true,
    cost_type=normal, patterns=hillclimbing(min_improvement=2,
    pdb_max_size=7349527, collection_max_size=233, max_time=relative time 32,
    num_samples=28))
--heuristic h_initial_state_potential=initial_state_potential(max_potential=1e8,
    cache_estimates=false, cost_type=normal)
--heuristic h_operatorcounting=operatorcounting(cache_estimates=false,
    cost_type=normal, constraint_generators=[feature_constraints(max_size=10),
    lmcut_constraints(), pho_constraints(patterns=hillclimbing(min_improvement=2,
    pdb_max_size=7349527, collection_max_size=233, max_time=relative time 32,
    num_samples=28)), state_equation_constraints()])
--heuristic h_pdb=pdb(pattern=greedy(max_states=6), cost_type=normal,
    cache_estimates=true)
--heuristic h_zopdbs=zopdbs(patterns=hillclimbing(min_improvement=2,
    pdb_max_size=7349527, collection_max_size=233, max_time=relative time 32,
    num_samples=28), cost_type=normal, cache_estimates=false)
--search astar(f_bound=compute, mpd=true, pruning=stubborn_sets_ec(
    min_pruning_ratio=0.0927145675045078), eval=max([h_all_states_potential,
    h_blind, h_cegar, h_deadpdbs_simple, h_initial_state_potential,
    h_operatorcounting, h_pdb, h_zopdbs]))

150,
--heuristic h_deadpdbs=deadpdbs(cache_estimates=false, cost_type=one,
    max_dead_ends=6, max_time=relative time 75, patterns=systematic(
    only_interesting_patterns=true, pattern_max_size=1))
--search unsolvable_search(heuristics=[h_deadpdbs], cost_type=one,
    pruning=stubborn_sets_ec(min_pruning_ratio=0.3918701752094733))
```

```
1549,
--heuristic h_deadpdbs=deadpdbs(cache_estimates=false, cost_type=one,
    max_dead_ends=63156737, max_time=relative time 4, patterns=ordered_systematic(
    pattern_max_size=869))
--heuristic h_merge_and_shrink=merge_and_shrink(cache_estimates=false,
    label_reduction=exact(before_shrinking=true, system_order=random,
    method=all_transition_systems_with_fixpoint, before_merging=false),
    cost_type=one, shrink_strategy=shrink_bisimulation(threshold=23,
    max_states_before_merge=29143, max_states=995640, greedy=false,
    at_limit=return), merge_strategy=merge_dfp(atomic_before_product=false,
    atomic_ts_order=regular, product_ts_order=new_to_old, randomized_order=false))
--search unsolvable_search(heuristics=[h_deadpdbs, h_merge_and_shrink],
    cost_type=one, pruning=null())
```