Merge-and-Shrink Heuristics for Classical Planning: Efficient Implementation and Partial Abstractions

Silvan Sievers
University of Basel
Basel, Switzerland
silvan.sievers@unibas.ch

Abstract

Merge-and-shrink heuristics are a successful class of abstraction heuristics used for optimal classical planning. With the recent addition of generalized label reduction, merge-and-shrink can be understood as an algorithm framework that repeatedly applies transformations to a factored representation of a given planning task to compute an abstraction. In this paper, we describe an efficient implementation of the framework and its transformations, comparing it to its previous implementation in Fast Downward. We further discuss partial merge-and-shrink abstractions that do not consider all aspects of the concrete state space. To compute such partial abstractions, we stop the merge-and-shrink computation early by imposing simple limits on the resource consumption of the algorithm. Our evaluation shows that the efficient implementation indeed improves over the previous one, and that partial merge-and-shrink abstractions further push the efficiency of merge-and-shrink planners.

Introduction

A* search (Hart, Nilsson, and Raphael 1968) with an admissible heuristic (Pearl 1984) is a state-of-the-art approach to optimally solving classical planning problems (Ghallab, Nau, and Traverso 2004). A popular method for obtaining admissible heuristics is based on abstractions. *Merge-and-shrink* (Dräger, Finkbeiner, and Podelski 2009; Helmert et al. 2014) is a state-of-the-art algorithm framework for computing abstractions of planning tasks. With the recent addition of generalized label reduction (Sievers, Wehrle, and Helmert 2014), the algorithm can be understood as a framework that repeatedly applies transformations to a given state space to compute an abstraction.

Broadly speaking, the key idea of merge-and-shrink is to operate on compact representations of large transition systems, called *factored transition systems*. Planning tasks induce factored transition systems that consist of *atomic factors* which each represent a single variable of the task. Starting from this factored transition system, the merge-and-shrink framework repeatedly either *merges* two factors by computing their product, *shrinks* a factor by applying an abstraction to it, *prunes* a factor by discarding dead states, or applies *label reductions* to the common label set of the factored transition system, which initially corresponds to the set of operators of the task. The computation ends if only one factor is left, which induces the merge-and-shrink heuristic.

Besides factored representations of transition systems, the merge-and-shrink framework also computes *factored mappings* (e.g., Sievers 2017) to represent the state mapping of the abstraction. Factored mappings are tree-like data structures that map states of the planning task to states of the factors of the transformed factored transition system.

In the literature on merge-and-shrink, the algorithmic aspect as well as the question of how to efficiently implement a merge-and-shrink planner has not been addressed a lot. Helmert et al. (2014) describe some techniques used in their implementation which we base ours on. Recently, Fan, Holte, and Müller (2018) suggested to compute very small merge-and-shrink abstractions quickly and to complement them with the usual, larger abstractions by computing the maximum heuristic over both.

In this paper, we address the algorithmic aspect as well as the question of how to efficiently compute merge-and-shrink abstractions by making the following contributions:

1. We provide an algorithmic description of how to compute exact generalized label reductions, which has only briefly been mentioned in the original work introducing generalized label reduction (Sievers, Wehrle, and Helmert 2014).
2. We describe an *efficient* implementation of the merge-and-shrink framework that exploits local label equivalence relations to compactly represent transition systems. In particular, we describe the implementation of merge-and-shrink in Fast Downward (Helmert 2006) and compare it to the previous implementation.
3. We discuss *partial* merge-and-shrink abstractions that do not cover all variables of a planning task, which we compute by imposing simple limits on the resource consumption of the merge-and-shrink computation, further pushing the efficiency of the overall approach.

Background

In this section, we briefly review the concept of classical planning and provide a rather extensive exposition of the merge-and-shrink framework to build a solid foundation based on which we describe algorithms and implementation.

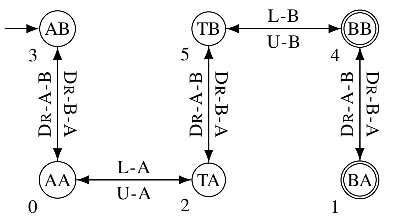
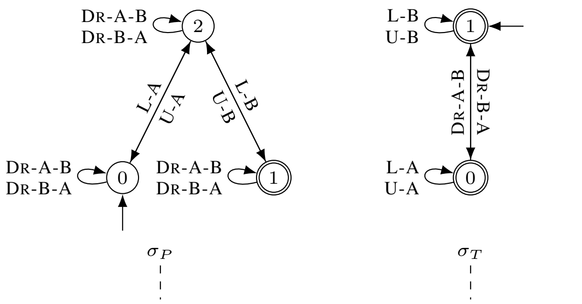


Figure 1: Induced transition system $\Theta(\Pi)$ of the example task. Also: product $\Theta_\otimes$ of $F(\Pi)$ (modulo names of states).



$dom(v_P)$	A	B	T
	0	1	2

$dom(v_T)$	A	B
	0	1

Figure 2: Top: induced factored transition system $F(\Pi)$ of the example task with two atomic factors Θ^P (left) and Θ^T (right). Bottom: F2F mapping Σ consisting of two F2N mappings σ_P (left) and σ_T (right), corresponding to Θ^P and Θ^T.

Classical Planning

A planning task in the SAS$^+$ formalism (Bäckström and Nebel 1995), augmented with action costs, is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with the following components. $\mathcal{V}$ is a finite set of *variables* v, each associated with a finite domain $dom(v)$. A *partial state* s is a partial assignment over a subset $vars(s) \subseteq \mathcal{V}$ of the variables. We write $s[v]$ to denote the value of $v \in vars(s)$. If $vars(s) = \mathcal{V}$, s is called a *state*. We write $\mathcal{S}(\mathcal{V})$ for the set of states over $\mathcal{V}$. Two partial states s and s' are *consistent* if $s[v] = s'[v]$ for all $v \in vars(s) \cap vars(s')$. $\mathcal{O}$ is a finite set of *operators* o, each with associated partial states $pre(o)$ and $eff(o)$, called *precondition* and *effect*, and an associated non-negative value $cost(o) \in \mathbb{R}_0^+$, called the *cost*. Finally, s_0 is a state called the *initial state*, and $S_\star$ is a partial state called the *goal*.

Example 1. *Consider a simple logistics planning task with truck* T *and package* P. *Variable* v_T *with* $dom(v_T) = \{\mathrm{A}, \mathrm{B}\}$ *represents the position of the truck, and* v_P *with* $dom(v_P) = \{\mathrm{A}, \mathrm{B}, \mathrm{T}\}$ *that of the package. There is an operator* DR-A-B *that drives the truck from* A *to* B *and has precondition* $pre(\text{DR-A-B}) = \{v_T \mapsto \mathrm{A}\}$ *and effect* $eff(\text{DR-A-B}) = \{v_T \mapsto \mathrm{B}\}$. *Analogously for* DR-B-A. *Similarly,* U-X *and* L-X *load the package at location* $\mathrm{X} \in \{\mathrm{A}, \mathrm{B}\}$. *We assume unit-cost. The goal is to have the package at* B, *which is initially at* A, *and the truck starts at* B.

The semantics of a planning task can be naturally defined in terms of a *labeled transition system*. A transition system is a tuple $\Theta = \langle S, L, c, T, s_0, S_\star \rangle$, where S is a finite state of *states*, L is a finite set of *transition labels*, $c : L \to \mathbb{R}_0^+$ is a *label cost function*, $T \subseteq S \times L \times S$ is a set of *labeled transitions*, $s_0 \in S$ is the *initial state*, and $S_\star \subseteq S$ is the set of *goal states*. We write $s \xrightarrow{\ell} s'$ to denote a transition $\langle s, \ell, s' \rangle$ from s to s' with label ℓ. A *path* from s to s' is a sequence $\pi = \langle t_1, \dots, t_n \rangle$ of transitions such that there exist states $s = s_0, \dots, s_n = s'$ with $t_i = \langle s_{i-1}, \ell_i, s_i \rangle \in T$ for all $i \in \{1, \dots, n\}$. The *cost* of such a path is the accumulated cost of the labels of the transitions, i.e., $\sum_{i=1}^n c(\ell_i)$.

Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. Its *induced transition system* $\Theta(\Pi)$ is defined as $\langle S, L, c, T, s_0, S_\star \rangle$, where $S = \mathcal{S}(\mathcal{V})$, $L = \mathcal{O}$, $c(\ell) = cost(\ell)$ for all $\ell \in L$, $s \xrightarrow{\ell} t \in T$ iff s is consistent with $pre(\ell)$ and t is the state consistent with $eff(\ell)$ and $t[v] = s[v]$ for all $v \notin vars(eff(\ell))$, and $S_\star = \{s \in S \mid s \text{ consistent with } s_\star\}$. A *plan* is a path from s_0 to a state in $S_\star$. *Optimal planning* deals with finding plans of *minimal cost* or proving that no plan exists.

Figure 1 shows the induced transition system $\Theta(\Pi)$ for the example task. Goal states are marked by double circles and the initial state has an ingoing arrow. In this example, we label states with two letters indicating the position of the package and the truck.

A heuristic for a transition system Θ with states S is a function $h_\Theta : S \to \mathbb{R}_0^+ \cup \infty$ that maps each state to an estimate of the cost of reaching a goal state from s. h_Θ is *perfect*, denoted h^*_Θ, if it computes the true cost for all states. It is *admissible* if $h_\Theta(s) \le h^*_\Theta(s)$ for all states s.

Merge-and-Shrink

The goal of the merge-and-shrink framework is to compute an abstraction heuristic for a large transition system such as the one induced by a planning task. To do so, it not only computes the abstract transition system, but also the state and label mappings of the abstraction. However, the given transition system is usually too large to be represented explicitly. A key aspect of the merge-and-shrink framework to address this issue is to work with *factored representations* of both transition systems and state mappings.

In particular, a *factored transition system* F is a tuple of transition systems (also called *factors* or *variables* of F), $F = \langle \Theta_1, \dots, \Theta_n \rangle$, where each element has the same label set and label cost function. It serves as a representation of the *synchronized product* (*product* for short) of all of its elements, which is the transition system defined as $\bigotimes F = \langle S^\otimes, L, c, T^\otimes, s_0^\otimes, S_\star^\otimes \rangle$, where $S^\otimes = \prod_{i=1}^n S^i$, i.e., the Cartesian product of the sets of states, $T^\otimes = \{\langle s^1, \dots, s^n \rangle \xrightarrow{\ell} \langle t^1, \dots, t^n \rangle \mid s^i \xrightarrow{\ell} t^i \in T^i \text{ for all } 1 \le i \le n\}$, $s_0^\otimes = \langle s_0^1, \dots, s_0^n \rangle$, and $S_\star^\otimes = \prod_{i=1}^n S_\star^i$.

A planning task Π induces a factored transition system $F(\Pi)$ which consists of *atomic* factors Θ^v that reflect the behavior of all variables v of Π: the states of Θ^v correspond to values of v and all operators induce transitions depending on their preconditions and effects on v. The top part of Figure 2 shows $F(\Pi)$ for the example task. It consists of two atomic factors for the two variables of the task. The product of $F(\Pi)$ corresponds to the transition system shown in Figure 1, apart from names of states, which formally would read, e.g., $\langle 0, 0 \rangle$ instead of AA and $\langle 2, 1 \rangle$ instead of TB.

Algorithm 1 Merge-and-shrink Framework

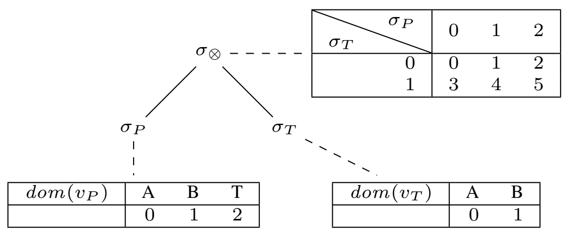
```
Input: Planning task $\Pi$, function SELECTTF that selects
    the next basic transformation to apply.
Output: Merge-and-shrink heuristic $h^{\text{M\&S}}_{\Pi}$ for $\Pi$.
1: function MERGEANDSHRINK($\Pi$, SELECTTF)
2:     $F \leftarrow F(\Pi)$, $\Sigma \leftarrow \text{id}^F$, $\lambda \leftarrow \text{id}^L$
3:     while $|F| > 1$ do
4:         $\langle F', \Sigma', \lambda' \rangle \leftarrow$ SELECTTF($F, \Sigma, \lambda$)
5:         $F \leftarrow F'$, $\Sigma \leftarrow \Sigma' \circ \Sigma$, $\lambda \leftarrow \lambda' \circ \lambda$
6:     end while
7:     $h^{\text{M\&S}}_{\Pi} \leftarrow h^*_{\Theta}$
8:     return $h^{\text{M\&S}}_{\Pi}$
9: end function
```

A *factored mapping* σ (Sievers 2017), also called cascading table (Helmert et al. 2014) or merge-and-shrink representation (Helmert, Röger, and Sievers 2015), is a tree-like data structure inductively defined over a set of variables $\mathcal{V}$. If σ is *atomic* (a leaf), then it has an associated variable $v \in \mathcal{V}$ and a table function σ^{tab} mapping from $dom(v)$ to an arbitrary value set $vals(\sigma)$. If σ is a *merge* (an inner node), then it has two component factored mappings σ_L and σ_R and a table function σ^{tab} mapping from $vals(\sigma_L) \times vals(\sigma_R)$ to an arbitrary value set $vals(\sigma)$. A factored mapping σ represents a function $[\![\sigma]\!] : \mathcal{S}(\mathcal{V}) \rightarrow vals(\sigma)$ inductively defined as follows: if σ is atomic with v, then $[\![\sigma]\!](\alpha) = \sigma^{\text{tab}}(\alpha[v])$. If it is a merge with components σ_L and σ_R, then $[\![\sigma]\!](\alpha) = \sigma^{\text{tab}}([\![\sigma_L]\!](\alpha), [\![\sigma_R]\!](\alpha))$.

We call such factored mappings σ *factored-to-non-factored (F2N)* mappings because $[\![\sigma]\!]$ maps states to single values. To represent state mappings between factored transition systems, we define *factored-to-factored (F2F)* mappings. An F2F mapping from a factored transition system F to a factored transition system $F' = \langle \Theta'_1, \dots, \Theta'_n \rangle$ is a tuple $\Sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ where each σ_i is an F2N mapping defined over F (viewing factors as variables with their states as values) which *corresponds* to Θ'_i, i.e., it maps to the states of Θ'_i. Σ thus represents the function $[\![\Sigma]\!] : \mathcal{S}(F) \rightarrow \mathcal{S}(F')$.

The bottom part of Figure 2 shows $\Sigma = \langle \sigma_P, \sigma_T \rangle$, an F2F mapping from $F(\Pi)$ to $F(\Pi)$. Its component F2N mappings σ_P and σ_T each project states of $F(\Pi)$ to their corresponding state in Θ^P and Θ^T. Σ thus represents the identity function on states of $F(\Pi)$. In the illustration, we only show the node of the F2N mapping and its table, omitting its value set which can be inferred from the the entries of the table.

We now discuss the pseudo-code of the merge-and-shrink framework shown in Algorithm 1. The algorithm first computes the induced factored transition system $F = F(\Pi)$ of the given task Π and further initializes the F2F mapping Σ to the identity mapping from F to F and the label mapping λ to the identity mapping on the label set L of F (line 2). In the main loop (lines 3–6), the algorithm then repeatedly selects a *transformation* (line 4) of the current factored transition system F using the user-specified SELECTTF function. Such transformations of F specify the transformed factored transition system F', the F2F mapping Σ from F to F', and the *label mapping* λ defined on L. To "apply" the selected transformation, the algorithm replaces F by F' and composes the state and label mappings with the previous one (line 5). When there is only one factor Θ of F left, the algorithm computes the heuristic which is defined as $h^{\text{M\&S}}_{\Pi}(s) = h^*_{\Theta}([\![\Sigma]\!](s))$ for all states $s \in \mathcal{S}(F(\Pi))$.



σ_T \\ σ_P	0	1	2
0	0	1	2
1	3	4	5

$dom(v_P)$	A	B	T
	0	1	2

$dom(v_T)$	A	B
	0	1

Figure 3: Merge factored mapping $\sigma_\otimes$.

We call the algorithm an algorithm framework rather than a concrete algorithm because it leaves one important choice point unspecified, namely how to select the transformations using function SELECTTF. Before discussing a concrete instantiation of the framework, we briefly review the four types of merge-and-shrink transformations.

A *shrink transformation* computes an abstraction α of the states of a given factor Θ of F and replaces Θ by the induced abstract factor $\alpha(\Theta)$. The F2N mapping corresponding to $\alpha(\Theta)$ represents the abstraction mapping α. An example shrink transformation of $F(\Pi)$ of the example task abstracts states 1 and 2 of Θ^P to the same state x, which thus inherits the transitions of both 1 and 2, turning, e.g., $1 \xrightarrow{\text{L-B}} 2$ into a self-looping transition. The transformed F2N mapping $\alpha(\sigma_P)$ maps both B and T to x (and still A to 0).

A *merge transformation* replaces two factors Θ, Θ' of F by their product $\Theta_\otimes$. The F2N mapping for the new factor is a merge factored mapping that maps pairs of states of Θ and Θ' to the corresponding product state in $\Theta_\otimes$. An example merge transformation for $F(\Pi)$ of the example task replaces the two atomic factors Θ_P and Θ_T by their product $\Theta_\otimes$ shown in Figure 1. The two corresponding factored mappings σ_P and σ_T are replaced by the merge factored mapping $\sigma_\otimes$ with components σ_P and σ_T, shown in Figure 3. It maps values as computed by σ_P and σ_T (which correspond to states of the corresponding factors) to values corresponding to the product states in the product system (we annotated states of $\Theta_\otimes$ in Figure 1 with corresponding numbers from the value set of $\sigma_\otimes$). For example, for the assignment $\{v_P \mapsto 2, v_T \mapsto 1\}$, $\sigma_\otimes$ first computes values 2 and 1 from its components and then retrieves the result 5 from its table, which indeed corresponds to the state TB.

A *prune transformation* combines unreachable states or states from which no goal can be reached of some factor Θ of F into an isolated dummy state, removing all corresponding transitions. In the corresponding F2N mapping, this change is reflected like for shrink transformations.

Finally, a *label reduction transformation* applies a label mapping λ to the label set L with label costs c of F, mapping it to some (usually smaller) label set L' with label costs c'. In all factors of the transformed factored transition system, all transitions $s \xrightarrow{\ell} t$ must be replaced by transitions $s \xrightarrow{\lambda(\ell)} t$.

Algorithm 2 The main loop of the merge-and-shrink framework as implemented in Fast Downward.

```
1: while $|F| > 1$ do
2:     $\Theta_1, \Theta_2, \leftarrow$ SELECT($F$)
3:     $\langle F, \Sigma, \lambda \rangle \leftarrow$ COMPOSETF(LABELRED($F$))
4:     $\langle F, \Sigma, \lambda \rangle \leftarrow$ COMPOSETF(SHRINK($F, \Theta_1, \Theta_2$))
5:     $\langle F, \Sigma, \lambda \rangle \leftarrow$ COMPOSETF(MERGE($F, \Theta_1, \Theta_2$))
6:     $\langle F, \Sigma, \lambda \rangle \leftarrow$ COMPOSETF(PRUNE($F, \Theta_\otimes$))
7: end while
```

Label costs must not increase to guarantee admissibility, i.e., $c'(\ell') := \min_{\ell \in \lambda^{-1}(\ell')}(c(\ell))$ for all $\ell' \in L'$. An example label reduction for $F(\Pi)$ of the example task reduces U-A, U-B, L-A, L-B to a new label X. In the transformed factors, transitions are relabeled accordingly, collapsing identical transitions. For example, $0 \xrightarrow{\text{L-A}} 0$ and $0 \xrightarrow{\text{U-A}} 0$ of Θ_T are combined into $0 \xrightarrow{\text{X}} 0$ in the transformed factor.

Sievers, Wehrle, and Helmert (2014) provide a full characterization of label reductions, describing sufficient and necessary conditions under which they are exact. *Exact* transformations preserve perfect heuristic values and are thus desirable. The exclusive condition used in previous work to compute exact generalized label reductions is based on Θ-combinability. Consider a factored transition system F with labels L and some factor Θ of F. Labels $\ell, \ell' \in L$ are *locally equivalent* in Θ if they label the same transitions in Θ, and they are Θ-*combinable* in F if they are locally equivalent in all factors $\Theta' \neq \Theta$ of F. A label reduction that only combines Θ-combinable labels of the same cost for some Θ of F is exact. In $F(\Pi)$ of the example task, labels DR-A-B and DR-B-A are Θ_T-combinable because they both induce the same self-looping transitions in the only other factor Θ_P.

We conclude this section with a description of the concrete instantiation of the merge-and-shrink main loop in Fast Downward (Helmert 2006). Algorithm 2 shows pseudocode.[1] The layout is based on the decision to perform one merge transformation in each iteration. Selecting the factors Θ_1, Θ_2 to merge is thus the first step (line 2), followed by possibly applying a label reduction (line 3), shrinking one or both factors that will be merged in order to satisfy a given size limit imposed on transition systems (line 4), performing the merge transformation by replacing Θ_1 and Θ_2 by $\Theta_\otimes$ (line 5), and finally pruning $\Theta_\otimes$ (line 6). Recall that performing a transformation means to compose it with the previous transformation as described in line 5 of Algorithm 1, for which COMPOSETF is a short-hand notation. This algorithm still has parameters, namely the *transformation strategies* that need to specify how to compute a specific transformation. For example, a shrink strategy needs to specify how to compute an abstraction for a given factor.

Computation of Exact Label Reductions

In the following, we detail out the computation of exact label reductions based on Θ-combinable labels (combinable labels for short) that Sievers, Wehrle, and Helmert (2014) only briefly mentioned. We begin with the following important observations. The relation on locally equivalent labels for a single factor is transitive; it is even an equivalence relation, called *local label equivalence relation* in the following. However, the combinable relation is the union over these local equivalence relations, and this union relation is *not* transitive. This means that if ℓ_1 and ℓ_2 are combinable and ℓ_2 and ℓ_3 are combinable for different factors of a factored transition system, this does *not* imply that ℓ_1 and ℓ_3 are combinable for any factor. Intuitively, after merging ℓ_1 and ℓ_2, the combined label is in general not locally equivalent with ℓ_3 in all (but one) factors because we now have transitions with the new label that originated from ℓ_1 and have no matching transition for ℓ_3. An immediate consequence of this observation is that if we want to apply all possible label reductions by computing combinable labels for all factors until there are no more such combinable labels, the result depends on the order in which we consider the factors.

Algorithm 3 Fixed point label reduction algorithm.

```
Input: Factored transition system $F$ with labels $L$ and label
    costs $c$; Order $O$ on factors of $F$.
Output: Transformed factored transition system $F$.
 1: function LABELREDUCTION($F$)
 2:     #UnsuccIt $\leftarrow 0$
 3:     while #UnsuccIt $< |F|$ do
 4:         $\Theta \leftarrow$ NEXT($F', O$)
 5:         eqRel $\leftarrow$ $\Theta$-COMBINABLEREL($F, \Theta$))
 6:         if eqRel is trivial then
 7:             #UnsuccIt $\leftarrow$ #UnsuccIt $+ 1$
 8:         else
 9:             $\lambda, c' \leftarrow$ LABELMAPPING($L$, eqRel)
10:             $F \leftarrow$ APPLY($F, \lambda, c'$)
11:             #UnsuccIt $\leftarrow 0$
12:         end if
13:     end while
14:     return $F$
15: end function
16: function $\Theta$-COMBINABLEREL($F, \Theta \in F$)
17:     eqRel $\leftarrow$ UNIVEQREL($L$)
18:     for $\Theta' \in F$ with $\Theta' \neq \Theta$ do
19:         eqRel $\leftarrow$ REFINE(eqRel, EQREL($\Theta', L$))
20:     end for
21:     return eqRel
22: end function
```

Due to the above observations, Fast Downward implements the fixed point algorithm illustrated in Algorithm 3 for computing exact label reductions based on Θ-combinability. It keeps track of the number of iterations in which no more labels could be combined (line 2) and reaches a fixed point if the number of such unsuccessful iterations equals the number of factors in the given factored transition system F, i.e., if there are no more combinable labels in any of the factors of F (line 3).

In each iteration, the algorithm chooses the next factor $\Theta \in F$ according to some user-specified *order* O in which

[1]In the implementation, pruning can be applied to all atomic factors once before the main loop.

factors of F should be considered (line 4). For the chosen factor, the algorithm computes the Θ-combinable equivalence relation using the method Θ-COMBINABLEREL. If the result is trivial in the sense that there are no Θ-combinable labels, the iteration counts as unsuccessful (line 7). Otherwise, the algorithm turns the equivalence relation into a label mapping λ (line 9) that maps all labels within a single equivalence class to a new label ℓ, setting $c'(\ell)$ to the minimum cost of all labels mapped to ℓ by λ. (To obtain an exact label reduction, we have to further split each equivalence class according to label costs.) Applying the label mapping to the factored transition system (line 10) means to re-label all transitions labeled by labels that have been reduced, potentially combining transitions. Finally, the algorithm continues with the next iteration or terminates, returning the transformed factored transitions system (line 14).

The function Θ-COMBINABLEREL for computing Θ-combinable labels starts with the universal label equivalence relation in which all labels are considered equal (line 17). It then subsequently refines this equivalence relation through the local label equivalence relations of all factors other than Θ (line 19). (As observed above, the order on the factors of F does *not* matter for this iterative refinement.)

We remark that the merge-and-shrink framework computes *single* transformations at each step and composes them (cf. lines 4–5 in Algorithm 1), which does not directly fit the fixed point algorithm that computes a sequence of label reductions. Hence, in this algorithm, each call to APPLY(F, λ, c') has to be understood as composing the label reduction transformation of the current iteration with the transformation maintained by the merge-and-shrink framework.

Secondly, we remark that the local label equivalence relation of a factor Θ (EQUIVREL(Θ, L)) can be computed in time polynomial in the number of label groups and their transitions in Θ. Refining an equivalence relation A through another one called B (REFINE(A, B)) is possible in time linear in the number of elements (here: labels) of A and B when using suitable data structures such as linked lists.

Finally, to avoid introducing any bias with the selection of an order O, the default order of the implementation in Fast Downward is a random one. This order has been used for producing all results reported in the literature to the best of our knowledge.

Efficient Implementation

In this section, we describe our *optimized* implementation of the merge-and-shrink framework for classical planning. This implementation is publicly available in Fast Downward (Helmert 2006). It has first been used for the results reported by Sievers, Wehrle, and Helmert (2016). All results reported prior to this work used the *previous*, already highly engineered implementation of merge-and-shrink, which Helmert et al. (2014) describe in Section 4.3, extended with a basic implementation of generalized label reduction. In the following, we distinguish clearly between optimizations present already in the previous implementation and those new to our optimized one.

We first remark that our optimized implementation is solely based on an improved representation of transition systems, and that the implementation of factored mappings follows the inductive definition in a straightforward way and is identical in both versions. To represent transition systems, both implementations do not store transitions as an adjacency list as it is commonly done to represent graphs, but rather store all transitions grouped by labels. This allows an efficient application of all merge-and-shrink transformations, as we will see below.

prev	opt
DR-A-B : $\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle\}$	{DR-A-B, DR-B-A} : $\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle\}$
DR-B-A : $\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle\}$	
L-A : $\{\langle 0,2\rangle\}$	{L-A} : $\{\langle 0,2\rangle\}$
U-A : $\{\langle 2,0\rangle\}$	{U-A} : $\{\langle 2,0\rangle\}$
L-B : $\{\langle 1,2\rangle\}$	{L-B} : $\{\langle 1,2\rangle\}$
U-B : $\{\langle 2,1\rangle\}$	{U-B} : $\{\langle 2,1\rangle\}$

Table 1: Representing Θ^P of the example (cf. Figure 2) in the previous (prev) and the optimized (opt) implementation.

The central difference to the previous implementation is that we store *label groups* of locally equivalent labels, disregarding their cost (the cost of a label group is the minimum cost of any participating label), and thus only have to store the transitions of locally equivalent labels once rather than separately for each label. This was not possible in the previous implementation where labels were associated with preconditions and effects of planning operators in order to enable the old, syntax-based theory of label reduction (Helmert et al. 2014). Table 1 shows the representation of Θ^P of the example in the the previous (prev) and optimized (opt) implementation, using the notation $x : y$ to the denote the set y of transitions of a label (prev) or a label group (opt) x.

Due to this more memory-efficient representation of transition systems, we can also represent so-called *irrelevant* labels,[2] i.e., have one group that represents irrelevant labels. The previous implementation could not store transitions of irrelevant labels explicitly due to a prohibitive memory consumption, which led to various error-prone special-casing. In Θ^P of the example, labels DR-A-B and DR-B-A are irrelevant and thus form a single label group (cf. Table 1).

A further, small conceptual difference of the two implementations is that the previous implementation computed pruning transformations as part of shrinking, whereas the optimized implementation only attempts pruning after merging, which is the only place where pruning opportunities can occur. Furthermore, we allow computing g- and h-values depending on which information is required by the transformation strategies, whereas the previous implementation always computed both. Both implementations compute distances of transition systems with Dijkstra's algorithm (Dijkstra 1959). This is the only place where we need an explicit adjacency list representation of transition systems.

We now turn our attention to the different merge-and-shrink transformations, beginning with shrinking, which is

[2]A label l is irrelevant in Θ if for all states s of Θ, it labels exactly one self-looping transition $s \xrightarrow{l} s \in \Theta$. A label l is relevant if it is not irrelevant.

done identically in both implementations. First, the shrink strategy computes a state equivalence relation for the given factor. An example shrink transformation combines states 0 and 1 of Θ^P. We transform this equivalence relation into an explicit state mapping, assigning a consecutive number to each equivalence class. In the example, $0 \mapsto 0$, $1 \mapsto 0$, and $2 \mapsto 1$. Then we use this state mapping for an in-place modification of the factor by going over all transitions and updating their source and target states. This can be done efficiently because we store transitions by labels or label groups, see Table 1. If we used an adjacency list representation of transition systems, we would possibly need to reorder all entries of the list, besides renumbering the entries within successor lists. Finally, we perform an in-place modification of the F2N mapping corresponding to the shrunk factor by by applying the state mapping to its table. In Figure 2, we would relabel the second row of σ_P to read 0, 0, and 1.

When merging two transition systems Θ_1 and Θ_2, the previous implementation considered each label separately and computed its transitions in the product by pair-wise combinations of its transitions in Θ_1 and Θ_2. In the optimized implementation, we do not compute this full product, because this would require to compute the local label equivalence relation from scratch afterwards. Instead, we use the following simple bucket-based approach to directly compute the refinement of the local label equivalence relations `EqRel1` and `EqRel2` of Θ_1 and Θ_2: for each label group x of `EqRel1`, first partition its labels according to `EqRel2`, and then, for each resulting new label group y, compute the product of the transitions of x and of the label group of Θ_2 that corresponds to y. We remark that computing product transitions is facilitated by storing transitions grouped by labels rather than using an adjacency list, which allows to quickly access the transitions of a label or a label group.

Computing the F2N mapping $\sigma_\otimes$ corresponding to the product is straightforward and identical to the previous implementation: it has two components, which are the two F2N mappings of the previous transformation that map to the factors Θ_1 and Θ_2, and it has a table that maps pairs of component states to their product state. Starting from the factored transition system shown in Figure 2, Figures 1 and 3 illustrate the result of a merge transformation.

When pruning, the prune strategy computes a set of to-be-pruned states for the given factor. We then modify the factor in-place by removing these states together with their transitions. The table of the corresponding F2N mapping is updated in-place to map removed states to a special symbol which is evaluated to ∞ by the heuristic. This implementation of pruning is identical to the previous one.

Computing an exact label reduction with Algorithm 3 is favored by the representation that stores the local label equivalence relation of all factors: the loop of the method Θ-COMBINABLEREL can simply use the already computed local label equivalence relations (instead of calling EQUIVREL(Θ', L) in line 19). The previous implementation needed to compute this information on demand and cache it over the iterations of the algorithm.

For efficient refinement of label equivalence relations (REFINE), we store label groups in linked lists as discussed in the description of the algorithm. Computing the label mapping (LABELMAPPING) is straightforward and is done like computing a state mapping from a state equivalence relation when shrinking. Applying the label mapping (APPLY) can be efficiently done as follows. When updating a factor other than Θ, all we have to do is to remove the old labels from their group (they are all in the same group) and to add the new label to it; the transitions remain. Otherwise, when updating Θ, we need to remove the old labels from their groups (potentially different ones) and add a new singleton group for the new label. While doing so, we collect the transitions of all to-be-removed labels and combine them to form the transitions of the new label. If label groups become empty, we remove them together with their transitions. We also need to recompute the costs of all modified groups. This part of label reduction is identical to the previous implementation. In the optimized implementation, we additionally recompute the local label equivalence relation to restore the compact representation of transition systems.

We remark that our optimized implementation also greatly benefits two transformation strategies of the merge-and-shrink toolbox. The state-of-the-art shrink strategy based on bisimulation (Nissim, Hoffmann, and Helmert 2011) is accelerated significantly due to only considering transitions of label groups rather than of individual labels. For the same reason, computing factored symmetries (Sievers et al. 2015) that can be used to enhance merge strategies is faster with the optimized implementation.

Partial Merge-and-Shrink Abstractions

The regular termination criterion of the merge-and-shrink framework is to stop when the transformed factored transition system only contains a single factor. Then we have exactly one abstraction for computing the heuristic. However, nothing prevents us from stopping the algorithm early, ending up with several factors and factored mappings, and hence with several abstractions that induce heuristics.

Paired with the observation that merge-and-shrink based planners fail to finish computing the heuristic in the given time and memory limits in a non-negligible number of cases (151–267 out of 1667 tasks for state-of-the-art-configurations[3]), we will evaluate two simple ways of stopping the merge-and-shrink computation by limiting resource consumption. Both have been used before by Torralba and Hoffmann (2015) to limit the merge-and-shrink computation. However, they did not compute merge-and-shrink heuristics but used the framework as a basis for computing label dominance relations.

The first limit restricts the number of transitions that a factor may have at any time. This is reasonable because this number has a much higher impact on the runtime of merge-and-shrink than the number of states, which is controlled by the shrink strategy: both computing exact label reductions and product systems is dominated by the number of transitions. However, controlling the number of transitions is not possible in a straightforward way with an approach similar to how shrinking controls the number of states because

[3]See rows "# constr" of column "base" of Table 3 or 4.

removing or ignoring transitions can result in inadmissible heuristics. Therefore, we terminate the merge-and-shrink algorithm once the limit is reached by any factor. The second limit restricts the runtime of the merge-and-shrink algorithm, which we allow to be terminated even during the computation of atomic factors.

In both cases, when terminating the merge-and-shrink computation early, we are left with a factored transition system $F = \langle \Theta_1, \dots, \Theta_n \rangle$ and an F2F mapping $\Sigma = \langle \sigma_1, \dots, \sigma_n \rangle$ that contain several elements. Thus, instead of computing $h^{\text{M\&S}}_{\Pi} = h^*_{\Theta}$ for the single remaining factor Θ as usually, for each factor Θ_i, we can compute the *factor heuristic* $h_i(s) = h^*_{\Theta_i}(\sigma_i(s))$ for all states s of the task. The decision we then face is to compute a (merge-and-shrink) heuristic from this set of heuristics.

The first, possibly most obvious variant we consider is to compute the *max-factor heuristic* $h^{\text{mf}}_F = \max_{1 \leq i \leq n} h_i$ that maximizes over all factor heuristics. The second, less expensive alternative is to choose a *single* factor heuristic (h^{sg}_F) and use it as the merge-and-shrink heuristic. We use the following simple rule of thumb for this choice: we prefer the heuristic with the largest estimate for the initial state (rationale: better informed heuristic), breaking ties in favor of larger factors (rationale: more fine-grained abstraction), and choose a random heuristic among all remaining candidates of equal preference. Other choices or tie-breakers are certainly possible, such as choosing the heuristic with the largest average estimate or the heuristic with the largest number of distinguishable estimates. We leave an evaluation of these options as future work.

Experiments

In the following, we first compare the previous to the optimized implementation of merge-and-shrink in Fast Downward and then evaluate partial merge-and-shrink abstractions.[4] The technical setup is the same for all experiments: we use all (optimal) benchmarks of all all IPCs up to 2014, a set comprised of 1667 planning tasks distributed across 57 domains.[5] We limit time to 30 minutes and memory to 3.5 GiB per task, using Downward-Lab (Seipp et al. 2017) for conducting the experiments on a cluster of machines with Intel Xeon Silver 4114 CPUs running at 2.2 GHz.

All results[6] are obtained with the state-of-the-art shrink strategy based on bisimulation with a size limit of 50000. We allow shrinking even if the size limit is not violated to allow exploiting potential perfect shrinking opportunities. We use full pruning of dead states. Label reductions are computed with the fixed point algorithm described earlier. In the first experiment, we can only use the merge strategies that were available already in the previous implementation. This includes two representatives of simple linear strategies, *causal graph goal level (CGGL)* (Helmert, Haslum, and Hoffmann 2007) and *reverse level (RL)* (Nissim, Hoffmann, and Helmert 2011), as well as the non-linear merge strategies *DFP* (Dräger, Finkbeiner, and Podelski 2009; Sievers, Wehrle, and Helmert 2014) and *maximum intermediate abstraction size minimizing (MIASM)* (Fan, Müller, and Holte 2014), the latter using DFP as fallback mechanism (*MIASMdfp* or *Mdfp* for short). In the second experiment, we also use the most recent, state-of-the-art non-linear merge strategies *score-based MIASM* (*sbM*), also called DYN-MIASM, and the strategy based on *strongly connected components* of the causal graph (Sievers, Wehrle, and Helmert 2016), which uses DFP for internal merging (SCCdfp).

Previous and Optimized Implementation

Recall that in this comparison we want to reproduce results of an older implementation of merge-and-shrink in Fast Downward. Since Fast Downward versions that are several years apart usually have huge differences in performance, we integrated the previous implementation of merge-and-shrink into the most recent version of Fast Downward, attempting to keep the required modifications at a minimum. We think that this is the best way to allow for a fair comparison to state-of-the art techniques.

We further remark that the previous implementation uses the first implementation of MIASM that its authors used for their original paper, whereas the optimized implementation uses a re-implementation of MIASM which its authors developed based on a newer version of Fast Downward. Unfortunately, integrating the newer implementation of MIASM into the previous implementation of merge-and-shrink was not possible without major changes. Hence the following comparison of MIASM has to be taken with a grain of salt.

Table 2 shows the comparison of the previous implementation (column prev) against the optimized one (column opt). For each merge strategy (vertical blocks), the table reports, in different rows, the number of solved tasks (coverage), the number of tasks for which the construction of the heuristic finished (# constr), the runtime of the construction in seconds (Constr time), aggregated using the geometric mean first over all tasks and then over all domains, the number of tasks for which the construction ran out of memory (oom) or time (oot), and the number of expansions (rounded to thousands) until the last f-layer, aggregated using the 75th percentile (E 75th perc). The table also includes columns displaying the difference (diff) between opt and prev, which is highlighted in bold if it is favorable for the optimized implementation, and the number of tasks for which opt is better (+) and worse (-) than prev.

We observe that the optimized implementation indeed leads to an improved efficiency of the merge-and-shrink computation: the smaller representation size of transition systems results in fewer tasks for which the computation runs out of memory, which also transfers to a great decrease of construction time on average.[7] This increase of the number of successful heuristic constructions in turn yields better

[4]Code: https://doi.org/10.5281/zenodo.1290524

[5]Suite "optimal strips" from `https://bitbucket.org/aibasel/downward-benchmarks`

[6]Dataset: https://doi.org/10.5281/zenodo.1290644

[7]However, for some tasks with a large number of variables or operators, the (expensive) fixed point computation of label reductions causes the heuristic construction to run out of time or memory in the optimized implementation because it explicitly represents irrelevant labels of each factor. We still prefer the cleaner implementation over an even more efficient computation for those few cases.

	prev	opt	diff	+	-	
Coverage	733	754	**21**	23	2	CGGL
# constr	1387	1467	**80**	85	5	
Constr time	104.79	55.75	**-49.03**	1146	236	
Constr oom	137	57	**-80**	98	18	
Constr oot	140	143	3	22	23	
Exp 75th perc	3117k	3117k	0	16	34	
Coverage	768	774	**6**	9	3	DFP
# constr	1419	1504	**85**	89	4	
Constr time	96.10	59.61	**-36.50**	1193	222	
Constr oom	106	21	**-85**	94	9	
Constr oot	139	142	3	16	17	
Exp 75th perc	1957k	1861k	**-96k**	44	44	
Coverage	778	804	**26**	32	6	MIASMdfp
# constr	1382	1480	**98**	98	0	
Constr time	80.61	59.95	**-20.66**	735	647	
Constr oom	117	68	**-49**	93	43	
Constr oot	161	119	**-42**	81	34	
Exp 75th perc	1048k	1047k	**-1k**	142	165	
Coverage	756	773	**17**	17	0	RL
# constr	1433	1515	**82**	85	3	
Constr time	80.09	53.70	**-26.39**	1148	282	
Constr oom	90	16	**-74**	82	8	
Constr oot	141	136	**-5**	15	8	
Exp 75th perc	1879k	2219k	340	27	27	

Table 2: Comparison of the previous against the optimized implementation of merge-and-shrink.

coverage of all configurations. The very similar number of expansions also confirms that the difference of the implementations is of a pure optimization nature.

Partial Merge-and-Shrink Abstractions

We now evaluate partial merge-and-shrink abstractions, i.e., heuristics computed based on the set of factor heuristics that remain when terminating the merge-and-shrink computation early. Recall that we use two approaches for computing the final heuristic given the set of factor heuristics: choosing a single one as described (h^{sg}), or maximizing over all (h^{mf}). We first evaluate imposing a limit on the number of transitions that, once hit by any factor, terminates the computation. Table 3 shows results for limits of 2, 5, and 10 millions of transitions (t2m, t5m, t10m).

With the exception of CGGL, limiting the number of transitions does not increase coverage. The reason comes apparent when comparing the number of successful heuristic constructions and the heuristic quality in terms of expansions: only for small limits on the number of transitions does the limit actually trigger frequently enough to effect a significant increase of successful heuristic constructions, but at the same, the number of expansions also increases significantly. For larger limits, the results are close to that of the baseline. We conclude that limiting the number of transitions does not serve to reliably stop the merge-and-shrink computation in those cases where it would be needed, because a low limit may unnecessarily reduce the quality of the computed heuristic and a larger limit may not stop the

		h^{sg}			h^{mf}			
	base	t2m	t5m	t10m	t2m	t5m	t10m	
Coverage	754	745	757	766	748	757	**767**	CGGL
# constr	1466	**1547**	1533	1529	**1547**	1532	1529	
E 75th perc	**3170k**	4378k	**3170k**	**3170k**	4378k	**3170k**	**3170k**	
Coverage	**775**	735	764	770	740	766	772	DFP
# constr	1506	1530	1516	1513	**1531**	1517	1515	
E 75th perc	**1084k**	2687k	**1084k**	**1084k**	2584k	**1084k**	**1084k**	
Coverage	**804**	775	791	801	775	791	801	Mdfp
# constr	1482	**1515**	1493	1490	**1515**	1493	1490	
E 75th perc	**1367k**	2268k	**1367k**	**1367k**	2124k	**1367k**	**1367k**	
Coverage	**773**	745	759	767	744	758	767	RL
# constr	1516	**1525**	1514	1518	1523	1517	1516	
E 75th perc	**1763k**	2467k	**1763k**	**1763k**	2421k	**1763k**	**1763k**	
Coverage	**802**	787	797	**802**	792	798	**802**	sbM
# constr	1400	**1453**	1422	1414	1452	1424	1417	
E 75th perc	**734k**	1569k	**734k**	**734k**	1474k	**734k**	**734k**	
Coverage	**813**	778	801	811	778	801	811	SCCdfp
# constr	1506	**1532**	1515	1514	**1532**	1515	1512	
E 75th perc	**1217k**	3631k	**1217k**	**1217k**	3338k	**1217k**	**1217k**	

Table 3: Comparison of the baseline against two versions of partial merge-and-shrink, using different limits on the number of transitions.

construction before running out of time.[8]

A second conclusion we draw from Table 3 is that there is no significant difference between h^{mf} and h^{sg}. While h^{mf} theoretically dominates any factor heuristic by definition, evaluating the former can be slightly more expensive. Furthermore, in scenarios where there is one large factor and many small (e.g., atomic) factors in the end, the large one likely dominates the others, and thus h^{sg} is equally informed as h^{mf}. This scenario always occurs with linear merge strategies because they maintain only one non-atomic factor.

We now evaluate adding a time limit to the computation. Table 4 shows the results for limits of 450s, 900s, and 1350s. As expected, this is a very effective measure for greatly increasing the number of successful heuristic constructions, which also directly transfers to a significant increase in coverage of all configurations, with $450s$ and $900s$ achieving slightly better results than 1350s. In contrast to limiting the number of transitions, stopping the computation early does *not* affect the heuristic quality at all. The likely reason is that with limiting the time, we catch precisely those tasks for which the construction otherwise does not terminate or terminates too late for a successful search. Tasks which we can already solve without imposing a time limit (base) usually require a rather short construction time, and therefore limiting the time to 900s or more does not stop the heuris-

[8]We also tested to only *exclude* a factor from further consideration if it violates the transition limit rather than aborting the computation. This is only possible with merge strategies such as DFP and sbMIASM that are able to base their selection on a subset of factors. In this variant, coverage decreased less compared to aborting the computation, but still did not increase for any tested limit.

		h^{sg}			h^{mf}			
	base	450s	900s	1350s	450s	900s	1350s	
Coverage	754	**790**	789	784	789	789	784	CGGL
# constr	1466	**1599**	1586	1571	1598	1585	1568	
E 75th perc	**3117k**	**3117k**	**3117k**	**3117k**	**3117k**	**3117k**	**3117k**	
Coverage	775	805	805	801	805	**806**	801	DFP
# constr	1506	**1622**	1620	1608	**1622**	1620	1604	
E 75th perc	**1739k**	**1739k**	**1739k**	**1739k**	**1739k**	**1739k**	**1739k**	
Coverage	804	**835**	832	827	**835**	833	826	Mdfp
# constr	1482	**1595**	1591	1568	1592	1590	1566	
E 75th perc	**1403k**	**1403k**	**1403k**	**1403k**	**1403k**	**1403k**	**1403k**	
Coverage	773	796	796	796	796	**797**	792	RL
# constr	1516	**1626**	**1626**	1617	**1626**	**1626**	1616	
E 75th perc	**2221k**	**2221k**	**2221k**	**2221k**	**2221k**	**2221k**	**2221k**	
Coverage	802	835	835	835	**836**	**836**	835	sbM
# constr	1400	**1637**	1628	1616	1636	1628	1615	
E 75th perc	**1342k**	1368k	**1342k**	**1342k**	1368k	**1342k**	**1342k**	
Coverage	813	844	844	840	844	**845**	840	SCCdfp
# constr	1506	**1622**	1620	1608	**1622**	1620	1610	
E 75th perc	**1860k**	**1860k**	**1860k**	**1860k**	**1860k**	**1860k**	**1860k**	

Table 4: Comparison of the baseline against two versions of partial merge-and-shrink, using different time limits.

tic computation early and hence does not reduce heuristic quality in these cases. Regarding the difference between h^{sg} and h^{mf}, it is again very small, showing no clear dominance trend.

Conclusions

In this paper, we provided an extensive theoretic and algorithmic description of the merge-and-shrink framework based on generalized label reduction. Furthermore, we described an efficient implementation in Fast Downward, highlighting improvements compared to a previous implementation. We also discussed partial merge-and-shrink abstractions and proposed imposing two simple limits on the algorithm for computing such abstractions. Our experimental evaluation showed that the optimized implementation clearly improves over the previous one. While limiting the number of transitions did not result in partial abstractions that are informed enough, imposing a time limit significantly pushed the performance of merge-and-shrink planners.

We think that partial merge-and-shrink abstractions have further potential. In particular, we would like to come up with a way of deterministically computing them rather than imposing a time limit which does not allow full reproducibility due to computational fluctuations. Furthermore, we want to investigate other alternatives for computing a merge-and-shrink heuristic given a set of partial factor heuristics, such as using cost partitioning techniques.

Acknowledgments

We thank the anonymous reviewers who helped improving the clarity of the paper. This work was supported by the European Research Council as part of the project "State Space Exploration: Principles, Algorithms and Applications".

References

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

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2009. Directed model checking with distance-preserving abstractions. *Software Tools for Technology Transfer* 11(1):27–37.

Fan, G.; Holte, R.; and Müller, M. 2018. Ms-lite: A lightweight, complementary merge-and-shrink method. In *Proc. ICAPS 2018*.

Fan, G.; Müller, M.; and Holte, R. 2014. Non-linear merging strategies for merge-and-shrink based on variable interactions. In *Proc. SoCS 2014*, 53–61.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM* 61(3):16:1–63.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS 2007*, 176–183.

Helmert, M.; Röger, G.; and Sievers, S. 2015. On the expressive power of non-linear merge-and-shrink representations. In *Proc. ICAPS 2015*, 106–114.

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

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *Proc. IJCAI 2011*, 1983–1990.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

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

Sievers, S.; Wehrle, M.; Helmert, M.; Shleyfman, A.; and Katz, M. 2015. Factored symmetries for merge-and-shrink abstractions. In *Proc. AAAI 2015*, 3378–3385.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized label reduction for merge-and-shrink heuristics. In *Proc. AAAI 2014*, 2358–2366.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An analysis of merge strategies for merge-and-shrink heuristics. In *Proc. ICAPS 2016*, 294–298.

Sievers, S. 2017. *Merge-and-shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation*. Ph.D. Dissertation, University of Basel.

Torralba, Á., and Hoffmann, J. 2015. Simulation-based admissible dominance pruning. In *Proc. IJCAI 2015*, 1689–1695.