Non-Orthogonal Factored Transition Systems for Merge-and-Shrink

Luka Obser

October 10, 2023

Abstract

Planning tasks can be used to describe many real world problems of interest. Solving those tasks optimally is thus an avenue of great interest. One established and successful approach for optimal planning is the merge-and-shrink framework, which decomposes the task into a factored transition system. The factors initially represent the behaviour of one state variable and are repeatedly combined and abstracted. The solutions of these abstract states is then used as a heuristic to guide search in the original planning task. Existing merge-and-shrink transformations keep the factored transition system orthogonal, meaning that the variables of the planning task are represented in no more than one factor at any point. In this thesis we introduce the clone transformation, which duplicates a factor of the factored transition system, making it non-orthogonal. We test two classes of clone strategies, which we introduce and implement in the Fast Downward planning system and conclude that, while theoretically promising, our clone strategies are practically inefficient as their performance was worse than state-of-the-art methods for merge-and-shrink.

Acknowledgements

I want to thank everyone who has made it possible for me to write this thesis. First, I want to thank Dr. Gabriele Röger for the opportunity of writing this thesis. I also want to thank my two supervisors Dr. Silvan Sievers and Remo Christen for their endless patience and support during the last seven months. Lastly, I want to thank my friends, family, and especially my partner Rebecca for their constant support, patience, and words of encouragments throughout my studies.

Contents

Chapter 1

Introduction

Planning has been one of the central areas of artificial intelligence research since the birth of the field. The idea of tasking a machine with finding the specific actions which may lead to a desired result has been mentioned as early as the first time the term "artificial intelligence" was proposed by McCarthy *et al.* [1] in the summer of 1955.

In short, planning is the task of finding a plan, a sequential set of actions, which will lead us from one world state to another. An intuitive example is route planning, where we start at point A, for example our home and wish to drive to a specific destination, such as the sea. The actions of our plan then specify how to act at each intersection we encounter. In this example, our goal state is characterized by having reached a beach.

The main problem we encounter when trying to solve planning tasks is one we are already familiar with from when we ourselves try to make plans. Most problems are simply too complex for an exhaustive search to be feasible, as there are usually too many different states to keep track of, even after only a handful of decisions. Considering every direction at every intersection from here to the sea will inevitably lead to us spending the summer planning a vacation instead of taking it. Depending on how landlocked we are, we might not even make it to the beach by next year simply because we are tracing side streets and alleys on our map in some big city which is only a few kilometers closer to the ocean than we are now.

This need for guidance in complex problems leads us to heuristics. Heuristics are essentially rules of thumb, methods which aim to be easy to compute and which may tell us whether we are making progress towards our goal or not. A heuristic for our task of planning a route to the beach could be to take a ruler and to measure the beeline from a point to the ocean. We will then consider the roads which will bring us to points closer to the ocean first. While this may not always work, as it could lead us down a dead end, we will be able to get to the beach quicker than when blindly tracing all streets.

Good heuristics are hard to come by, but once obtained, they can greatly increase the efficiency of informed search algorithms such as A* [2], which yields optimal solutions, provided the heuristic fulfills a few simple conditions.

The last two decades saw a huge increase in interest for heuristic search and this trend does not seem to fade. The method we will investigate and expand upon in our work are merge-and-shrink heuristics. Originally a method developed for model checking by Dräger *et al.* [3], the method was quickly adapted for optimal planning in 2007 by Helmert *et al.* [4]. More recently, Sievers and Helmert [5] combined previous work on merge-and-shrink into a modular framework which interprets the method in a more general context.

In our thesis we expand upon this framework by introducing a clone transformation. This allows the algorithm to consider non-orthogonal factored transition systems. This makes it possible to express more nuanced relations between variables, which in turns opens up the possibility of more accurate heuristics. Clone transformations in the context of merge-and-shrink heuristics have only been mentioned as a side note [6], or as a possible venue of future work [7]. Related notions exist for factored transition systems in general [8].

We will begin by providing the necessary background before formally defining the clone transformation in the terminology of Sievers and Helmert's framework [5]. We then consider two classes of strategies which decide when and what to clone. While, in theory, we may obtain better results due to the increased flexibility of non-orthogonal factored transition systems, our experiments have shown

that it is difficult to automate the choice of which factor to clone when. As such, the performance of our strategies was inferior to state-of-the-art methods we used as comparison. Although the performance of our cloning strategies was unable to keep up with the state of the art, we see promise in cloning more selectively. This is especially promising in combination with cost partitioning, which is already an established improvement for merge-and-shrink heuristics as shown by Sievers *et al.* [9].

Chapter 2

Background

This chapter provides definitions of concepts and methods referenced throughout the thesis. Since this thesis aims to be self-contained, many of the more basic definitions might already be familiar to some readers. For those, we suggest to treat this chapter as a glossary to look up notations or to freshen up on details if questions arise during the reading of the subsequent chapters.

Unless otherwise specified, the definitions on these pages are taken directly from the journal paper on the merge-and-shrink framework [5], or from the introductory lectures on artificial intelligence [10] and planning and optimization [11] taught at the University of Basel.

Before jumping into definitions of planning tasks, let us recount some basics regarding functions as well as partitions of sets.

Definition 1 *(Inverse of a Function; based on [5]). Let $f\colon X \to Y$ be a function. For $y \in Y$, the inverse of f is defined as the set $f^{-1}(y) = \{x \in X \mid f(x) = y\}$. We extend this definition to subsets $Y' \subseteq Y$ by defining $f^{-1}(Y') = \{x \in X \mid f(x) \in Y'\}$.*

Definition 2 *(Composition of Functions; based on [5]). Let $f\colon X \to Y$ and $g\colon Y \to Z$ be two functions. The composition of f and g, is the function $h\colon X \to Z$, where $h = g \circ f$, such that $h(x) = g(f(x))$.*

Definition 3 *(Partition). Let X be a non-empty set. The partition of X is a set $P = \{A_1, \ldots, A_n\}$, where $A_1, \ldots, A_n$ are non-empty subsets of X, such that for all elements $x \in X$, x is contained in exactly one element of P.*

2.1 Classical Planning as Heuristic Search

We start with the definition of planning task. For this, we need to get a few technicalities out of the way so let us first define variable spaces and assignments.

Definition 4 *(Variable Space; based on [5]). A variable space is a tuple $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ of n variables, each with a finite domain of arbitrary values ($dom(v)$).*

Throughout our definitions we will oftentimes treat variable spaces as sets where convenient. To allow this we assume that each variable has a separate identity, that is, $\mathcal{V}$ does not contain duplicates. We may thus write, for example, $v \in \mathcal{V}$ to express that $\mathcal{V}$ contains the variable v. The reason for defining variable spaces as tuples is in order to retain a total order on the variables contained which helps in expressing algorithms over variable spaces in an unambiguous way.

Definition 5 *(Assignment, Partial Assignment, Consistent Assignment; based on [5]). Let $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ be a variable space. A partial assignment for $\mathcal{V}$ is a function α defined on $V \subseteq \mathcal{V}$, that maps each element $v \in V$ to some element in its domain $dom(v)$. We write $\alpha[v]$ for this element of $dom(v)$. We write $vars(\alpha)$ to denote the set of variables on which α is defined. A partial assignment with $vars(\alpha) = \mathcal{V}$ is called an assignment. Two partial assignments α and α' are consistent if $\alpha[v] = \alpha'[v]$ for all $v \in vars(\alpha) \cap vars(\alpha')$. We write $\mathcal{A}(\mathcal{V})$ for the set of all assignments for $\mathcal{V}$.*

With these definitions out of the way we can now formally define a planning task.

Definition 6 *(Planning Task; based on [5]). A planning task is a tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$. *$\mathcal{V}$ denotes a variable space whose variables are called state variables. (Partial) assignments of $\mathcal{V}$ are called (partial) states. $\mathcal{O}$ is a finite set of operators, where each operator $o \in \mathcal{O}$ has associated partial states for the precondition ($pre(o)$), and for the effect ($eff(o)$). Each operator also has a non-negative cost ($cost(o)$). The state s_I is called the initial state and s_G is a partial state called the goal.*

One common method of interpreting planning tasks are transition systems. Transition systems offer an intuitive graph based representation of planning tasks.

Definition 7 *(Transition System; based on [5]). A transition system is a tuple* $\Theta = \langle S, L, c, T, S_I, S_G \rangle$. *$S$ is a finite set of states and L is a finite set of transition labels. $c\colon L \to \mathbb{R}_0^+$ is a label cost function which maps each transition label to a non negative cost. $T \subseteq S \times L \times S$ is a set of labeled transitions between states. $S_I \subseteq S$ is the set of initial states (which usually consists of only one state), and $S_G \subseteq S$ is the set of goal states.*

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ induces a transition system $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ which can be seen as a directed graph where the vertices are the states $S = \mathcal{A}(\mathcal{V})$ and the arcs are the transitions $\langle s, l, s' \rangle = t \in T$, where s and s' denote predecessor and successor and $l \in L$ corresponds to an operator $o \in \mathcal{O}$. The label cost function c keeps track of the costs of the operators ($cost(o)$) associated with their labels, that is, it assigns weights to the arcs. The concepts of initial and goal states are equivalent in semantics but different in syntax since planning tasks keep track of states via (partial) assignments rather than sets.

Definition 8 *(Paths, Costs, Plans; based on [5]). Let* $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ *be a transition system. We write $s \xrightarrow{l} s'$ to denote a transition $\langle s, l, s' \rangle \in T$ from s to s' with label l. We may write $s \xrightarrow{l} s' \in \Theta$ rather than $s \xrightarrow{l} s' \in T$ and $s \in \Theta$ for $s \in S$ whenever Θ is not specified further.*

A path from $s \in S$ to $s' \in S$ is a sequence $\pi = \langle t_1, \ldots, t_n \rangle$ of transitions such that there exist states $s = s_0, \ldots, s_n = s'$ with $t_i = (s_{i-1} \xrightarrow{l_i} s_i) \in T$ for all $1 \leq i \leq n$. As a special case, the empty path $\langle\rangle$ is a path from s to s for all $s \in S$. The cost of a path $c(\pi)$ is the accumulated cost of the labels of the transitions, i.e., $c(\pi) = \sum_{i=1}^{n} c(l_i)$. We allow empty paths $\pi = \langle\rangle$ iff $s = s'$. The cost of the empty path is always 0.

A path from some state s to some goal state $s' \in S_G$ is also called an s-plan. An s-plan for some initial state $s \in S_I$ is also called a plan. If a plan (or s-plan) has minimal cost among all plans (or s-plan), it is called optimal.

We will also consider the causal graph of a planning task, as defined by Knoblock [12], later on. The causal graph represents dependencies and influences variables have on one another.

Definition 9 *(Causal Graph of a Planning Task; based on [12]). Let* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ *be a planning task. We call $CG(\Pi)$ the causal graph of Π. The causal graph is a directed graph with vertices $V = \mathcal{V}$ and arcs A. There is an arc between variables u and v iff $u \neq v$ and there exists an operator $o \in \mathcal{O}$ such that $v \in vars(eff(o))$, and either $u \in vars(pre(o))$ or $u \in vars(eff(o))$.*

In natural language, a causal graph has two types of edges. The first being "*pre* $\rightarrow$ *eff*" edges, which are from variable u to variable v if there is an operator that has u in its precondition and v in its effect. This means that there is an operator that affects v which is dependent on variable u. The other types of edges are called "*eff* $\rightarrow$ *eff*" edges, which exist if there is an operator that affects both u and v.

As mentioned in the introduction, one central problem in solving transition systems induced by planning tasks is that these transition systems, while highly intuitive, are usually very large in size. If we consider a "small" planning task with 10 variables, each of which have a domain of size 10, we are already left with 10'000'000'000 different possible states. Granted, many of these may never be reached from the initial state, but it is nonetheless a considerable challenge to find a path through this system, let alone an optimal one. As such, a successful search algorithm would be to greedily choose to go to the successor state of our current (or initial) state which is closest to the goal. But to know the exact distance to the goal we would need to know the path to the goal which is what is so hard, that

is, infeasible to compute to begin with. So, rather than computing exact distances we use heuristics. Heuristics aim to approximate the actual distance to the goal, just like our example of considering the beeline to approximate distances on a map.

Definition 10 *(Heuristic; based on [5]). Let Θ be a transition system with states S and label cost function c. A heuristic for Θ is a function $h_\Theta \colon S \to \mathbb{R}_0^+ \cup \infty$. A heuristic h_Θ is called*

- *perfect if $h_\Theta(s) = h^*_\Theta(s)$ for all states s of Θ, where $h^*_\Theta(s)$ is the cost of an optimal s-plan or ∞ if no s-plan exists.*
- *goal-aware if $h_\Theta(s) = 0$ for all goal states s of Θ.*
- *safe if $h_\Theta(s) = \infty$ for all states s of Θ for which no s-plan exists.*
- *consistent if $h_\Theta(s) \leq c(l) + h_\Theta(t)$ for all transitions $s \xrightarrow{l} t \in \Theta$*
- *admissible if $h_\Theta(s) \leq h^*_\Theta(s)$ for all states s of Θ.*

We may denote heuristics by h instead of h_Θ if the transition system is clear from context and we call the values produced by a heuristic for state $s \in S$ "heuristic estimate" or "heuristic value" for s.

The merge-and-shrink heuristic, which is the central topic of this thesis, is part of a class of heuristics called abstraction heuristics. Abstraction heuristics are based on the idea of solving a simplified version of the transition system, called the abstract transition system, and to use the cost of the solutions of the abstract system as a heuristic value for the original, also called concrete, transition system.

An additional advancement in heuristic search came from being able to combine multiple heuristics in an additive way using cost partitioning [13] [14].

Definition 11 *(Cost Partitioning; based on [11]). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ be a planning task. A cost partitioning for Π is a tuple $\langle cost_1, \ldots, cost_n \rangle$, where*

- *$cost_i \colon \mathcal{O} \to \mathbb{R}_0^+$ for $1 \leq i \leq n$ and*
- *$\sum_{i=1}^n cost_i(o) \leq cost(o)$ for all $o \in \mathcal{O}$.*

The cost partitioning induces a tuple $\langle \Pi_1, \ldots, \Pi_n \rangle$ of planning tasks, where each Π_i is identical to Π except that the cost of each operator o is $cost_i(o)$.

Cost partitioning creates several slightly modified versions of a planning task. The solution costs of these may then be added up to obtain an admissible heuristic which may be better than using only the solution cost of one of the modified tasks. In the case of abstractions, we may thus use cost partitioning to combine heuristics obtained from multiple abstractions of the same original planning task in an admissible way.

For our experiments later on we will use saturated cost partitioning [15], which computes a cost partitioning quickly by iterating through each abstraction. For each abstraction, the cost function is then adapted such that all operators are assigned the minimum costs necessary to preserve the heuristic values of all states. The costs assigned to those operators are then subtracted from the original costs of the operators before the next abstraction is being assigned as much of the remaining costs as is necessary to best preserve its heuristic values. Thus, the abstraction chosen first will yield its exact heuristic values while those chosen later while likely not.

2.2 The Merge-and-Shrink Framework

The main idea of the merge-and-shrink framework lies in transformations of transition systems. Transformations define mappings of states and labels which allow us to transform one transition system into another, which is ideally simpler to solve or more compact while having plans of the same cost.

Definition 12 *(Transformation; based on [5]). Let Θ be a transition system with states S and labels L. Let Θ' be a transition system with states S' and labels L'. $\tau = \langle \Theta', \sigma, \lambda \rangle$ is called a transformation of Θ into Θ', where Θ' is called the transformed transition system, $\sigma \colon S \to S'$ is called the state mapping and $\lambda \colon L \to L'$ is called the label mapping. Θ is called the original transition system of τ. σ and λ may be partial functions.*

Transformations induce heuristics.

Definition 13 *(Heuristic Induced by a Transformation; based on [5]). Let $\tau = \langle \Theta', \sigma, \lambda \rangle$ be a transformation of a transition system Θ into transition system Θ'. The heuristic for Θ induced by τ is called h^τ_Θ or h^τ for short. It is defined as $h^\tau_\Theta(s) = h^*_{\Theta'}(\sigma(s))$ for all $s \in dom(\sigma)$, and $h^\tau_\Theta(s) = \infty$ for all other states $s \in \Theta$.*

We will now introduce properties a transformation may have.

Definition 14 *(Properties of Transformations; based on [5]). Let $\tau = \langle \Theta', \sigma, \lambda \rangle$ be a transformation of a transition system $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ into transition system $\Theta' = \langle S', L', c', T', S'_I, S'_G \rangle$. The following list defines properties that τ may have, along with a short-hand name for each property. For example, we say that τ satisfies $\mathbf{CONS_S}$ if τ is state-conservative.*

- $\mathbf{CONS_S}$ *τ is state-conservative if $dom(\sigma) = S$, that is, σ is a total function.*
- $\mathbf{CONS_L}$ *τ is label-conservative if $dom(\lambda) = L$, that is, λ is a total function.*
- $\mathbf{CONS_C}$ *τ is cost-conservative if $\forall l \in L \colon l \in dom(\lambda) \rightarrow c'(\lambda(l)) \leq c(l)$.*
- $\mathbf{CONS_T}$ *τ is transition-conservative if $\forall s \xrightarrow{l} t \in T \colon s \in dom(\sigma) \wedge t \in dom(\sigma) \wedge l \in dom(\lambda) \rightarrow \sigma(s) \xrightarrow{\lambda(l)} \sigma(t) \in T'$.*
- $\mathbf{CONS_I}$ *τ is initial-state-conservative if $\forall s \in S_I \colon s \in dom(\sigma) \rightarrow \sigma(s) \in S'_I$.*
- $\mathbf{CONS_G}$ *τ is goal-state-conservative if $\forall s \in S_G \colon s \in dom(\sigma) \rightarrow \sigma(s) \in S'_G$.*
- $\mathbf{IND_S}$ *τ is state-induced if σ is sujective, that is, if $\forall s' \in S' \exists s \in S \colon s \in \sigma^{-1}(s')$.*
- $\mathbf{IND_L}$ *τ is label-induced if λ is surjective, that is, if $\forall l' \in L' \exists l \in L \colon l \in \lambda^{-1}(l')$.*
- $\mathbf{IND_C}$ *τ is cost-induced if $\forall l' \in L' \exists l \in L \colon l \in \lambda^{-1}(l') \wedge c(l) = c'(l')$.*
- $\mathbf{IND_T}$ *τ is transition-induced if $\forall s' \xrightarrow{l'} t' \in T' \exists s \xrightarrow{l} t \in T \colon s \in \sigma^{-1}(s') \wedge t \in \sigma^{-1}(t') \wedge l \in \lambda^{-1}(l')$.*
- $\mathbf{IND_I}$ *τ is initial-state-induced if $\forall s' \in S'_I \exists s \in S_I \colon s \in \sigma^{-1}(s')$.*
- $\mathbf{IND_G}$ *τ is goal-state-induced if $\forall s' \in S'_G \exists s \in S_G \colon s \in \sigma^{-1}(s')$.*
- $\mathbf{REF_C}$ *τ is cost-refinable if $\forall l' \in L' \forall l \in \lambda^{-1}(l') \colon c(l) = c'(l')$.*
- $\mathbf{REF_T}$ *τ is transition-refinable if $\forall s' \xrightarrow{l'} t' \in T' \forall s \in \sigma^{-1}(s') \exists s \xrightarrow{l} t \in T \colon t \in \sigma^{-1}(t') \wedge l \in \lambda^{-1}(l')$.*
- $\mathbf{REF_G}$ *τ is goal-state-refinable if $\forall s' \in S'_G \forall s \in \sigma^{-1}(s') \colon s \in S_G$.*

Based on these basic properties, we define the following derived properties. In general, $\mathbf{A} = \mathbf{B} + \mathbf{C}$ means that τ has property $\mathbf{A}$ if it has properties $\mathbf{B}$ and $\mathbf{C}$. We group together related properties like $\mathbf{CONS_X} + \mathbf{CONS_Y}$ by writing them as $\mathbf{CONS_{X+Y}}$.

- *conservative:* $\mathbf{CONS} = \mathbf{CONS_{S+L+C+T+I+G}}$
- *induced:* $\mathbf{IND} = \mathbf{IND_{S+L+C+T+I+G}}$
- *refinable:* $\mathbf{REF} = \mathbf{REF_{C+T+G}}$

Conservative transfromations ($\mathbf{CONS}$) are also called abstractions. Abstractions that are also induced ($\mathbf{CONS} + \mathbf{IND}$) are called induced abstractions. Abstractions that are refinable ($\mathbf{CONS} + \mathbf{REF}$) are called exact transformations. An exact induced transformation combines all three properties ($\mathbf{CONS} + \mathbf{IND} + \mathbf{REF}$).

Informally speaking conservativeness describes that the transformation does not change the behaviour of the transition system. This means that all states of the original system ($\mathbf{CONS_S}$), as well as all transitions and their labels ($\mathbf{CONS_{T+L}}$) are mapped to states, transitions, and labels in the transformed system. Thus all states, transitions, and labels are still accounted for. Additionally it is required that the initial and goal states of the original system are still initial and goal states in the transformed system ($\mathbf{CONS_{I+G}}$). The final condition is that the costs of the actions have not increased, they may, however, decrease without the transformation losing this property ($\mathbf{CONS_C}$).

The next property is inducedeness, which simply requires that no new states ($\mathbf{IND_S}$), transitions ($\mathbf{IND_T}$), or labels ($\mathbf{IND_L}$) are introduced, as well as that no states become initial or goal states, which were not ones in the original system ($\mathbf{IND_{I+G}}$). For a transformation to be cost-induced ($\mathbf{IND_C}$), it must hold that for all transformed labels, one of the labels they were transformed from has the same cost as the new one.

The final property is refinability. Refinability requires that all transitions in the transformed system are able to be uniquely translated back into their original transitions ($\mathbf{REF_T}$). This means that for all transitions from a state s to a state t in the transformed system, there are transitions with the corresponding untransformed label between all states of the original system which are mapped to s and t. Additionally, it is required that the cost of all labels is unchanged by the transformation ($\mathbf{REF_C}$) and that no state which was not a goal state before becomes one in the transformed system ($\mathbf{REF_G}$).

Transformations are composable, which means that we may chain different transformations.

Definition 15 *(Composition of Transformations; based on [5]). Let $\tau = \langle \Theta', \sigma, \lambda \rangle$ be a transformation of a transition system Θ into a transition system Θ'. Let $\tau' = \langle \Theta'', \sigma', \lambda' \rangle$ be a transformation of Θ' into a transition system Θ''. The composition of τ' and τ is the transformation of Θ into Θ'' defined as $\tau' \circ \tau = \langle \Theta'', \sigma' \circ \sigma, \lambda' \circ \lambda \rangle$.*

Properties of these composite transformations are easily deduced from the properties of the individual transformations they are composed of. Suppose we have transformations τ, which transforms a transition system Θ into a transition system Θ', and τ', which transforms Θ' into a transition system Θ''. If τ and τ' both have any property from Definition 14, then the composition $\tau'' = \tau' \circ \tau$ also has that property.

One important theorem that stems from this framework is that if a transformation τ of a transition system Θ is refinable ($\mathbf{REF}$), then the heuristic h^τ for Θ induced by τ is lower-bounded by h^* ($h^*(s) \leq h^\tau(s)$ for all states s), making it admissible. If τ is additionally conservative ($\mathbf{CONS}$), then the heuristic induced by τ is perfect since if τ satisfies $\mathbf{CONS}$, all solutions of the untransformed system are still solutions of the transformed system, meaning that an optimal solution in the transformed system is as most as expensive as the optimal solution of the untransformed system ($h^\tau(s) \leq h^*(s)$ for all states s).

Rather than transforming the concrete transition system of the planning task, the merge-and-shrink algorithm works with a factored transition system.

Definition 16 *(Factored Transition System; based on [5]). A factored transition system is a tuple $F = \langle \Theta^1, \ldots, \Theta^n \rangle$ of transition systems where each transition system has the same set of labels and the same label cost function.*

Planning tasks induce factored transition system. Factors of this induced factored transition system are called atomic factors, each of which representing the behaviour of exactly one state variable.

Definition 17 *(Atomic Factor, Induced Factored Transition System; based on [5]). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ be a planning task. The atomic factor for variable $v \in \mathcal{V}$ is the transition system $\Theta^v = \langle dom(v), \mathcal{O}, c, T^v, S_I^v, S_G^v \rangle$, where c maps each label $l \in \mathcal{O}$ to the cost $cost(l)$ of the operator. $T^v = \{d \xrightarrow{l} d' \mid (v \notin vars(pre(l)) \vee pre(l)[v] = d) \wedge ((v \notin vars(eff(l)) \wedge d' = d) \vee eff(l)[v] = d')\}$, $S_I^v = \{s_I[v]\}$, and $S_G^v = \{s_G[v]\}$ if $v \in vars(S_G)$ and $S_G^v = dom(v)$ otherwise.*

The induced factored transition system of a planning task Π with the state variables $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ is the factored transition system $F(\Pi) = \langle \Theta^{v_1}, \ldots, \Theta^{v_n} \rangle$.

The states of atomic factors correspond to the values the associated state variable may take. A transition $\langle d, l, d' \rangle$ between those values exists if two conditions hold. First, the operator corresponding to the label l must either have no precondition on v or it requires the value to be d. Second, the operator

either does not affect v or it sets v to d' in its effect. If an operator does not change the variable, there are self-looping transitions for each state with that corresponding label. Finally, initial and goal states of the transition system are the values that the variable takes in the initial and goal states of the planning task.

A factored transition system may yet be understood as one single transition system by computing the synchronized product of the factored transition system.

Definition 18 *(Synchronized Product of a Factored Transition System; based on [5]). Let* $F = \langle \Theta^1, \ldots, \Theta^n \rangle$ *be a factored transition system with* $\Theta^i = \langle S^i, L, c, T^i, S^i_I, S^i_G \rangle$ *for all* $1 \leq i \leq n$*. The synchronized product (or simply product) of* F *is the transition system defined as* $\otimes F = \langle S^\otimes, L, c, T^\otimes, S^\otimes_I s, S^\otimes_G \rangle$*, where* $S^\otimes s = \prod_{i=1}^n S^i$*,* $T^\otimes = \{\langle s^1, \ldots, s^n \rangle \xrightarrow{l} \langle t^1, \ldots, t^n \rangle \mid s^i \xrightarrow{l} t^i \in T^i \text{ for all } 1 \leq i \leq n\}$*,* $S^\otimes_I = \prod_{i=1}^n S^i_I$*, and* $S^\otimes_G = \prod_{i=1}^n S^i_G$*. With* $\prod_{i=1}^n A^i$ *indicating the Cartesian product of the sets* A^i*, that is* $\prod_{i=1}^n A^= A^1 \times \cdots \times A^n$*.*

If we compute the product of a factored transition system induced by a planning task we obtain the transition system induced by the planning task. This is very important as this means that the behaviour of the transition system is preserved in its factored form.

In addition to factored representations of transformation systems, we also need factored representations of state mappings in order to keep track of which states of the concrete factored transition system correspond too which states of the abstract factored transition system. This is achieved by factored mappings, which represent arbitrary functions on variable spaces in a tree-like data structure.

Definition 19 *(Factored Mapping; based on [5]). Factored mappings over a variable space* $\mathcal{V}$ *are inductively defined as follows. A factored mapping* σ *has an associated non-empty value set* $vals(\sigma)$ *and an associated table* σ^{tab}*.* σ *is either atomic or a merge.*

- *If* σ *is atomic, then it has an associated variable* $v \in \mathcal{V}$*. Its table is a partial function* $\sigma^{tab} \colon dom(v) \to vals(\sigma)$*.*
- *If* σ *is a merge, then it has a left component factored mapping* σ_L *and a right component factored mapping* σ_R*. Its table is a partial function* $\sigma^{tab} : vals(\sigma_L) \times vals(\sigma_R) \to vals(\sigma)$*.*

Factored mappings may either be atomic, that is, their table stores a partial function over one variable, or they may be merges. Merge factored mappings, much like nodes in a binary tree, have a left and a right component factored mapping. The table of a merge factored mapping then stores a partial function over the product of its component factored mappings' variables. The functions represented by factored mappings are computed as follows.

Definition 20 *(Represented Function; based on [5]). Let* σ *be a factored mapping over a variable space* $\mathcal{V}$*.* σ *represents the partial function* $[\![\sigma]\!] \colon \mathcal{A}(\mathcal{V}) \to vals(\sigma)$ *which is inductively defined as follows:*

- *If* σ *is atomic with associated variable* v*, then* $[\![\sigma]\!](\alpha) = \sigma^{tab}(\alpha[v])$*.*
- *If* σ *is a merge, then* $[\![\sigma]\!](\alpha) = \sigma^{tab}([\![\sigma_L]\!](\alpha), [\![\sigma_R]\!](\alpha))$*. If either* $[\![\sigma_L]\!](\alpha)$ *or* $[\![\sigma_R]\!](\alpha))$ *are undefined, then* $[\![\sigma]\!](\alpha)$ *is undefined as well.*

Finally, if we wish to represent factored transformations we need factored mappings that map from one variable space to another, rather than mapping a factored input to a non-factored output.

Definition 21 *(Factored-to-Factored Mapping; based on [5]). Let* $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ *and* $\mathcal{V}' = \langle v'_1, \ldots, v'_m \rangle$ *be variable spaces. Let* $\sigma_1, \ldots, \sigma_m$ *be factored mappings where each* σ_j *is defined over* $\mathcal{V}$ *and maps to* $dom(v'_j)$*.* $\Sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$ *is called a factored-to-factored mapping from* $\mathcal{V}$ *to* $\mathcal{V}'$*. It represents the partial function* $[\![\Sigma]\!] \colon \mathcal{A}(\mathcal{V}) \to \mathcal{A}(\mathcal{V}')$ *defined as* $[\![\Sigma]\!](\alpha) = \langle \sigma_1(\alpha), \ldots, \sigma_m(\alpha) \rangle$*. If any* $\sigma_i(\alpha)$ *is undefined, then* $[\![\Sigma]\!](\alpha)$ *is undefined as well.*

Factored to factored mappings are, just like the transformations of transition systems, composable. We can now fully define factored transformations, which we may call transformations when it is clear from context that they are transforming a factored transition system.

Definition 22 *(Factored Transformation; based on [5]). A factored transformation of a factored transition system F with label set L into a factored transition system F' with label set L' is a tuple $\tau_F = \langle F', \Sigma, \lambda \rangle$ where F' is the transformed factored transition system, Σ is a factored-to-factored mapping from F to F' called the state mapping, and $\lambda \colon L \to L'$ is a partial function called the label mapping. The transformation induced by τ_F is a transformation of $\otimes F$ into $\otimes F'$ which is defined as $\tau = \langle \otimes F', [\![\Sigma]\!], \lambda \rangle$.*

Let us now take a look at two important factored mappings. First, the factored-to-nonfactored projection mapping, which takes a variable space and returns an atomic factored mapping of one of the variables. Second, the factored-to-factored identity mapping, which takes a variable space and applies the projection mapping to each of them. These two factored mappings are of interest if we do not want to transform factors of the factored transition system.

Definition 23 *(Projection Factored-to-Nonfactored Mapping and Identity Factored-to-Factored Mapping; based on [5]). Let $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$ be a variable space, and let $1 \leq i \leq n$. The projection of $\mathcal{V}$ to v_1 is an atomic factored mapping $\pi_i^{\mathcal{V}}$ that is defined over $\mathcal{V}$, has v_i as its associated variable, has the value set $dom(v_i)$, and a table function mapping d_i to d_i for all $d_i \in dom(v_i)$. The identity mapping for $\mathcal{V}$ is the factored-to-factored mapping $id^{\mathcal{V}} = \langle \pi_i^{\mathcal{V}}, \ldots, \pi_n^{\mathcal{V}} \rangle$. We omit $\mathcal{V}$ from the notation and write π_i and id where $\mathcal{V}$ is clear from context.*

Using what we have defined so far we may now define the factored transformation framework in Algorithm 1. The factored transformation framework takes a factored transition system, as well as functions SelectTransformation and Terminate. Once finished, the framework returns the factored transformation which transform the input factored transition system into another factored transition system. This output is built iteratively until the Terminate function decides that the computation is finished. Originally the identity transformation, the output is updated in each iteration by composing it with a factored transformation chosen by the SelectTransformation function. The SelectTransformation function uses the intermediary factored transformation to make its decision.

Algorithm 1 Factored Transformation Framework; based on [5]

Input: Factored transition system F, function SelectTransformation that selects the next basic transformation to apply, function Terminate that decides when to terminate.
Output: Factored transformation $\langle F', \Sigma, \lambda \rangle$ of F into F'.

1: **function** FactoredTransformationFramework(F, SelectTransformation, Terminate)
2: ▷ Set the current transformation $\tau_F = \langle F', \Sigma, \lambda \rangle$ to the identity transformation of F.
3: $F' \leftarrow F$
4: $\Sigma \leftarrow id^F$
5: $\lambda \leftarrow id^F$, where L is the set of labels of F
6: **while** not Terminate($\langle F', \Sigma, \lambda \rangle$) **do**
7: $\langle F'', \Sigma', \lambda' \rangle \leftarrow$ SelectTransformation($\langle F', \Sigma, \lambda \rangle$)
8: ▷ Update the current transformation $\tau_F = \langle F', \Sigma, \lambda \rangle$ to be the composition of τ_F with the selected transformation $\langle F'', \Sigma', \lambda' \rangle$.
9: $F' \leftarrow F''$
10: $\Sigma \leftarrow \Sigma' \circ \Sigma$
11: $\lambda \leftarrow \lambda' \circ \lambda$
12: **end while**
13: **return** $\langle F', \Sigma, \lambda \rangle$
14: **end function**

We can consider the merge-and-shrink framework as an instance of the factored transformation framework that has the goal of producing a heuristic. Since it would be infeasible to compute an "global" heuristic which would use the product of the factored transition system, we must first define a more easily computed heuristic induced by a factored transition system.

Definition 24 *(Factor Heuristics and Local Heuristics; based on [5]). Let $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ be a factored transition system. The factor heuristic for $\Theta_i \in F$ is defined as $h_i(s) = h^*_{\Theta_i}(s[\Theta_i])$ for all $s \in \mathcal{A}(F)$. The max-factor heuristic of F is defined as $h^{mf}_F = max_{1 \leq i \leq n} h_i$.*

Let $\tau_F = \langle F', \Sigma, \lambda \rangle$ be a factored transformation of a factored transition system F. The local heuristic for F induced by τ_F is defined as $h_F^{loc,\tau_F}(s) = h_{F'}^{mf}([\![\Sigma]\!](s))$ for all $s \in \mathcal{A}(F)$. If $s \notin dom([\![\Sigma]\!])$, we define $h_{F'}^{mf}([\![\Sigma]\!](s)) = \infty$.

These heuristics are much easier to compute compared to the one induced by the synchronized product of the factored transition system. In the context of the local and max factor heuristic we keep track of one additional property of transformations, which, just like the previous ones, is composable.

Definition 25 *(Locally Conservative Factored Transformation; based on [5]). Let τ_F be a factored transformation of a factored transition system F. The following list defines properties that τ_F may have, along with a short-hand name for each property.*

- **LOC$_\leq$** *τ_F is locally nonincreasing if $h_F^{loc,\tau_F}(s) \leq h_F^{mf}(s)$ for all $s \in \mathcal{A}(F)$.*
- **LOC$_\geq$** *τ_F is locally nondecreasing if $h_F^{loc,\tau_F}(s) \geq h_F^{mf}(s)$ for all $s \in \mathcal{A}(F)$.*
- **LOC$_=$** *τ_F is locally equal if $h_F^{loc,\tau_F}(s) = h_F^{mf}(s)$ for all $s \in \mathcal{A}(F)$.*

In words, a locally equal transformation does not change the heuristic values obtained from the max factor heuristic, meaning that the transformation did not change the quality of the local heuristic. If the transformation were to result in an admissible heuristic, it being locally nondecreasing would mean that the quality of our heuristic would improve through its application.

Equipped with these heuristics we may now properly define the merge-and-shrink framework in Algorithm 2.

Algorithm 2 Merge-and-Shrink Heuristic; based on [5]

Input: Planning task Π, function SelectTransformation which selects the next basic transformation to apply, function Terminate which determines when to stop applying transformations.
Output: Merge-and-shrink heuristic $h_\Pi^{M\&S}$ for Π.

```
1: function MergeAndShrink(Π, SelectTransformation, Terminate)
2:     ▷ Compute the induced factored transition system of Π.
3:     F ← F(Π)
4:     ▷ Call the factored transformation framework.
5:     ⟨F', Σ, λ⟩ ← FactoredTransformationFramework(F, SelectTransformation, Terminate)
6:     ▷ Compute the merge-and-shrink heuristic, where τ_F = ⟨F', Σ, λ⟩.
7:     h_Π^{M&S} ← h_F^{loc,τ_F}
8:     return h_Π^{M&S}
9: end function
```

The merge-and-shrink framework takes a planning task Π and the functions Terminate and SelectTransformation we are familiar with from the factored transformation framework. It computes the induced factored transition system F of Π and then uses the factored transformation framework to obtain a factored transformation τ_F. It then returns the merge-and-shrink heuristic for Π which is simply the local heuristic for F induced by τ_F.

2.3 Merge-and-Shrink Framework Transformations

At its core, the merge-and-shrink framework constructs a heuristic from a planning task by computing the factored transition system induced by it. It does so by using specific transformations. The first is the merge transformation, which takes two factors of a factored transition system and replaces them with their product. A visual example of merging two transition systems can be seen in Figure 2.1.

Definition 26 *(Merge Transformation; based on [5]). Let $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ be a factored transition system. The merge transformation for $j \neq k$ with $1 \leq j, k \leq n$ is the factored transformation $\tau_F = \langle F', \Sigma, \lambda \rangle$ of F into the factored transition system F' where:*

- *$F' = \langle \Theta_{i_1}, \ldots, \Theta_{i_{n-2}}, \Theta_\otimes \rangle$ with $i_1, \ldots, i_n - 2 = indices(\{1, \ldots, n\} \setminus \{j, k\})$ and $\Theta_\otimes = \Theta_j \otimes \Theta_k$.*

- $\Sigma = \langle \pi_{i_1}, \ldots, \pi_{i_{n-2}}, \sigma_\otimes \rangle$ *with* $i_1, \ldots, i_{n-2} = indices(\{1, \ldots, n\} \setminus \{j, k\})$*, where* $\sigma_\otimes$ *is a merge factored mapping with left component* π_j*, right component* π_k *and* $\sigma_\otimes^{tab}(s_j, s_k) = \langle s_j, s_k \rangle$ *for all states* $s_j \in \Theta_j$ *and* $s_k \in \Theta_k$*.*
- $\lambda = id$ *is the identity label mapping.*

Merge transformations are exact induced, that is, they satisfy $\mathbf{CONS} + \mathbf{IND} + \mathbf{REF}$*. Merge transformations are also locally nondecreasing, that is, they satisfy* $\mathbf{LOC}_{\geq}$*.*

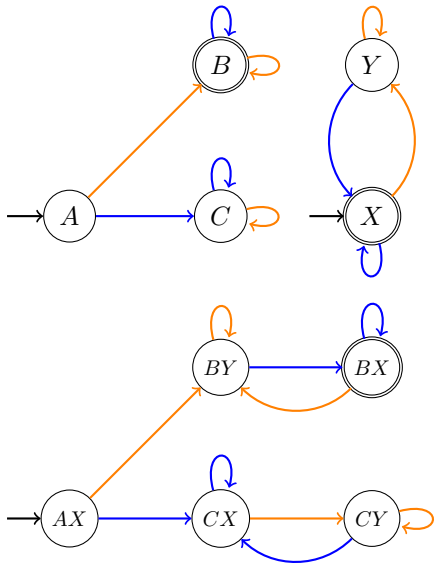

Figure 2.1: Example of a merging transformation. The two upper systems are atomic factors. Edges of the same color correspond to the same action. Initial states are denoted by an incoming arrow, goal states are denoted by a double circle. The third system at the bottom shows the result of merging both atomic factors.

In order to not obtain the full transition system induced by the planning task another transformation may be applied: shrinking. Shrinking, as the name suggests, reduces the size of a transition system. The shrink transformation affects one factor of the factored transition system and it reduces the amount of states in it by mapping multiple states of the original factor to the same state in the transformed factor.

Definition 27 *(Local Abstraction; based on [5]). Let* $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ *be a transition system. The local abstraction* α *of* Θ *is a total state mapping defined on* S*. The transition system induced by the abstraction* α *and the transition system* Θ *is defined as* $\Theta^\alpha = \langle \alpha(S), L, c, \{\langle \alpha(s), l, \alpha(t) \rangle \mid \langle s, l, t \rangle \in T\}, \alpha(S_I), \alpha(S_G) \rangle$*.*

Definition 28 *(Shrink Transformation; based on [5]). Let* $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ *be a factored transition system. Let* α *be a local abstraction for* Θ_k *for some* $1 \leq k \leq n$*. The shrink transformation for* α *and* Θ_k *is the factored transformation* $\tau_F = \langle F', \Sigma, \lambda \rangle$ *of* F*, where:*

- $F' = \langle \Theta'_1, \ldots, \Theta'_n \rangle$ *with* $\Theta'_i = \Theta_i$ *for all* $i \neq k$ *and* $\Theta'_k = \Theta^\alpha_k$*.*
- $\Sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ *where* $\sigma_i = \pi_i$ *for all* $i \neq k$ *and* σ_k *is an atomic factored mapping with variable* Θ_k *and* $\sigma_k^{tab}(s_k) = \alpha(s_k)$ *for all* $s_k \in \Theta_k$*.*
- $\lambda = id$ *is the identity label mapping.*

Shrink transformations are induced abstractions, that is, they satisfy $\mathbf{CONS} + \mathbf{IND}$*. They are also cost-refinable, that is, they satisfy* $\mathbf{REF_C}$ *and they are locally nonincreasing, that is, they satisfy* $\mathbf{LOC}_{\leq}$*.*

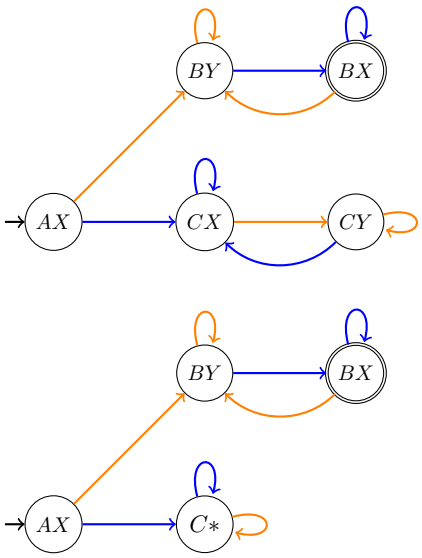

Figure 2.2: Example of a shrinking transformation. We begin with the merged system from Figure 2.1, and shrink it by combining the states CX and CY into C*.

Figure 2.2 graphically visualizes how the shrink transformation works by combining two states of a transition system into one. These two transformations are what give the framework its name. However, state-of-the-art merge-and-shrink uses two additional transformations. The first being label reduction. Label reduction, visualized in Figure 2.3, does for labels what shrinking does for states. Given a set of labels, it takes all transitions that have any of the labels in the set and relabels them with the same new label, which oftentimes simply combines parallel transitions into one. One crucial difference to shrinking is that label reduction is globally applied, meaning it affects all factors of the factored transition system.

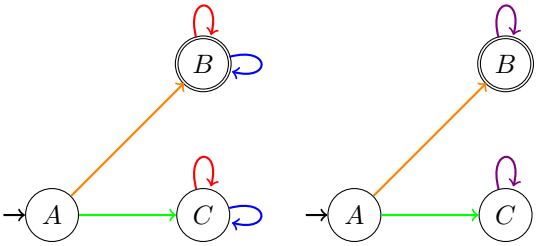

Figure 2.3: Example of a label reduction transformation. On the left a system before label reduction and on the right after combining the red and blue labels.

Definition 29 *(Transition System Induced by Label Mapping; based on [5]). Let* $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ *be a transition system,* λ *a (total) label mapping defined on* L*, and* c' *a cost function defined on* $\lambda(L)$*. The transition system induced by* Θ*,* λ*, and* c' *is defined as* $\Theta^{\lambda,c'} = \langle S, \lambda(L), c', \{\langle s, \lambda(l), t\rangle\} \mid \langle s, l, t\rangle \in T, S_I, S_G\rangle$*.*

Definition 30 *(Label Reduction; based on [5]). Let* $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ *be a factored transition system with label set* L *and label cost function* c*. Let* λ *be a total label mapping defined on* L*. The label reduction transformation (label reduction for short) for* λ *is the factored transformation* $\tau_F = \langle F', \Sigma, \lambda \rangle$ *of* F *into a factored transition system* F' *with label set* $\lambda(L)$*, where:*

- $F' = \langle \Theta_1^{\lambda,c'}, \ldots, \Theta_n^{\lambda,c'} \rangle$*, where* $c'(l') = min_{l \in \lambda^{-1}(l')} c(l)$ *for all* $l' \in \lambda(L)$*.*
- $\Sigma = \langle \pi_1, \ldots, \pi_n \rangle$ *is the identity mapping.*

τ_F is called atomic if λ combines only two labels, that is, if there exist $l_1, l_2 \in L$ with $l_1 \neq l_2$ such that $\lambda(l_1) = \lambda(l_2) = l_{12}$ (where $\lambda_{12} \notin L$ is a fresh label) and $\lambda(l) = l$ for all $l \in L \setminus \{l_1, l_2\}$. It is called general otherwise

All label reductions are abstractions, that is, they satisfy **CONS**. *They also satisfy* $\mathbf{IND_{S+L+C+I+G}}$, $\mathbf{REF_G}$, *and are locally nonincreasing, that is, they satisfy* $\mathbf{LOC_{\leq}}$.

An atomic label reduction combining labels $l_1, l_2 \in L$ is exact, that is, it satisfies $\mathbf{CONS + IND + REF}$, *iff $c(l_1) = c(l_2)$ and:*

- *all transitions in all factors in F, labeled with l_1 have corresponding transitions labeled with l_2, or*
- *all transitions in all factors in F, labeled with l_2 have corresponding transitions labeled with l_1, or*
- *l_1 and l_2 label the same transitions in all but one factor, or*
- *there exists a factor $\Theta \in F$ in which both l_1 and l_2 label no transition.*

Whether a label reduction is exact can be checked in polynomial time, making it feasible to compute and perform them. The other additional transformation formalized by the framework is pruning, which removes states of factors. This is beneficial if there are states that are irrelevant to the task of finding solutions from the initial state, such as ones that can not be reached from the initial state or those from which no goal state can be reached.

Definition 31 *(Pruned Transition System, Prune Transformation; based on [5]). Let $\Theta = \langle S, L, c, T, S_I, S_G \rangle$ be a transition system. Given a subset $K \subseteq S$ of the states of Θ, the transition system Θ prunde to K is defined as $\Theta^K = \langle K, L, c, \{\langle s, l, s' \rangle \mid \langle s, l, s' \rangle \in T \text{ and } s, s' \in K, S_I \cap K, S_G \cap K\rangle$. We call K the set of kept states and $S \setminus K$ the set of pruned states.*

Let $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ be a factored transition system, let S_k be the set of states of Θ_k for some $1 \leq k \leq n$, and let $K \subseteq S_k$. The prune transformation for K and Θ_k is the factored transformation $\tau_F = \langle F', \Sigma, \lambda \rangle$ of F where:

- *$F' = \langle \Theta'_1, \ldots, \Theta'_n \rangle$ with $\Theta'_i = \Theta_i$ for all $i \neq k$ and $\Theta'_k = \Theta_k^K$.*
- *$\Sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$, where $\sigma_1 = \pi_i$ for all $i \neq k$, and σ_k is an atomic factored mapping with variable Θ_k, $\sigma_k^{tab}(s_k) = s_k$ for all $s_k \in K$, and $\sigma_k^{tab}(s_k)$ is undefined for all $s_k \notin K$.*
- *$\lambda = id$ is the identity label mapping.*

The choice of which transformation to perform is made by so called strategies. The first such strategy is to determine when to terminate the algorithm, that is, the function Terminate of the factored transformation framework in Algorithm 1. We also need to determine which transformation is selected by the function SelectTransformation, as well as how the transformation is instantiated. We thus need a general strategy, which decides which transformation, if any, is performed next. We also need transformation strategies which, depending on the transformation chosen by the general strategy, instantiate their respective transformations.

The merge strategy's task is to decide which two factors are to be combined. For this, it may either precompute an order in which the factors are combined or it can decide based on the current state of the factored transition system. The shrink strategy decides which states of which factor (usually the ones selected to be merged) are to be mapped to the same states. Bisimulation based shrinking, introduced by Nissim *et al.* [16], makes sure that the resulting transformation is exact by combining only bisimilar states. The states s and s' are bisimilar if they are both goal states or both not goal states, and if all transitions from s and s' with the same label lead to states t and t' which are also bisimilar. Label-reduction strategies are tasked with determining which labels to combine and pruning strategies decide which states of which factors may be pruned. As described above, exact label reductions can be efficiently computed and intuitive choices for states to be pruned include states that may not be reached from the initial state and states from which no goal state can be reached.

Chapter 3

The Clone Transformation

Using only the transformations above, each state variable is accounted for at most once in the factored transition system. We call such a factored transition system, where all leaves of all factored mappings represent distinct variables, orthogonal. We now define the clone transformation, which, as the name suggests, clones a factor of the factored transition system. The transformed system is then no longer orthogonal, since variables may be represented in more than one factor. This can be interesting if we do not wish to compute a fully merged system where only one factor remains. A non-orthogonal system with multiple factors can then be more expressive than an orthogonal one if we combine the factors using methods such as cost partitioning. Say we have variables u, v, and w, and we know that factors uv and uw would be informative. We might not want to consider the entire merged system of uvw as it may be too big, and merging u with either v or w in the current framework leads to us missing out on considering the other combination. If we clone u before merging, we may construct a composition of transformations which leads to us being able to obtain both uv and uw in our factored system.

Definition 32 *(Clone Transformation). Let $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ be a factored transition system. The clone transformation which clones factor Θ_k with $1 \leq k \leq n$ is the factored transformation $\tau_F = \langle F', \Sigma, \lambda \rangle$, where:*

- *$F' = \langle \Theta'_1, \ldots, \Theta'_n, \Theta'_{n+1} \rangle$, where $\Theta'_i = \Theta_i$ for all $i \neq n+1$, and $\Theta'_{n+1} = \Theta_k$.*
- *$\Sigma = \langle \sigma_1, \ldots, \sigma_n, \sigma_{n+1} \rangle$, where $\sigma_i = \pi_i$ for all $i \neq n+1$, and $\sigma_{n+1} = \pi_k$.*
- *$\lambda = id$ is the identity label mapping.*

The clone transformation is always conservative. It is also always label- and cost-induced, as well as cost- and goal-state-refinable. However, it is not always (goal-)state-induced and by extension not always transition-induced and not transition-refinable. We lose the properties $\mathbf{IND_{S+G+T}}$ and $\mathbf{REF_T}$, when cloning a non-deterministic factor. Even if we assume that the original transition system is deterministic, as is usually the case, its factored representation may include non-deterministic factors. In fact, such factors are common as an action may affect a variable differently, depending on another variable. Even if we do not allow for conditional effects, we still commonly produce non-deterministic transition systems by shrinking. The clone transformation is always locally equal. Proofs for these properties are found in Section 3.1 below.

Without cloning, the framework can only ever consider orthogonal factored transition systems. This restriction is very reasonable as non-orthogonal systems come with a few caveats. Suppose we have the atomic factor of a variable v and an exact copy of it v'. The product of the factored transition system would now contain states where v and v' are assigned the same value. These states make perfect sense. However, there would also be states where v and v' take different values, which is simply not possible in the original planning task. Fortunately, these spurious states should not be reachable, as the values for v and v' are identical in the initial state and since actions taken from any state where v and v' agree on the same value would affect v and v' in the same manner. As such, the new states introduced by cloning could be easily pruned. But as mentioned before, shrinking may lead to non-deterministic transition systems in which case it would be possible to reach states where v and v' take different

values. However, this is of little concern since all new states introduced by cloning and made reachable by other transformations are little more than states where the exact assignment for v is disregarded. An example for this would be new goal states which may be introduced since cloning does not satisfy $\mathbf{IND_G}$. But since cloning satisfies $\mathbf{REF_G}$, the only new goal states we have are states where v and v' disagree on their value but both values taken by v and v' are the values v can take in the goal state of the planning task.

3.1 Proofs of Properties of the Clone Transformation

Let $F = \langle \Theta_1, \ldots, \Theta_n \rangle$ and $F' = \langle \Theta'_1, \ldots, \Theta'_n, \Theta'_{n+1} \rangle$ be factored transition systems. Let $\tau_F = \langle F', \Sigma, \lambda \rangle$ be the clone transformation of F into F' and let Θ_k be the factor cloned by τ_F.

Let $\tau = \langle \otimes F', [\![\Sigma]\!], \lambda \rangle$ be the transformation of $\otimes F$ into $\otimes F'$ induced by τ_F.

Conservativeness

- $\mathbf{CONS_S}$: $[\![\Sigma]\!]$ is a total function.
- $\mathbf{CONS_L}$, $\mathbf{CONS_C}$: These properties are satisfied since $\lambda = id$, which means that the transformation τ_F leaves the labels and their costs unchanged.
- $\mathbf{CONS_T}$: Consider any transition $s \xrightarrow{\ell} t \in \otimes F$. Due to the definition of the product $\otimes F$ it holds that $s[\Theta_i] \xrightarrow{\ell} t[\Theta_i] \in \Theta_i$ for $1 \leq i \leq n$. Since for all $i \neq n+1$, $\Theta'_i = \Theta_i$ and since $\Theta'_{n+1} = \Theta_k$, it holds that $[\![\sigma_i]\!](s) \xrightarrow{\ell} [\![\sigma_i]\!](t) \in \Theta'_i$ for all $1 \leq i \leq n+1$. By the definition of the product $\otimes F'$ it thus holds that $[\![\Sigma]\!](s) \xrightarrow{\ell} [\![\Sigma]\!](t) \in \otimes F'$, which shows that τ_F is transition-conservative.
- $\mathbf{CONS_I}$, $\mathbf{CONS_G}$: Consider any initial or goal state $s \in \otimes F$. Due to the definition of the product $\otimes F$, $s[\Theta_i]$ has to be an initial or goal state respectively for all $1 \leq i \leq n$. Since for all $i \neq n+1$, $\Theta'_i = \Theta_i$ and since $\Theta'_{n+1} = \Theta_k$, it holds that $[\![\sigma_i]\!](s)$ is an initial or goal state in Θ'_i for all $1 \leq i \leq n+1$. Thus, by the definition of the product $\otimes F'$, it holds that $[\![\Sigma]\!](s)$ is an initial or goal state respectively and that τ_F is indeed initial-state- and goal-state-conservative.

Inducedness

- $\mathbf{IND_L}$, $\mathbf{IND_C}$: These properties are satisfied since $\lambda = id$, which means that the transformation τ_F leaves the labels and their costs unchanged.
- Not $\mathbf{IND_S}$, $\mathbf{IND_I}$, $\mathbf{IND_G}$: Let us consider the toy example depicted in Figure 3.1. In it, the product of a non-deterministic transition system with itself is taken. We can see that the states ut and tu have no preimage in the original transition system, cloning is thus not state-induced and by extension not initial-state- and goal-state-induced.
- Not $\mathbf{IND_T}$: Since the cloning transformation is not state-induced, it also is not transition induced, since the states making up a transition may not have a preimage, such as the states tu and ut in Figure 3.1.

Refinability

- $\mathbf{REF_C}$: These properties are satisfied since $\lambda = id$, which means that the transformation τ_F leaves the labels and their costs unchanged.
- Not $\mathbf{REF_T}$: Consider any transition $s' \xrightarrow{\ell} t' \in \otimes F'$. A transformation is transition-refinable if for all $s \in [\![\Sigma]\!]^{-1}(s')$ there is a transition $s \xrightarrow{\ell} t \in \otimes F$ with $t \in [\![\Sigma]\!]^{-1}(t')$ and $l \in \lambda^{-1}(l')$. Since, for the clone transformation, λ is the identity function, the last part is always given. However, since the clone transformation introduces spurious states and, more importantly, spurious transitions between non-spurious states and spurious states, it is not generally transition-refinable. Consider the transition $ss \xrightarrow{\ell} tu$ in the right transition system in Figure 3.1. The preimage of the state ss is the state s and the preimage of tu is empty. We thus have no state corresponding to tu,

and likewise no corresponding transition to $ss \xrightarrow{\ell} tu$, in the left transition system. τ_F is thus not transition-refinable.

- $\mathbf{REF_G}$: We prove this property by considering non-spurious and spurious states in the transformed system. Non-spurious states, such as ss and tt in Figure 3.1, have exactly one element in their preimage. This state in the original system is a goal state if the state in the transformed system is one as well since the transformation satisfies $\mathbf{CONS_G}$. Spurious states, such as tu in Figure 3.1, have no preimage, i.e. $[\![\Sigma]\!]^{-1}(tu) = \emptyset$. Since the preimage is empty, it holds trivially that elements of the preimage are also goal states in $\otimes F$. τ_F is thus goal-state-refinable.

Local Properties of Induced Heuristics

- $\mathbf{LOC_=}$: Since τ_F does not modify any existing factors, i.e. $\Theta'_i = \Theta_i$ for $1 \leq i \leq n$ and $\Theta'_{n+1} = \Theta_k$, the max-factor heuristics of both F and F' are identical. Meaning that $h^{mf}_F(s) = h^{mf}_{F'}([\![\Sigma]\!](s))$ holds for all $s \in \mathcal{A}(F)$, which makes τ_F locally equal.
- $\mathbf{LOC_\leq}$, $\mathbf{LOC_\geq}$: Given since τ_F is locally equal.

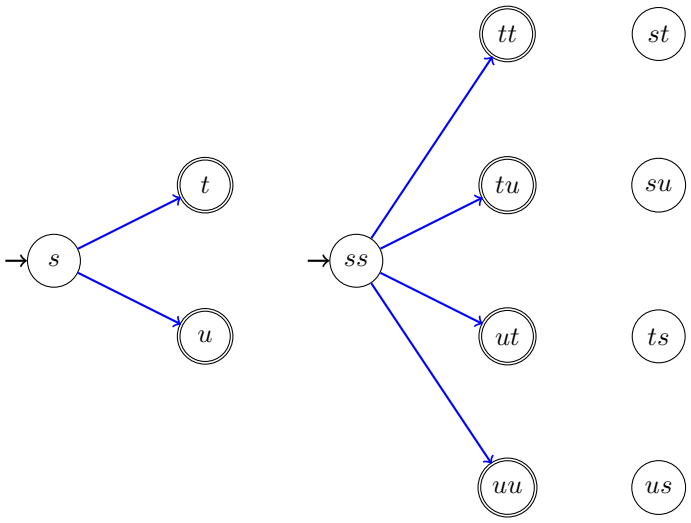

Figure 3.1: Example of a non (goal-)state- and non transtition-induced clone transformation. On the left system Θ and on the right the product of Θ with itself. Note that states st, su, ts, and us are part of the product transition system but are not connected to any other states.

Chapter 4

Clone Strategies

We will now take a look at two classes of cloning strategies we have devised. Similar to the transformation strategies described at the end of Section 2.3, cloning strategies specify how the clone transformation is instantiated, that is, which factor is to be cloned. Before we dive into our strategies we will cover a few technical details regarding the implementation of the merge-and-shrink framework and the clone transformation. We then describe the ad hoc cloning strategy, which will only use information available at each iteration of the main loop of the merge-and-shrink algorithm. It will perform clones ad hoc, meaning that it will only clone if that clone is to be used in a merge shortly after. Afterwards we will consider a precomputed cloning strategy, which will determine which factors to clone before the main loop of the merge-and-shrink algorithm begins to transform the factored transition system.

4.1 Implementing the Cloning Transformation

Throughout the work on this thesis our concrete implementations of the clone transformation underwent a few iterations. Inside the Fast Downward system, factored transition systems are represented by multiple lists which keep track of the factors, their corresponding factored mappings, and distances of interest within the transition system.

Algorithm 3 Fast Downward Merge-and-Shrink Main Loop

Input: Factored transition system F, functions MergeStrategy, ShrinkStrategy, LabelReductionStrategy, and PruningStrategy.

```
1: while F has more than one factor do
2:     ▷ The merge strategy determines the two factors that are to be merged.
3:     ⟨MergeFactor1, MergeFactor2⟩ ← MergeStrategy.nextMerge()
4:     LabelReductionStrategy.reduceLabels(F)
5:     ShrinkStrategy.shrink(MergeFactor1)
6:     ShrinkStrategy.shrink(MergeFactor2)
7:     LabelReductionStrategy.reduceLabels(F)
8:     ▷ The two merged factors are removed from F and their product is added.
9:     F ← (F \ {MergeFactor1, MergeFactor2}) ∪ {MergeFactor1 ⊗ MergeFactor2}
10:    PruningStrategy.prune(F)
11: end while
```

The main loop of the merge-and-shrink implementation, which is described in pseudo code in Algorithm 3, begins with the factored transition system obtained by taking all atomic transition systems of the concrete transition system of the task. It then runs a loop which continues until either time or memory limits are reached or until only one factor remains in the factored transition system. Inside this loop, multiple transformations may be performed after the merge strategy computes which factors are to be merged in this iteration. This is achieved either by a precomputed merge strategy, which specifies in which order all merges occur or by using a set of scoring functions. Given a factored transition system F, the scoring functions assign a score to each pair of factors $\langle \Theta_i, \Theta_j \rangle \in F \times F, i \neq j$. The first scoring function returns the pairs of factors it deems best to merge, which oftentimes are more

than one as is the case with the `goal_relevance()` scoring functions, which assigns the best score to pairs where at least one factor contains a non-goal state. The next scoring functions then filters the pairs chosen by the previous ones until there is only one pair left which is the pair that will be merged next. To make sure that there is always only one pair left after the scoring functions are finished, it is necessary to include a tiebreaking function such as one that selects one pair at random. Before merging, other transformations may occur such as shrinking of the factors which are to be merged and label reduction before or after said shrinking. Then the factors are merged by computing their product and removing them from the aforementioned lists which keep track of the factored transition system. The new factor consisting of transition system, factored mapping, and distances is then appended too the vectors representing the factored transition system. After merging, the product may be pruned before the iteration is complete.

Our first idea was to simply copy the transition system, mapping, and distances, and to create a new entry at the end of the lists, corresponding to the new clone. While this idea is very simple, its implementation was not. Existing classes abused the fact that new products of merges would be appended in set positions. Clones, of which we may have up to two in each iteration of the main loop, are then sitting at these positions, which is not communicated to aforementioned classes. This resulted in bugs which occurred in almost every iteration of our implementations.

A different idea that came to mind was the preserving clone. Recall that the Fast Downward planner tracks the factored transition system via lists, whose elements are being removed whenever they are being used in a merge. A way to implicitly clone would be to simply skip the step where we remove the used factors. While computationally more efficient, this unfortunately leads to occasional inconsistencies, as the factors may be affected by a shrink transformation before being merged, which, while not technically invalidating the framework as a whole, is not quite how a clone transformation should fit into the main loop. However, this implementation has its uses in configurations where shrinking or pruning before merging are not dependent on the other factor of the merge, that is, if we would always transform the clone and its original exactly the same, regardless of the factors we may merge either with.

Algorithm 4 Fast Downward Non-Orthogonal Merge-and-Shrink Main Loop

Input: Factored transition system F, functions MergeStrategy, ShrinkStrategy, LabelReductionStrategy, and PruningStrategy.

while F has more than one factor **do**
  ▷ The merge strategy determines the two factors that are to be merged, as well as whether they are to be cloned. Additionally, the merge strategy may decide to exit the main loop before all factors are merged into one.
  ⟨Factor1, Factor2, cloneFirst, cloneSecond, abort⟩ ← MergeStrategy.nextMerge()
  if abort **then**
    break
  end if
  ▷ If either factor is to be cloned, we add a copy of them to F before.
  if cloneFirst **then**
    $F \leftarrow F \cup \{\text{CopyOfFactor1}\}$
  end if
  if cloneSecond **then**
    $F \leftarrow F \cup \{\text{CopyOfFactor2}\}$
  end if
  LabelReductionStrategy.reduceLabels(F)
  ShrinkStrategy.shrink(Factor1)
  ShrinkStrategy.shrink(Factor2)
  LabelReductionStrategy.reduceLabels(F)
  ▷ The two merged factors are removed from F and their product is added.
  $F \leftarrow (F \setminus \{\text{Factor1, Factor2}\}) \cup (\text{Factor1} \otimes \text{Factor2})$
  PruningStrategy.prune(F)
end while

Ultimately we decided to go with the first idea. As seen in Algorithm 4, cloning is the first transformation performed in each iteration. As for cloning strategies, which determine when and what to clone, we decided to incorporate cloning into the process of selecting which factors to merge, which means that the merge strategy in Algorithm 4 now additionally needs to specify whether each of the factors ought to be cloned. We have also given the merge strategy the option to decide that it does not want to merge anymore. This way, we can ensure that the factors we clone are the ones used in that iteration. Otherwise we would be left with multiple semantically identical factors which would only add overhead to the other transformations, since they would be treated as separate "interesting" factors, although they are not. Binding the cloning tightly to the merge strategy also made it easier to ensure that the cloning transformation is used in a (semi)informed way. We thus do not have cloning strategies which exist individually but we have rather merge strategies which incorporate a cloning strategy. To differentiate between existing merge strategies and our new merge strategies, we will refer to ours as cloning strategies.

One extremely crucial aspect of cloning is the huge strain it can have on both time and memory constraints. While the cloning itself can be implemented highly efficiently, the time it takes to run the merge-and-shrink algorithms grows with the increase in transition systems. In a factored transition system with n factors, cloning a factor with x states may be quick, but the synchronized product of the factored transition system now has x times as many states as before. If our goal is to let the main loop run until there is only one transition system left, we also need an additional iteration of it for each time we clone. Additionally, we now have n additional potential merge we could execute, meaning that the scoring functions' work load increases with each clone, especially if we were to clone before the main loop begins. It is thus important that we limit the amount of clones and that we either clone as late as possible or ensure otherwise that the scoring functions do not consider semantically identical factors multiple times.

4.2 Ad Hoc Cloning

We devised two classes of cloning strategies which are inspired by existing, successful merge strategies. The first, "ad hoc" method, extends a stateless merge strategy which uses only the information available through the current state of the factored transition system. The basic idea being that we clone on demand whenever the merge strategy deems multiple merges equally interesting. This way, the merge strategy can (occasionally) have its cake and eat it too as the factor used to merge will be a clone rather than the original one, allowing it to perform all merges it would want to perform without needing to tiebreak.

To achieve this, our ad hoc cloning strategy extends the existing stateless merge strategy of the Fast Downward planner. This merge strategy uses one or more scoring functions in order to choose the merge with the best score for the next iteration of the main loop. This selection process is then repeated in the next iteration using the updated factored transition system. This means that in every iteration of the main loop, scores need to be computed anew for each pair of factors. As mentioned earlier, a very common occurrence for the stateless merge strategy is that multiple different merges are determined to be equal in quality. This means that we need to tiebreak between many different prospective merges, which may be done by a scoring function which randomly picks one potential merge to assign the best value to.

Our strategy differs from this since we allow the merge strategy to perform multiple merges "simultaneously". not literally of course, but in a simulated manner. For this the merge strategy keeps a list of future merges as determined by the scoring functions. Rather than requiring that the scoring functions return exactly one pair out of all possible merges, we now allow them to return multiple. We then ensure that this list does not contain too many duplicates by counting how often each factor occurs. If a factor occurs in multiple possible merges, we would need to clone it to perform all of them. To regulate how often the algorithm clones we limit the amount of clones with a budget. Each time a clone transformation is performed, one token is taken out of the budget. If the list of merges contains too many duplicate factors, we tiebreak by choosing a random merge and recompute the list of best merges in the next iteration of the main loop. But if the budget allows us to perform all the merges in the list, all of them are being performed in random order, one per iteration of the main loop.

Technically, this new strategy is not exactly stateless, as it keeps track of the list across multiple iterations of the main loop. However, we are simulating a simultaneous execution of the merges over

multiple iterations of the main loop. We only ever shrink clones, unless the factor is not used in any other of the planned merge later on. We also do not recompute the scores so the product of a merge is not considered until the other merges have been performed. It is important to note that the label reduction, which may be performed in each iteration, does affect subsequent merges and the order of the merges thus defines the results of the label reduction. We thus decided to randomize the order in which the merges are performed.

First ideas for our ad hoc cloning strategy allowed re-computing the best merges after each merge in the list is performed, that is, it was truly stateless. However, this may lead to loops, wherein the merge strategy selects the same merge over and over since merges involving the new factor may not always be more interesting, that is, be preferred by the scoring functions. To alleviate this issue, we have created a new scoring function that favors merges which would lead to factors whose sets of atomic components are not already present in the current factored representation. This is achieved by searching through the factored mappings of the factored transition system and is, fortunately, efficient due to the structure of those mappings. If a factor's factored mapping contains the same set of atomic components as the union of the sets of atomic components of the factors that are candidates for a merge, we give this candidate a bad score.

Using this new scoring function generally improved coverage, meaning that the algorithm was able to solve more instances of the benchmark than without. This is despite the fact that loop should not occur when performing merges "simultaneously".

4.2.1 Experiment Setup

For all our experiments we used the a benchmark suite consisting of IPC benchmark instances for the optimal track of the competition[1].

For our orthogonal baseline we employ non-greedy bisimulation based shrinking and we always call the shrink strategy before merging. We perform exact label reduction before shrinking and we allow transition systems of sizes up to 50'000 states. We use a stateless merge strategy which chooses the next merge using either the goal relevance and DFP scoring function or the MIASM scoring function with the same shrinking parameters as before. In both cases we tiebreak among multiple equally good potential merges at random. The DFP scoring function was adapted from Dräger *et al.* [3] by Sievers *et al.* [17] for the use as a non-linear merge strategy. It prefers merges of factors which will synchronize on labels close to goal states, that is, both factors have transitions with the same label close to goal states. The MIASM scoring function was developed by Fan *et al.* [18] and it prefers those merges that result in factors that may be heavily pruned.

In addition to different scoring functions we also consider different methods for saturated cost partitioning. We consider a base case without cost partitioning, in which we use the max factor heuristic if more than one factor remain in the factored transition system once the main loop concludes computation, which may be the case if time or memory limits are met. We also consider both offline and online saturated cost partitioning. In the case of the online cost partitioning we compute two cost partitions, one before starting the main loop, thus a cost partitioning of the atomic factors, and one after the main loop finishes. To determine the heuristic value of a state we then take the maximum of the heuristic values of these two cost partitions. The offline cost partitioning computes a single cost partition over the atomic factors and the factors remaining after the main loop has finished.

To test our ad hoc cloning strategy we kept shrinking, upper limits for sizes of transition systems, label reduction, and cost partitioning identical to the baseline. We used our non-orthogonal "stateless" merge strategy with the same scoring functions as for the baseline, with the addition of additionally testing whether the scoring function which disfavors merges which create transition systems that are, in terms of atomic components, already present (`avoid_existing()` for short). We used cloning budgets of 15, 50, and 100 tokens. We also let the non-orthogonal strategy run with a budget of 0 tokens to get a direct comparison to the baseline which we used to determine the overhead incurred by our implementation.

[1] Found at: https://github.com/aibasel/downward-benchmarks

4.2.2 Results

The main takeaway of our ad hoc cloning strategy was that, put bluntly, it failed with current state-of-the-art constellations of merge selectors. We compiled the results discussed here in Appendix A. The main metric of success we considered is the coverage which indicates how many planning task instances were solved by each combination of parameters. We also included the amount of instances which hit time or memory limits during the construction of the abstraction and during heuristic search. To give an idea of how long abstraction construction and search took respectively, we included the average for both. Additionally we included the score metric which assigns a normalized score depending on how few expansions, evaluations, and generations of states were performed during search. This also includes scores to compare how much memory and time were used. A higher score means that there were fewer evaluations, expansions, and generated states or that less memory or time was required to solve the planning task. In order to determine the quality of the solutions found we rely on the amounts of expansions needed during search to find the solution. There, smaller values are desirable as this implies that the search is following a more direct path to the solution.

Using the `avoid_existing` scoring function improved coverage but the results were still poorer than the baseline. One confusing occurrence was that the new scoring function reduced coverage for the MIASM runs using a budget of 50 clones compared to not using the `avoid_existing` scoring function. Since the coverage was better for runs with budgets of 15 and 100 clones, we believe this is an outlier and not a consequence of using the scoring function. The new scoring function impaired the construction time of the heuristic by less than a tenth of a second on average for the base case. It reduced the mean time taken to construct the heuristic for DFP constellations but increased it for MIASM constellations. Based on this we consider the `avoid_existing` scoring function to be efficient and in our figures and tables, except for the orthogonal baseline, only runs in which it was used.

As seen in Figure 4.1, the orthogonal baseline outperformed runs with 15, 50, and 100 available clones. For 15 tokens alone, we have lost between 7 and 10% coverage, with higher budgets for cloning resulting in consistently worse coverage. For reference, overhead from our implementation reduced coverage by up to roughly 3% for DFP configurations and by up to roughly 7% for configurations using MIASM with offline saturated cost partitioning.

This loss in coverage is explained by increased resource requirements in both time and space when cloning without improvement of heuristic quality. This lack of improvement of the heuristic can be seen by the expansion score which, along with all other scores, steadily declines as we increase the budget for cloning. Generally speaking, all available tokens were used with a few exceptions, which explains the increase in instances reaching the memory limit. Situations in which not all tokens may have been used arise when either a), there is no need to clone, or b) the amount of clones to be performed simultaneously surpasses the available budget. We have observed b) many times and earlier experiments showed that even with 200 or 500 tokens, the algorithm will fully deplete the given budget. This obviously creates a huge increase in necessary space and ultimately leads to the run ending due to lack of memory. There were some domains on which configurations with MIASM did not make use of all tokens for all three budgets. This may indicate that there was no further need to clone but is not a guarantee of this. This was the case for some instances of the `floortile`, `miconic`, `mystery`, `nomystery`, `psr-small`, and the `visitall` domains. In all cases there was no noticeable advantage in coverage or quality (as measured by expansions needed to reach the solution during search) when using cloning.

Despite its poor performance, we think that our ad hoc method can be improved to add value to the merge-and-shrink framework. We believe that most of the issues leading to the decrease in coverage when cloning may be alleviated with scoring functions that are a better fit for the non-orthogonal merge-and-shrink algorithm. With the configurations used by us the algorithm will always completely deplete the available budget of clones which indicates that cloning is not used in an informed way. Cloning this much leads to issues such as loops where the same merges are executed again and again until the budget is depleted. These clones and merges add nothing to the quality of the heuristic and thus essentially only waste time and memory. We observe that even in the baseline, the more restrictive MIASM scoring function leads to higher coverage. This, combined with the clear improvement achieved by the `avoid_existing` scoring function suggests that, moving forward, it would perhaps be best to consider ad hoc cloning only in connection with new and more restrictive scoring functions or with more informed merge strategies altogether.

Figure 4.1: Coverage and aborts due to memory or time for different scoring functions and cost partitioning constellations, as well as the expansion score measuring how few expansions were needed during search to find a solution.

4.3 Precomputed Cloning

The second class of cloning strategies we came up with is a precomputed one, meaning that the decision of which factors to clone is made before the merge-and-shrink algorithm begins its work on the factored transition system. For this we take inspiration from the strongly connected component (SCC) merge strategy which partitions the set of atomic factors such that the partition corresponds to the strongly connected components of the causal graph of the task [19]. The algorithm then combines these subsets of atomic factors into one factor each before combining these factors according to a stateless merge strategy.

Generalizing the idea of the SCC strategy, we have devised a merge strategy which creates a subset for each state variable of the planning task. These subsets, which we call clusters, consist of the variable and the variables in its neighborhood in the causal graph. Unlike the SCC strategy, the clusters created may overlap, as each variable is present in its own cluster, as well as the clusters of its neighbors. Both depth of the neighborhood and choice of edges used to construct it may be specified. Additionally, we allow limiting the amount of cloning transformations to be performed by using a budget, similar to the ad hoc strategy. If there is too much overlap among the clusters, that is, we would need to clone more atomic factors to provide each cluster with their own atomic factor than allowed by the budget, we iteratively combine two clusters until the budget's restriction is met. The clusters, initially consisting of (cloned) atomic factors, transform their factors into one single transition system each using a stateless merge strategy. If at any point an atomic factor is to be merged which is also required to compute another cluster, a clone of it is used instead. Once only one factor remains in all clusters we use cost partitioning to obtain our final heuristic.

4.3.1 Experiment Setup

The baseline reference used for this experiment is the SCC merge strategy using the goal relevance and DFP scoring functions with random tie breaking to construct the factors representing the strongly connected components of the causal graph. As before, we use non-greedy bisimulation based shrinking and always shrink before merging to maintain a limit of 50'000 states per factor. We also kept using exact label reduction before shrinking. As for cost partitioning methods, we again considered offline and online saturated cost partitioning in the exact same way as we did for the ad hoc strategy. Since our precomputed cloning strategy always terminates the main loop before only one factor remains, we have additionally adapted the SCC strategy, which we use as a baseline reference, to also use cost partitioning once the factors of the strongly connected components are constructed.

Our precomputed cloning strategy uses identical scoring functions, shrinking parameters, label reduction methods, and cost partitionings as the reference. As for cluster creation, we considered neighborhoods of depths 1, 2, and 4, as well as a reference of depth 0 which yields an orthogonal run resulting in a cost partitioning of the factored transition system consisting only of the atomic factors.

We considered using predecessors, successors, or both when creating the clusters from the causal graph, as well as combining the largest clusters, the smallest clusters, or random clusters when it is necessary to reduce the overall need for cloning. To clone, we allocated budgets of 15, 50, and 100 tokens.

Compared to the ad hoc experiments, we did not consider the MIASM scoring function as the amount of configurations already was already too large.

4.3.2 Results

Table 4.1 shows that, in terms of coverage, the depth 0 baseline slightly outperforms the SCC baseline. However, if we look at the expansions to measure the quality of the obtained heuristics, we can see that the SCC strategy is considerably better than the depth 0 baseline.

Unsurprisingly, the choice of cost partitioning has little effect on the baseline of depth 0 since the cost partitionings should be identical as the algorithm does not transform the factored transition system at all. If we look at the SCC baseline, we see a trend that is also noticeable in the results of the ad hoc cloning. Online cost partitioning performs better than offline cost partitioning. This is also true for the precomputed cloning strategy, where online cost partitioning outperforms offline cost partitioning regardless of depth, method of cluster creation, and clone budget. Since online cloning

	SCC, Offline CP	SCC, Online CP	Precomputed Cloning, Depth 0, Offline CP	Precomputed Cloning, Depth 0, Online CP
Coverage	776	827	831	830
Construction out of Time	167	169	0	0
Construction out of Memory	10	8	13	7
Search out of Time	1	4	0	0
Search out of Memory	855	801	965	972
Construction time mean	4.06	3.95	0.05	0.03
Search time mean	0.49	0.33	0.766	0.73
Score Expansions	340.89	416.04	348.32	349.66
Score Memory	389.82	426.75	438.60	446.88
Score Search Time	660.78	713.11	698.47	700.10
Score Total Time	586.97	632.95	696.15	698.87

Table 4.1: Coverage and scores of the baselines of the precomputed cloning strategy.

universally outperformed offline cloning in our experiments, we will only consider configurations with online cost partitioning from here on.

Let us now take a look at the budget allocated for cloning. We might suspect that, similar to the ad hoc cloning, increasing the budget will reduce coverage due to computational constraints but this

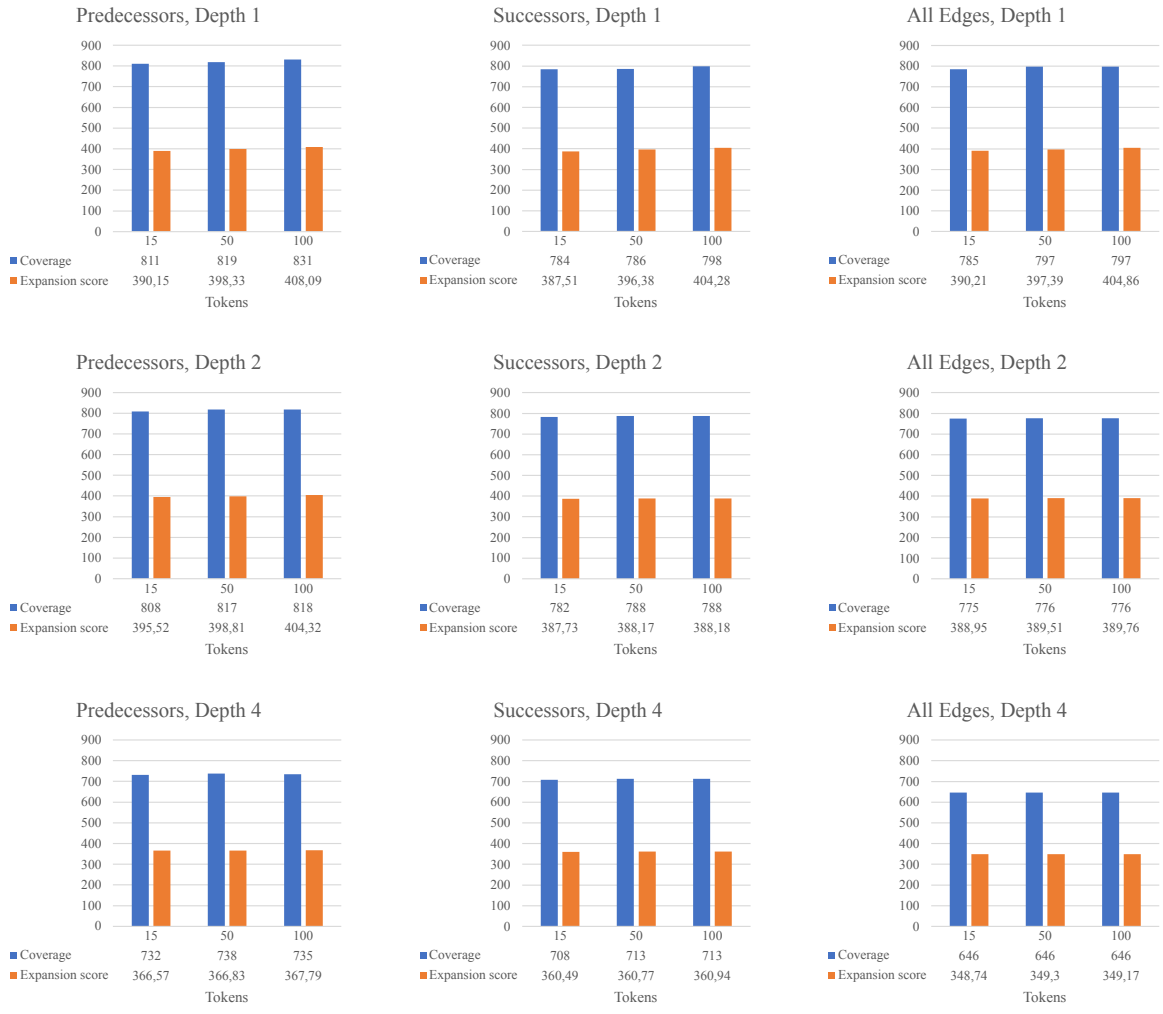

Figure 4.2: Coverage in instances and expansion scores for varying tokens, depths, and choice of edges during cluster construction. Combines random clusters to reduce the amount of clones necessary to realize the clusters.

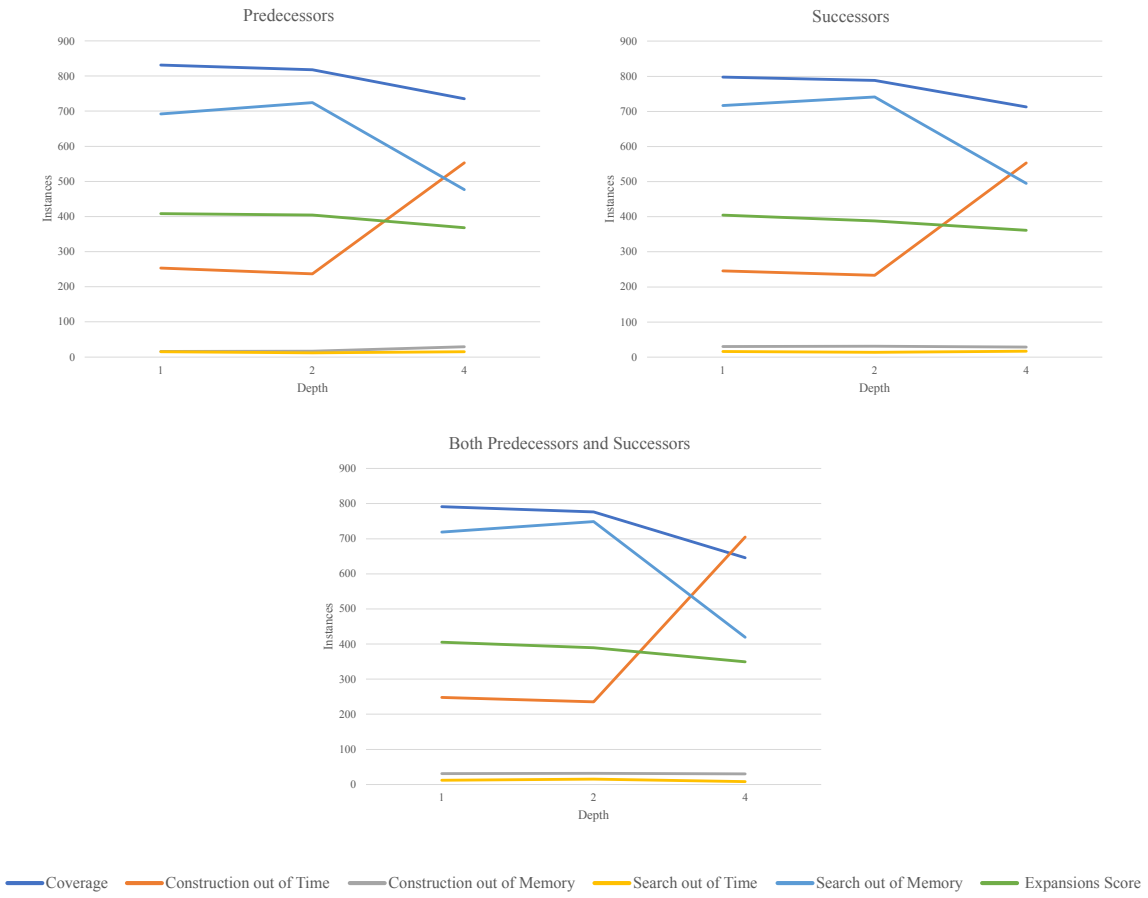

Figure 4.3: Coverage, aborts, and expansion scores for varying depths and choice of edges during cluster construction. Allows up to 100 clones and combines clusters at random if more than 100 clones are necessary.

is generally not the case. Figure 4.2 shows how coverage and expansion score behave if we increase the amount of clones that we permit. The general trend that we observe is that an increase in tokens leads to a slight increase in coverage. This is not universally true as coverage declines slightly if we increase the depth of the neighborhoods considered which is likely due to the increased depth as discussed later on. Although the coverage is generally worse than our 0 depth baseline, the expansion scores are generally better by a considerable margin. From this we conclude that, rather than limiting tokens, we should limit the depth.

Figure 4.3 shows how depth impacts coverage and expansion score. We can clearly see that at depth 4, we substantially lose coverage, with the main reason being that the construction time hits the limit. The difference in coverage and expansion score between depth 1 and 2 is not as drastic but we notice that neighborhoods of depth 1 outperform those of depth 2 consistently. One big surprise is that the construction time is significantly lower for depth 2. This is likely due to the fact that the larger depth facilitates the need to combine more clusters, reducing the amount of clusters and thus clones, which subsequently means that we perform fewer merges.

Next, we take a look at the different methods for combining clusters to reduce the total overlap, that is, the amount of clones we would need to realize all clusters. As can be seen in Table 4.4, there is little difference between combining at random, combining the two largest clusters, and combine the two smallest clusters until the overlap has been reduced. Obviously, combining the largest clusters means that more time is needed to construct the abstraction, as calculating merges for bigger clusters takes longer due to the exponentially increasing amount of potential merges within a cluster, as well as, on average, larger factors. This is why the construction time for the baseline examples of depth 0 from Table 4.1 was almost instant, bar the overhead from "clustering". Despite there only being

Combining Clusters Randomly	Predecessor	Successor	All Edges
Coverage	831	798	797
Construction out of Memory	16	90	31
Construction out of Time	253	246	248
Search out of Memory	692	717	719
Search out of Time	15	16	12
Expansion Score	408.09	404.28	404.86
Search Time Score	709.74	686.22	686.53
Total Time Score	586.74	564.38	560.31
Combining Smallest Clusters	Predecessor	Successor	All Edges
Coverage	831	793	800
Construction out of Memory	13	28	31
Construction out of Time	245	235	234
Search out of Memory	703	737	727
Search out of Time	15	14	15
Expansion Score	405.96	400.79	401.66
Search Time Score	707.83	680.54	682.85
Total Time Score	586.62	564.2	560.49
Combining Largest Clusters	Predecessor	Successor	All Edges
Coverage	827	799	803
Construction out of Memory	28	27	28
Construction out of Time	295	282	277
Search out of Memory	646	685	687
Search out of Time	11	14	13
Expansion Score	407.97	407.28	407.72
Search Time Score	708.71	689.8	689.95
Total Time Score	586.87	566.05	563.61

Figure 4.4: Coverage, aborts, and scores for different methods of creating and combining clusters.

a slight difference in quality and coverage, we believe that it would be best to combine the largest clusters when it is necessary to combine. This way, we are likely to achieve the largest reduction in the required amount of clones. This also hints at the next possible improvement which may be made to this strategy, which would be to combine the clusters with the largest overlap, or largest overlap relative to their size. While this is somewhat cumbersome to implement efficiently, it is a promising avenue for further improvement of the precomputed cloning strategy.

Lastly, we take a look at which edges we may use to construct the clusters. As we can see in Table 4.4 (and already saw in Figures 4.2 and 4.3), using both predecessor and successor nodes to construct the neighborhoods puts a strain on the memory taken to construct the factored transition system. Surprisingly, using only successors puts almost the exact same strain on the memory as using both predecessors and successors. Overall, all methods yield roughly equal heuristics quality wise, as measured by expansion scores. We would still recommend using only predecessors, as this generally puts the least strain on memory, while preserving heuristic quality.

One final experiment we have performed was to evaluate how computing a cost partitioning at each iteration of the main loop would affect the performance of the precomputed cloning strategy. For this we compared an optimized configuration of the SCC strategy with a few of the most promising constellations for our precomputed cloning strategy. Changes to the SCC strategy compared to the baseline used in Table 4.1 consist of computing a cost partitioning at each iteration of the main loop and limiting the main loop by a time limit of 15 minutes. For our precomputed cloning strategy we always combine the largest clusters and give a budget of 100 clones. We vary how clusters are created by considering either predecessors, successors, or both. We also considered neighborhoods of depths 1, 2, and 4. We also compute a cost partitioning at every iteration of the main loop and limit the time of the main loop to 15 minutes.

In Table 4.2 we can see that the optimized SCC call heavily outperforms the best of the 9 constellations tested against it. As expected, the precomputed cloning strategy performed best when

considering predecessors of up to depth 2 when creating the clusters. All other constellations were worse, with decreases in coverage ranging from roughly 4% for constellations of depths 1 and 2 up to roughly 20% for constellations of depth 4.

The decrease in coverage and quality of heuristics compared to the constellations from previous experiments is not at all surprising. The main advantage of computing cost partitionings at each iteration of the main loop lies in the availability of many different factors which may be considered during search. While it makes a lot of sense to use cost partitioning to combine the clusters obtained from our precomputed cloning strategy, our design philosophy intended for the clusters to be combined after they have been fully built from their atomic factors. As such it seems reasonable that we lose performance when computing cost partitionings throughout construction.

Compared to ad hoc cloning, the precomputed cloning strategy is significantly more promising. We have seen that it is, at least in terms of coverage, able to keep up with a very simple instantiation of the SCC merge strategy. While the overall quality of the heuristics obtained by the precomputed cloning strategy is poorer than even the unoptimized SCC strategy's we see promise in the flexibility of our approach.

As with the ad hoc cloning strategy, we are certain that there is much room for improvement for the precomputed cloning strategy. We believe that there is significant overhead in our implementation during the computation of the clusters and their combining which likely has a negative impact on our results. We are certain that there are many promising methods of creating clusters from the causal graph which do not only consider the neighborhood of a node, but also focus the overlap on the dominating set or on cliques of the causal graph. This way we would more likely end up cloning factors whose presence in multiple factors is beneficial.

	SCC	Precomputed Cloning
Coverage	901	815
Construction out of Time	27	102
Construction out of Memory	17	208
Search out of Time	508	265
Search out of Memory	356	418
Construction time mean	1.00	1.39
Search time mean	0.15	0.29
Score Expansions	505.08	391.02
Score Memory	451.95	350.61
Score Search Time	740.12	662.45
Score Total Time	637.19	564.74

Table 4.2: Coverage and scores of an optimized SCC call creating a cost partitioning at each iteration of the main loop, as well as the best performing constellation of parameters for the precomputed cloning which also creates a cost partitioning at each iteration of the main loop.

Chapter 5

Conclusion

In this thesis we have defined a new transformation for the merge-and-shrink framework which clones a factor of the factored transition system, making the latter non-orthogonal. The non-orthogonal merge-and-shrink algorithm may create more expressive intermediary factored transition systems, which in conjunction with cost partitioning may yield heuristics of better quality.

We considered two classes of merge-and-shrink algorithms, one which decided ad hoc whether a factor is to be cloned, and one where clones were prepared before the merge-and-shrink algorithm began to transform the factored transition system. Both strategies performed worse than their respective state-of-the-art baselines. The ad hoc strategy shows little promise as the interplay between the different transformation strategies would need to be redesigned from ground up to exploit the possibility of creating non-orthogonal factored transition systems.

Considering the class of merge-and-shrink algorithms which precompute when to clone, we are much more optimistic. Although it also failed to compete with its optimized state-of-the-art counterpart, we still believe that there is room for improvement without having to redesign the entire existing implementation. The main difference between the implementation of the precomputed cloning and the ad hoc cloning is that our precomputed cloning may be seen in a more modular light. Each of the clusters we create is orthogonal, meaning we may exploit well performing methods for transforming that cluster into one factor. The methods for creating clusters are also implemented on top of the existing code, meaning that it would be easy to implement a new method for clustering and even combining clusters to reduce the amount of cloning required.

For future work on this topic we see two lanes, the first, more arduous and less promising path would design a merge strategy which keeps track of and exploits the fact it may merge the same factor multiple times with others. An entry point for this would be development of a new scoring function.

The second, and in our opinion more promising, lane is to develop new methods for the creation of clusters. While the causal graph offers a lot of interesting ideas for creating clusters, it would certainly be of interest to consider taking inspiration from pattern selection for pattern databases, as this feels like a natural interpretation of the task of creating multiple overlapping factors in the factored transition system, each of which ideally yields an informative heuristic.

Bibliography

[1] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence," 1955.

[2] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[3] K. Dräger, B. Finkbeiner, and A. Podelski, "Directed model checking with distance-preserving abstractions," *Proceedings of the 13th International SPIN*, pp. 19–34, 2006.

[4] M. Helmert, P. Haslum, and J. Hoffmann, "Flexible Abstraction Heuristics for Optimal Sequential Planning," in *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, 2007, pp. 176–183.

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

[6] M. Helmert, P. Haslum, J. Hoffmann, and R. Nissim, "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces," *Journal of the ACM*, vol. 61, no. 3, 2014.

[7] S. Sievers, "Merge-and-Shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation," Ph.D. dissertation, University of Basel, 2017.

[8] B. Bonet and M. Van Den Briel, "Flow-Based Heuristics for Optimal Planning: Landmarks and Merges," in *Proceedings of the 24th International Conference on Automated Planning and Scheduling*, 2014, pp. 47–55.

[9] S. Sievers, F. Pommerening, T. Keller, and M. Helmert, "Cost-partitioned merge-and-shrink heuristics for optimal classical planning," in *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, 2020, pp. 4152–4160.

[10] M. Helmert and T. Keller, *Lecture Foundations of Artificial Intelligence*, https://dmi.unibas.ch/de/studium/computer-science-informatik/lehrangebot-fs20/lecture-foundations-of-artificial-intelligence/, Accessed: 2023–10–06, 2020.

[11] M. Helmert and G. Röger, *Lecture Planning and Optimization*, https://dmi.unibas.ch/de/studium/computer-science-informatik/lehrangebot-hs21/lecture-planning-and-optimization/, Accessed: 2023–10–06, 2021.

[12] C. Knoblock, "Automatically generating abstractions for planning," *Artificial Intelligence*, vol. 68, pp. 243–302, 1994.

[13] M. Katz and C. Domshlak, "Optimal Additive Composition of Abstraction-based Admissible Heuristics," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2008, pp. 174–181.

[14] F. Yang, J. Culberson, R. Holte, U. Zahavi, and A. Felner, "A General Theory of Additive State Space Abstractions," *Journal of Artificial Intelligence Research*, vol. 32, pp. 631–662, 2008.

[15] J. Seipp, T. Keller, and M. Helmert, "Saturated cost partitioning for optimal classical planning," *Journal of Artificial Intelligence Research*, vol. 67, pp. 129–167, 2020.

[16] R. Nissim, J. Hoffmann, and M. Helmert, "Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011, pp. 1983–1990.

[17] S. Sievers, M. Wehrle, and M. Helmert, "Generalized label reduction for merge-and-shrink heuristics," in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014, pp. 2358–2366.

[18] G. Fan, M. Müller, and R. Holte, "Non-linear merging strategies for merge-and-shrink based on variable interactions," in *Proceedings of the 7th Annual Symposium on Combinatorial Search*, 2014, pp. 53–61.

[19] S. Sievers, M. Wehrle, and M. Helmert, "An analysis of merge strategies for merge-and-shrink heuristics," in *Proceedings of the 26th International Conference on Automated Planning and Scheduling*, 2016, pp. 294–298.

Appendix A

Results of Ad Hoc Cloning

A.1 Orthogonal Baseline

	DFP	DFP, Offline CP	DFP, Online CP	MIASM	MIASM, Offline CP	MIASM, Online CP
Coverage	787	795	827	825	803	860
Construction out of Time	172	169	171	211	208	211
Construction out of Memory	33	35	33	12	15	12
Search out of Time	4	4	5	12	11	9
Search out of Memory	812	805	772	746	769	714
Construction time mean	5.94	6.09	6.02	12.49	12.64	12.57
Search time mean	0.31	0.45	0.25	0.21	0.33	0.18
Score Evaluations	378.73	330.34	397.27	426.88	354.72	444.13
Score Expansions	419.95	369.46	443.00	472.82	399.19	494.14
Score Generated	345.81	287.19	363.59	395.62	313.59	412.42
Score Memory	397.03	388.26	414.09	424.09	406.86	440.09
Score Search Time	679.60	673.38	715.27	720.41	690.25	753.05
Score Total Time	597.15	584.85	620.94	590.49	563.41	605.78

Figure A.1: Results of the orthogonal baseline using a stateless merge strategy with DFP or MIASM merge selectors, with and without Cost Partitioning (CP). Times are in seconds.

A.2 Baseline Comparison, Budget Size 0

	DFP	DFP, Offline CP	DFP, Online CP	MIASM	MIASM, Offline CP	MIASM, Online CP
Coverage	788	775	827	811	748	844
Construction out of Time	172	170	172	296	291	295
Construction out of Memory	30	32	30	14	17	14
Search out of Time	5	3	4	12	10	13
Search out of Memory	814	829	776	673	740	640
Construction time mean	4.11	4.18	4.17	14.87	14.97	14.95
Search time mean	0.24	0.40	0.19	0.15	0.40	0.12
Score Evaluations	371.46	306.52	390.11	421.38	297.69	437.49
Score Expansions	411.99	340.88	435.22	463.97	331.15	484.28
Score Generated	337.42	260.14	355.48	388.53	251.84	404.39
Score Memory	397.78	381.83	414.80	421.56	380.07	436.40
Score Search Time	679.47	659.49	715.15	709.41	637.27	740.05
Score Total Time	598.80	574.13	622.06	571.64	517.09	584.48

Figure A.2: Results of the non-orthogonal baseline with a cloning budget of 0 clones. We use a stateless merge strategy with DFP or MIASM merge selectors, with and without Cost Partitioning (CP). The scoring function avoiding already existing merges is always included. Times are in seconds.

A.3 Budget Size 15

	DFP	DFP, Offline CP	DFP, Online CP	MIASM	MIASM, Offline CP	MIASM, Online CP
Coverage	736	752	778	779	738	810
Construction out of Time	172	170	172	312	313	315
Construction out of Memory	72	74	74	28	31	28
Search out of Time	124	3	112	39	12	35
Search out of Memory	704	809	672	648	712	618
Construction time mean	6.66	6.74	6.75	25.15	25.29	25.23
Search time mean	0.39	0.34	0.29	0.25	0.34	0.19
Score Evaluations	339.21	303.15	358.72	382.24	293.04	396.60
Score Expansions	376.47	336.50	400.93	421.66	325.60	440.47
Score Generated	302.49	257.62	321.38	345.62	247.44	359.86
Score Memory	322.64	319.63	333.73	372.54	342.77	383.29
Score Search Time	614.89	643.37	652.04	669.63	628.82	699.61
Score Total Time	527.60	539.05	551.30	497.25	464.56	507.26

Figure A.3: Results of the non-orthogonal baseline with a cloning budget of 15 clones. We use a stateless merge strategy with DFP or MIASM merge selectors, with and without Cost Partitioning (CP). The scoring function avoiding already existing merges is always included. Times are in seconds.

A.4 Budget Size 50

	DFP	DFP, Offline CP	DFP, Online CP	MIASM	MIASM, Offline CP	MIASM, Online CP
Coverage	705	714	747	706	666	723
Construction out of Time	173	171	173	447	445	450
Construction out of Memory	213	216	214	105	108	106
Search out of Time	228	2	205	176	34	166
Search out of Memory	489	705	469	372	553	361
Construction time mean	9.37	9.42	9.47	55.74	55.97	56.11
Search time mean	0.35	0.25	0.28	0.27	0.25	0.23
Score Evaluations	332.17	294.68	350.56	364.67	273.17	378.22
Score Expansions	367.99	325.57	391.45	403.36	303.66	419.59
Score Generated	298.05	251.57	315.65	331.68	230.03	345.29
Score Memory	287.35	283.87	297.87	317.85	291.29	322.83
Score Search Time	575.66	611.18	611.61	589.75	567.06	610.14
Score Total Time	472.99	489.41	496.94	377.62	362.79	382.83

Figure A.4: Results of the non-orthogonal baseline with a cloning budget of 50 clones. We use a stateless merge strategy with DFP or MIASM merge selectors, with and without Cost Partitioning (CP). The scoring function avoiding already existing merges is always included. Times are in seconds.

A.5 Budget Size 100

	DFP	DFP, Offline CP	DFP, Online CP	MIASM	MIASM, Offline CP	MIASM, Online CP
Coverage	674	675	711	556	538	575
Construction out of Time	180	178	180	771	769	771
Construction out of Memory	284	287	287	215	217	217
Search out of Time	365	1	332	134	38	125
Search out of Memory	305	667	298	130	244	118
Construction time mean	13.53	13.59	13.60	96.72	96.58	97.56
Search time mean	0.31	0.20	0.26	0.34	0.20	0.28
Score Evaluations	332.16	285.72	348.78	301.14	231.76	315.52
Score Expansions	366.48	315.75	387.78	329.90	254.00	346.25
Score Generated	299.98	243.59	315.85	273.35	195.60	287.55
Score Memory	260.52	259.49	269.00	238.08	224.30	240.96
Score Search Time	533.19	584.82	566.11	461.74	458.54	481.44
Score Total Time	412.05	437.09	432.82	239.83	237.56	244.45

Figure A.5: Results of the non-orthogonal baseline with a cloning budget of 100 clones. We use a stateless merge strategy with DFP or MIASM merge selectors, with and without Cost Partitioning (CP). The scoring function avoiding already existing merges is always included. Times are in seconds.