UNIVERSITÄT BASEL

Analysing and Combining Static Pruning Techniques for Classical Planning Tasks

Master Thesis

Faculty of Science, University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
http://ai.cs.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle

Michaja Pressmar
m.pressmar@unibas.ch
11-059-011

May 14, 2016

Abstract

In classical AI planning, the state explosion problem is a reoccurring subject: although the problem descriptions are compact, often a huge number of states needs to be considered. One way to tackle this problem is to use static pruning methods which reduce the number of variables and operators in the problem description before the planning.

In this work, we discuss the properties and limitations of three existing static pruning techniques with a focus on satisficing planning. We analyse these pruning techniques and their combinations, and identify synergy effects between them and the domains and problem structures in which they occur. We implement the three methods into an existing propositional planner, and evaluate the performance of different configurations and combinations in a set of experiments on IPC benchmarks. We observe that static pruning techniques can increase the number of solved problems, and that the synergy effects of the combinations also occur on IPC benchmarks, although they do not lead to a major performance increase.

Contents

1. Introduction

Classical Planning is a field in Artificial Intelligence in which, given a specific problem, the aim is to find a solution in form of an action sequence which can be applied to reach a defined goal. One approach to find a solution is *state space search*, where a problem is modeled as a state space in which executing an action equals a transition to a different state. A solution to the problem, called a *plan*, is a sequence of actions leading from the initial state to a goal state. We distinguish between *satisficing* search, where any plan needs to be found for the problem, and *optimal* search, where the plan to be found needs to have minimal cost among all plans.
A reoccurring problem that has been subject to much research is the *state explosion problem*, i.e. the problem that the size of the search space quickly explodes when problems become bigger, and often a solution cannot be found anymore within reasonable time and memory bounds, especially in optimal search. One possibility to decrease the number of states that need to be considered is to use *heuristic search*, which is based on computing a heuristic function to estimate the distance of a state to a goal state. Heuristic search has been well studied and very good heuristics were discovered in the last years, both for optimal (Helmert and Domshlak, 2009; Haslum et al., 2007; Pommerening et al., 2014) and for satisficing planning (Hoffmann and Nebel, 2001; Helmert and Geffner, 2008).
An orthogonal method to tackle the state explosion problem is to *prune* parts of the search space, in order to reduce the number of states that need to be expanded, thus easing the effort of the search. Some pruning methods work in a *dynamic* way, meaning they are active during the search, while *static* pruning methods reduce the problem size before the search by modifying the problem description. Several of these static pruning methods were discovered, but they are not fully explored yet.

In this work, we explore to what degree and in which domains problem solving algorithms can benefit from using recent static pruning methods that preserve at least one plan for the problem. Further, we investigate possible synergies between techniques, and the costs in time and loss of plan optimality resulting from combining them.

The thesis is organized as follows.
In Chapter 2, we formally introduce planning tasks and other concepts which are relevant for this work. Chapter 3 describes and explains the three static pruning techniques, and also analyses some of their properties and limitations. In Chapter 4, we examine each combination of two pruning methods and discuss synergy effects by means of examples

or prove that no synergy effects exist.
We then describe the implementation of the three methods into an existing propositional planner in Chapter 5. The implementation is used in Chapter 6 for a row of experiments on IPC benchmarks, where we first evaluate the techniques separately and then in combination with each other. We conclude the thesis with an idea for a possible extension of one of the three techniques in Chapter 7, and summarize our results in Chapter 8.

Related Work

We conclude this introduction with a brief summary of the past research on static pruning methods.

A central concept of this work is *Safe Abstraction*, which has been discovered by Helmert (2006), later on applied to model checking (Wehrle and Helmert, 2009) and further researched by Haslum (2007). The idea of this static pruning method is to identify *safe* variables, which can be abstracted from the problem in order to tackle the state-space explosion. The property of a safe variable ensures that no state is reachable in the abstract problem which is not reachable in the concrete problem. Consequently, every plan for the abstract problem can be modified so it becomes a plan for the original problem.

The second pruning method relevant to this work is *Redundant Operator Reduction*. A *redundant* operator can be replaced in any plan by a sequence of different operators. As shown by Haslum and Jonsson (2000), redundant operators can be removed from the planning task before planning in order to reduce the problem complexity, and the problem remains solvable.

Another method to reduce the size of the search space is *Dominance Pruning*, which has recently been researched by Torralba and Hoffmann (2015). The authors propose to detect states that dominate other states by computing simulation relations on Merge-and-Shrink abstractions. If a state is dominated by a previously explored state, it can be pruned during the search, meaning that it is not explored. This optimality-preserving technique can also be used in a static way, to remove operators from the planning task whose transitions are irrelevant as other transitions exist which lead to better states (Kissmann and Torralba, 2015). This static pruning method is the third technique we investigate in this work.

To complete the picture, we describe additional static pruning techniques in the following, which are not directly relevant to the work.

As some problems have structural symmetries between variables and between actions, the search effort can be reduced by exploiting them. Fox and Long (1999) detect symmetric objects by forming symmetric groups. This information can be used to extend the problem description with symmetric-breaking predicates as shown by Crawford et al. (1996), or during the search to avoid looking at plans which differ from already considered plans only in symmetric objects (Pochter et al., 2011; Domshlak et al., 2012, 2013).

Another method, called *Tunnel Macros*, has been proposed by Junghanns and Schaeffer (2001) in the context of solving the Sokoban puzzle. Tunnel Macros have been generalized in a recent work by Nissim et al. (2012) and allow the composition of several actions, which naturally should be executed subsequently, into a single action, either dynamically by pruning states, or statically by adding new operators to the problem. Nissim et al. (2012) also discovered a method to prune the search space following the idea to finish first what has been started. Their *Partition-based Path Pruning* uses a decomposition of the planning task, which allows grouping actions that somehow belong together, and should be executed after another. The authors also suggest a way to decompose a given problem based on a similarity score, and present a rule how the Partition-based Path Pruning can be combined with Tunnel Macros while preserving optimality.

Lastly, *Partial Order Reduction* prunes the search space by only considering a subset of the applicable transitions in each state. To identify the transitions to be considered, Valmari (1989) proposes *stubborn sets*, which contain all applicable transitions but those independent of the transitions in the set, which can therefore still be applied later on. Godefroid (1996) describes an orthogonal concept, so-called *sleep sets*, which contain transitions that can be ignored since the states they lead to can be reached through other paths. Both of these methods have recently found interest in the context of planning (Wehrle and Helmert, 2012; Holte et al., 2015; Alkhazraji et al., 2012; Wehrle and Helmert, 2014).

2. Background

In this chapter, we will introduce the relevant concepts and definitions concerning planning tasks.

In classical planning, a problem is represented by a set of variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, where each Variable $v \in \mathcal{V}$ is associated with a finite domain D_v. A specific *state* s of the problem is captured by a distinct assignment of values to all variables:
$s = \{(v_1, s(v_1)), \ldots, (v_n, s(v_n))\},\ s(v_i) \in D_{v_i}$, where each variable-value pair $(v, s(v))$ in the set represents an assignment $v \leftarrow s(v)$ in the state s. A *partial state* p is an assignment of values to a subset of variables denoted by $V(p) = \{v_1, \ldots, v_k\}$, $V(p) \subseteq \mathcal{V}$:
$p = \{(v_1, p(v_1)), \ldots, (v_k, p(v_k))\},\ p(v_i) \in D_{v_i}$.

The definition of a *planning task* (or *planning problem*) used in this work is based on the SAS$^+$ formalism (Bäckström and Nebel, 1995), extended with operator costs:

Definition 1. *A* Planning Task *is a 4-tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ *consisting of*

- *a finite set of* variables $\mathcal{V}$*,*
- *a finite set of* operators $\mathcal{O}$*. Each operator* $o \in \mathcal{O}$ *is defined as a triple* $\langle pre_o, eff_o, c(o) \rangle$ *where* pre_o *and* eff_o *are partial states capturing the* preconditions *and* effects *of the operator, and* $c(o) \in \mathbb{R}_0^+$ *is its* cost *value,*
- *an* initial state s_0*,*
- *and a partial state* $s_\star$*, called the* goal*. For every variable* $v \in V(s_\star)$*, we call* $s_\star(v)$ *its* goal value*.*

The semantics of a planning task on the basis of states is formalized in a *Labeled Transition System.*

Definition 2. *A* Labeled Transition System (LTS) *for a given planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ *is a tuple* $\Theta(\Pi) = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{S}^G \rangle$ *containing*

- *a set of all possible* states *of the planning task,* $\mathcal{S}$*,*
- *a set of* labels*,* $\mathcal{L}$*, which correspond to the operators* $\mathcal{O}$ *in the planning task. We denote the cost of the operators corresponding to a label* $l \in \mathcal{L}$ *by* $c(l)$*,*

- *a set of* transitions, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. *A transition* $t = (s, l, u)$, $s, u \in \mathcal{S}, l \in \mathcal{L}$, *written as* $s \xrightarrow{l} u$, *is in* $\mathcal{T}$, *iff an operator corresponding to* l *is applicable in* s *and leads to* u. *An operator* o *is* applicable *in* s, *iff* $\forall v \in V(pre_o) : \; s(v) = pre_o(v)$. *Applying an operator* o *in a state* s *leads to a state* u *in which the following assignment holds:*
  $$\forall v \in \mathcal{V} : \; u(v) = \begin{cases} \mathit{eff}_o(v) & \text{if } v \in V(\mathit{eff}_o) \\ s(v) & \text{otherwise} \end{cases}$$

- *the initial state,* $\mathcal{I} \in \mathcal{S}$,

- *and the set of goal states,* $\mathcal{S}^G \subseteq \mathcal{S}$, *with* $s_g \in \mathcal{S}^G$ *iff* $\forall v \in V(s_\star) : \; s_g(v) = s_\star(v)$.

An LTS can be visualized as a directed graph (a *state transition graph*), where a state is represented by a node, and the transitions as edges.

A planning task can be decomposed into several LTSs. We consider the formalism by Sievers et al. (2014), where a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ is represented by a set of LTSs, $X = \{\Theta_1, \ldots, \Theta_k\}$, using the same labels $\mathcal{L}$ that correspond to the operators $\mathcal{O}$. X is initialized with *Atomic Transition Systems* (ATS), one for every variable in the planning task: $X = \{ATS_v(\Theta(\Pi)) \mid v \in \mathcal{V}\}$.

Definition 3. *Given the LTS* $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{S}^G \rangle$ *corresponding to a planning task and a variable* v *in that task, we define the projection of a (partial) state* s *into the ATS of* v *as a function*

$$p_v(s) = \begin{cases} \{(v, s(v))\} & \text{if } v \in V(s) \\ \varnothing & \text{otherwise} \end{cases}$$

The Atomic Transition System *of* v *is a labeled transition system* $ATS_v(\Theta) = \langle \mathcal{S}_v, \mathcal{L}, \mathcal{T}_v, \mathcal{I}_v, \mathcal{S}_v^G \rangle$, *where* $\mathcal{S}_v = \{p_v(s) \mid s \in \mathcal{S}\}$, $\mathcal{T}_v = \{(p_v(s), l, p_v(u)) \mid (s, l, u) \in \mathcal{T}\}$, $\mathcal{I}_v = p_v(\mathcal{I})$ *and* $\mathcal{S}_v^G = p_v(\mathcal{S}^G)$.

Two LTSs can then be *merged*, meaning they are replaced in X with their *synchronised product.*

Definition 4. *Given two labeled transition systems with the same set of labels,* $\Theta_i = \langle \mathcal{S}_i, \mathcal{L}, \mathcal{T}_i, \mathcal{I}_i, \mathcal{S}_i^G \rangle$ *and* $\Theta_j = \langle \mathcal{S}_j, \mathcal{L}, \mathcal{T}_j, \mathcal{I}_j, \mathcal{S}_j^G \rangle$, *we define their* Synchronised Product, *written as* $\Theta_i \otimes \Theta_j$, *as* $\langle \mathcal{S}_i \times \mathcal{S}_j, \mathcal{L}, \mathcal{T}^\otimes, (\mathcal{I}_i \mathcal{I}_j), \mathcal{S}_i^G \times \mathcal{S}_j^G \rangle$, *where* $(s_i s_j) \xrightarrow{l} (t_i t_j) \in \mathcal{T}^\otimes$ *iff* $s_i \xrightarrow{l} t_i \in \mathcal{T}_i$ *and* $s_j \xrightarrow{l} t_j \in \mathcal{T}_j$.

The synchronisation of all atomic transition systems yields one single LTS which completely represents the planning task, as described in Definition 2. We call this LTS the *state space* of the problem.

We further define the *domain transition graph*, to capture the internal structure of a variable, and the *causal graph*, to capture the relations between variables:

Definition 5. *Given a planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$*, we define the* Domain Transition Graph (DTG) *of a variable* $v \in \mathcal{V}$ *using* $\mathcal{O}$*, written as* $DTG_{\mathcal{O}}(v)$*, as a directed graph containing one node for every value* $d \in D_v$*. The graph contains an edge from the node corresponding to* $d_1 \in D_v$ *to the node corresponding to* $d_2 \in D_v$*, iff an operator* $o \in \mathcal{O}$ *exists where* $v \in V(\mathit{eff}_o)$*,* $\mathit{eff}_o(v) = d_2$ *and either* $v \notin V(\mathit{pre}_o)$ *or* $\mathit{pre}_o(v) = d_1$*.*
We define the free DTG *of a variable* $v \in \mathcal{V}$ *as* $DTG_{\mathcal{O}_v}(v)$*, where* $\mathcal{O}_v$ *is the subset of operators which have no precondition and no effect on any other variable:*
$\mathcal{O}_v = \{o \in \mathcal{O} : \; V(\mathit{pre}_o) \subseteq \{v\}, V(\mathit{eff}_o) = \{v\}\}$*.*

Definition 6. *Given a planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$*, we define the* Causal Graph *of the task as the directed graph* $CG(\Pi) = \langle \mathcal{V}, E \rangle$*, where* $\mathcal{V}$ *are the nodes and* E *the edges.* E *contains an edge* (v_1, v_2)*, iff there exists an operator* $o \in \mathcal{O}$ *where* $v_2 \in V(\mathit{eff}_o)$ *and* $v_1 \in (V(\mathit{pre}_o) \cup V(\mathit{eff}_o))$*.*

We call a variable $v_1 \in \mathcal{V}$ *irrelevant*, if there exists no path in the causal graph from that variable to any variable $v_2 \in \mathcal{V}$ where $v_2 \in V(s_\star)$, meaning that v_1 has no influence on any variable for which a goal value is defined. We call a variable *static*, if no state can be reached in which an operator is applicable that has an effect on this variable.

A solution to a planning task, called a *plan*, is a finite sequence of operators, $\langle o_1, \ldots, o_n \rangle$, which, if applied successively starting from the initial state, lead to a goal state. A planning task is *solvable*, if such a plan exists, and *unsolvable* otherwise. The cost of a plan equals the sum of the operator costs in the sequence. A plan is called *optimal*, if it has minimal cost among all plans for the planning task.

Satisficing Planning is the problem of finding any plan for a given planning task, *optimal planning* is the problem of finding an optimal plan for the task. We call a static pruning method *solution-preserving*, if at least one plan exists after the application of the method, and *optimality-preserving*, if at least one optimal plan is preserved.

3. Static Pruning Methods Revisited

In this chapter, we explain three existing pruning techniques, which will be examined in this work; two of which are solution-preserving, and one optimality-preserving. Further, we investigate some of their properties and possible extensions.

3.1. Safe Abstraction

The technique of Safe Abstractions has firstly been described by Helmert (2006). Safe Abstraction (SA) aims to abstract variables in a problem, so the search algorithm can run on a smaller, abstracted problem without dependencies on these variables. We formalize the abstraction of a variable from a planning task as follows:

Definition 7. *Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ and a variable $v \in \mathcal{V}$, we define the effect of abstracting v from the planning task on a (partial) state s as a function*

$$a_v(s) = \begin{cases} s \setminus \{(v, s(v))\} & \textit{if } v \in V(s) \\ s & \textit{otherwise} \end{cases}$$

Using that function, the planning task that results from abstracting v from Π is defined as follows: $SA_v(\Pi) = \langle \mathcal{V} \setminus v, \mathcal{O}', a_v(s_0), a_v(s_\star) \rangle$ where the set of operators is defined as $\mathcal{O}' = \{\langle a_v(pre_o), a_v(eff_o), c(o) \rangle \mid o \in \mathcal{O}\}$.

A special property of Safe Abstractions is that they fulfill the *downward refinement* property: Every plan for the abstract problem can be extended into a plan for the concrete problem, meaning no additional states are reachable in the abstract problem compared to the concrete problem. This property is used in the formalization of SA by Haslum (2007), to define *safe* (*safely abstractable*) variables:

Definition 8. *Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, we call a variable $v \in \mathcal{V}$* SAFE, *if $SA_v(\Pi)$ fulfills the* downward refinement *property, meaning that a plan for $SA_v(\Pi)$ can be refined into a plan for Π by inserting operators from the set $\{o \in \mathcal{O} \mid V(pre_o) \subseteq \{v\}, V(eff_o) = \{v\}\}$ which only depend on v and only affect v.*

The original concept of SA by Helmert (2006) used the following conditions for safe variables:

Theorem 1. (Helmert, 2006)
Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, a variable $v \in \mathcal{V}$ is SAFE, *if its domain*

transition graph $DTG_{\mathcal{O}}(v)$ is strongly connected and v is a source node in the causal graph $CG(\Pi)$.

The intuition behind these conditions is that if a variable can arbitrarily take on any value independently of any other variable, it is ensured that any plan for the problem resulting from removing that variable, called an *abstract plan*, can be modified to a plan for the original problem, called a *concrete plan.* Whenever an operator in the abstract plan has a precondition on the value of an abstracted variable, the precondition can be satisfied by inserting operators into the plan that change the variable's value as required. If there exists a goal condition on the abstracted variable, it can be satisfied in a similar way by inserting operators at the end of the abstract plan. We call this process of modifying an abstract plan to be a concrete plan *refining* the abstract plan.

The process of abstracting a set of safe variables $\{v_1, \ldots, v_k\} \subseteq \mathcal{V}$ from a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ removes the safe variables and any precondition, effect, or goal condition on them, creating an abstract planning task $\Pi' = \mathrm{SA}_{v_k}(\mathrm{SA}_{v_{k-1}}(\ldots(\mathrm{SA}_{v_1}(\Pi))\ldots))$. It is possible that further variables become safe in Π' and can be abstracted, since some operators lost dependencies they had in Π. We call this process a *cascading abstraction.* It can be repeated, until no more variables become safe.

The search is then executed on the most abstract planning task, yielding a plan for it if the problem is solvable. It is possible that after the abstraction process every variable with a goal condition has been abstracted, in which case the abstract problem is solved by a plan of zero length. No actual search is required to solve the problem in this case, we say that the problem has been *solved by Safe Abstraction.*

Since variables which were abstracted lastly have preconditions on previously abstracted variables, the refining of the abstract plan also needs to be performed in a successive way, inserting the abstracted variables in the reversed ordering. With every inserted variable, the plan becomes more concrete.

As Haslum (2007) explains, the conditions by Helmert are stronger than they need to be to make a variable safe. To ensure that every abstract plan can be refined, it is not necessary that every abstracted variable is a source node in the causal graph, and only a subset of the variable's values needs to be connected in the variable's free DTG. Haslum identifies the relevant conditions based on the following definitions:

Definition 9. *Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ and a variable $v \in \mathcal{V}$, we call a value $d_2 \in D_v$* FREE REACHABLE *from another value $d_1 \in D_v$, iff a path exists from d_1 to d_2 in $DTG_{\mathcal{O}_v}(v)$.*

We call a value $d \in \mathcal{D}_v$ EXTERNALLY REQUIRED, *iff there exists an operator $o \in \mathcal{O}$ such that $v \in V(pre_o)$, $pre_o(v) = d$ and $V(eff_o) \not\subseteq \{v\}$, meaning o has d as a precondition on v, and at least one effect on a variable other than v.*

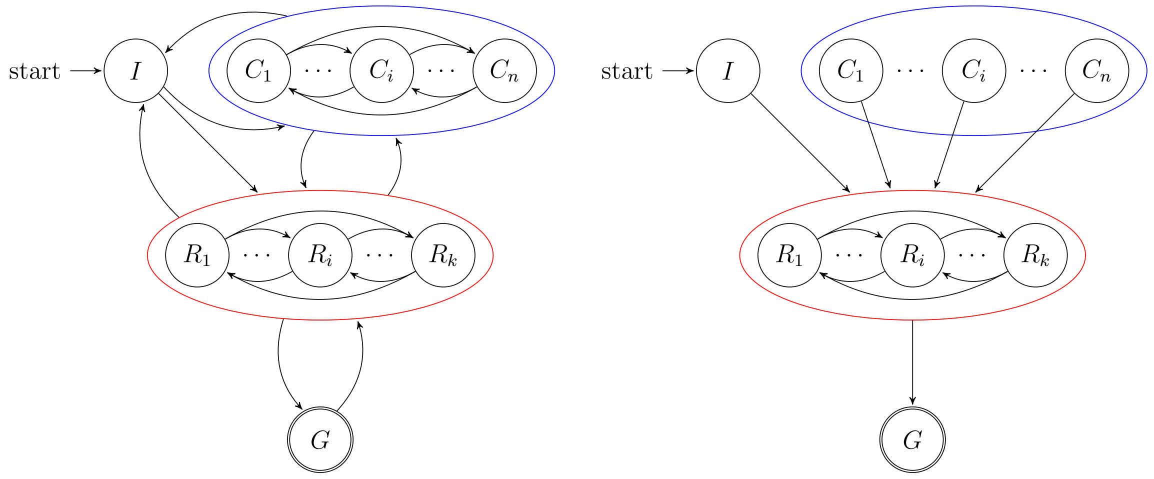
We call a value $d \in \mathcal{D}_v$ EXTERNALLY CAUSED*, iff there exists an operator $o \in \mathcal{O}$ such that $v \in V(\mathit{eff}_o)$, $\mathit{eff}_o(v) = d$ and $V(\mathit{eff}_o) \not\subseteq \{v\}$, meaning o has d as an effect on v, and at least one effect on a variable other than v.*

Theorem 2. (Haslum, 2007)
Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, a variable $v \in \mathcal{V}$ is SAFE*, if all externally required values of v are strongly connected in its free DTG and free reachable from every externally caused value of v, and the goal value of v (if any) is free reachable from each externally required value.*

The intuition behind Theorem 2 is that every value the abstract plan might require a previously abstracted variable to have must be free reachable from every value the variable might take on as a consequence of the plan. This ensures that whenever a value of the variable is needed, an operator sequence exists which assigns this value to the variable and is independent of other variables.

Figure 3.1 visualizes the different conditions for Safe Abstraction on the free DTG of a variable. The two depicted graphs show which structure has to exist in the free DTG of the variable to make it safe. With Helmert's condition, shown in Subfigure (a), the free DTG must be strongly connected, meaning that every value has to be free reachable from every other. Since free reachability is a transitive relation, at least one of the caused/required values has to be free reachable from values outside of the group, which is depicted by a single edge to and from the group of values.
As Helmert's condition is stronger than it needs to be, some of the connections in the graph are not necessary to make the variable safe. For example, it is not necessary that other values are free reachable from the goal state G, since that value is only required at the end of the plan - except G was an externally required value.
In Haslum's condition, shown in Subfigure (b), fewer connections are required. The initial value and the externally caused values do not need to be free reachable, except they overlap with the externally required values or contain the goal value. Similarly, the values of the variable do not need to be free reachable from the goal value.

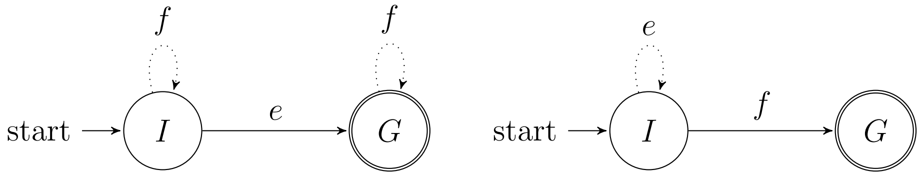
Considering Theorem 2, one special case exists we'd like to discuss. When a variable has no externally required values, and the goal value is not free reachable from the initial state or an externally caused value, then the variable is safe according to the theorem. If it is abstracted, however, the previously described straight-forward refining algorithm is not necessarily going to produce a plan for the original problem. Figure 3.2 shows an example instance in which this applies, a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ with the variables $\mathcal{V} = \{v_1, v_2\}$, the domains $D_{v_1} = D_{v_2} = \{I, G\}$, the initial state $s_0 = (I, I)$, the goal state $s_\star = (G, G)$, and the operators $\mathcal{O} = \{e, f\}$ where e has the preconditions $\mathrm{pre}_e(v_1) = I, \mathrm{pre}_e(v_2) = I$ and the effect $v_1 \leftarrow G$, f has the precondition $\mathrm{pre}_f(v_2) = I$ and the effect $v_2 \leftarrow G$.



(a) Helmert's condition

(b) Haslum's condition

Figure 3.1.: Visualization of the difference between Helmert's and Haslum's conditions for an example variable with the initial value I, the goal value G, the externally caused values $C_1, \dots, C_n$ and the externally required values $R_1, \dots, R_k$. An edge represents the free reachability of one value from another, by applying one or several operators which do not have any precondition or effect on any other variable. The externally caused values are grouped together (marked in blue), as are the externally required values (marked in red).



(a) DTG of variable v_1. f (dotted transitions) has no precondition on this variable.

(b) DTG of variable v_2. e (dotted transition) is only applicable if v_2 has the value I.

Figure 3.2.: Example instance where Theorem 2 allows to abstract v_1 which is unsafe.

According to Haslum's condition, v_1 is safe, since every part of the condition is based on the free reachability or strong connectedness of the externally required values, and no externally required value exists in v_1. After abstracting v_1, a plan for the resulting problem is $< f >$. Refining that plan would insert e at the end of the plan, since no value of the abstracted variable is required throughout the plan except for the goal value in the end. $< f, e >$ is however not a plan for the concrete problem, since e is not applicable after applying f. As we can form a plan by inserting e before f, the downward refinement property holds here. However, the intuition behind safe variables is violated since it is not possible to reach G in v_1 independently of v_2. To refine this

plan correctly, the algorithm would need to look ahead, and detect that e needs to be applied before f. In this example, v_2 would be safely abstractable and v_1 subsequently without refinement problems, but with more than two variables this is not necessarily the case.
We propose to capture this special case with a further condition in Theorem 2: If the considered variable has no externally required values but a goal value, then the goal value must be free reachable from every externally caused value and the initial value.

3.2. Redundant Operator Reduction

The aim of the technique described by Haslum and Jonsson (2000), which we call *Redundant Operator Reduction* (ROR), is to remove *redundant* operators from the planning task. An operator is called redundant, if we can replace its application in any plan by a sequence of different operators and the resulting sequence, although potentially longer, will still be a plan for the problem. The state-transition graph of a problem with redundant operators contains many transitions which are not directly necessary to make the problem solvable, meaning it is denser than it needs to be. Removing those operators results in a decrease of the branching factor, so fewer edges have to be considered. This can potentially speed up the search, as long as the solution depth (the plan length) does not increase too much.

The original description of the technique by Haslum and Jonsson (2000) is based on the `STRIPS` problem formulation. In `STRIPS`, every variable is binary, and preconditions/goal conditions only exist on the `True`-value of the variables. Consequently, an effect making a variable `True` can only increase the number of applicable operators, which is called an *add*-effect, while a *delete*-effect sets a variable to `False` and possibly causes operator/goal conditions to not be satisfied anymore. As we consider non-binary variables in this work, such a split is not possible. Every effect is simultaneously an add-effect for the assigned value and a delete-effect for every other value of the variable's domain. For that reason, we rephrase the original definitions (Def. 3 and 4 in Haslum and Jonsson (2000)) in a `SAS+` formulation.

To detect an operator $o \in \mathcal{O}$ as redundant, an operator sequence must exist that *implements* o, meaning it is applicable in all states in which o can be applied, and it must lead to the same state o would lead to. A concept needed to define this implements relation is that of *cumulative* effects and preconditions of an operator sequence, which is defined inductively in Definition 10: When the sequence is extended by an operator, its cumulative preconditions grow by those variable assignments which aren't ensured by the cumulative effects of the sequence already. Only when these cumulative preconditions are met in a state, the entire operator sequence can be applied successively. The cumulative effects are extended by the effects of the new operator if the sequence had no effect on

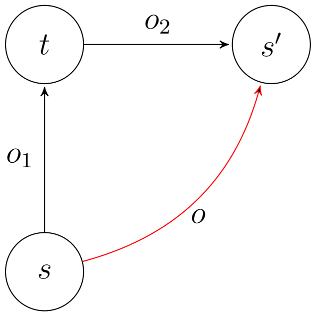


Figure 3.3.: Redundant operator: o (marked red) is implemented by $\langle o_1, o_2 \rangle$.

the respective value, and overwritten by the new value if the sequence had an effect on this variable. The cumulative effects and preconditions together form the minimal assignment which is met after applying the operator sequence.

Definition 10. *We define the* CUMULATIVE PRECONDITIONS *and* EFFECTS *of a sequence of* SAS⁺ *operators, denoted* $pre_{\dots}$, $eff_{\dots}$ *respectively, as*
$pre_{o_1,\dots,o_n,o} = pre_{o_1,\dots,o_n} \cup \{(var, val) \in pre_o \mid \nexists(var, val) \in eff_{o_1,\dots,o_n}\}$
$eff_{o_1,\dots,o_n,o} = eff_o \cup \{(var, val) \in eff_{o_1,\dots,o_n} \mid var \notin V(eff_o)\}$

Definition 11. *We say a sequence of operators* $o_1, \dots, o_n$ IMPLEMENTS *an operator* o*, iff*
(i) o *does not occur in the sequence,*
(ii) $pre_{o_1,\dots,o_n} \subseteq pre_o$
(iii) $eff_o = eff_{o_1,\dots,o_n}$

Based on Definition 11, a *redundant* operator can then be defined as an operator $o \in \mathcal{O}$ for which a sequence of operators $o_1, \dots, o_n \in (\mathcal{O} \setminus \{o\})$ exists which implements o. Figure 3.3 visualizes the state-transition graph of a problem containing a redundant operator, which is implemented by a sequence of length 2. Removing a redundant operator from the planning task means that a short-cut transition is removed in the state-transition graph, but the state is still reachable by applying a sequence of other operators, meaning no state is pruned: In the example, all three states are still reachable after pruning o.

Definition 11 differs from the formulation found in Haslum and Jonsson (2000) in the third condition, where add- and delete-effects are not separable. Also, in the original version it was allowed for an operator sequence to have *more* delete-effects than the operator it implements, iff the additional values it deletes (changing the variable's value to false) are *incompatible* with the preconditions of the operator. A variable's value is incompatible with the preconditions, if no state can be reached in which this value is assigned and the preconditions of the operator are satisfied. Consequently, deleting that value in the operator sequence does not change anything, since the value does not hold. Haslum and Jonsson explain this special case with an example from

the Blocksworld domain. The operator Move(block A, from block B, to block C) is redundant with the operator sequence of applying Unstack(block A, from block B) and Stack(block A, on block C). The operator sequence has a cumulative delete-effect, ¬ ontable(block A), as the block was moved from the table onto block C. The Move-operator does not have this delete-effect, the effect is, however, incompatible with the precondition onblock(block A, on block B), since a block can't be on top of another block and on the table simultaneously.
As in SAS$^+$ problem formulations such mutually exclusive facts are commonly expressed as different values of the same variable, the value ontable(block A) is overwritten by the value onblock(block A, on block C) in the cumulative effects of the described operator sequence, which is a value the implemented Move-operator also assigns to the variable. Therefore this special case does not need to be considered in the SAS$^+$ formulation. In the case of other mutually exclusive facts which span across multiple variables being known, they could be incorporated into Definition 11 as a modification of the third condition: If an operator sequence $o_1, \dots, o_n$ has one effect on a variable, $v \leftarrow d,\ d \in D_v$, on which the operator o has no effect, Definition 11 does not allow the sequence to implement the operator. However, if every other value $e \in (D_v \setminus \{d\})$ is mutually exclusive with at least one precondition of o, whenever o is applicable, v must already have the value d. In that case, permitting the additional effect of $o_1, \dots, o_n$ would allow to detect o as redundant. Since this exceptional case depends on the availability of additional information about mutually exclusive facts, and to keep things simple, it is not considered in Definition 11.

Haslum and Jonsson define a *reduced set* of operators as a set which does not contain any redundant operators. Often operators are responsible for implementing each other, and by removing a redundant operator another previously redundant operator is not implemented anymore. Depending on the order of operators removed, the size of the resulting reduced set can differ. As shown in the paper, finding a *minimal* reduced set is NP-hard, so in practice they make use of a greedy algorithm that removes a redundant operator directly when it is detected. To further reduce the complexity of the algorithm, they define a maximum operator sequence length, and show that in the examined instances (Blocks, Logistics, Grid) a limit of 2 operators produces almost as small operator sets as higher limits do.

As Haslum and Jonsson (2000) point out, every reduced operator brings a reduction of the branching factor, but also potentially an increase of the plan length. This means the gain from using the technique is not consistent across all planning tasks, and can lead to a speedup of the search as well as a slowdown. Consider Figure 3.4, which shows the atomic transition system of a variable in a Miconic instance. In this domain, an elevator moves up and down between floors, allowing passengers to board at their starting floor and depart at their destination. The goal is reached when all passengers are at their destination floor. An optimal plan for this problem would be to first move to the floor f_1, collect passenger p_1, drop him off on floor f_3 while boarding p_0 at the same time, and then drop off the second passenger one floor lower.

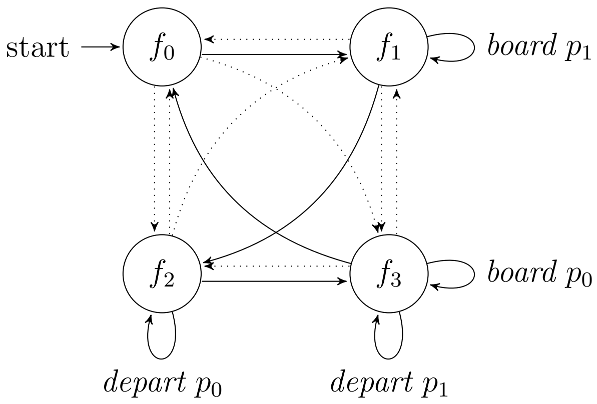


Figure 3.4.: ATS of the elevator location variable in a `Miconic` instance. The label notations at the transitions between floors have been left out, because they are not relevant. The dotted edges correspond to operators which are removed by ROR. *board* p_i and *depart* p_i are operators with effects on other variables and conditions on the elevator variable.

The elevator's position is encoded in a variable whose values are strongly connected in the graph of its ATS, so many redundant operators can be removed by ROR - in this example, it removes 8 operators. This leads to an increase in the number of states that need to be expanded (from 9 to 12 states), since some shortcuts, such as moving from floor f_1 directly to f_3, are not possible anymore.

The above example shows that it is not always beneficial in planning to remove a redundant operator. In that case, ROR is removing too many operators, since it only uses redundancy as criterion. One idea regarding this problem is that ROR could be extended with knowledge similar to what is used in the Tunnel Macros by Nissim et al. (2012). If the dynamic method of action tunneling was used after removing the redundant operators in this problem, it would tunnel the ascend from floor f_1 to f_3, since without having passenger p_0 boarded, there is nothing to be done on floor f_2 except for moving on to a different floor. A dynamically pruning version of ROR with this knowledge could avoid pruning the $f_1 \rightarrow f_3$ transition, since it is a tunnel for the implementing sequence.

3.3. Dominance Pruning

Another pruning method to speed up the search, recently researched by Torralba and Hoffmann (2015), is *Dominance Pruning* (DP), which is in contrast to the previously explained techniques optimality-preserving. It is based on the idea that some states are better to be in than others, and that a state does not need to be explored when a better state was previously considered already. Definition 12 describes the label-dominance simulation that is used to define this state relation. It is calculated on labeled transition systems, $\{\Theta_1, \ldots, \Theta_n\}$, which represent the problem, and contains those pairs of states (s, t) in an LTS where state t *simulates* s, written as $s \succeq t$. Denoting the cost of the shortest path for a given state p to a goal state as $h^*(p)$ (the *perfect heuristic*), $s \succeq t$ means that $h^*(t) <= h^*(s)$ holds, so t is "at least as good" as s. Note, that Definition 12 is based on the concept of label domination (Definition 13) and vice versa. They form a cyclic definition, which has to be accounted for using an iterative implementation.

Definition 12. (Label-Dominance Simulation). Def. 5 in Torralba and Hoffmann (2015) *Let $\mathcal{X} = \{\Theta_1, \ldots, \Theta_k\}$ be a set of LTSs sharing the same labels. Denote the states of Θ_i by S_i. A set $\mathcal{R} = \{\succeq_1, \ldots, \succeq_k\}$ of binary relations $\succeq_i \subset S_i \times S_i$ is a* LABEL-DOMINANCE SIMULATION *for $\mathcal{X}$ if, whenever $s \succeq_i t$, $s \in S_i^G$ implies that $t \in S_i^G$, and for every transition $s \xrightarrow{l} s'$ in Θ_i, there exists a transition $t \xrightarrow{l'} t'$ in Θ_i such that $c(l') <= c(l)$, $s' \succeq_i t'$, and, for all $j \neq i$, l' dominates l in Θ_j given $\succeq_j$.*

We call $\mathcal{R}$ the COARSEST *label-dominance simulation if, for every label-dominance simulation $\mathcal{R}' = \{\succeq'_1, \ldots, \succeq'_k\}$ for $\mathcal{X}$, we have $\succeq'_i \subseteq \succeq_i$ for all i.*

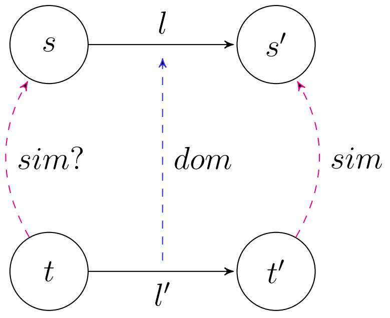


Figure 3.5.: State simulation according to Definition 12 for one example transition starting from s, for which a transition in t needs to exist which leads to a state at least as good as s'.
Whether t simulates s (*sim*?) depends on whether l' dominates l in all other LTS (*dom*) and whether t' simulates s' (*sim*)

Figure 3.5 visualizes the conditions for a pair of states to be an element in the label-dominance simulation: The state t simulates the state s, if for every transition from s to a different state s' there exists a transition from t to a different state t' which simulates s' (meaning the transition leads to a state which is at least as good as s in this LTS), and the label of the second transition *dominates* the label of the first transition in all other LTS.

The concept of label dominance is described in Definition 13, and visualised in Figure 3.6. As the definition of label dominance has a condition on the simulation relation $\succeq$, and the label-dominance simulation requires the computation of label dominance, the simulation has to be iteratively re-computed until no more change occurs. Initially, all pairs of states are in the simulation relation, except for those which can clearly never fulfill the conditions: When s is a goal state, and t is not, then it's clear that s cannot be simulated by t.

Definition 13. (Label Dominance). Def. 4 in Torralba and Hoffmann (2015)
Given an LTS $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{I}, \mathcal{S}^G \rangle$ and a label-dominance simulation $\succeq$ for a set of LTS including Θ, we say a label $l' \in \mathcal{L}$ DOMINATES *$l \in \mathcal{L}$ in Θ given $\succeq$ if for every transition $s \xrightarrow{l} s' \in \mathcal{T}$ there exists a transition $s \xrightarrow{l'} t \in \mathcal{T}$ s.t. $s' \succeq t$.*

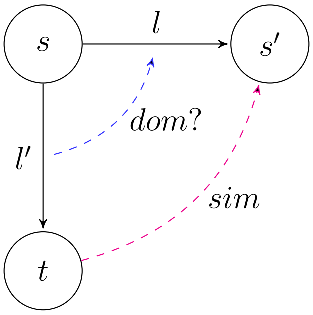


Figure 3.6.: Label dominance according to Definition 13 for one example transition $s \xrightarrow{l} s'$.
Label l' dominates label l (*dom*?) in this LTS if for every such transition of l a transition of l' exists which leads to a state that is better (*sim*) than s'.

The intuition behind Definition 13 is that whenever a label l is dominated by another label l' in one LTS, every time (an operator corresponding to) l could be applied, applying l' is possible and leads to a better state in this LTS than applying l would lead to. As shown by Kissmann and Torralba (2015), the label-dominance simulation can be utilized to prune transitions which are *subsumed*, meaning there exists a transition which is applicable in the same states and leads to a better state in all LTS. Therefore, removing the subsumed transition does not destroy any optimal plan for the problem. Definition 14 describes the conditions needed for a transition to be subsumed, which are visualized in Figure 3.7.

Definition 14. (Subsumed Transitions). Definition 2 in Kissmann and Torralba (2015) *Given a set of LTS* $\mathcal{T} = \{\Theta_1, \dots, \Theta_k\}$*, where* $\Theta_i = \langle \mathcal{S}_i, \mathcal{L}, \mathcal{T}_i, \mathcal{I}_i, \mathcal{S}_i^G \rangle$*, a transition* $s_i \xrightarrow{l} t_i \in \mathcal{T}_i$ *is* SUBSUMED *if and only if there exists another transition* $s_i \xrightarrow{l'} t_i' \in \mathcal{T}_i$ *such that* $t_i \succeq t_i'$ *and* l' *dominates* l *in all* Θ_j *for* $j \neq i$*. In that case, we say that transition* $s \xrightarrow{l} t$ *is subsumed by* $s \xrightarrow{l'} t'$*.*

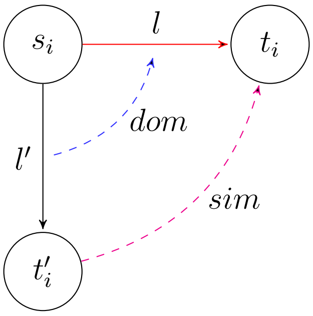


Figure 3.7.: A subsumed transition according to Definition 14. $s_i \xrightarrow{l} t_i$ (marked in red) is subsumed by the transition $s_i \xrightarrow{l'} t_i'$ if t_i' simulates t_i, and l' dominates l in all other LTS.

In some cases, the pruning of transitions allows to remove entire operators from the planning task if the corresponding label in a transition system becomes *dead*, meaning there exist no more transitions of that label. That makes it possible to use DP as a static preprocessing method, while maintaining at least one optimal plan for the problem.
As mentioned in Kissmann and Torralba (2015), we can obtain a coarser simulation by adding a *noop*-Operator without costs, that is applicable in every state and leads to no state change. Transitions which only lead away from good states to worse ones can then be subsumed by *noop*-Transitions.
By merging the labeled transition systems successively, potentially more transitions can be detected to be subsumed. Instead of then recomputing the entire simulation on the new set of LTS, Kissmann and Torralba (2015) suggest an incremental computation, in which only the simulation of the new LTS is computed. This strategy saves a lot of computation overhead, although with the drawback of obtaining a less coarse simulation.
Also, the simulation relation does not need to be computed for every state pair in the new LTS, the relation of some of them can be concluded from the known simulation in the LTS which are being merged. Assume two LTS Θ_i and Θ_j are merged, and the simulation for a state pair (s, t) is to be determined, where s and t are the synchronizations of states in the two merged LTS: $s = (s_1 s_2),\ t = (t_1 t_2),\ s_1, t_1 \in \mathcal{S}_i,\ s_2, t_2 \in \mathcal{S}_j$. It is a sufficient criterion for $s \succeq_{ij} t$, if $s_1 \succeq_i t_1$ and $s_2 \succeq_j t_2$ holds. If one of the relations does not hold, the simulation needs to be computed for the state pair.

One important question when using DP as a static preprocessing method is when subsequent pruning of entire operators is solution- and optimality-preserving. As Kissmann and Torralba (2015) point out, after pruning a single subsumed transition it is in general not safe to prune further transitions which were previously detected to be subsumed. We try to answer the question when subsequent pruning is safe in the following theorem.

Theorem 3. *A subsumed transition t_x is still subsumed after pruning a subsumed transition t_r, except t_r was only subsumed by t_x.*

Lemma 1. *Pruning a subsumed transition t_r does not remove any state pairs from the label-dominance simulation $\succeq$.*

Proof. for Lemma 1
We assume the contrary, i.e. there exists a state pair $(s, t) \in \succeq$ which is not in $\succeq$ after pruning t_r. According to Definition 12, (s, t) is in $\succeq$, if for every transition $s \xrightarrow{l} s'$ a transition $t \xrightarrow{l'} t'$ exists such that $s' \succeq t'$, l' has less or equal costs than l and dominates l in all other LTS.

If (s, t) was in $\succeq$ and is no longer in it after pruning t_r, there must exist a transition $s \xrightarrow{l} s'$, for which after pruning t_r no longer any such transition $t \xrightarrow{l'} t'$ exists, meaning either $s' \succeq t'$ does not hold anymore, or l' does not dominate l anymore in some other LTS, or the transition $t \xrightarrow{l'} t'$ was pruned. The first two cases would assume that $\succeq$ changed already for some other state pair, which has to be at some point caused by the pruning of a transition; we can assume without loss of generality that (s, t) is the first pair for which the simulation changes. For that to be the case, for some $s \xrightarrow{l} s'$, t_r needs to be the only transition $t \xrightarrow{l'} t'$ which fulfills that $s' \succeq t'$ and l' dominates l in all other LTS. Once t_r is pruned, no such transition exists anymore.
As we assumed the removed transition t_r was subsumed, that means there exists a transition $t_s = t \xrightarrow{k} u$ such that $t' \succeq u$ and k had less or equal costs than and dominated l' in all other LTS. From definitions 12 and 13 we can conclude that $\succeq$ and the domination relation are transitive, so if $s' \succeq t'$, and $t' \succeq u$, then $s' \succeq u$. Similarly for label dominance, k must also dominate l. Therefore, for the transition $s \xrightarrow{l} s'$, t_s takes over the role of t_r, as a transition $t \xrightarrow{k} u$ such that $s' \succeq u$, k has less or equal costs than and dominates l in all other LTS. That contradicts that there exists no such transition other than t_r. Therefore, all pairs (s, t) stay in the label-dominance simulation $\succeq$ (see Figure 3.8 for a visualization).

A very similar reasoning could be applied to show that the label-dominance simulation does not change at all, meaning no additional state pairs are added to it by pruning t_r. We do not show it here, as it is not needed to prove the above theorem. □

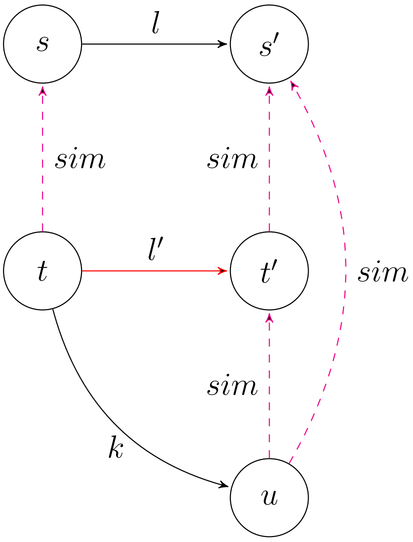


Figure 3.8.: Visualization for proving Lemma 1.
Transition $t \xrightarrow{l'} t'$ is subsumed by transition $t \xrightarrow{k} u$, whose existence ensures that $s \succeq t$ after pruning $t \xrightarrow{l'} t'$

Proof. for Theorem 3

To contradict the theorem, we assume a previously subsumed transition $t_x = s \xrightarrow{l} t$ is not subsumed anymore after pruning t_r, and that the transition which subsumed t_r is not equal to t_x. Since t_x was subsumed before pruning t_r, another transition $s \xrightarrow{l'} t'$ existed such that $s' \succeq t'$ and l' dominated l in all other LTS. t_x is assumed not to be subsumed, no such transition exists anymore. As shown in Lemma 1, $s' \succeq t'$ cannot change by pruning a subsumed transition, and therefore the dominance of l by l' does not change, either (see Definition 13). The only way for t_x not to be subsumed anymore is therefore if the subsuming transition was pruned, meaning $t_r = \; s \xrightarrow{l'} t'$, and that there exists no other transition that subsumes t_x.

As we assumed that the pruned transition t_r is subsumed itself, there exists a transition $t_s = \; s \xrightarrow{k} u'$ such that $t' \succeq u'$ and k dominates l' in all other LTS. This means that also $s' \succeq u'$ and k dominates l in all other LTS, meaning t_s also subsumes t_x. This is a contradiction to the above assumption, that t_x is not subsumed anymore. The only exception, as according to Definition 14 a transition can't subsume itself, is $t_s = t_x$, which is the case we excluded from above's theorem (see Figure 3.9. for a visualization) □

This theorem will be useful for the implementation: In successive transition pruning, for every transition we prune we only need to ensure that at least one transition that subsumes it continues to exist in the problem.

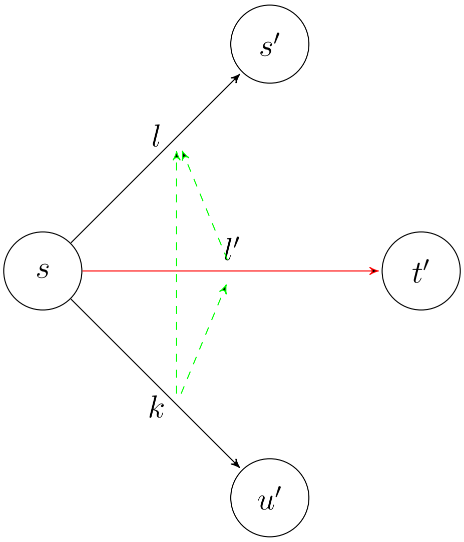


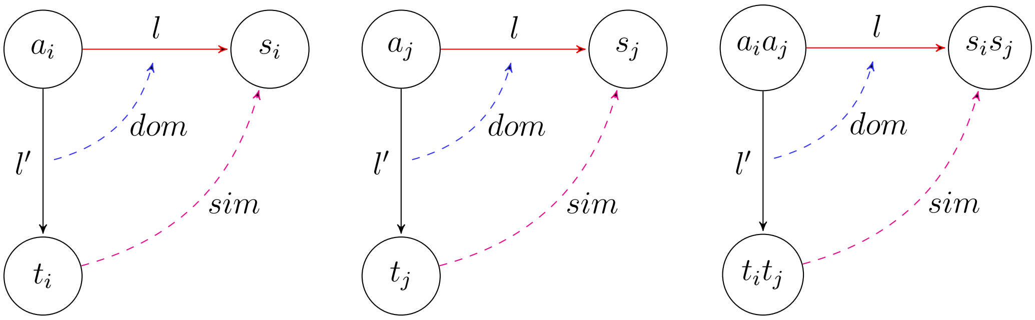
Figure 3.9.: Visualization for proving Theorem 3.
Transition $s \xrightarrow{l'} t'$ is subsumed by transition $s \xrightarrow{k} u'$, whose existence ensures that $s \xrightarrow{l} s'$ is still subsumed after pruning $s \xrightarrow{l'} t'$
Subsumption relation is marked in green.

Another question of importance for the later implementation is whether it is beneficial to prune transitions when they are subsumed, even though the corresponding label does not become dead at that point. Theorem 4 prepares for that question by showing that the pruning of subsumed transitions can be delayed to a later point in time, at which the entire label becomes dead.

Theorem 4. *Any subsumed transition is still subsumed after merging two labeled transition systems.*

Proof. Assuming a planning task represented by a set of labeled transition systems $\{\Theta_1, \dots, \Theta_n\}$, we denote the subsumed transition by $a_i \xrightarrow{l} s_i \in \mathcal{T}_i$ and the subsuming transition by $a_i \xrightarrow{l'} t_i \in \mathcal{T}_i$ such that $s_i \succeq t_i$ and l' dominates l in all other LTS.
In the case that the transition system Θ_i, in which the transition is subsumed, is merged with another LTS, Θ_j, we can conclude from l being dominated in Θ_j that for every transition $a_j \xrightarrow{l} s_j$ a transition $a_j \xrightarrow{l'} t_j$ exists such that $s_j \succeq t_j$. According to the definition of the synchronized product (see Definition 4), two transitions will therefore exist in the merged LTS: $(a_i a_j) \xrightarrow{l} (s_i s_j)$, and $(a_i a_j) \xrightarrow{l'} (t_i t_j)$. As Kissmann and Torralba (2015) discussed, we can then conclude from $s_i \succeq t_i$ and $s_j \succeq t_j$ that also $(s_i s_j) \succeq (t_i t_j)$ holds. Together with the given label domination of l by l' in the other LTS, which stayed the same during the merging process, the necessary conditions for subsuming the transition are met.

In the case that Θ_i is not among the merged transition systems, the same reasoning applies. Assuming that Θ_j is merged with Θ_k, for every transition $(a_j a_k) \xrightarrow{l} (s_j s_k)$ there exists a transition $(a_j a_k) \xrightarrow{l'} (t_j t_k)$ such that $(s_j s_k) \succeq (t_j t_k)$, which means that l stays dominated by l' in all LTS except Θ_i after the merge (see Figure 3.10 for a visualization). □



(a) Θ_i before merging. (b) Θ_j before merging. (c) $\Theta_i \otimes \Theta_j$

Figure 3.10.: Transition $a_i \xrightarrow{l} s_i$ (marked in red) is subsumed by transition $a_i \xrightarrow{l'} t_i$ before merging Θ_i and Θ_j. The corresponding transition $a_i a_j \xrightarrow{s_i s_j}$ stays subsumed after the merge in $\Theta_i \otimes \Theta_j$.

4. Combinations and Synergy Effects

In this chapter we will examine the previously introduced pruning methods in combination with each other. As all three techniques prune the search space in a different way, potentially an even smaller search space can be obtained by applying two techniques subsequently. Particularly interesting are synergies between two pruning methods, where the set of operators/variables pruned by the combination differs from the union of the pruning achieved by each technique individually. We say a pruning technique has a *positive synergy* on a subsequently applied technique, if the second method benefits from the pruning of the first one and is able to prune more than it is able to on the original problem. Equivalently, a *negative synergy* lets the second technique prune less than it would have been able to on the original problem, although less pruning is *not* considered a negative synergy when the first technique already pruned the operator/variable which the second method would have been able to. We abbreviate the three pruning techniques as SA (Safe Abstraction), ROR (Redundant Operator Reduction) and DP (Dominance Pruning), and denote them, when applied successively to a planning task, with a plus. `ROR+SA` therefore describes an application of Redundant Operator Reduction, followed by Safe Abstraction.

For some of these combinations theoretical results can prove that no negative synergies exist. In combinations where negative or positive synergies are possible, we find example instances from the IPC benchmarks or construct instances in which they occur. In this chapter, we consider Haslum's conditions for Safe Abstraction. Since the set of variables abstracted with Helmert's condition is a subset of the variables abstracted with Haslum's, proofs which show that negative synergies do not exist also hold for the Helmert version. Similarly, we consider Dominance Pruning using the coarsest simulation.

4.1. Safe Abstraction and Redundant Operator Reduction

Negative synergies of Safe Abstraction on ROR do not occur, as Theorem 5 shows. SA+ROR allows ROR to remove all operators it could have on the original task, except for those which SA already removed.

Theorem 5. *The planning task resulting from the application of SA+ROR does not contain any operator which is not contained in the planning task resulting from applying Redundant Operator Reduction.*

Proof. We assume the contrary, i.e. there exists an operator o in the planning task which is redundant, has at least one effect on a variable which is not safe abstractable (otherwise Safe Abstraction could remove this operator), and is not redundant anymore after applying Safe Abstraction.
Then, according to the definition of redundancy as implementation by an operator sequence, and Definition 11, there must exist a sequence of operators $o_1, \ldots, o_n$ in the original planning task which implements o, and after abstracting a variable v this is not the case anymore. This could have the following causes:

1. The operator sequence has a (cumulative) precondition which o does not have:
Clearly this means that a precondition of o was removed, as no preconditions are added by the abstraction process. Only preconditions on v are removed by Safe Abstraction, which means the preconditions in the operator sequence on v are also removed and the operator o stays redundant.

2. The (cumulative) effects of the operator sequence differ from those of o:
In analogy to the preconditions, only effects on v are removed, and in both the operator sequence and the operator o.

3. An operator o_i in the operator sequence is removed by Safe Abstraction:
If an operator is removed by Safe Abstraction, its only effects were on the variable v. As the preconditions on v are removed by Safe Abstraction, a shorter operator sequence without o_i now implements o and therefore o can still be removed.

□

Potentially, more operators can be removed by ROR after applying Safe Abstraction, as described in Theorem 6.

Theorem 6. *SA+ROR can have positive synergy effects, if a sequence of operators would implement an operator apart from preconditions or effects on safe variables.*

Proof. An example that shows this type of positive synergy can be found in a `Tpp` instance, where two trucks are available to move to different locations and buy goods. For each good, each location and each truck, one `buy` operator is available which moves the truck to that location and loads the good into the store. After Safe Abstraction removes the variables encoding the truck location, every two of these `buy` operators implement each other, since they become equivalent when the location of the truck is not stored; which results in the removal of all `buy` operators for one of the two trucks. □

In reversed order, `ROR+SA`, negative synergies are again not possible as shown in Theorem 7.

Theorem 7. *Safe Abstraction, executed* after *ROR, can abstract at least as many variables as Safe Abstraction can on the original task.*

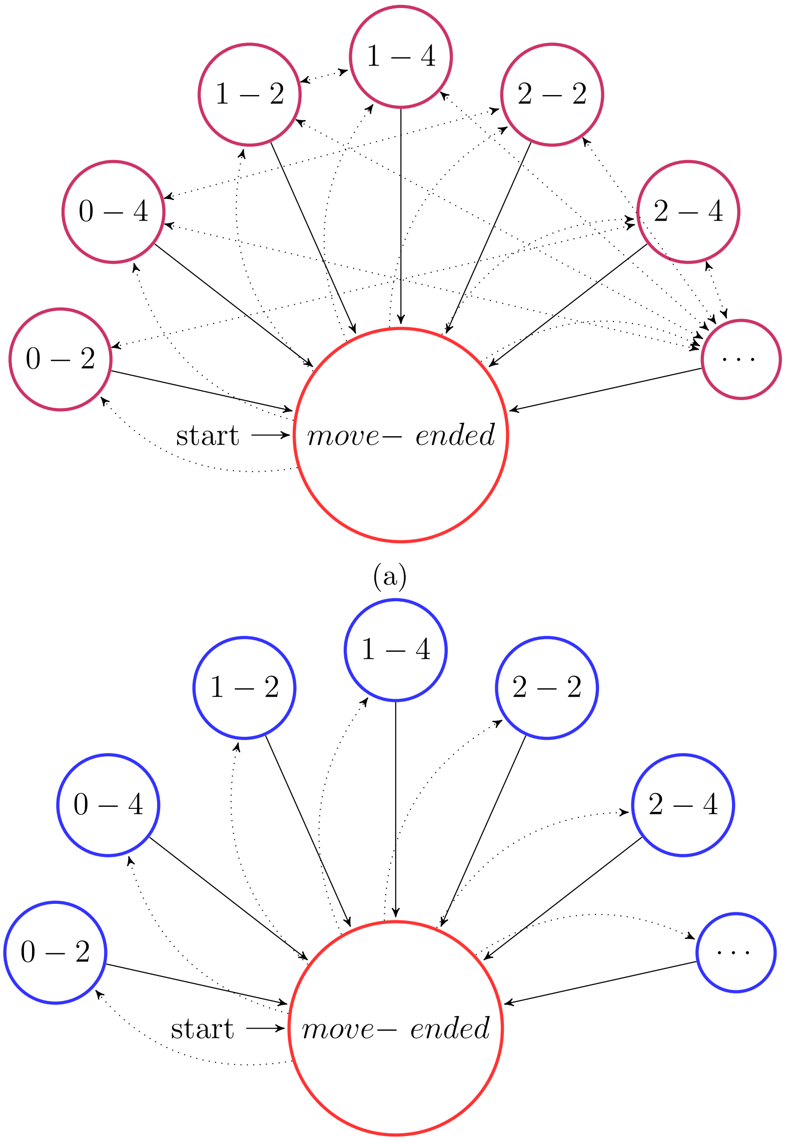
Lemma 2. *If there exists a variable v which is safe and an operator o which is redundant, then after removing o from the planning task, v is still safe.*

Proof. for Lemma 2
As explained in Theorem 2, the conditions for Safe Abstraction only depend on the free reachability of values of v. Under the assumption that after the removal of o the variable is not abstractable anymore, we can draw the conclusion that o needs to be part of the free DTG, otherwise it could not have an effect on the safe abstractability.
Being part of the free DTG of v, o does not have any precondition nor effect on any other variable. As o was assumed to be redundant, it is implemented by a sequence of operators $o_1, \ldots, o_n$, which, according to Definition 11, cannot have preconditions nor effects on variables o has not. Therefore the sequence $o_1, \ldots, o_n$ is also part of the free DTG of v. Definition 11 also ensures that $o_1, \ldots, o_n$ is applicable in at least all the states in which o is applicable, and leads to the same target states. That means the free reachability of states of v does not change by removing o from the planning task, and v can still be abstracted.
Theorem 7 directly follows from Lemma 2. □

Positive synergies of `ROR+SA` are not straight-forward to see, but do exist as the following theorem shows.

Theorem 8. *`ROR+SA` can have positive synergy effects, if redundant operators are responsible for values in a variable being externally caused or required.*

Proof. Figure 4.1 shows an example of a `Pegsol` problem instance where positive synergies are possible. We examine the DTG of a variable which keeps track of the last move that was made, omitting the details about other variables for the sake of simplicity. The straight edges resemble operators which belong to the free DTG of the variable, while the dotted edges mark operators with effects or preconditions on other variables (*external* operators).



(a)

(b)

Figure 4.1.: DTG of a variable from a `Pegsol` instance that becomes safe after removing redundant operators. The labels corresponding to the transitions are not denoted in the graphs, since they are irrelevant.

The centered value *move* – *ended* (a, marked in red) is externally required by the external operators which lead to the other values in this DTG (bent dotted edges), making those values externally caused. The other values in the DTG are also externally required

by the external operators which connect them with each other in this DTG (straight dotted edges). The variable is not safe abstractable, since the externally required values marked in purple are not free reachable from the $move - ended$ value.
All connections of the outer values (straight dotted edges) are redundant with a detour to the center value, therefore ROR can remove them. After removing the redundant operators, the outer values are not externally required anymore (b, marked in blue), hence they do not need to be free reachable from the $move - ended$ value. The variable became safe. □

4.2. Dominance Pruning and Safe Abstraction

The following two sections explore the application of Dominance Pruning in combination with a satisficing pruning technique. The consequence of such combination is that the optimality-preserving property of Dominance Pruning is lost. A decrease in time needed to solve instances this way is not to be expected, since the gain of an optimality-preserving method applied to domains used for satisficing planning is often too small compared to the overhead. Still, it is of interest to investigate their relative pruning capacities, including potential synergies, on a conceptual level. In this section, we investigate DP combined with SA.

The combination of `SA+DP` removes at least the same operators from the problem as just applying DP does: Theorem 9 shows this by proving that no operator can exist in a problem which DP is able to prune, and that is still existent and not prunable anymore after applying SA. Since DP does not actively remove any variables from the problem, but variables can become static if the relevant operators are pruned, the same can be said for variables: The combination of `SA+DP` never leaves behind a variable which could have been removed if just DP had been applied.

Theorem 9. *Applying `SA+DP` removes at least the same operators from the planning task as only applying DP to it does.*

Proof. Let $\mathcal{X} = \{\Theta_1, \dots, \Theta_n\}$ be a set of LTS representing a planning task, and let $\Theta_i \in \mathcal{X}$ be the atomic transition system of an abstracted variable v_i. In all other LTS $\Theta_j \neq \Theta_i$, Safe Abstraction only removes those transitions $s_j \xrightarrow{l} t_j \in \mathcal{T}_j$ which belong to an operator it removed. Safe Abstraction only removes an operator from the problem if it had no effect and no precondition on any non-abstracted variable. Since the removed operators had no condition and no effect on any of the remaining variables, the removed labels were dominated by the noop-label in all $\Theta_j \neq \Theta_i$. This means that the state simulation relation $\succeq_j$ does not change in any $\Theta_j \neq \Theta_i$, since for every removed transition there exists a noop-transition with the same origin, destination, and equal or less cost.

Assume Theorem 9 to be false, i.e. there exists an operator o which can be pruned by DP in the original planning task, but not anymore after Safe Abstraction has been applied abstraction, and it has not been removed by Safe Abstraction. This means a transition must exist which was previously subsumed, but is not subsumed anymore after Safe Abstraction was applied.
A transition t_o is not subsumed anymore according to Definition 14, if either the simulation relation changed, or the subsuming transition is not existent anymore. As shown above, the simulation relation is unchanged. Further, if t_o was previously subsumed by a transition t_{SA} that has been removed by Safe Abstraction, it has to be subsumed by a noop-transition t_{noop} now, since t_{SA} was subsumed by t_{noop}, and the subsumption relation is transitive (since label domination and state simulation are transitive).

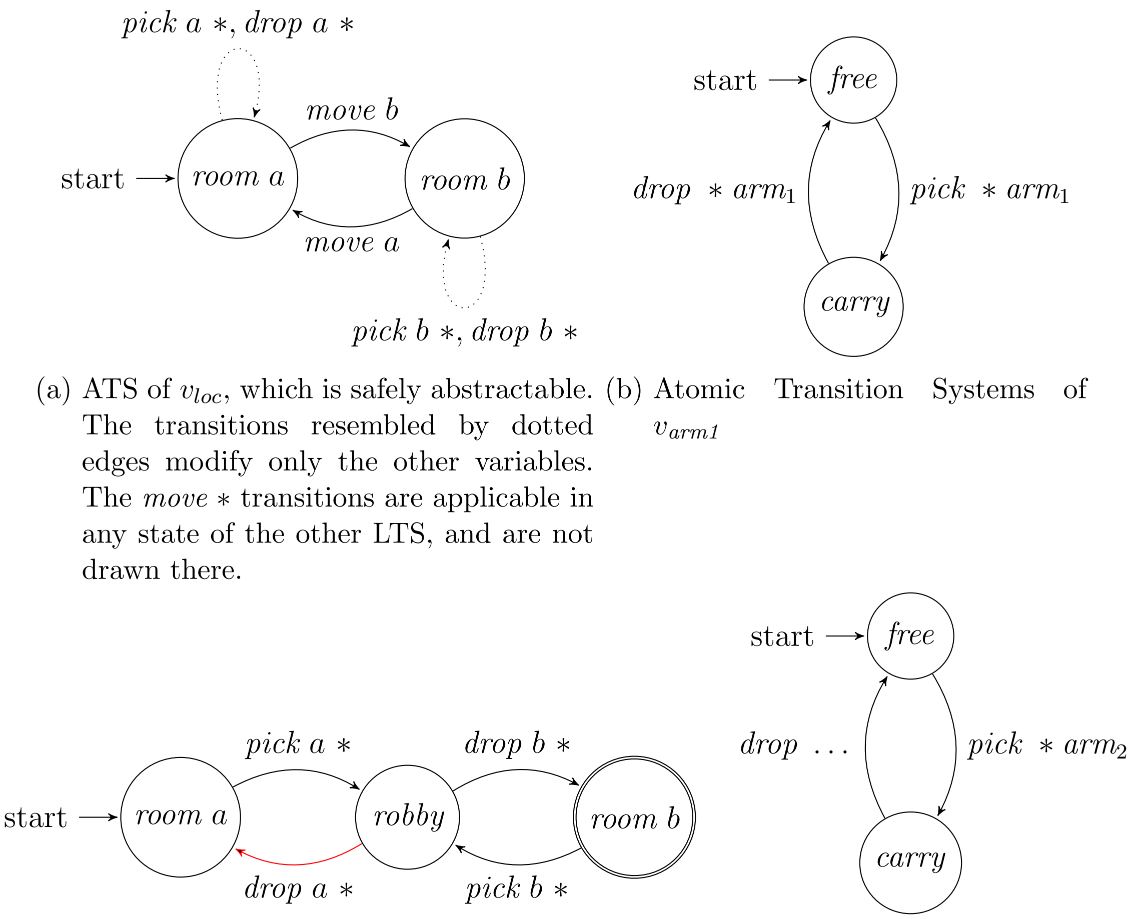
So, if a transition was subsumed before abstracting v_i and its corresponding operator was not removed by Safe Abstraction then it is still subsumed after the abstraction process. That means abstracting *one* variable does not reduce the set of operators that can be removed from the planning task by DP, and therefore abstracting an arbitrary amount of variables does not do so, either, which proves Theorem 9. □

Positive synergies of Safe Abstraction on DP are possible, meaning Dominance Pruning is able to prune operators it couldn't have otherwise.

Theorem 10. *`SA+DP` can have positive synergy effects, if for subsuming the transitions of one label, label dominance is not given in some atomic transition systems, and Safe Abstraction abstracts the variables corresponding to those ATS.*

Proof. Figure 4.2 shows an example from a `Gripper` instance, in which a ball needs to be transported from *room a* to *room b*. Four variables exist in the instance, v_{loc} encodes the roboter's location, v_{ball} the ball's location, and two arms of the roboter, v_{arm1} and v_{arm2} are either free or carry the ball. Two operators exist, *move a* and *move b*, to move the roboter from either room to the other, and for every room and every arm there exists an operator *pick a/b arm1/arm2*, to pick up the ball in that room with that arm, and equivalently *drop a/b arm1/arm2* to drop it in that room. To simplify the figure, some edges are labeled containing a placeholder $(*)$ and represent one transition for every room $\{a, b\}$, equivalently for the arms $\{arm_1, arm_2\}$.

Before abstracting the roboter's location v_{loc}, DP can't prune any operators. The *drop a* $*$ transitions (in ATS c, edge marked in red) lead to a worse state than *drop b* $*$, but the corresponding label is not dominated in ATS (a) since *drop b* $*$ is not applicable in the state *drop a* $*$ is applicable in. The label is dominated in ATS (b) and (d), so after abstracting v_{loc}, the transitions can be pruned and the labels *drop a* $*$ become dead, the operators can be removed. □



(a) ATS of v_{loc}, which is safely abstractable. The transitions resembled by dotted edges modify only the other variables. The *move* ∗ transitions are applicable in any state of the other LTS, and are not drawn there.

(b) Atomic Transition Systems of v_{arm1}

(c) Atomic Transition System of v_{ball}.

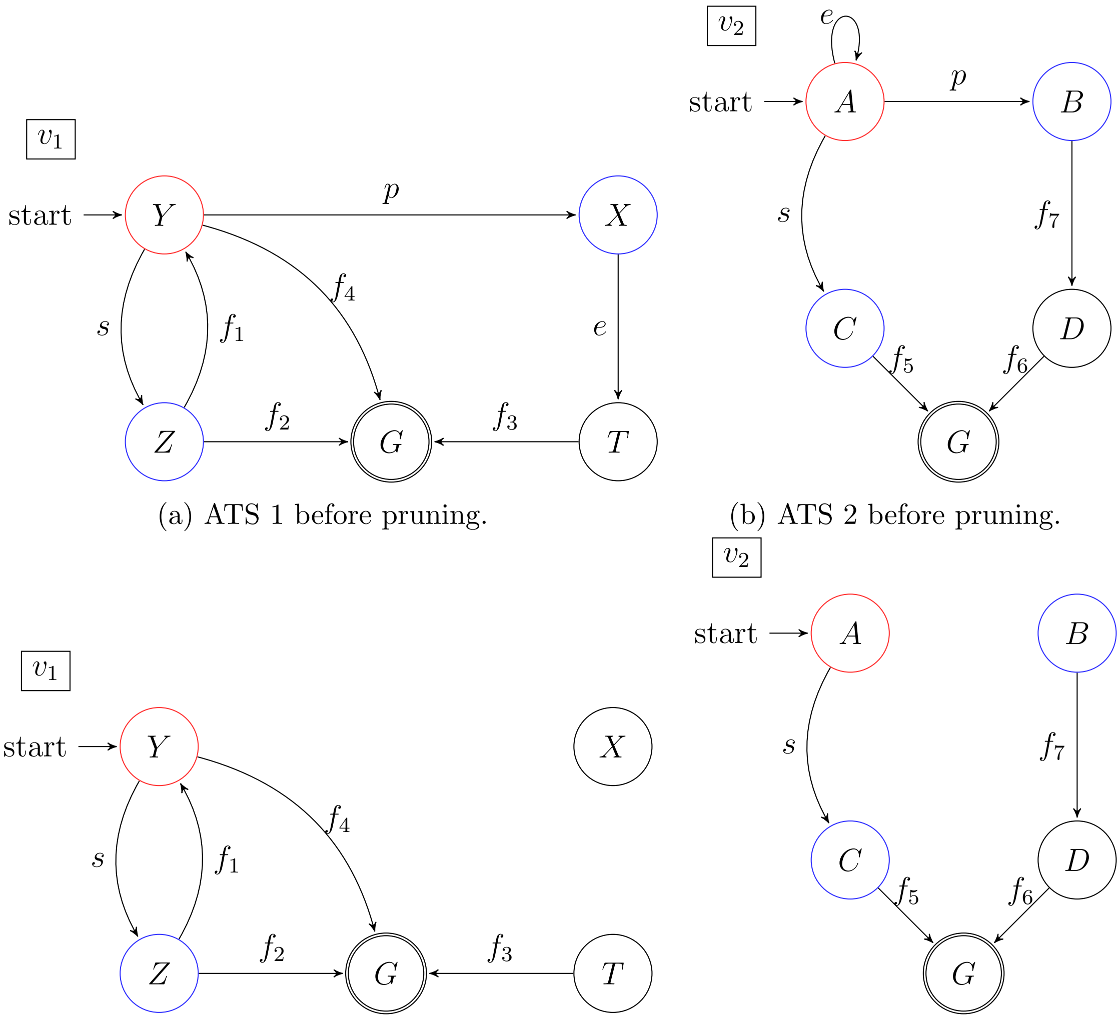
(d) Atomic Transition Systems of v_{arm2}

Figure 4.2.: Example of a positive synergy of SA on DP from the `Gripper` domain.

In the reverse direction, `DP+SA`, both positive and negative synergies are possible, as shown in Theorem 11 and 12.

Theorem 11. *`DP+SA` can have positive synergy effects, where a variable becomes safe by pruning operators.*

Proof. Figure 4.3 visualizes a positive synergy in a constructed example. The instance contains two variables, v_1 and v_2, whose ATS are depicted in (a) and (b). The operators f_1, f_2, f_3, f_4 do not have any precondition or effect on v_2, and are not depicted in the subfigure (b) since they're applicable in each of the states. Equivalently, the operators f_5, f_6, f_7 are not drawn in the ATS 1.
If Safe Abstraction is applied to that example, it is not able to abstract any variable, but



(a) ATS 1 before pruning.

(b) ATS 2 before pruning.

(c) ATS 1 after pruning.

(d) ATS 2 after pruning.

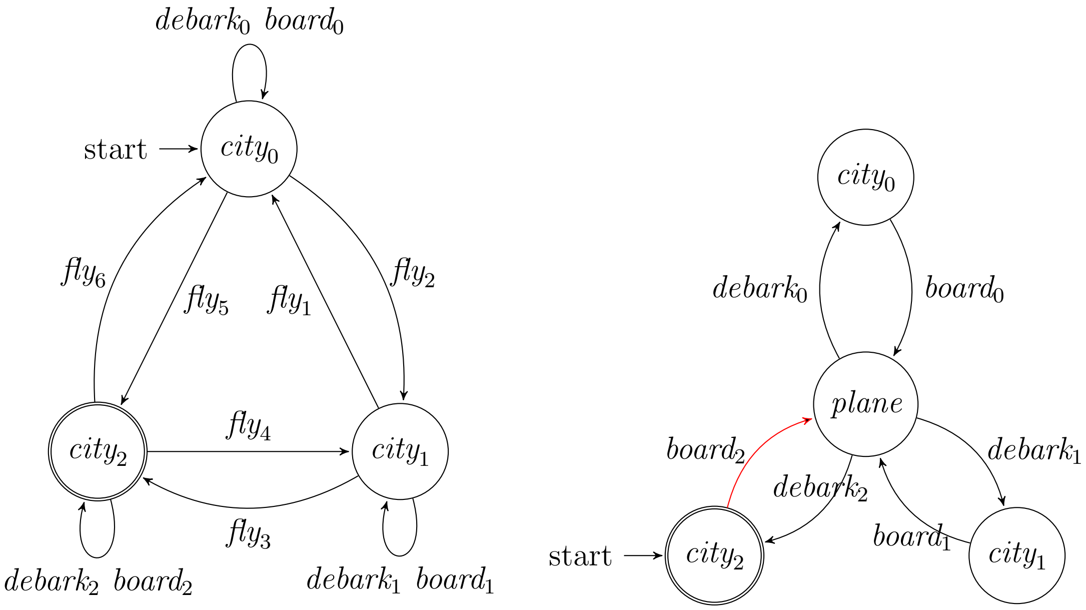
Figure 4.3.: Constructed example to show the positive synergy of DP on SA. Externally required values are marked in red, externally caused values in blue.

after applying Dominance Pruning it can do so. v_1 is not safe (a), as the goal value G is not free reachable from the externally caused value X. v_2 is also not safe (b), as the externally required value A is not free reachable from the externally caused values B and C.

Transition $Y \xrightarrow{p} X$ is subsumed in ATS 1 by $Y \xrightarrow{s} Z$, and the label s dominates label p in ATS 2, so after applying Dominance Pruning, p becomes dead in this LTS. After merging the two transition systems it is possible to prune the operator e as well.

After pruning the p transition in ATS 1 (c), X is not externally caused anymore, with the consequence that v_1 becomes safe: Y is free reachable from Z, and G is free reachable from Y. v_2 is still not safe (d), as the goal value G is not free reachable from the externally required value A, although the variable becomes safe after abstracting v_1. □

In practice, these positive synergies often appear when a variable becomes static through Dominance Pruning, an example from the IPC benchmarks is shown in Figure 4.4. In the `Zenotravel` instance depicted, a plane can fly between three cities (a), allowing a passenger to board and debark from it (b). In this instance, the passenger is already at his destination initially, and only the plane needs to reach city$_2$. In the passenger LTS (b), the transition $\mathit{city}_2 \xrightarrow{\mathit{board}_2} \mathit{plane}$ (marked in red) is subsumed by $\mathit{city}_2 \xrightarrow{\mathit{noop}} \mathit{city}_2$, since the transition leads away from a goal state. After removal of the transition, all *board* and *debark* operators become inapplicable and can be removed. There are no operators left with conditions/effects on the passenger variable, and the initial state is equal to the goal state, thus the variable can be safely abstracted. A positive synergy with Safe Abstraction apart from static variables is not common, it appears in none of the examined domains (more details later in Chapter 6).



(a) Airport LTS

(b) Passenger LTS. The fly_* transitions are applicable in every state and have not been drawn.

Figure 4.4.: Example instance from the `Zenotravel` domain, where a variable becomes static after applying DP.

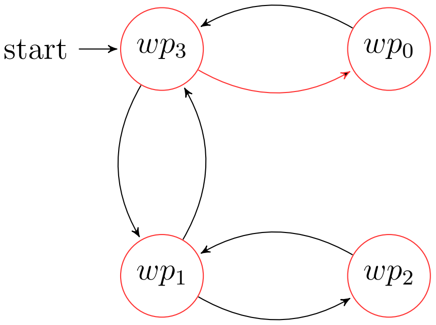


Figure 4.5.: Example instance of a negative synergy of DP on SA from the `Rovers` domain.

Theorem 12. *`DP+SA` can have negative synergy effects, when Dominance Pruning removes an operator from the planning task which is necessary to keep a variable safe. We can conclude that the removed operator was needed to ensure the free reachability of a value, so it had no effects and conditions on any other variable.*

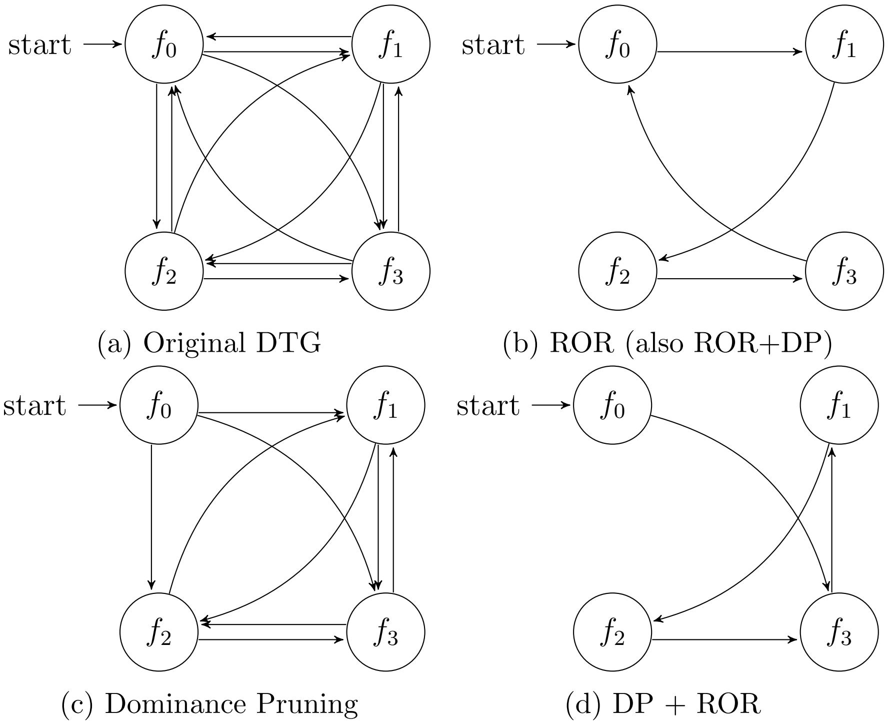
Proof. An example of this behavior is given in Figure 4.5, which shows the free DTG of a `Rovers` variable that becomes unsafe after applying Dominance Pruning to the problem. The variable encodes the rover's location at 4 waypoints $wp_0, \ldots, wp_3$, all of which are externally required values. The labels are not denoted in the graph, since they are not relevant; we again omit the details about other variables in this instance.
The operator corresponding to the transition marked in red can be removed by Dominance Pruning, rendering the variable unsafe as wp_0 is not free reachable anymore.

In the case of this instance, the value wp_0 becomes unreachable and could be removed from the problem, making the variable safe again in turn. □

4.3. Redundant Operator Reduction and Dominance Pruning

The pruning power of these two methods seems to overlap much more than is the case with Safe Abstraction in the sense that often for one operator that DP prunes, ROR can remove one operator less - though not necessarily the same operator - and the same is true in the reverse direction. This is especially the case with strongly connected DTGs, as an example in Figure 4.6 shows. Since Dominance Pruning is an optimality-preserving method, the operators it leaves behind are better than those left by ROR, in the sense that the plans do not get any longer.

Theorem 13. *Both positive and negative synergies of combining ROR with DP are possible, in both directions.*



(a) Original DTG

(b) ROR (also ROR+DP)

(c) Dominance Pruning

(d) DP + ROR

Figure 4.6.: Example `Miconic` instance, showing the overlapping pruning/reducing of DP and ROR. The labels corresponding to the transitions are not denoted, since they are irrelevant.
(a) DTG of the variable encoding the elevator's location.
(b) Applying ROR to the problem thins out the strongly connected DTG, after which DP is not able to prune anything.
(c) Applying Dominance Pruning first, it is detected that a return to the state f_0 is not necessary and the operators leading to it can be pruned.
(d) When ROR is applied after DP, it again thins out the DTG, the same number of operators stays in the problem as when just using ROR, but the plan can potentially be shorter, as reaching f_2 from f_3 does not need the detour over f_0 as it does in (b).

We do not show examples of these synergies here, since that would require to explain the label-dominance simulation in detail for instances with a bigger number of variables.

One interesting case to look at are the positive synergies of DP on ROR, which are only possible when by pruning an operator a variable becomes static and can be removed. In that case, it is possible for an operator to become redundant, if its conditions/effects differed only on that variable from the cumulative conditions/effects of a now implementing sequence, and since the variable is removed, the sequence is able to implement the operator. Otherwise, positive synergies are not possible, as shown in Theorem 14.

Theorem 14. *An operator o, which is not redundant, cannot become redundant by applying DP to the problem, except a variable becomes static and can be removed.*

Proof. The effect of DP on the planning task, if no variable becomes static, is a reduction of the set of operators $\mathcal{O}$ to $\mathcal{O}'$. As o is assumed not to be redundant, no sequence of operators in $\mathcal{O}$ exists which implements it, and since $\mathcal{O}' \subseteq \mathcal{O}$, no such sequence can exist in $\mathcal{O}'$. □

4.4. Summary on Synergy Effects

To conclude this chapter, we summarize the information gathered about the synergies. Consider Table 4.1, which lists each of the combinations.

Combination	Positive Synergies	Negative Synergies
SA+ROR	yes	no
ROR+SA	yes	no
SA+DP	yes	no
DP+SA	(yes)	yes
ROR+DP	yes	yes
DP+ROR	(yes)	yes

Table 4.1.: Summary of synergy effects. Parenthesized entries indicate a synergy which is often or always caused by variables becoming static.

Positive synergies exist for each combination, and appear in instances of the IPC benchmarks. In `DP+SA`, the synergies are limited on these benchmarks in the sense that DP often makes variables safe when it also makes them static. In `DP+ROR`, a variable has to become static to cause positive synergies. Negative synergies are possible between ROR and DP in either direction, but also in `DP+SA`.

Considering the benefit for the search, we can conclude from these synergies which technique should preferably be applied first to the problem.
For both `SA+ROR` and `ROR+SA` no negative synergies exist, and both allow positive synergies. Since removing redundant operators is not consistently beneficial for the search as shown in Section 3.2, it can be argued that the positive synergies of `ROR+SA` are likely to be more important for reducing search effort than the positive synergies of `SA+ROR`.
Since `DP+SA` allows negative synergies, and the positive synergies are rare, `SA+DP` should be able to remove more variables and operators from any planning task. Additionally, abstracting variables before running Dominance Pruning has the benefit of speeding up the heavy simulation computation.

Both `ROR+DP` and `DP+ROR` allow negative synergies, the positive synergies in `DP+ROR` however are very infrequent. Additionally, similarly to `SA+ROR`, it can be argued that the positive synergies on DP are more important than the positive synergies on ROR since DP does not cause an increase in the plan length.

5. Implementation

In this chapter, we describe the implementation of Safe Abstraction, Redundant Operator Reduction and Dominance Pruning as static preprocessing methods into the Fast Downward (Helmert, 2006) planning system. Fast Downward is a propositional planner which consists of three components: The *Translator* takes as input a problem description specified in PDDL and translates it into a SAS$^+$-like (Bäckström and Nebel, 1995) formalism using finite-domain state variables. The *Preprocessor* performs a relevance analysis, removes irrelevant and static variables, and computes data structures such as the causal graph and the domain transition graphs. The *Search* component then runs heuristic search on the output of the preprocessor.

5.1. Safe Abstraction

Although Haslum (2007) describes other preprocessing methods than Safe Abstraction - Reformulation of problem descriptions and composition of operators - they have not been implemented in this work, as they were at least partly performed manually and it is not clear whether and how they can be fully automated.

Safe Abstraction has been implemented into the preprocessor component of Fast Downward, since after abstracting, some data structures need to be computed, for which methods are available in that component. The translator output file is parsed, and the data is being stored inside a task object, which is then passed on to the abstraction.

For every variable of the task the conditions for safe abstractability are checked. At this point, the two conditions for Safe Abstraction demand different properties: The one described by Helmert (2006) requires all values in the variable to be strongly connected, and the variable to be a source node in the causal graph. Haslum (2007) also needs a test for connectedness and reachability, but with only a selection of the values. For that purpose, the operators are separated into *external* operators, which have an effect on another variable than the examined one, and *internal* operators, which have no condition nor effect on other variables. The values of the variable's domain are separated into *externally required* (precondition to an external operator) and *externally caused* (effect of an external operator, or initial state value).

To detect whether each externally required value and the goal value are free reachable from all externally required and caused values, graph connectedness algorithms, such as the one by Tarjan (1972), can be used. While Helmert's condition can be implemented straight-forwardly with an algorithm for strong connectedness, it is not as easy to capture all relevant cases in Haslum's condition. For that reason, this implementation uses an algorithm that creates a 2-dimensional array whose entries represent free reachability. Every value is free reachable from itself, and every internal operator further adds reachability entries. As reachability is a transitive relation, the filling of the array was realized as a recursive method. The reachability-array approach has a higher complexity than Tarjan's algorithm, but it can easily be adapted to check either of the two conditions.

If at least one variable can be abstracted, a new task is generated in which the abstractable variables as well as conditions and effects of operators and goal conditions on these variables, along with entries in mutually exclusive groups, are removed. Since by abstracting variables, other variables can become safe, the conditions for safe variables are then again evaluated on all variables in the new task object, a repetitive process until either no more variable can be abstracted or no more goal conditions remain (meaning the problem is solved without search). Thus, each time the problem is abstracted, a new task is created, which is stored inside a task vector.

To be able to later on refine the abstract plan found by the search, the information is needed which variable was abstracted in which order, and which operator has preconditions and effects on the abstracted variables. For that reason, each abstracted task in that vector is saved to a file. As the preprocessor automatically performs a relevance analysis for the remaining variables, it is possible that an abstracted problem loses variables which weren't directly abstracted. These irrelevant variables cannot be part of any plan, though.

The process of refining an abstract plan loads the vector of abstracted tasks, and traverses the abstract plan (also loaded from a file output by the search engine) once for each abstraction layer. It detects which variables have been abstracted, and keeps track of them with a variable-value mapping, which is updated every time an operator in the plan requires the variable to change its value. When such a precondition on the variable exists, an operator sequence is inserted into the plan right before the value is needed. This operator sequence is determined by finding the shortest path (using the algorithm by Dijkstra (1959)) from the variable's old value to its new one in the free domain transition graph. The same procedure is done after traversing the plan, adding operator sequences to the end of it to ensure that every abstracted variable gets assigned its goal value.

Complexity Analysis of the Implementation

The complexity of determining whether a variable is abstractable is dominated by the free reachability analysis. In the recursive array implementation used, assuming n_v as the size of the variable domain, there are maximally $n_v(n_v - 1)$ write operations on the array, as the diagonal is already filled (every value is reachable from itself). For every

write operation, all possible predecessors of the source, and all possible successors of the target need to be read, leading to $O(2(n_v - 1))$ reads per write operation. Therefore, the total complexity is dominated by the number of write operations, which is $O(n_v(n_v - 1) * 2(n_v - 1)) = O(n_v^3)$. If Tarjan's strongly connectedness algorithm was used for Helmert's condition, this complexity could be reduced to $O(n_v + n_v^2)$.
In the worst-case, every variable can be abstracted, but each one only after the previous has been abstracted. With v variables that leads to $v + (v-1) + (v-2) + \cdots + 1 = \frac{v*(v+1)}{2}$ abstractability checks, meaning a total complexity for the abstraction process of $O(v^2 n_v^3)$ or, with Tarjan's algorithm, $O(v^2 n_v^2)$.

Refining a plan requires to loop through the abstract plan once for every abstraction layer but the last one. Each iteration, for each operator in the plan, the preconditions and effects on each abstracted variable need to be considered. In the worst case, each operator in the plan has a precondition on each abstracted variable, so the shortest operator sequence from the old to the new value of that variable needs to be found. The shortest-path algorithm from Dijkstra has a complexity of $O(n_v^2)$, leading to a total complexity of $O(v * p * n_v^2)$ with p as the maximum plan length of the refined plan.

5.2. Redundant Operator Reduction

The implementation of Redundant Operator Reduction is for the most part straightforward. Same as Safe Abstraction, it is located in the preprocessor part of Fast Downward.

In a first preparation step, the effects and preconditions of operators are transformed from condition/effect vectors into maps using the variable as key. This allows for faster and simpler access later on.

Then, for every operator o in the problem, a recursive method is called to look for an implementing operator sequence, using the operator sequence limit as maximum recursion depth. It takes a sequence $\langle o_1, \ldots, o_i \rangle$ as parameter, evaluates whether the sequence fulfills the conditions of implementing o (Definition 11), and returns the sequence if that's the case. The conditions used differ from those found by Haslum and Jonsson (2000), where an implementing operator sequence is allowed to have additional effects which are incompatible with preconditions of o. Because of the multi-valued domain variables and the mutually exclusive value detection of the translator, most of these incompatible value pairs appear in the problem as two values of one variable, so e.g. in the previously described `Blocks` example, the deletion of the axiom `ontable(block)` is implicitly contained in the `move` operation.

The method is initially called with an empty operator sequence $\langle\rangle$. If the conditions for implementation of o are not satisfied by the given sequence, the method tries out every possibly applicable operator (which doesn't contradict a cumulative effect of the sequence up to that point), except for o, appends the new operator to the sequence and accumulates the cumulative effects and preconditions before recursively calling itself again. One criterion to abort appending operators to the sequence, besides the recursion depth, is when a precondition of an operator is neither a precondition of o, nor is it ensured by a cumulative effect of the operator sequence, because this sequence can never be extended to one which implements o (Haslum and Jonsson, 2000).

It is important to note that this is a greedy implementation, which can produce very different results on the same planning task depending on the order of operators.

Complexity Analysis of the Implementation

Let d be the maximum recursion depth (being the maximum length of an implementing operator sequence), n_v the number of variables and n_o the number of operators. Then, the recursive method to find an implementing operator sequence is externally called exactly n_o times (once for every operator), calling itself $(n_o - 1) * (n_o - 2) * \cdots * (n_o - d) = (n_o - \frac{d+1}{2})^d$ times for each in the worst case. Each call, the cumulative effects/preconditions need to be compared to the effects/preconditions of the operator which is to be implemented. In the worst case, the operator and the operator sequence have effects/preconditions on every variable (so n_v in total), leading to a total complexity for the reduction of operators of $O(n_o * (n_o - \frac{d+1}{2})^d * n_v) = O(n_o^{d+1} * n_v)$.

5.3. Dominance Pruning

Unlike the other techniques, Dominance Pruning was implemented into the search component of Fast Downward because much of the required functionality regarding transition systems could be reused from the implementation of the Merge&Shrink heuristic. Nevertheless, it is implemented as a static preprocessing method. Therefore it effectively only removes operators, whose labels become dead, and outputs a problem description without them.
The implementation used in this work is based on the algorithm described by Kissmann and Torralba (2015). The label-dominance simulation is initialized with all pairs of states, except for goal states which can't be simulated by non-goal states. State pairs are then removed successively if they do not fulfill the conditions for simulation, an iterative process until the simulation does not change anymore. Once the simulation is computed, the subsumed transitions can be identified. As Kissmann and Torralba (2015) point out, it is in general not safe to prune subsumed transitions successively. When pruning a transition, the label-dominance simulation can change. To avoid having to recompute it, they do not prune transitions which could cause such change. For example, it is in general not safe to prune a transition of label A in one LTS, that is only subsumed by a transition of label B, and later remove label B from the planning task because it became dead in a different LTS.
Such caution is necessary when single transitions of a label are pruned. However, in our application of Dominance Pruning as a static preprocessing method, it is not beneficial to prune single transitions if the corresponding label does not become dead. Although we then wouldn't need to take those transitions into account in recomputations of the simulation, directly pruning them has the drawback that mutually subsuming transitions cannot be pruned at all, as mentioned above. To maximize the achievable pruning of operators, the implementation used does not prune single transitions but an entire label from the problem, if all of its transitions in one LTS are subsumed by transitions of other labels.

Theorem 3 shows that it is safe to prune several subsumed transitions successively, if we ensure that for each one at least one subsuming transition continues to exist. In practice, this is realized by storing a list of (subsumed, subsuming) transition pairs. When every transition of a label in one LTS is subsumed, the label is removed from the planning task together with all of its transitions. Since we don't allow transitions of the same label to subsume each other, all transitions of one label can be safely removed successively as they are subsumed by transitions which still exist. After the removal, the list of subsumed transitions is cleaned from those transitions which were subsumed by a just-removed transition. This ensures that we prune no further transition which is not subsumed anymore. As an example, if two transitions subsume each other, and the label of the first transition is removed because all its transitions are subsumed, the second transition is not subsumed anymore, and the second label stays - except if there

was a third transition subsuming both of these transitions, in which case it is possible to also remove the second label.
One might think that by not pruning transitions right away when they are subsumed, information about subsumed transitions might be lost. Theorem 4 shows, though, that subsumed transitions stay subsumed throughout the merging process, and therefore we can wait with the pruning until the entire label becomes dead without losing information.

In difference to Kissmann and Torralba (2015), who compute the coarsest simulation before the first merging step yet only prune transitions after it, this implementation does perform pruning on the atomic transition systems. After the pruning step, two LTS are merged together using the DFP merging strategy (Sievers et al., 2014), and labels are reduced if possible. Exact shrinking is not used, since it potentially reduces the number of operators pruned, and irrelevance pruning is not performed during the pruning, either.
The label-dominance simulation then needs to be updated. Kissmann and Torralba (2015) suggest to use an incremental computation: Instead of recomputing the coarsest simulation for all LTS after every merge step, only the simulation for the new LTS is computed. Only when all LTS are merged together, the coarsest simulation is computed. This reduces the work needed for pruning, but has the potential downside of missing subsuming transitions because the intermediate simulations are not the coarsest possible. Only after the final merge, the coarsest simulation is computed - but if the pruning method time-outs before that, the pruning achieved is smaller. In this implementation, both versions were used; the one always calculating the coarsest simulation, to determine how much more pruning is possible, and the one using the incremental computation, which would be more viable in practice.
The mapping of reduced labels to the corresponding operators is stored during the merging process. Since reduced labels are locally equivalent in a merged LTS, in case one label becomes dead in an LTS it can be translated back to the corresponding operators, which are then removed from the planning task.
One option that was added in this implementation is to ignore the label costs, meaning a transition can be subsumed by a transition of a label with higher costs. Clearly, optimality of DP is lost using this option, but the pruning power could increase.

Potentially, after removing an operator, a variable can become static and be removed in the preprocessor, and certain data structures such as the causal graph have to be recomputed. For that reason, the dominance pruning component writes an output file in the format of the translator output, containing the planning task without the pruned operators. This output file can then be read from the preprocessor which is thus run two times, once before dominance pruning and once before the search.

Complexity Analysis of the Implementation

The most complex component of dominance pruning is the coarsest dominance simulation computation. As described by Torralba and Hoffmann (2015), the simulation is initialised with every two states simulating each other, except for goal states which are not simulated by non-goal states. Let n be the number of labeled transition systems for which the simulation is to be computed, $t_{/LTS}$ the maximum number of transitions in one LTS, $s_{/LTS}$ the maximum number of states in an LTS, $t_{/s}$ the maximum number of transitions of a state, and $t_{/l}$ the maximum number of transitions of a label in an LTS. The initial state simulation relation then contains $O(n \times s^2_{/LTS})$ state pairs which need to be checked for simulation at least once. For each state pair, in the worst case every transition of one state needs to be compared to each transition of the other state, and for each of these transition pairs the label dominance of the corresponding labels needs to be calculated, resulting in a complexity of $O(t^2_{/s} \times dom)$ per state pair, with dom being the complexity of calculating the label dominance relation between two labels.
To calculate this relation, in every LTS minus one every transition of one label needs to be compared to every transition of the other label, yielding a complexity of $O(n \times t^2_{/l})$ with the state simulation lookup being of constant complexity.
The domination of one label over the other in one LTS however doesn't need to be recomputed for every state comparison, because it can only change when a state simulation pair from that LTS is removed. This means if we iterate over all state pairs in one LTS before the next one, we can store all previously computed label dominations for the other LTS and only have to recompute them for this LTS: so every label domination only needs to be computed once per iteration, yielding a total complexity per iteration of $O(n \times s^2_{/LTS} \times t^2_{/s} + n \times l^2 \times n \times t^2_{/l})$. Using $l \times t_{/l} = t_{/LTS}$, this complexity can be rephrased as $O(n \times (s^2_{/LTS} \times t^2_{/s} + n \times t^2_{/LTS}))$, which becomes $O(n \times t^2_{/LTS} \times (1 + n))$ when using the equality of $s_{/LTS} \times t_{/s} = t_{/LTS}$. Finally, we can replace $n \times t_{/LTS}$ by the total number of transitions summed over all LTS, which leads to the final complexity of $O(t^2_{total})$ per iteration. Although theoretically, $n \times s^2_{/LTS}$ iterations are possible, in practice after few iterations a static point is reached in which nothing changes anymore. Since with very big problems the simulation computation will probably not be finished within a reasonable time and memory limit, a complexity threshold was introduced for the implementation which always calculates the coarsest simulation, to abort the computation if the effort would be too great. This complexity number is calculated as $n \times (s^2_{/LTS} \times t^2_{/s} + n \times t^2_{/LTS})$.

In the case of the incremental computation, only the simulation for the merged LTS needs to be computed, as well as the label dominance relation. The complexity of computing the simulation quickly increases during the merging process, though, as the number of states and transitions rises.

6. Experiments

In this chapter, the implemented techniques are evaluated in a row of experiments. Section 6.1 describes how the experiments were performed, and which values were measured. In section 6.2 we evaluate different conditions and parameters and choose those which seem best-suited for our purposes. Those parameters are then used in the successive experiments in section 6.3, where we investigate synergy effects and verify how well the theoretical results hold, and in section 6.4, where we compare the overall performance of different combinations with each other. Finally, we measure the performance of each pruning technique on a set of unsolvable instances in section 6.5.

As in Chapter 4, we abbreviate the subsequent application of two pruning techniques as `A+B`, where `A` and `B` are either `SA` (Safe Abstraction), `ROR` (Redundant Operator Reduction), or `DP` (Dominance Pruning).

6.1. Configuration

All experiments were performed on computers with Intel Xeon E5-2660 CPUs running at 2.2 GHz and with a time bound of 30 minutes (if not specified otherwise) and a memory bound of 3 GB.

The experiments using Dominance Pruning alone were run on the optimal International Planning Competition (IPC) tasks up to IPC2011 (46 domains with a total of 1456 tasks), using $A^\star$ search (Hart and Raphael, 1968) together with the LM-Cut heuristic (Helmert and Domshlak, 2009). Other experiments were run on satisficing and optimal IPC tasks up to IPC2011 (92 domains with a total of 1886 tasks) using greedy search together with the FF heuristic (Hoffmann and Nebel, 2001).

The time limit for the pruning techniques used is noted in seconds in the plots in brackets. If no time limit is stated, then none was used. If several techniques are combined, then each received the time limit noted in brackets separately. When the time limit is reached, the pruning method is aborted, and the pruning information gathered up to that point is used.

Measures

In plots where the domain information is relevant, instances are displayed with a symbol indicating the domain. Appendix A lists all domains used, together with the symbol representing that domain in the plots.
The values displayed in the plots are:

- **Reduced Operators**: The number of redundant operators which could be removed by ROR itself.
- **Pruned Operators**: The number of operators which could be removed by DP itself because of subsumed transitions.
- **Search Operators/Variables**: The number of operators/variables remaining after all preprocessing steps, indicating the size of the problem that needs to be searched.
- **Planning Time**: The CPU time in seconds needed to solve a problem/prove it is unsolvable, including the preprocessing (Fast Downward preprocessor and pruning methods), the actual search, and, if necessary, the plan refining process of Safe Abstraction. Instances which exceeded the overall time limit are located in the plots at the value of the time limit (around 1800 seconds, above 10^3). Instances which exceeded the memory bound during the search are located at the edges of the plot (at 10^4).
- **Expanded States**: The number of states expanded in the search before a solution was found. For experiments only using Dominance Pruning, the value of expansions before the last layer is used to avoid tie-breaking noise. Instances which could be solved without any search have zero expanded states. Instances with time- and mem-outs are both located at the edges of the plot.
- **Plan Costs**: The sum of all operator costs in the final plan (after potential refining). Problems without a plan (unsolvable, Time- or Mem-out) are located at the edges of the plot.

One further important value to measure the benefit of a pruning technique is the *coverage*, meaning the number of problems which could be solved in the given time bound.

6.2. Pruning Parameters

The experiments in this section evaluate the three pruning techniques separately. First, different parameter and conditions are compared with each other. The configuration obtaining the best result is then compared to the performance achieved without pruning, using the same heuristic function and search algorithm on the same domains. We call this second run the *baseline* for the configuration. For Dominance Pruning, the baseline uses A^* search with the LM-Cut heuristic, for the other two pruning techniques and any combination of two techniques it uses greedy search with the FF heuristic.

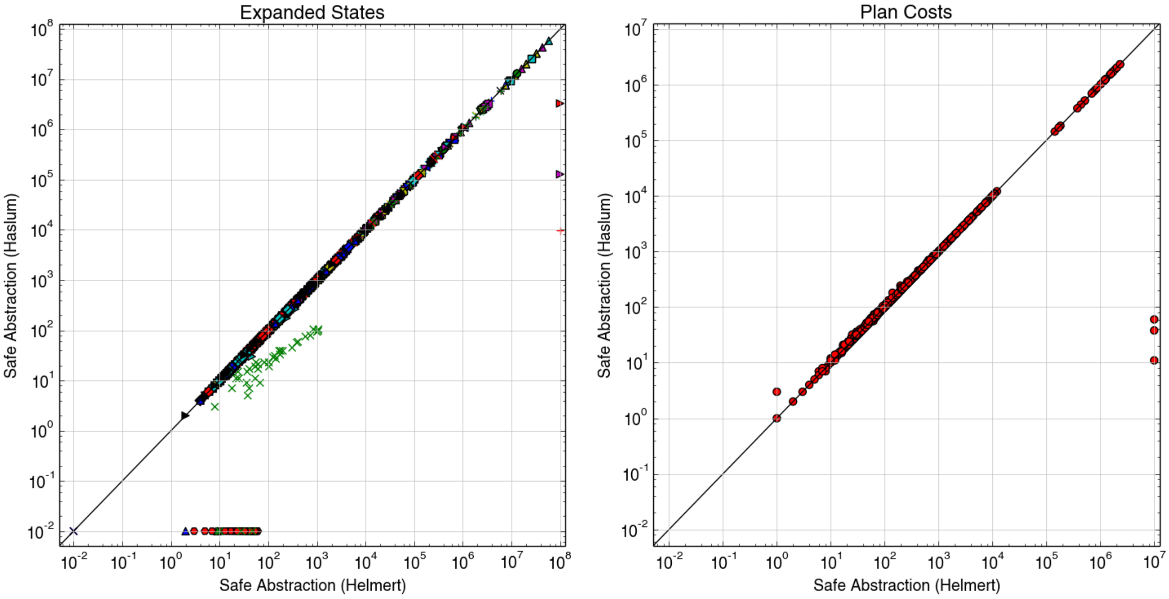
6.2.1. Safe Abstraction

As there exist two different conditions for Safe Abstraction, it is of interest to determine in which domains and by how much their abstraction power differs. Table 6.1 shows a comparison of Helmert's and Haslum's conditions in coverage, abstraction achieved, and plan overhead.

	Coverage	Solved by SA	Abstractable	Safe Variables	Plan overhead
Baseline	1413				
Helmert	1489	4.23%	35.19%	22.08%	10.98%
Haslum	1491	16.63%	58.08%	49.69%	20.33%

Table 6.1.: Comparison between Helmert's and Haslum's conditions. An instance is called *solved by SA* if no goal conditions remain after the abstraction, so no search is needed. It is called *abstractable*, if at least one variable in it is safe. The last two values measure for abstractable instances which percentage of the variables is safe with either condition and by how much the plan costs increase.

The coverage increases by using Safe Abstraction as a preprocessing method with either condition, the difference in coverage between them is marginal. Using the one by Haslum (2007) allows to abstract variables in many more instances, and also more variables in average: In more than half of the instances, roughly half of the variables can be abstracted. The plans become more expensive, but only by about 20% on average when an instance can be abstracted. Figure 6.1 visualizes the comparison. Since some instances are solved without search with Haslum's condition, but not with Helmert's, they do not require any states to be expanded and are located at the bottom of the plot (a). A further considerable improvement is achieved in the `Rovers` domain. As plot (b) shows, using Haslum's condition causes a relatively small plan cost increase (about 10%).



(a) Bottom edge: `Miconic`, `Movie`
Green crosses: `Rovers`

(b)

Figure 6.1.: Comparison of both Safe Abstraction conditions.

The two conditions also differ in how often cascading abstraction is possible. The comparison is shown in Table 6.2. Zero cascading abstractions mean that variables can be abstracted in a planning task, but abstracting them causes no further variables to become safe. While with Helmert's condition only one cascading abstraction is possible, up to 3 cascading abstractions are possible with Haslum's condition, mostly in `Rovers` and `Miconic` instances.

Cascading abstractions	No safe variables	0	1	2	3
Helmert	64.81%	30.96%	4.23%		
Haslum	41.92%	38.16%	6.77%	10.46%	2.68%

Table 6.2.: Comparison of cascading power between Helmert's and Haslum's condition. The numbers denote the percentage of total instances and add up row-wise to 100%.

Table 6.3 shows a domain-wise comparison between the two conditions. While in some domains the two conditions produce very similar or equal abstraction results, the condition by Haslum is considerably stronger in several domains. `Movie` and `Miconic` can be completely solved by Safe Abstraction, which is not possible with the condition by Helmert.

Domain	Helmert-Cond.	Haslum-Cond.	Haslum (2007)
Gripper	1 – 15%	1 – 15%	**Solved**
Logistics	**Solved**	**Solved**	**Solved**
Movie	None	**Solved**	**Solved**
Grid	None	None	**50%**
Blocksworld	None	None	0 – 20%
Elevator	**25 – 40%**	**25 – 40%**	**Solved**
Depots	1 – 15%	1 - 15%	1 – 10%
DriverLog	None	0 - 30%	0 – 25%
Rovers	5 – 10%	**75 – 95%**	**60 – 90%**
Satellite	1 – 5%	**50 – 80%**	**Solved**
Airport	None	**45 – 55%**	**40 – 60%**
Pipesworld-tankage	None	0 – 40%	None
PSR-small	None	**0 – 60%**	**0 – 50%**
Barman	None	0 – 15%	
Floortile	5 – 15%	5 – 15%	
Miconic	**1 – 35%**	**Solved**	
Parcprinter	None	**5 - 70%**	
Pathways	None	**20 – 35%**	
Storage	None	**5 – 55%**	
Tpp	1 – 20%	1 – 20%	
Transport	**10 – 35%**	**10 – 35%**	
Trucks	None	**85 – 97%**	
Visitall	None	**0 – 55%**, one solved	
Woodworking	None	0 – 15%	
Zenotravel	None	**0 – 50%**	

Table 6.3.: Safe Abstraction performance per domain. The values state how high the minimum and maximum percentage of variables abstracted per instance in the domain are. Domains marked with None do not contain safe variables, Solved instances were solved by Safe Abstraction.
The following domains contained no safe variables under either condition: Mystery, Mprime, FreeCell, Pipesworld-notankage, Openstacks, Parking, Pegsol, Scanalyzer, Sokoban, Tidybot. The first four of them were also used by Haslum (2007) with the same results.

The table also contains the experimental results presented by Haslum (2007), although they are not directly comparable, as they were produced in combination with other techniques such as (partly manual) reformulation of the problem encoding and composition of operators. Apparently this considerably increases the abstraction power in domains such as Grid, Gripper, Elevators and Satellite. It is not clear why the performances reported by Haslum (2007) are worse in Depots, DriverLog, Rovers and Pipesworld-tankage.

Since the abstraction power of Haslum's condition is considerably stronger, it is used in the successive experiments. In the following experiment, shown in Figure 6.2, it is compared to the baseline. Recall that the baseline is defined to be an equivalent run with the same search algorithm and heuristic, which is greedy search with the FF heuristic in this case. The attributes depicted are explained in detail in Section 6.1.

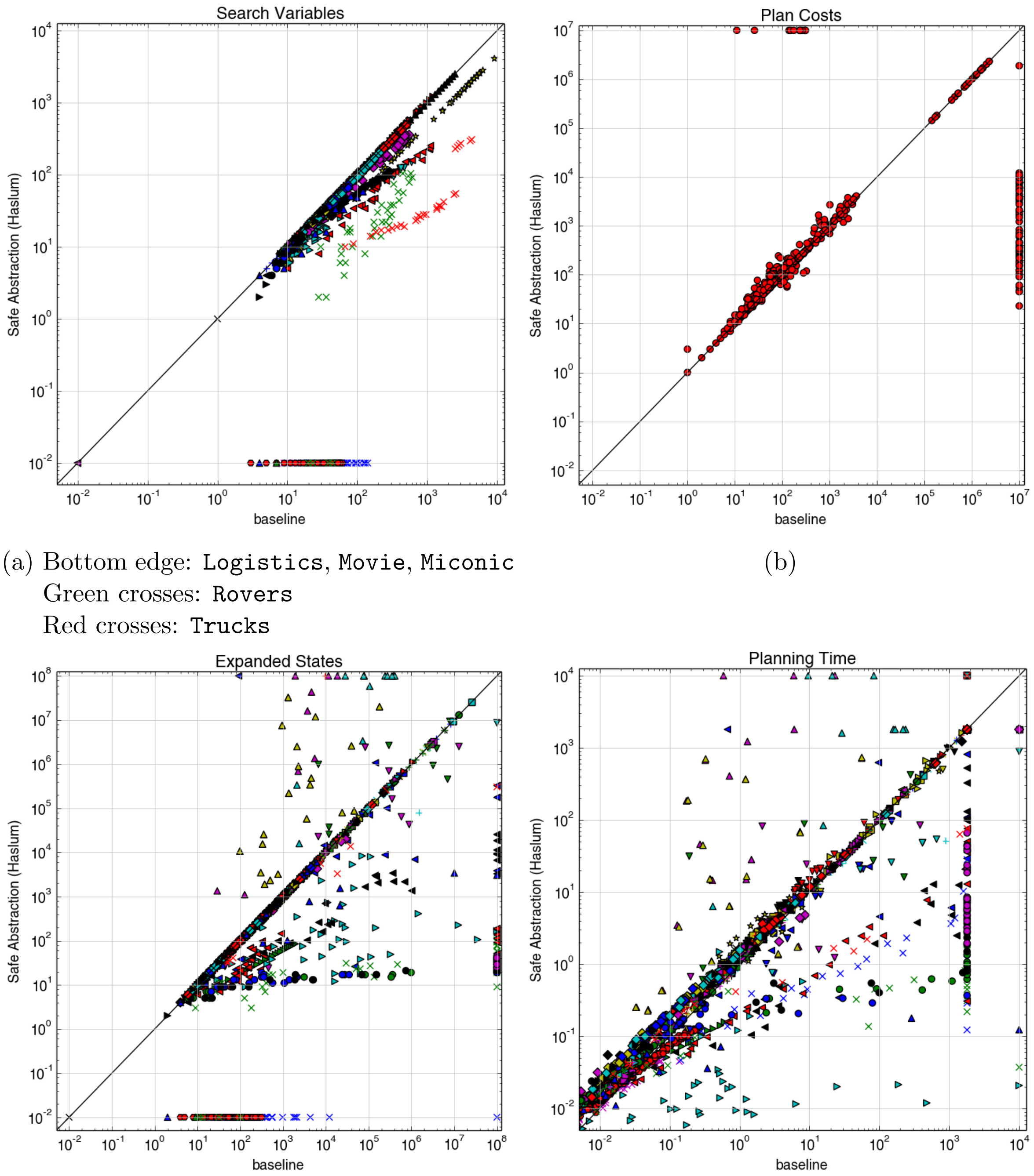
Subfigure (a) plots an overview over the domains in which at least one safe variable exists, corresponding to the numbers noted before in Table 6.3. The instances located at the bottom were solved without search. In general, the plan costs (b) are not drastically higher than in the baseline. Some noise is introduced by using suboptimal search, for which reason some plans become shorter. In (c), the clear benefit of Safe Abstraction is shown for the states which need to be expanded to find a solution. Many instances which could previously not be solved become solvable. Again, in some instances, which are solved without search, no states need to be expanded. Instances above the diagonal are caused by technical reasons (e.g. variable order variations). A slight time overhead (d) is introduced in small instances, as Safe Abstraction's conditions are checked, but with bigger instances a clear time benefit can be observed. As mentioned previously, the instances located at the edge of the plot ran out of memory.

6.2.2. Redundant Operator Reduction

One important parameter in Redundant Operator Reduction is how long the operator sequences that can implement an operator are allowed to be. Haslum and Jonsson (2000) have shown on a very small test suite that the number of redundant operators detected increases very little with a bound higher than 2. Table 6.4 shows the comparison of limit 2, 3 and 4 in our experiments. Note that a sequence limit of 3 does not imply the sequences are exactly of length 3, it also detects redundant operators which are implemented by a sequence of two or one operators.

In most domains, a length greater than 2 does not improve the number of operators that can be removed, though there is a significant increase in the Rovers domain where only sequences of 3 operators seem to implement another operator. In Miconic instances, a greater length actually reduces the number of operators that can be removed slightly. This is caused by the greediness of the algorithm, as sometimes an operator which is implemented by a sequence of three operators could be used in a sequence to implement two different operators. Allowing a sequence length of 3, that operator is removed and thus the two previously redundant operators are not implemented anymore.

In Figure 6.3 we show a comparison between limit 2 and 3. Only a small benefit for



(a) Bottom edge: `Logistics`, `Movie`, `Miconic`
Green crosses: `Rovers`
Red crosses: `Trucks`

(b)

(c) Bottom edge: `Logistics`, `Movie`, `Miconic`

(d)

Figure 6.2.: Performance of Haslum's condition.

redundancy reduction (a) is observed in domains besides `Rovers` when using the higher sequence limit. In some instances (located at the bottom edge) the time limit was

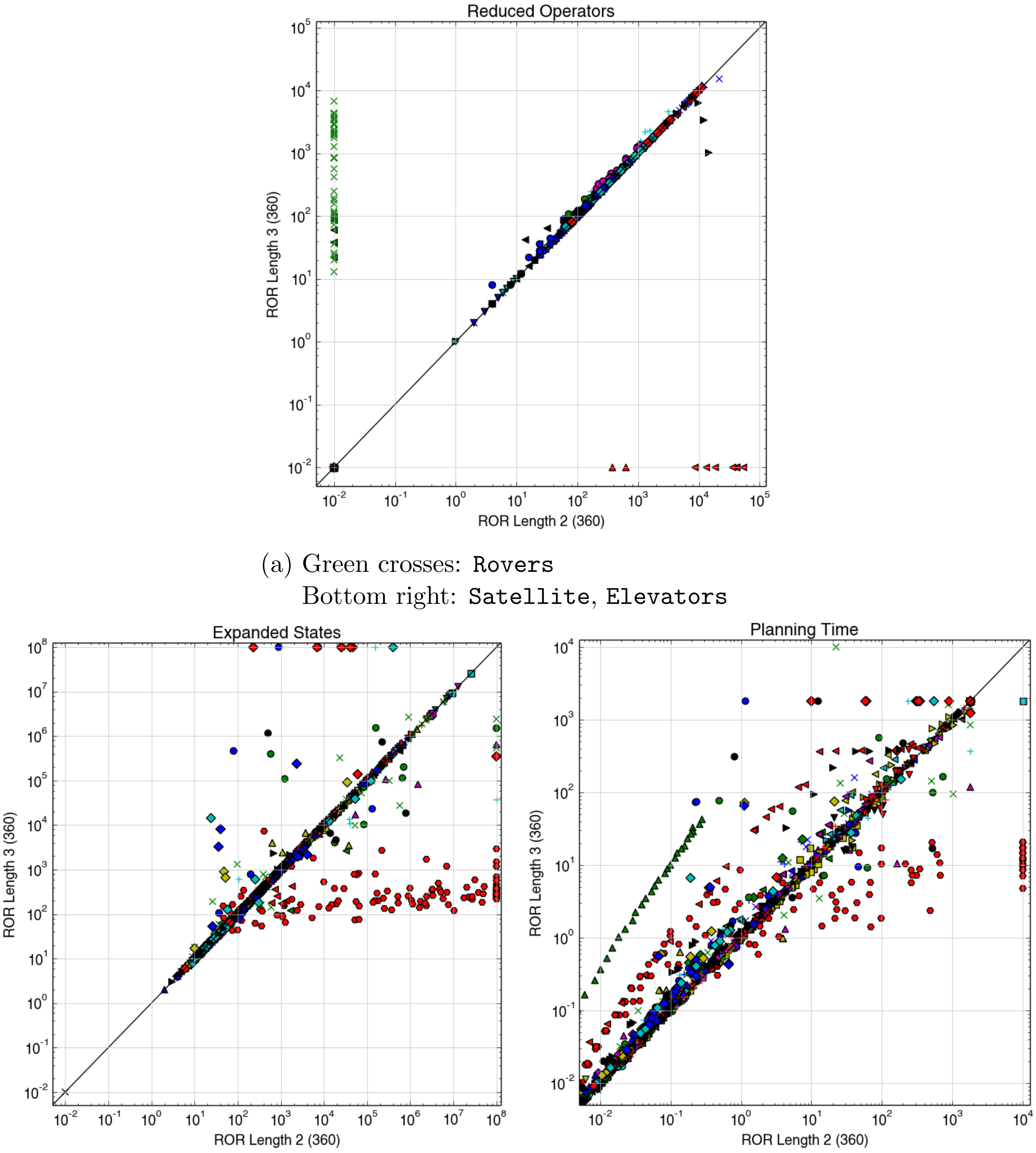
Sequence Limit	<= 2	<= 3	<= 4
Barman			+ 3%
Depot	0-8%		
Driverlog	7-28%	+ 1-9%	+ 1-11%
Elevators	5-20%		
Grid	76-81%	+ 1%	+ 1%
Logistics00	0-4%		
Logistics98	1-53%		
Miconic	0-96%	- 1-0%	- 1-0%
Movie	74-96%		
NoMystery	0-5%	+ 0-7%	+4-31%
Parcprinter	0-3%		
Pegsol	39-41%		
Psr-small	0-98%		
Rovers		+18-51%	+18-51%
Satellite	59-96%		
Tpp	0-1%	+ 0-1%	+ 0-1%
Transport	0-5%	+ 0-1%	+ 0-5%
Trucks	1-5%		
Woodworking	26-53%	+ 0-20%	+ 0-20%
Zenotravel	56-78%	+ 3%	+ 4-9%

Table 6.4.: Range of operators removed per domain (in percentage). Column 3 and 4 are relative to column 2, empty entries indicate that nothing changed in that domain.
In the following domains, no redundant operators could be removed: Airport, Blocks, Floortile, Freecell, Gripper, Mprime, Mystery, Openstacks, Parking, Pathways-noneg, Pipesworld-notankage, Pipesworld-tankage, Scanalyzer, Sokoban, Storage, Tidybot, Visitall

reached before redundant operators were found, since the number of operator sequences increases exponentially with the limit. In terms of expanded states (b), there is no clear benefit of using a longer sequence limit, except for the Miconic domain. Accordingly, the planning time increases in all other domains (c).

The increase in complexity with a higher limit does not pay off, as in most instances the number of operators does not change drastically. The Miconic domain is an exception, as using a sequence limit of 3 has the effect that fewer operators are reduced and fewer search runs are aborted because of exceeded memory. It appears that the effects of using this technique can go either way, depending on the instance structure. The effect in Miconic is big enough for the limit of 3 to achieve a higher coverage.
As apart from Miconic the limit of 3 does not pay off, we will use ROR with limit 2 in

the successive experiments.



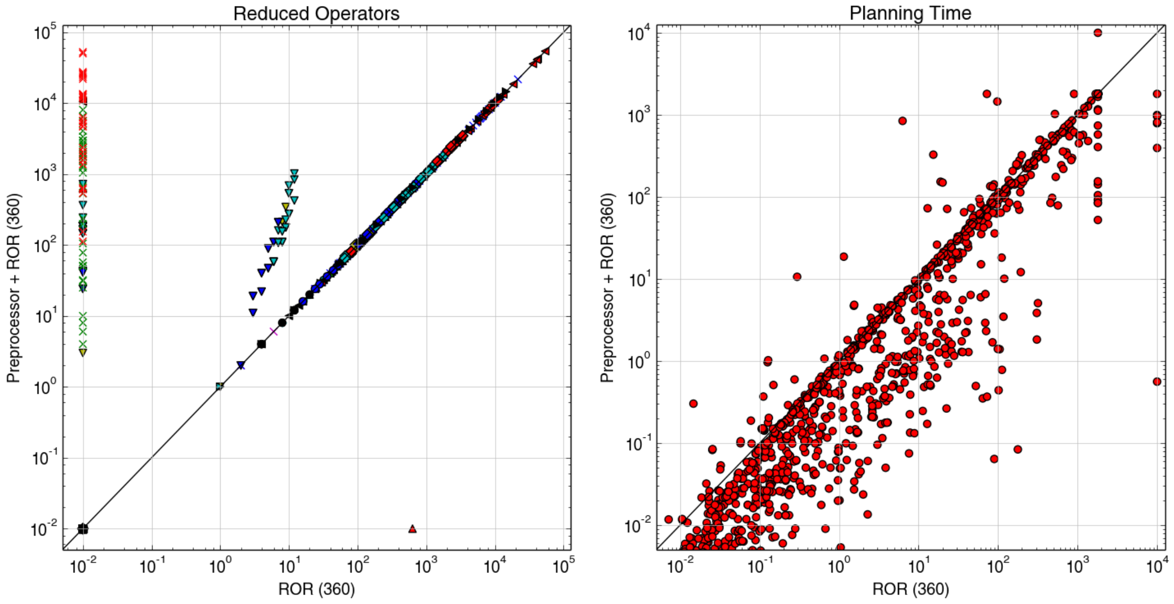
(a) Green crosses: `Rovers`
Bottom right: `Satellite`, `Elevators`

(b) Red hexagons: `Miconic`

(c)

Figure 6.3.: Comparison of sequence lengths for ROR.

Since ROR was originally implemented to be applied to a problem *before* irrelevant and static variables are removed, one further experiment was conducted to see how beneficial it is to remove these variables before applying ROR. This was not considered for Safe Abstraction, as it is capable of abstracting irrelevant and static variables itself. In the case of Dominance Pruning, static variables are always removed before the pruning since DP is implemented in the search component of Fast Downward.



(a) Red crosses: `Rovers`
Green triangles: `Satellite`
Golden triangles: `Trucks`

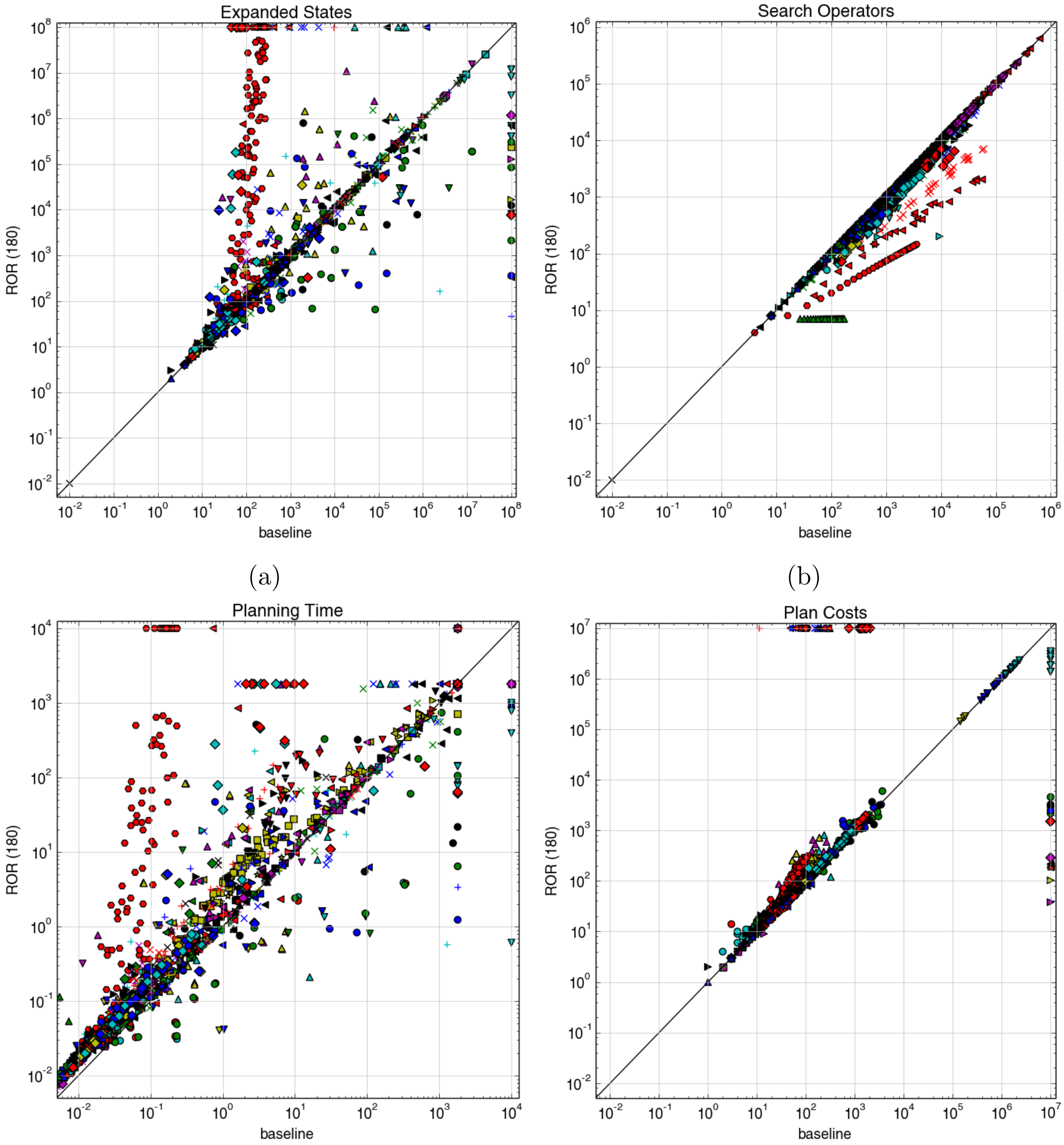
(b)

Figure 6.4.: Comparison of ROR with and without previous static variable removal.

Instances	solved	aborted (time)	aborted (memory)	unsolvable
Baseline	1406	401	71	8
ROR 2	1368	411	99	8
ROR 3	1382	418	78	8
P+ROR 3	1397	410	71	8

Table 6.5.: Comparison of solved and aborted instances in different ROR configurations out of 1886 total instances. `ROR 2` uses an operator sequence limit of 2, `P+ROR 3` a limit of 3 and the removal of static variables.

Figure 6.4 shows the positive effect on reduced operators (a), together with a significant benefit in runtime (b). The increased coverage is shown in Table 6.5, although higher than in the previous experiments, it is still lower than the baseline coverage. An



(a) (b)

(c) (d)

Figure 6.5.: Redundant Operator Reduction (sequence limit 2, with static variable removal)

increased number of timeouts during the search is the cause of this, sometimes having fewer redundant operators available actually slows down the search. Considering the obvious advantages of using static variable removal before ROR, it is performed in the successive experiments with ROR.

We compare the performance using ROR with an operator sequence limit of 2 against the baseline in Figure 6.5. In a broad range of domains, redundant operators can be removed (b), but having fewer redundant operators available can have both positive and negative effects on the number of expanded states (a) and the planning time needed (c). Many instances can be solved only without redundant operators, others can only be solved with them. Except for `Miconic`, a clear positive-negative distinction between domains is not possible. A considerable plan cost overhead (d) results from using ROR in many cases.

6.2.3. Dominance Pruning

As Dominance Pruning is an optimality-preserving method, experiments in this subsection were run on a smaller test suite using optimal search, as noted in Section 6.1.

The first experiments with Dominance Pruning are using an implementation that calculates the coarsest simulation after every merge step, as described in Section 5.3. In practice, this full calculation is less viable because of the increased time complexity, but potentially more operators can be pruned that way. Table 6.6 shows the configurations used in these experiments, with varying complexity and time limits, one experiment where merging was disabled, and one in which optimality was disabled, as described in Section 5.3. The results of these experiments are summarized in Table 6.7.

	Complexity Limit	Time limit	Merging enabled	Optimality on
Low C-Limit	1.000.000.000	30 min	Yes	Yes
High C-Limit	10.000.000.000	30 min	Yes	Yes
No C-Limit	None	30 min	Yes	Yes
High T-Limit	None	4 h	Yes	Yes
No Merging	None	30 min	No	Yes
Suboptimal	1.000.000.000	30 min	Yes	No

Table 6.6.: Configurations used for DP with the coarsest simulations. The first column contains the label we will use to refer to that configuration. Complexity limit prevents DP from calculating too big simulations, the Time Limit is for the entire search, including DP.

The coverage becomes worse with a higher complexity limit, the effort needed to prune operators this way is too high compared to the gain, but the amount of pruning which is possible when investing enough time is considerable (compare runs *Baseline*, *Low*

	Solved	Prunable	H_1	H_2	Operators	Price
Baseline	820					
Low C-Limit	818	18.22%	85.94%	99.51%	40.18%	0%
High C-Limit	624	59.62%	66.19%	96.79%	41.00%	0%
No C-Limit	182	90.11%	69.78%	86.26%	52.45%	0%
High T-Limit	204	91.18%	70.10%	87.25%	51.90%	0%
No Merging	650	52.92%			34.29%	0%
Suboptimal	816	19.00%	85.42%	99.51%	40.52%	1.06%

Table 6.7.: Coverage with different DP configurations, percentage of instances which contained at least one prunable operator, in which H_1 and H_2 hold. The last two columns inform about how high the percentage of pruned operators and the increase in plan costs is in average when the problem contains at least one prunable operator.

C-Limit, *High C-Limit*, *No C-Limit*). The complexity of constantly recalculating the simulation is so high that even a time limit of 4 hours (see run *High T-Limit*) hardly increased the coverage. Using the suboptimal approach did not result in a big change (compare runs *Suboptimal* with *Low C-Limit*), only in two domains, `Scanalyzer` and `Woodworking`, it made a small difference so that more operators could be pruned - at a small average plan cost increase of 1%, if pruning is possible. Since the increase in pruning using this modification is very limited, it is not used in the later experiments.
When merging is disabled (see run *No Merging*), most of the instances in which pruning is possible are also pruned, though not as many operators can be removed as with enabled merging. Two hypotheses were evaluated in these runs: The first one, H_1, states that if operators can be pruned before merging, operators can also be pruned after merging. The second one, H_2, states that if no operators can be pruned before merging, no operators can be pruned after merging. As the H_1 results show, in general it does pay off to merge. The results of H_2 show that the performance of DP before merging seems to be a good indicator on whether it will be possible to prune after merging.

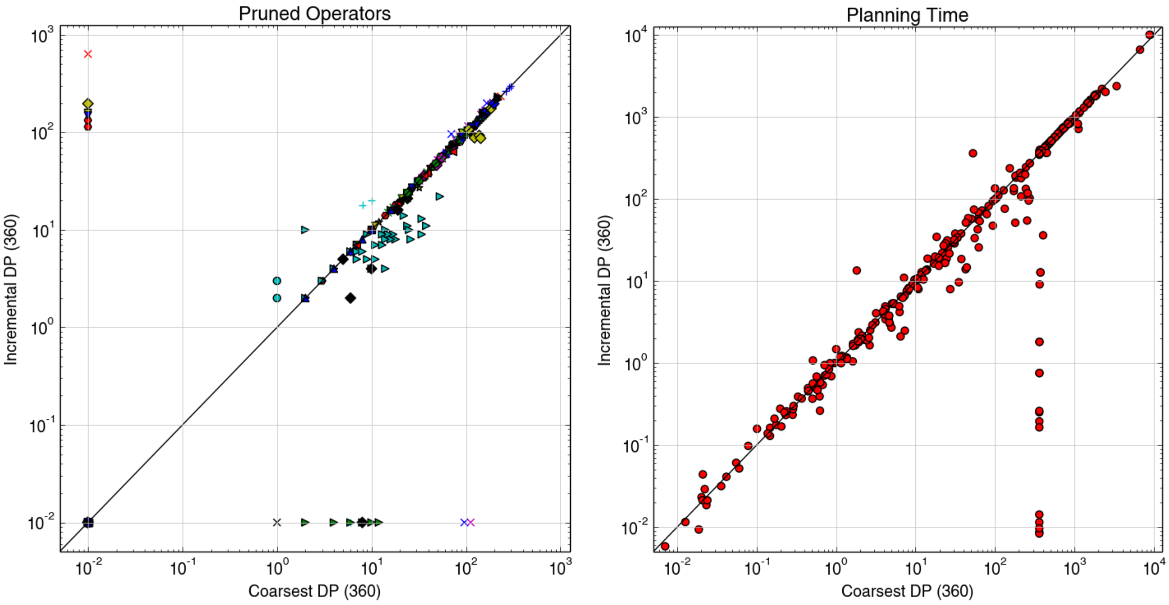
Table 6.8 shows the pruning that could be achieved in these runs in each domain. In some domains (`Blocks`, `Gripper`, `Scanalyzer`, `Storage`, `Transport`) no operators could be pruned without merging, while pruning was possible after merging, so for these domains merging seems to be vital to achieve any pruning. In reverse, merging did not help to find any more operators in the `Movie` domain.
In a row of domains, pruning was not possible in any of the configurations used (`Barman`, `Elevators`, `Freecell`, `Grid`, `Mprime`, `Openstacks`, `Parking`, `Pegsol`, `Pipesworld-tankage`, `Sokoban`, `Tidybot`), though Kissmann and Torralba (2015) report possible pruning in `Elevators`, `Freecell` and `Sokoban`. In general, there is no guarantee that with a longer run time no further operators could be pruned, though if an instance was

solved in a run without complexity limit, it has been completely merged, and pruning is definitely not possible there (such instances exist in Miconic, Openstacks, Tpp and Visitall). Compared to the results of Kissmann and Torralba (2015), by calculating the coarsest simulation often a much higher percentage of operators can be pruned, but the time invested for that pruning is so high that only very few problems can be solved before reaching a time limit.

Domain	#Instances	Low C-Limit	High C-Limit	No Merging	No C-Limit	High T-Limit	P_i
Airport	50	0-78%, 2/25	3-85%, 11/11	3-13%, 13/13	32-85%, 6/6	32-85%, 6/6	13.00%
Barman	20	0/4	0/4	0/4	0/0	0/0	
Blocks	35	0/28	0/28	0/28	31%, 3/3	31%, 3/3	
Depot	22	0/7	0/6	0/7	0/0	44%, 1/1	
Driverlog	20	0-5%, 4/14	1-41%, 13/13	1-8%, 13/13	41%, 1/1	41%, 1/1	1.00%
Elevators	50	0/40	0/40	0/40	0/0	0/0	1 - 10%
Floortile	20	0/7	28-29%, 9/9	28-29%, 9/9	0/0	29%, 2/2	2 - 28%
Freecell	80	0/15	0/2	0/2	0/0	0/0	28.00%
Grid	5	0/2	0/0	0/0	0/0	0/0	
Gripper	20	0/7	0-47%, 4/7	0/7	47-48%, 3/3	47-48%, 3/3	11.00%
Logistics	63	0-85%, 10/26	26-85%, 27/27	20-47%, 27/27	59-85%, 7/7	58-85%, 10/10	42, 67%
Miconic	150	0-45%, 30/141	0-45%, 88/93	11-45%, 90/95	0-45%, 20/25	0-45%, 24/29	58.00%
Movie	30	74-96%, 30/30	74-96%, 30/30	74-96%, 30/30	74-96%, 30/30	74-96%, 30/30	
Mprime	35	0/22	0-1%, 1/6	0-1%, 1/6	0/0	0/0	0.00%
Mystery	30	0-1%, 1/17	0-2%, 3/5	0-1%, 1/7	41%, 1/1	41%, 1/1	0.00%
NoMystery	20	0/15	2-53%, 11/11	2-5%, 9/9	3-53%, 10/10	3-53%, 12/12	49.00%
Openstacks	80	0/44	0/36	0/36	0/7	0/7	0.00%
Parcprinter	50	0-59%, 6/34	28-67%, 39/39	11-63%, 11/11	28-57%, 12/12	28-61%, 16/16	38 - 77%
Parking	20	0/3	0/0	0/0	0/0	0/0	
Pathways	30	0-60%, 4/5	47-73%, 5/5	47-60%, 5/5	59-73%, 2/2	59-73%, 2/2	
Pegsol	50	0/46	0/46	0/46	0/0	0/0	0.00%
Pipesworld-not.	50	0/17	0-3%, 1/6	0/9	0/0	0/0	0.00%
Pipesworld-t.	50	0/12	0/6	0/6	0/0	0/0	0.00%
Psr-small	50	0-48%, 37/49	0-55%, 44/45	0-40%, 37/45	16-66%, 38/38	16-66%, 40/40	85.00%
Rovers	40	0-48%, 7/9	14-53%, 9/9	14-50%, 9/9	35-53%, 4/4	35-53%, 4/4	71.00%
Satellite	36	0-40%, 3/7	10-50%, 7/7	10-50%, 7/7	10-40%, 3/3	10-40%, 4/4	50.00%
Scanalyzer	50	0/28	9-99%, 14/14	0/13	99%, 4/4	99%, 4/4	1, 0%
Sokoban	50	0/50	0/27	0/42	0/0	0/0	8 - 9%
Storage	30	0-63%, 2/15	0-68%, 6/15	0/15	1-72%, 4/4	41-72%, 4/4	
Tidybot	20	0/14	0/0	0/0	0/0	0/0	
Tpp	30	0-13%, 2/7	0-13%, 2/7	9-13%, 2/7	0/4	0/4	25.00%
Transport	50	0/17	0/17	0/17	23-44%, 3/3	23-44%, 3/3	0.00%
Trucks	30	0/10	67-87%, 7/7	67-87%, 7/7	73-79%, 2/2	73-79%, 2/2	90.00%
Visitall	20	0-75%, 4/11	0-75%, 6/11	0-50%, 6/11	0-75%, 3/5	0-75%, 3/5	6.00%
Woodworking	50	0-87%, 3/29	48-89%, 22/22	26-47%, 32/32	61-87%, 4/4	61-87%, 5/5	85 - 89%
Zenotravel	20	0-58%, 4/13	42-58%, 13/13	42-53%, 13/13	55-58%, 4/4	55-58%, 4/4	43.00%

Table 6.8.: Domain-wise performance comparison of different DP configurations calculating the coarsest simulations. The first and second value state the range between minimum and maximum percentage of operators pruned per instance in that domain. The third value (behind the comma) states how many instances contained at least one prunable operator, out of the instances which were solved (4th value).
The last column shows the average percentage of operators pruned in the P_i configuration in Kissmann and Torralba (2015), which used incremental computation and a time limit of 30 minutes.

Further experiments were conducted using the incremental calculation described by Kissmann and Torralba (2015). Additionally, the previously mentioned H_2 criterion was evaluated, where, if no operators could be pruned before merging LTS together, Dominance Pruning is aborted. The results of the incremental computation are visualized in Figure 6.6. Almost as many operators are often pruned (a), sometimes it is even possible to prune more operators, when with calculating the coarsest simulation the time limit is reached before merging far enough. The simulation computation is sped up considerably in some instances (b).



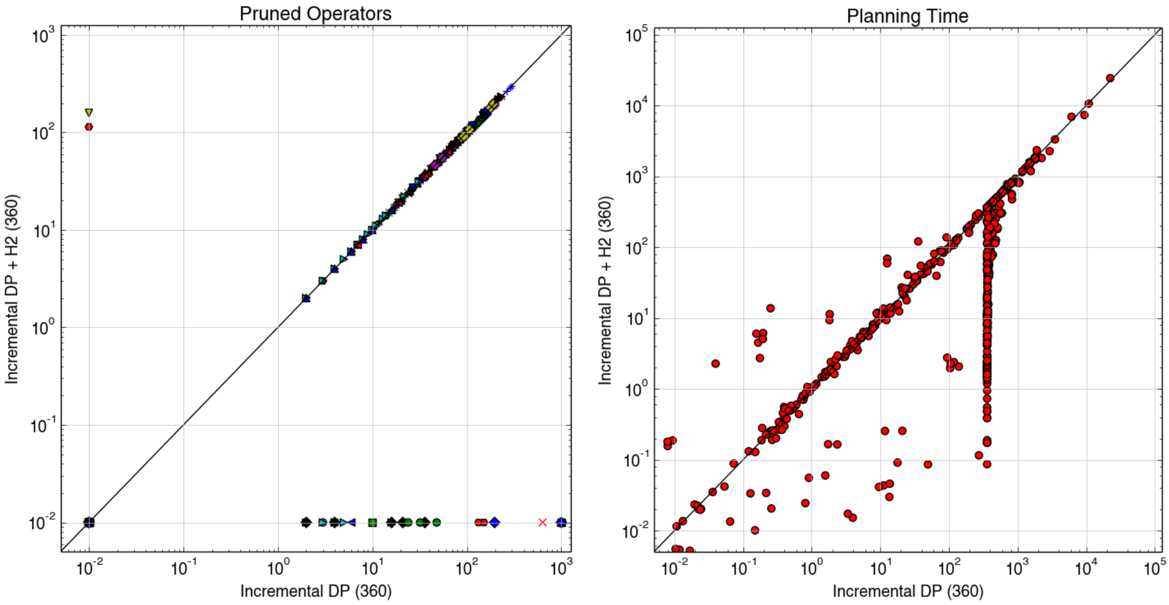
(a)

(b)

Figure 6.6.: Incremental against full (coarsest) computation

The effect of using the H_2 criterion is visualized in Figure 6.7. Only in a few instances, less operators are pruned (a). These instances where merging is required to prune any operators are located at the bottom edge. The instances located on the top left are caused by time fluctuation, as sometimes DP finishes the simulation calculation just before the time limit is reached, in other times it doesn't. Using the abort criterion, the time overhead (b) can be reduced in many instances in which previously the time limit for DP was reached without gain for the search.

Both methods sacrifice some pruning power but potentially speed up the pruning process. As Table 6.9 shows, neither the incremental calculation nor the usage of H_2 helped to improve the coverage compared to the baseline. Since they both decrease the planning time, though, Dominance Pruning is used in combination with the two in the successive experiments.



(a)

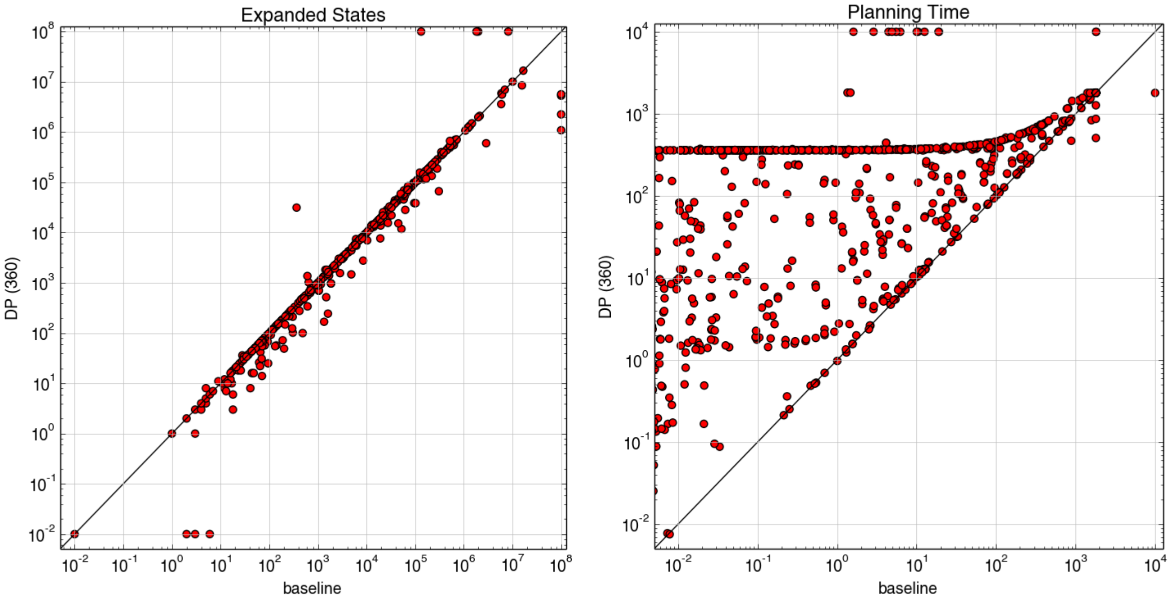
(b)

Figure 6.7.: Incremental Dominance Pruning using the H_2 abort criterion.

Instances	solved	aborted (time)	aborted (memory)	unsolvable
Baseline	820	627	2	7
Full DP	818	628	3	7
Incremental DP	818	630	1	7
Incremental DP + H_2	819	630	0	7

Table 6.9.: Coverage of different DP configurations compared to the baseline.

Figure 6.8 shows the results when comparing Dominance Pruning against the baseline. The number of expanded states (a) is considerably smaller, a few exceptions (above the diagonal) result from using the inconsistent LM-Cut heuristic. An increase in coverage is not achieved with DP, though, as even with the usage of the two previously described modifications, the time overhead (b) is immense - very often the time limit of 360 seconds is reached. This conclusion matches the results described by Kissmann and Torralba (2015).



(a) (b)

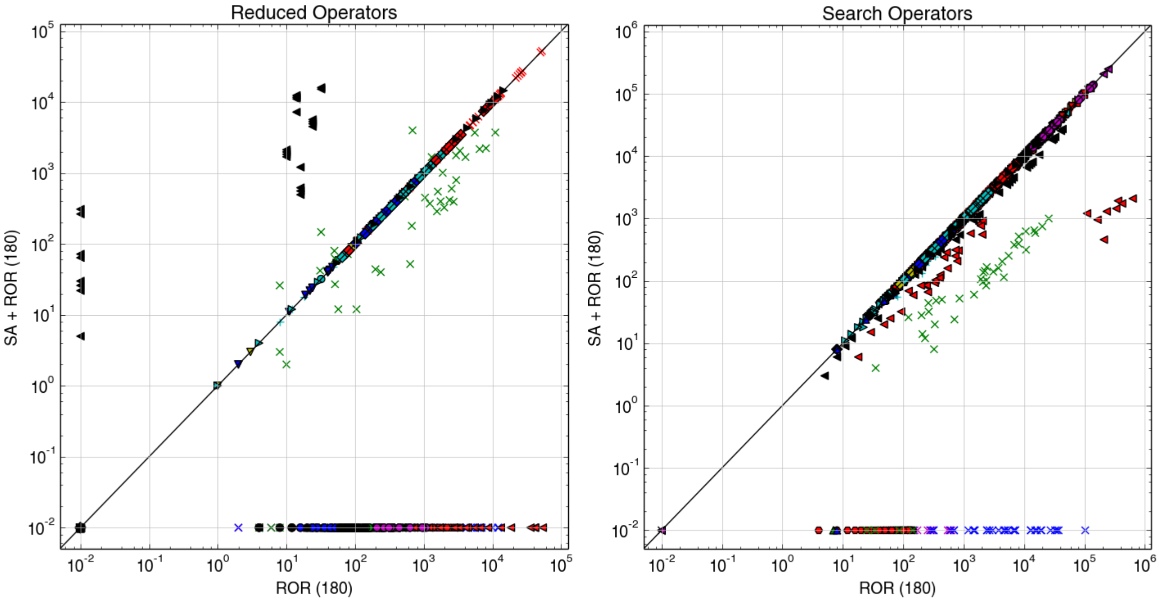
Figure 6.8.: Performance of DP with H_2 and incremental computation.

6.3. Pruning Power Synergies

In this section, we examine synergy effects by comparing the pruning power of each technique on its own with the performance in combination with a different pruning method. In order to isolate the influence of synergies, each of the techniques in this section receives a time limit of 180 seconds, meaning a combination of two techniques has a time limit of 360 seconds in total.

6.3.1. Safe Abstraction and Redundant Operator Reduction

As Figure 6.9 shows, positive synergies of `SA+ROR`, meaning that ROR can reduce more operators, occur only in `Tpp` and some `Rovers` instances. As Safe Abstraction never causes ROR to remove less operators, except for those which are already removed by SA (proven in Theorem 5), the number of search operators remaining in the problem is never greater when using the combination of both - though often less, when Safe Abstraction was active enough (`Satellite`, `Rovers`, `Logistics`,..).

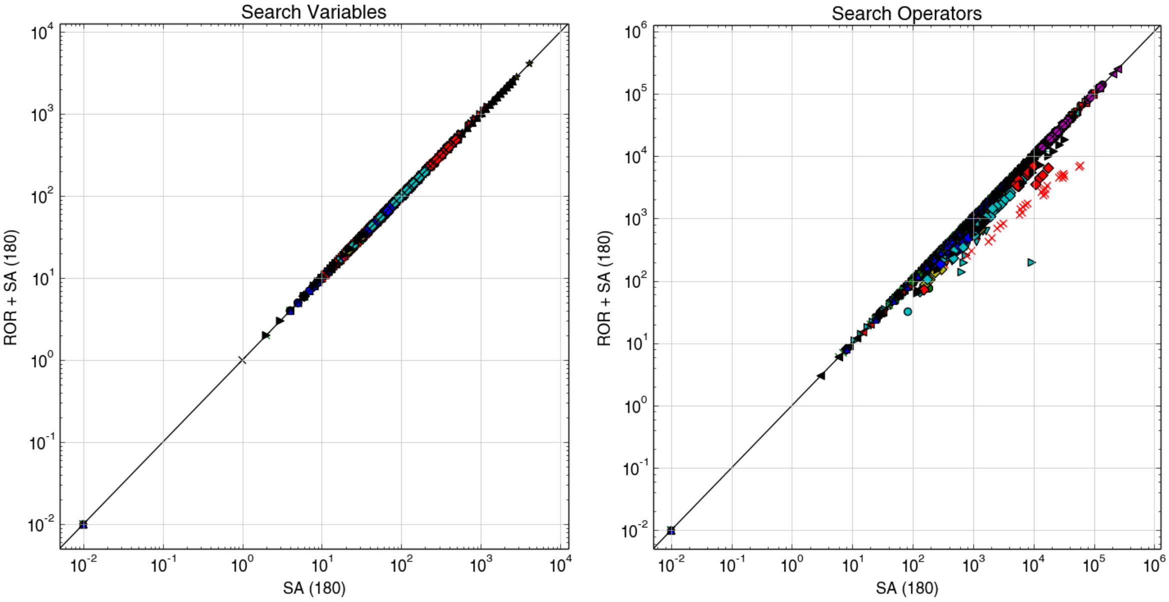


(a) Black triangles: `Tpp`
Green crosses: `Rovers`

(b) Bottom edge: `Logistics`, `Miconic`, `Movie`
Red triangles: `Satellite`

Figure 6.9.: Synergies of using Safe Abstraction before ROR.

The reverse direction, `ROR+SA`, is visualized in Figure 6.10, although the positive synergies are too small to be visible in the plot. In a large number of `Woodworking`, `Pegsol`, and `Trucks` instances, a small number of around 1-3 variables can be additionally abstracted when ROR has removed the redundant operators. Negative synergies do not



(a) (b)

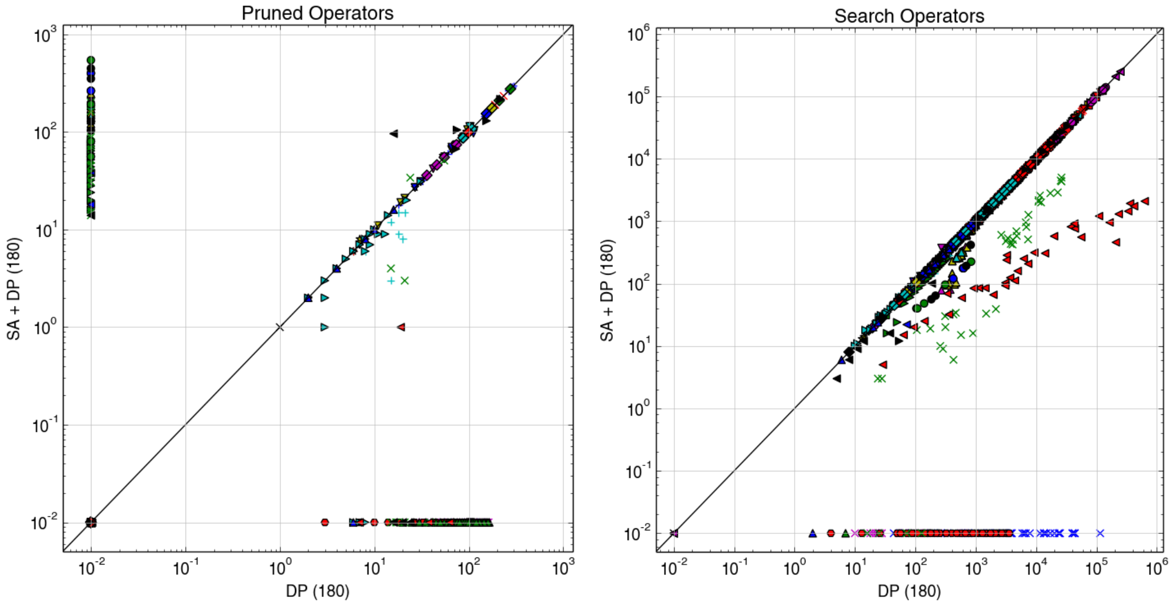
Figure 6.10.: Synergies of using ROR before Safe Abstraction

occur, meaning Safe Abstraction is never less active after ROR, as shown in Theorem 7.

6.3.2. Dominance Pruning and Safe Abstraction

Figure 6.11 compares SA+DP against using only DP. Positive synergies of applying Safe Abstraction before DP appear in many instances, so that more operators can be pruned (a, above the diagonal) - in the domains Depot, Elevators, Transport, Gripper, Rovers, Tpp, Driverlog and one Zenotravel instance. In some instances, less operators are pruned by DP, not because of negative synergies but because the operators are removed by Safe Abstraction. As the experimental results in (b) show, the combination does not leave behind more operators than using only DP. This supports the previously proven Theorem 9. A few exceptions (above the diagonal) are caused by slight time variances in the computation (noise).

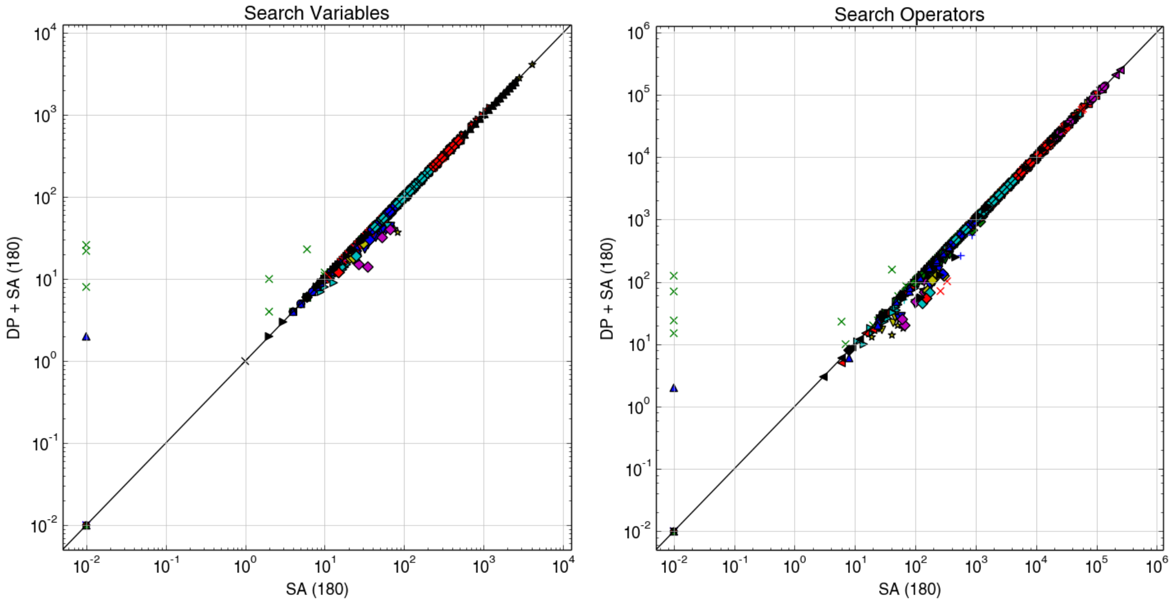
The results of the equivalent experiment for DP+SA are shown in Figure 6.12. With the exception of some Rovers and one Visitall instance (above the diagonal), which can be abstracted before, but not after Dominance Pruning, the positive synergies seem to outweigh the negative, both in operators and variables removed. Safe Abstraction can abstract more variables after DP in Psr-small, Woodworking, Parcprinter, Pathways, Airport, and Satellite instances.



(a) Bottom edge: `Logistics`, `Miconic`, `Movie`

(b)

Figure 6.11.: Synergy effects of `SA+DP`.



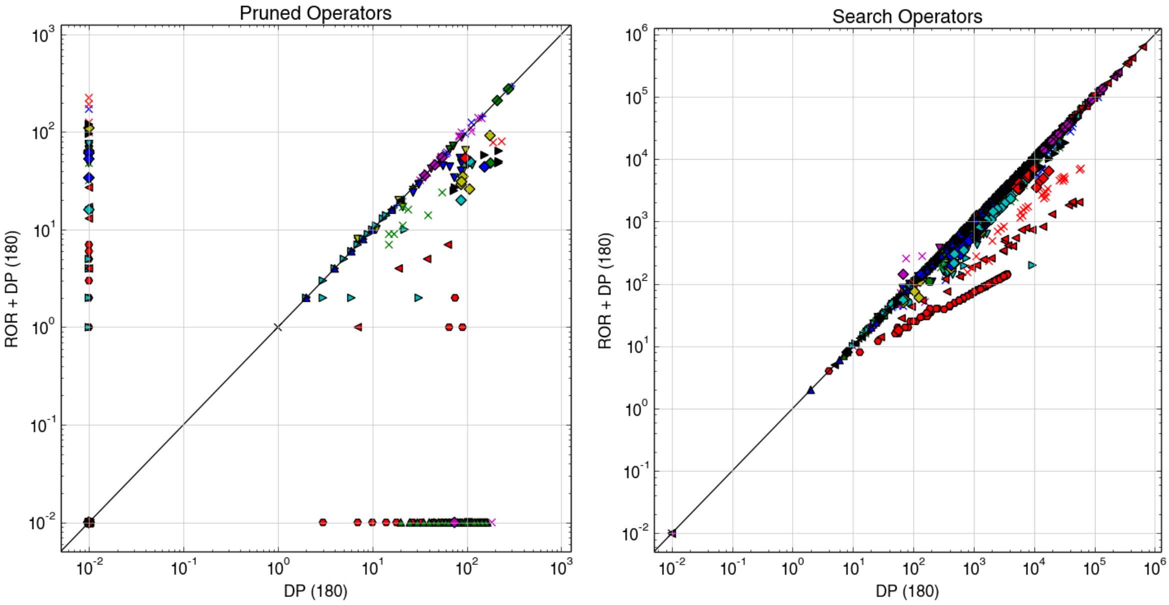
(a) Green crosses: `Rovers`
Blue triangle: `Visitall`

(b)

Figure 6.12.: Synergy effects of `DP+SA`.

6.3.3. Redundant Operator Reduction and Dominance Pruning

Figure 6.13 visualizes the experiment which explores the synergies `ROR+DP`. Positive synergies are not infrequent (a), in a row of instances (above the diagonal) DP benefits from less available redundant operators. Although nothing can be proven regarding the negative synergies of this combination, the number of search operators left in the problem is almost exclusively lower when using the combination (b). That's the case, because in most cases for one operator that is reduced by ROR, there's maximally one operator which Dominance Pruning is not able to prune anymore. Instances in which DP can prune more operators after redundant operators are removed (positive synergies) mostly include `Miconic` instances, but also instances from various other domains, such as `Parcprinter`, and `Woodworking`.

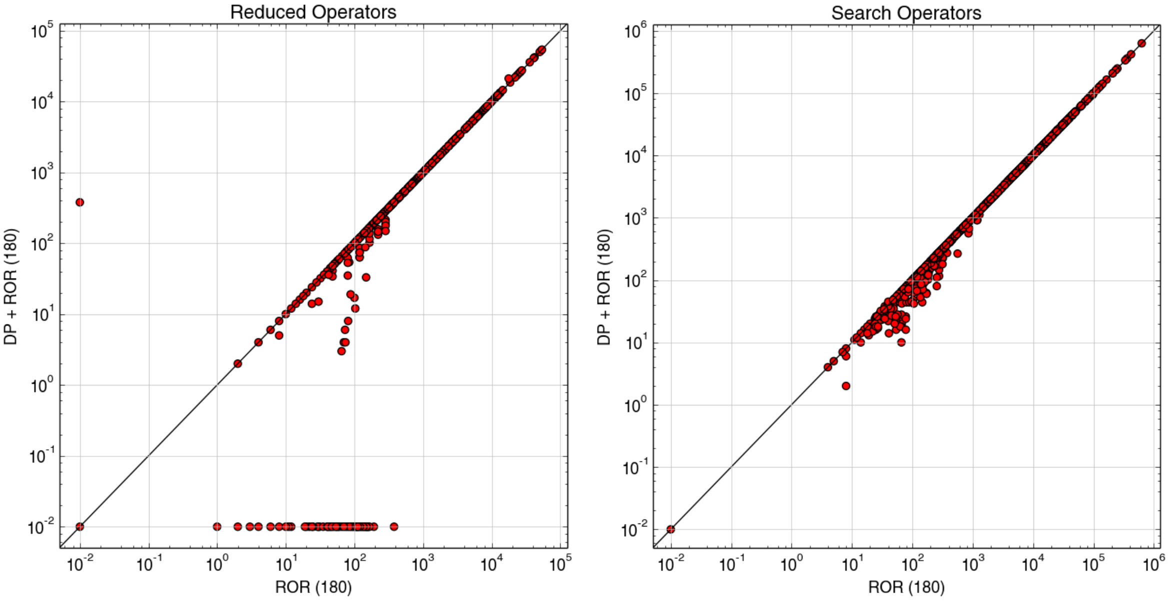


(a)

(b)

Figure 6.13.: Synergy effects of `ROR+DP`.

The experimental results of the reversed order, `DP+ROR`, are visualized in Figure 6.14. Similarly to the case described above, we achieve having less search operators left (b) than with ROR alone in almost all instances. Although there are negative synergies, usually there is maximally one less operator which ROR can remove for every operator pruned by DP. Only in a few `Miconic` the total number of operators removed is smaller than what ROR alone is capable of removing, since by pruning one operator, several operators are not removable by ROR anymore. The number of operators removed by ROR is in general never higher when applying Dominance Pruning first, only in two instances (above the diagonal in (a)) ROR is more active. These, however, are artifacts of a different variable ordering: Dominance Pruning did not remove any operator from the tasks. As shown in Theorem 14, positive synergies would be possible, but only in rare cases - since a variable needs to become static, which then has to cause an operator to become redundant - this did not appear in any domain.



(a)

(b)

Figure 6.14.: Synergy effects of DP+ROR.

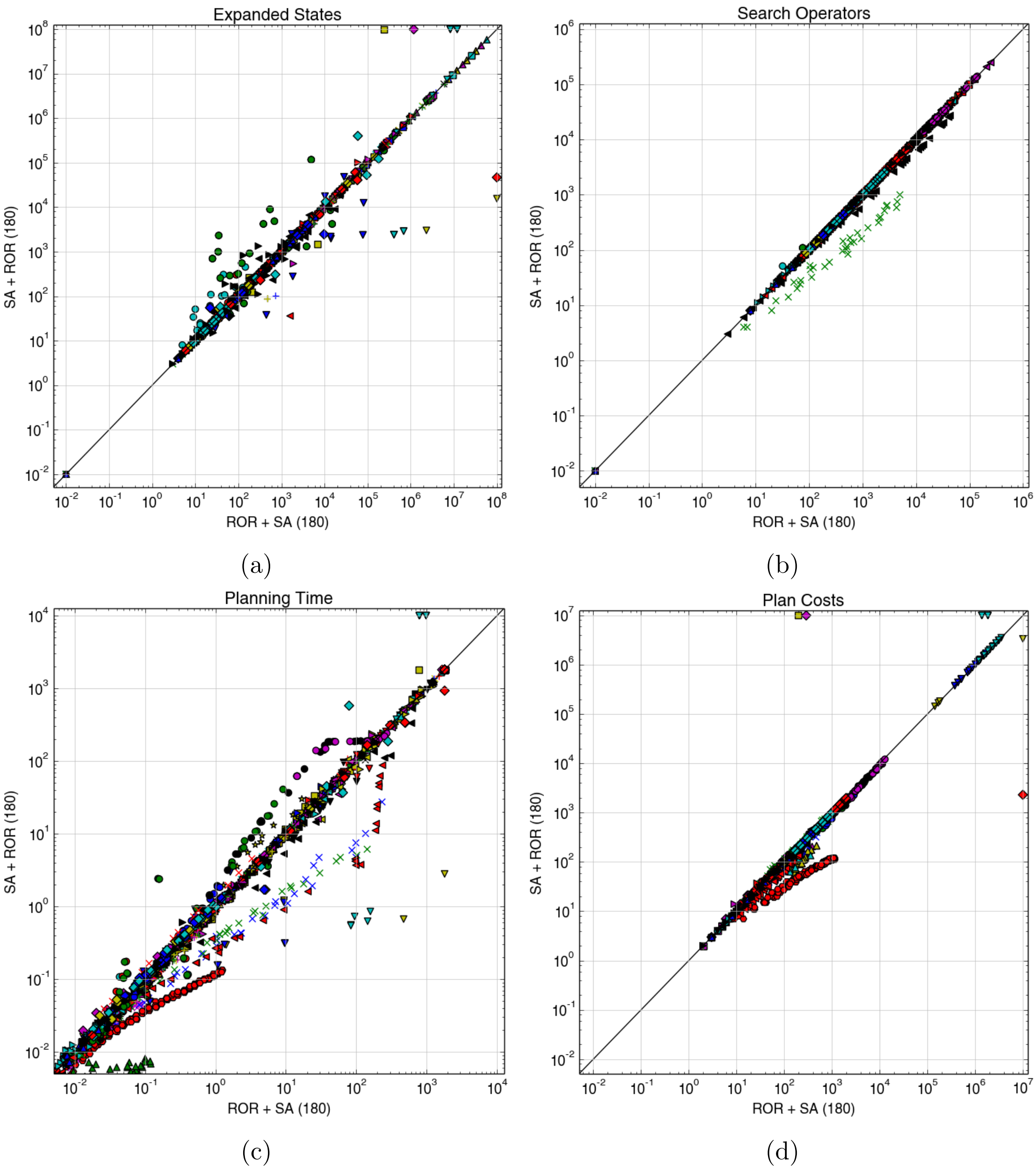
6.4. Performances in combination

In this section, we take a look at the different combinations, compare them with each other and quantify how important the synergies are for achieving a better pruning result. Every technique in this section is given a time limit of 180 seconds, meaning a combination of two techniques has a total time limit of 360 seconds.

Figure 6.15 compares the two possible combinations of SA with ROR. Although the number of search operators (b) is in general smaller when applying SA first (because of the positive synergy the abstracted variables have on ROR), and the planning time (c) is also often smaller, the coverage is slightly greater when ROR + SA is used. The increased coverage (shown in Table 6.10) is not caused by positive synergies of ROR on SA, though - as we've seen, they only appear in `Pegsol` and `Woodworking` instances. Since static and irrelevant variables are removed before `ROR+SA`, but not in `SA+ROR` (since Safe Abstraction abstracts those variables), the variables are saved in a different order, which allowed a few additional instances to be solved. Although fewer search operators are available when applying Safe Abstraction first, the plan costs (d) are lower in most instances compared to the reversed order.

The two combinations of DP with SA are compared with another in Figure 6.16. Applying SA before DP produces in general better results than the reversed order, meaning that more operators are removed (b) and less states are expanded (a). The positive synergies of DP on SA seem to be outweighed by its negative effects, which appear mainly in `Rovers`, and by the positive synergies of SA and DP. The planning time plot (c) visualizes two interesting effects: Instances in which Safe Abstraction is very active take much shorter when SA is applied before DP (below the diagonal); in the vertically aligned instances DP reaches its time limit on the original instance, but is very fast on the abstracted instance. The horizontally aligned instances above the diagonal are a side-effect of using an early-abort criterion: DP cannot prune any operators on the original tasks and therefore aborts before merging, while after application of SA some operators can be pruned and DP takes up its full time limit. Concerning the planning cost (d), no major differences can be observed.

Lastly, the combinations of DP with ROR are shown in Figure 6.17. While there is no significant difference in the expanded states (a), there are a few less search operators (b) left when ROR is applied first, since the positive synergies in the opposite direction are rare. In the planning time (c) we do not see a clear preference for one of the two. On the one hand, ROR is faster in removing operators than DP, on the other it potentially removes operators which are needed for a faster search time. The horizontally aligned instances above the diagonal are again a side-effect of using the H_2 abort criterion.



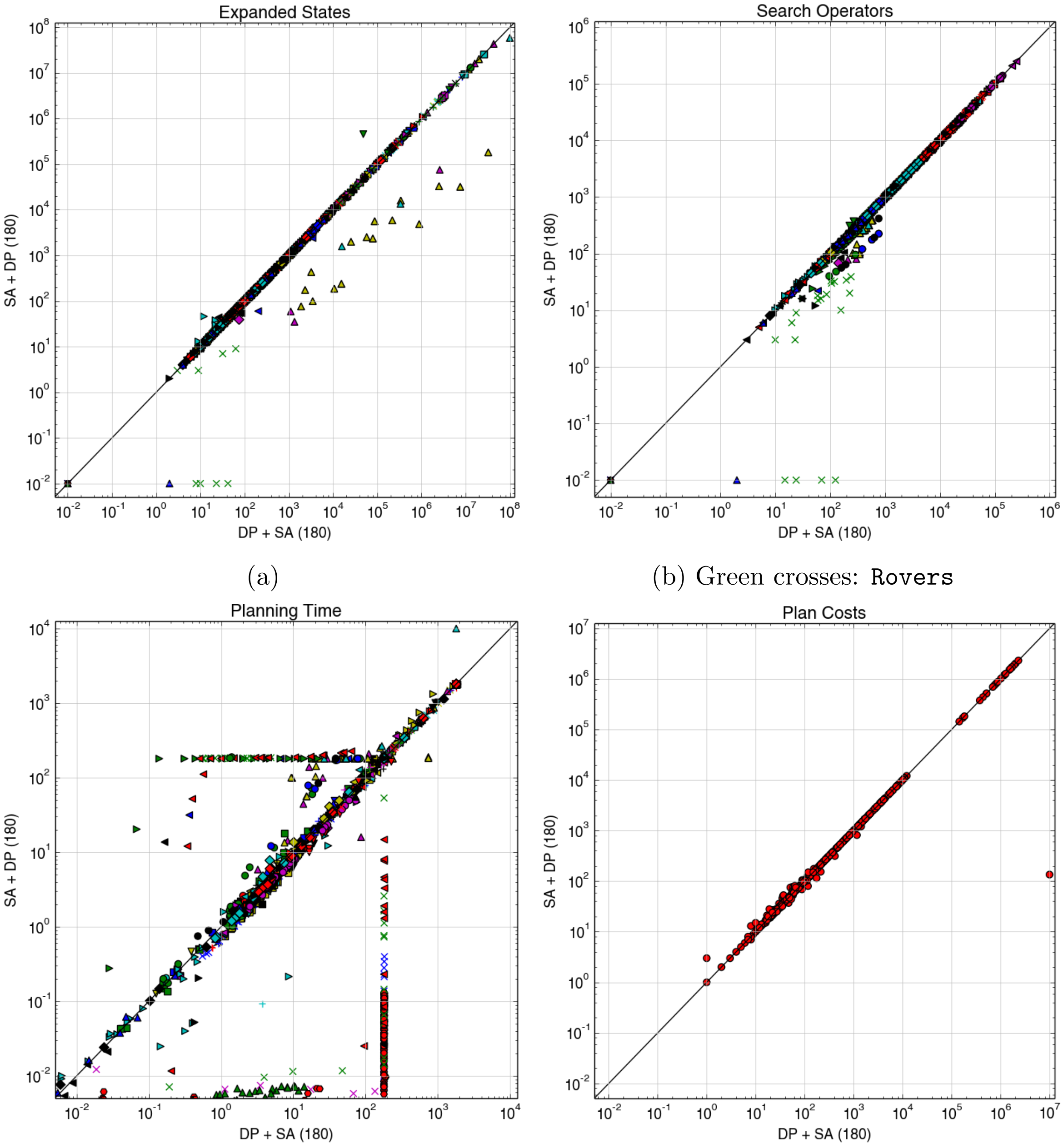
(a)

(b)

(c)

(d)

Figure 6.15.: Comparison of both combinations with SA and ROR.



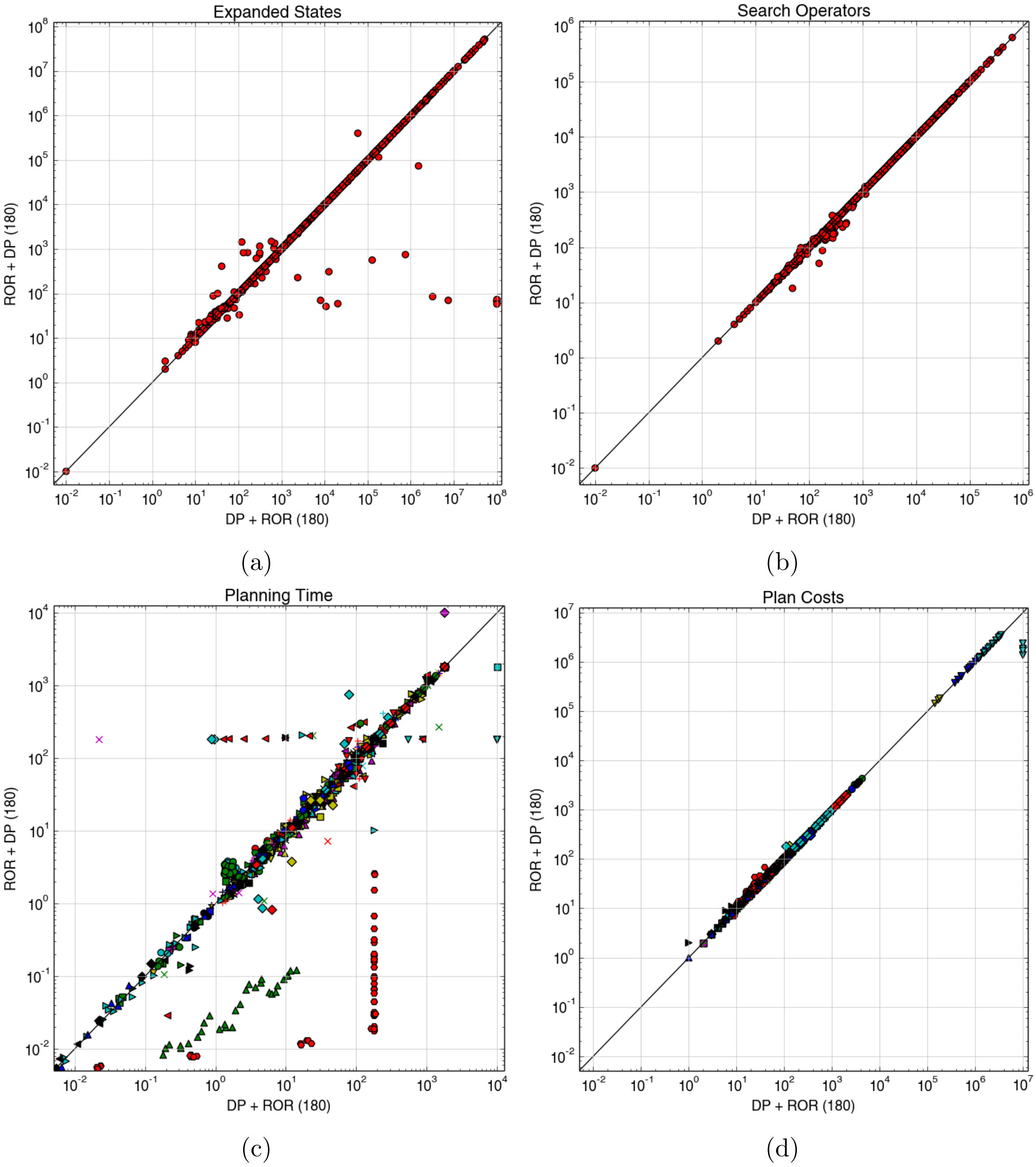
(a)

(b) Green crosses: `Rovers`

(c)

(d)

Figure 6.16.: Comparison of both combinations with DP and SA.



(a)

(b)

(c)

(d)

Figure 6.17.: Comparison of both combinations with DP and ROR.

Table 6.10 shows the coverage of the mentioned combinations along with the single techniques. Dominance Pruning does not impact the coverage by a lot, a small reduction can be the case. The worst results are achieved by using ROR, alone or in either combination with Dominance Pruning, though together with Safe Abstraction the coverage did not suffer. Safe Abstraction clearly improves the coverage the most, and so do the combinations of Safe Abstraction with other techniques, though there seems to be no real benefit of using the other techniques together with Safe Abstraction. A small increase in coverage was observed when redundant operators were removed before Safe Abstraction, but as mentioned a different variable ordering was identified as the reason, not a positive synergy of the two.

	Solved	aborted (time)	aborted (memory)	unsolvable
Baseline	1414	388	76	8
SA	**1484**	312	82	8
ROR	1390	394	94	8
DP	1413	409	56	8
SA + ROR	**1484**	316	78	8
ROR + SA	**1490**	316	72	8
SA + DP	**1486**	322	70	8
DP + SA	**1486**	322	70	8
ROR + DP	1394	403	81	8
DP + ROR	1384	404	90	8

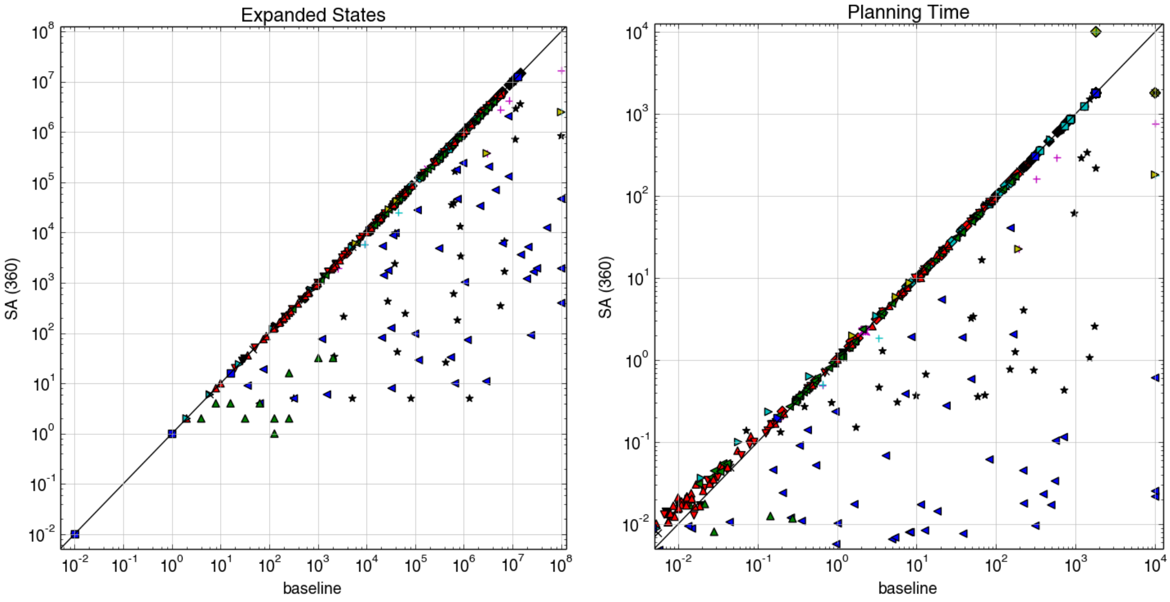
Table 6.10.: Solved and aborted problems out of 1886 instances with different combinations. Preprocessing had a time limit of 360 seconds in total for every experiment (except for the baseline). Combinations/techniques with increased coverage compared to the baseline are marked in bold.

6.5. Unsolvable instances

In this section, we evaluate the three pruning techniques on a set of problem domains in which all instances are unsolvable. We use the unsolvable benchmarks by Hoffmann et al. (2014) as well as unsolvable benchmarks by Jendrik Seipp, Florian Pommerening, Chris Fawcett, Silvan Sievers, Yusra Alkhazraji and Martin Wehrle (the latter benchmarks have been generated in the context of their portfolio planner submission to the unsolvability IPC 2016).

The aim of a planner in such an instance is to prove that it is unsolvable, for which every state in the state space needs to be expanded. Since all three pruning techniques are solution-preserving, proving that the pruned problem is unsolvable is sufficient to prove the same for the original problem.

Figure 6.18 shows the expanded states and the planning time when using Safe Abstraction against a run without it (both with greedy search and FF heuristic). The domains represented by the symbols are listed in Figure 6.19.



(a)

(b)

Figure 6.18.: Performance of Safe Abstraction on unsolvable problem instances.

The advantage of Safe Abstraction in this context is that refining a plan is not necessary, since no plan exists. We can see that in a row of domains Safe Abstraction reduces the number of states in the search space considerably (a), while in other domains it has no effect at all. Similarly, the planning time (b) needed to prove that a problem is unsolvable is reduced for the same instances, it also completes expanding the search space

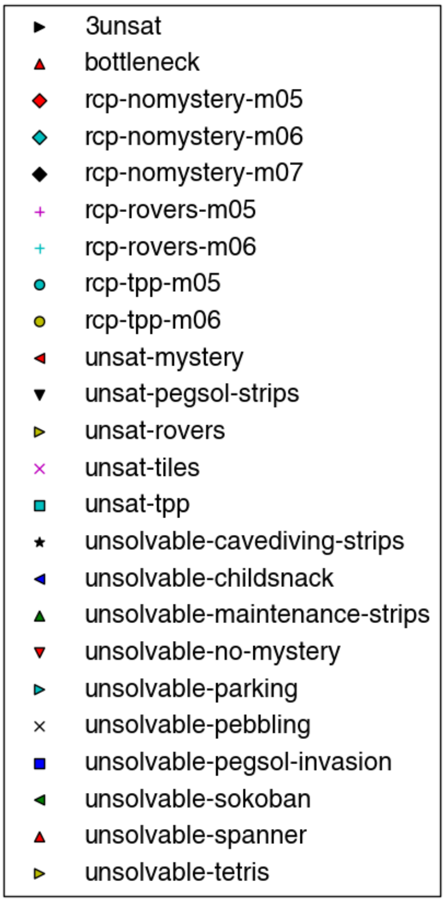


Figure 6.19.: Legend for the plots in Figure 6.18.

in some instances where without Safe Abstraction the time limit is reached. (instances on the right edge).

As every state in the search space needs to be expanded, Redundant Operator Reduction does not have any benefit since it does not prune any state. Dominance Pruning allows to prune less states only in a small number of instances and at the same time introduces a huge overhead in time for the computation, so often the search space cannot be completely expanded anymore. Both these techniques do not seem to be well suited to be used in the context of unsolvable instances.

7. Discussion on Safe Abstraction

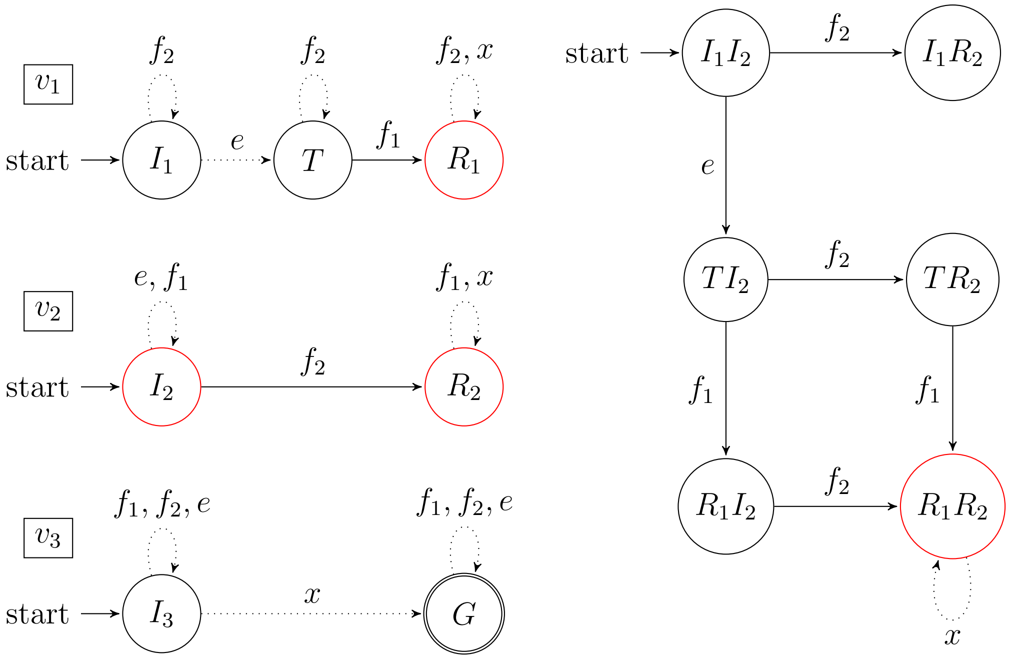
As the experiments in the previous chapter have shown, despite the existing positive synergies between the pruning techniques, applying two of them in combination does not yield a better coverage than either of the techniques does separately (when excluding noise effects). Safe Abstraction is responsible for the improved coverage in each of its combinations, and it also causes that increase on its own. Being the most promising of the three techniques, we discuss one possible extension to Safe Abstraction in this chapter. This extension has not been implemented in this work, we describe the general idea and the conditions for Safe Abstraction in an informal way.

Although the conditions for Safe Abstraction are defined on the domain transition graphs of variables, they could as well be defined on the atomic transition systems, since both contain the relevant information. The states in the ATS are then the equivalent of the values of the variable. One extension that could possibly improve the power of Safe Abstraction further is to merge these transition systems together and to compute Safe Abstractions on the so-achieved LTSs, abstracting several variables together which are not safe on their own, but whose merged transition system is safe. An example of such an instance is shown in Figure 7.1.

After merging and abstracting (v_1, v_2) together, v_3 also becomes safe, and a search is not needed therefore. Refining that plan would first form a less abstract task by reinserting v_3, for which $\langle x \rangle$ is a plan. Then (v_1, v_2) would be reinserted. For the operator x in the abstract plan, a path must be found in the merged LTS from the initial state $I_1 I_2$ to $R_1 R_2$, as x has that requirement. Either $\langle e, f_1, f_2, x \rangle$ or $\langle e, f_2, f_1, x \rangle$ are plans that would be created by the refinement process.

In the extreme case in which all LTS are merged together, no operator has any condition/effect on an external variable, meaning the only condition for Safe Abstraction is that a goal state is reachable from the initial state. All variables are safely abstractable together if a plan exists. However, there is no benefit in merging up to that point, as the entire effort of the search would be moved to the refinement process, in which a path from the initial state to a goal state needs to be found. Considering that there is the additional overhead of finding out whether a path exists, to fulfill the conditions for Safe Abstraction, the work would be doubled compared to a simple search. Clearly a measurement needs to be found to determine at which point this strategy is not beneficial anymore.

The example shown in Figure 7.1 is too small, so that any abstraction slows down



(a) ATS of three variables v_1, v_2, v_3. No variable is safe, as R_1 / I_2 / G are not free reachable from I_1 / R_2 / I_3.

(b) LTS of $v_1 \otimes v_2$. Safely abstractable, as e, f_1 and f_2 have no effect/condition on the third variable.

Figure 7.1.: Problem instance (with variables v_1, v_2, v_3) in which Safe Abstraction would be possible on merged transition systems. Externally required values are marked in red. The dotted edges correspond to operators which have no effect on that variable.

the search. The state space of the original problem that needs to be searched contains 7 reachable states, including the initial state. Abstracting (v_1, v_2) means that the remaining abstract problem contains 2 states (those of v_3) that need to be searched. For the refinement, the merged LTS needs to be searched for a path, containing 6 reachable states. With the additional overhead for abstraction (merging the LTS, detection whether it's safe, building the abstract problem) there cannot be any gain.
In general, when abstracting an LTS with n states, the abstracted problem's search space (including unreachable states) has n times fewer states. The costs of this operation are the reachability checks for abstraction and shortest-path searches (both on those n states) for every operator that needs to be inserted into the plan. If the total number of states in the search space is considerably larger than n, then abstracting the LTS should prove to be useful.

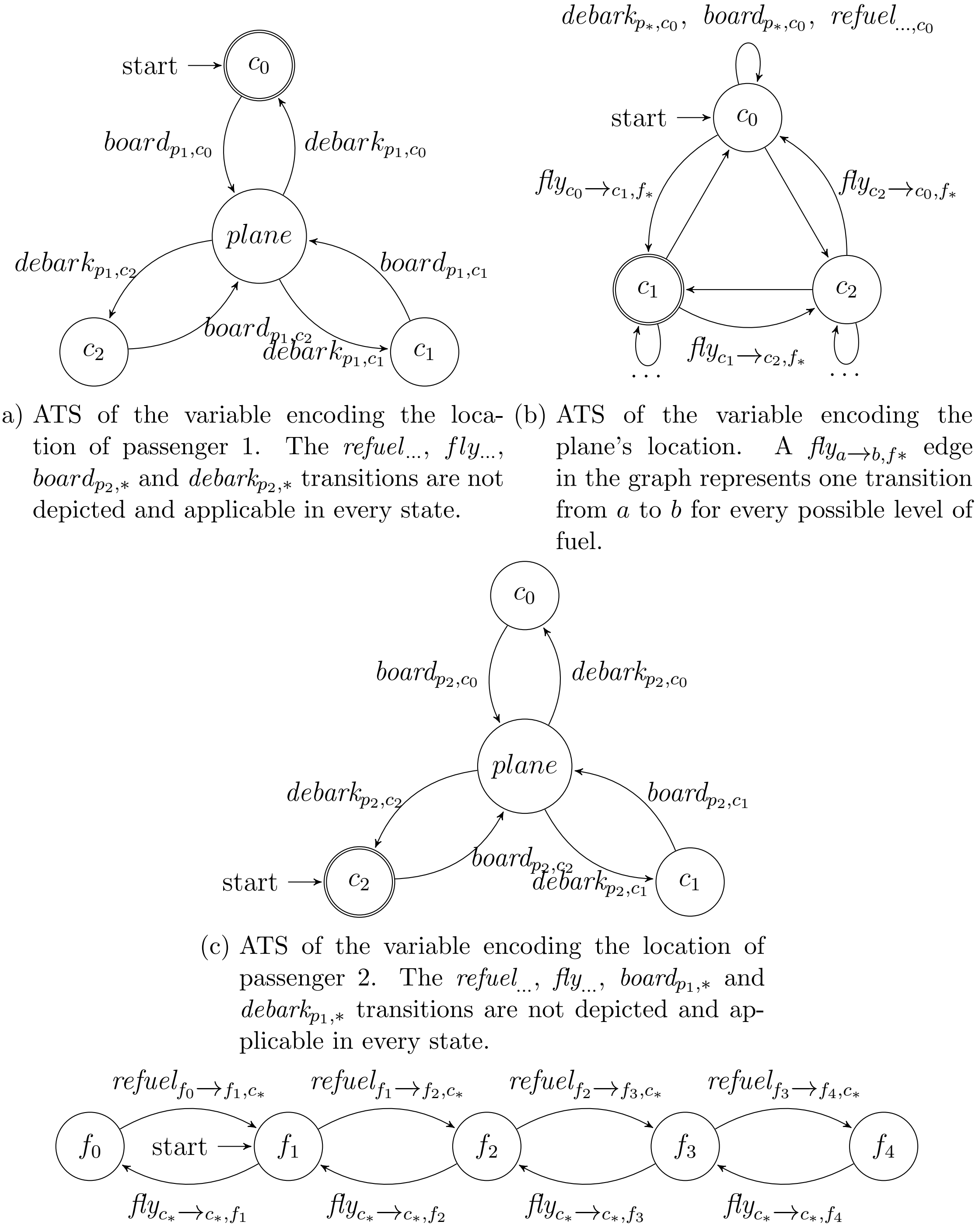
To abstract several variables together, the conditions for safe variables need to be reconsidered. In the above example, it is obvious that the state R_1R_2 is externally required, since it is the only state in which the operator x is applicable. If the operator was applicable in several states, though, meaning it had a precondition on only one of the variables, defining all of them to be externally required states would be very restrictive. The same applies to goal states, if only one of the two variables has a goal value, and to externally caused states. If all of these states have to be connected according to Haslum's condition, not many additional variables can be abstracted in this way. Out of the domains examined in this work, there's only one instance in which two variables exist which are not safe on their own, but could be abstracted using this concept. That instance from the `Zenotravel` domain is shown in Figure 7.2.

In this `Zenotravel` instance, a plane can travel (fly) between 3 cities, c_0, c_1 and c_2 (b). Two passengers (a, c) can board the plane when the plane is located at their starting city, and debark the plane later on again. Every flight costs the plane one level of fuel (d), it can be refueled at any city. The goal is for the passengers to arrive at their destination, and for the plane to arrive at the city c_1. A plan for this problem is $\langle \mathit{fly}_{c_0 \rightarrow c_1, f_1} \rangle$, since the passenger are already at their destination in the initial state.

The two variables encoding the passenger's location (a, c) are both safe: Neither of them has an externally required value, and their initial state is equal to their goal state. The two variables encoding the plane's location and its level of fuel cannot be abstracted, since the fly and refuel operators have preconditions and effects on both of them. Merging the two ATS together, every state $c_* f_*$ in the resulting LTS is free reachable from any other state. Therefore, abstracting those two variables together is possible. Since no variables are left in the abstract problem, no search is needed. To refine the plan, a path must be found in the merged LTS from the state $c_0 f_1$ to any goal state $c_1 f_*$, either by directly flying from c_0 to c_1, or by first refueling or making a detour to c_2.

Requiring free reachability of all states, in which operators are applicable that work on other variables, appears to be too restricting. The concept behind safe variables is, that every abstract plan can be refined to a plan for the concrete problem. This refining is still possible when not all relevant states are freely reachable, but it requires a more complex refining algorithm. Consider Figure 7.3 for an example, which depicts a problem instance with three variables. None of the variables is initially safe: In v_1, the externally required value R_1 is not free reachable from the initial state. In v_2, the externally required value I is not free reachable from R_2. In v_3, the goal value is not free reachable from the initial state.

Subfigure (b) shows the LTS resulting from merging the ATS of v_1 and v_2 together. Using the strict condition mentioned above, the two variables are not safely abstractable, since some externally required values are not free reachable from IR_2. However, if the variables were abstracted, every plan for the resulting abstract problem could be refined to a concrete plan, as long as the refining algorithm ensures that the state IR_2 is not reached, since from that state, no other state is free reachable in which y is applicable.



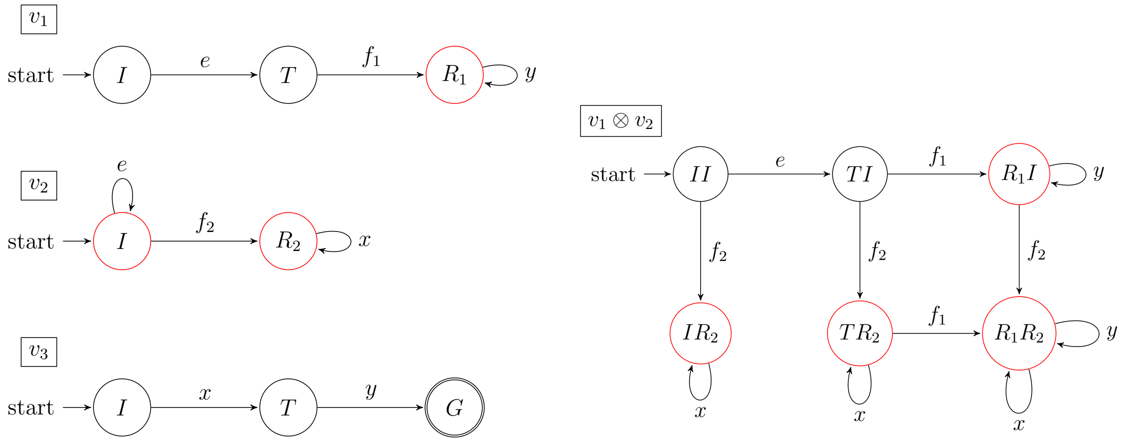
(a) ATS of the variable encoding the location of passenger 1. The $refuel_{\ldots}$, $fly_{\ldots}$, $board_{p_2,*}$ and $debark_{p_2,*}$ transitions are not depicted and applicable in every state.

(b) ATS of the variable encoding the plane's location. A $fly_{a \to b,f*}$ edge in the graph represents one transition from a to b for every possible level of fuel.

(c) ATS of the variable encoding the location of passenger 2. The $refuel_{\ldots}$, $fly_{\ldots}$, $board_{p_1,*}$ and $debark_{p_1,*}$ transitions are not depicted and applicable in every state.

(d) ATS of the variable encoding the level of fuel. A $fly_{c_* \to c_*,f_a}$ edge in the graph represents one transition for every two cities, $refuel_{f_a \to f_b,c_*}$ equivalently. The The $board_{\ldots}$ and $debark_{\ldots}$ transitions are not depicted and applicable in every state.

Figure 7.2.: `Zenotravel` instance in which a merged LTS allows the safe abstraction of two variables



(a) ATS of the three variables v_1, v_2, v_3. The transitions of f_1, f_2 and e are not depicted in the LTS in which they do not have a precondition or an effect.

(b) ATS of v_1 and v_2 merged together.

Figure 7.3.: Example instance with three variables, in which merging and abstracting v_1 and v_2 requires a different refining algorithm.

Removing v_1 and v_2, the abstract plan is $\langle x, y \rangle$. When refining, the preconditions of x on v_2 need to be satisfied first. A naive refining algorithm might apply f_2 directly, since then the preconditions are satisfied. Since IR_2 must not be visited, either the state TR_2 or R_1R_2 can be considered.

To ensure that every abstract plan can be refined, a definition of Safe Abstraction on merged LTS needs to contain an *at least one*-semantic. For every externally required value of one variable, at least one state s_r in the merged LTS assigning that value must be free reachable from at least one state for every other externally required value and from the initial state. If externally caused values exist, s_r must also be free reachable from at least one state s_c for each externally caused value. Further, denoting the operator which causes that externally caused value with o, we need to ensure that one state s_o must be free reachable, in which a transition $s_o \xrightarrow{o} s_c$ exists. Otherwise, if the operator o is applied in the abstract plan, it might not be possible for the refining algorithm to end up in s_c after o is applied but in some other state caused by the operator, from which the externally required states are not free reachable.

In summary, a cluster of states must exist in the merged LTS which allows every operator to be applied and every goal condition to be satisfied. The refining algorithm must then ensure that while inserting operators into the abstract plan, it never runs into a dead end (like the state IR_2 in Figure 7.3.b).

8. Conclusion

To conclude, we first briefly review the subjects covered in this work and then recapitulate the results of the thesis in more detail.

In this work, we have discussed three different existing static pruning techniques, Safe Abstraction, Redundant Operator Reduction, and Dominance Pruning. We analysed how each of them prunes the search space, and proposed modifications and extensions to them in order to further increase their pruning power. We investigated the possible synergy effects in each combination of two pruning methods, and observed that most of the synergies also occur in instances of IPC benchmarks. We described the implementation of the three techniques into an existing propositional planner and evaluated the methods on IPC benchmarks. We measured the performance of different configurations for each technique, and the magnitude of the synergy effects between the methods.

As two conditions for Safe Abstraction have been proposed, we have compared their performances with each other. Both lead to a considerable increase of coverage, Haslum's condition allows to abstract more variables in many domains and to solve some additional domains without search. When used in the context of unsolvable instances, Safe Abstraction reduces the number of states which need to be expanded, consequently speeding up the search process. To further extend this technique, we discussed an idea in the last chapter based on calculating Safe Abstractions on merged atomic transition systems. This approach has not been implemented yet, but is an interesting point for future research.

The technique of Redundant Operator Reduction has so far only been used on a small set of domains. In this work, we have evaluated it on a bigger set. Although ROR is capable of removing many operators from the planning tasks, its pruning is not consistently beneficial for the search. In many instances, having fewer redundant operators available leads to an increase in the number of states which have to be expanded and in the search time. A dynamic version of ROR, which avoids to prune transitions based on knowledge used in Tunnel Macros, might be able to reduce these negative effects.

We observed that ROR can be combined together with Safe Abstraction without either of the techniques having negative effects on the pruning power of the other. While positive synergies do exist, they are too weak to increase the coverage.

As a third pruning method, we have studied the static version of Dominance Pruning.

We proposed an implementation that allows to prune transitions which subsume each other in order to increase the number of operators that can be removed. The technique is able to remove many operators, consequently fewer states need to be expanded, but the preprocessing overhead does not pay off in terms of overall planning time, even when using the incremental computation and an early-abort criterion. If Dominance Pruning is allowed to ignore label costs, only very few operators can be pruned additionally.

Despite being an optimality-preserving method, we were able to identify positive synergy effects on Dominance Pruning when the two satisficing techniques are used before it. In practice, the methods would not be combined because the optimality-preserving property is lost, and the overhead of DP is too high as a satisficing preprocessing method. If a faster, suboptimal variant of DP was found, it could be used for satisficing search in combination with Safe Abstraction.

A. Plot legend

★	airport
×	barman-opt11-strips
×	barman-sat11-strips
■	blocks
◀	depot
+	driverlog
▲	elevators-opt08-strips
▲	elevators-opt11-strips
▲	elevators-sat08-strips
▲	elevators-sat11-strips
▼	floortile-opt11-strips
▼	floortile-sat11-strips
■	freecell
◀	grid
▶	gripper
×	logistics00
×	logistics98
●	miconic
▲	movie
+	mprime
×	mystery
+	nomystery-opt11-strips
◆	nomystery-sat11-strips
■	openstacks-opt08-strips
■	openstacks-opt11-strips
■	openstacks-sat08-strips
■	openstacks-sat11-strips
■	openstacks-strips
▼	parcprinter-08-strips
▼	parcprinter-opt11-strips
▼	parcprinter-sat11-strips
▶	parking-opt11-strips
▶	parking-sat11-strips
◆	pathways-noneg
●	pegsol-08-strips
●	pegsol-opt11-strips
●	pegsol-sat11-strips
▶	pipesworld-notankage
▶	pipesworld-tankage
▶	psr-small
×	rovers
◀	satellite
◀	scanalyzer-08-strips
◀	scanalyzer-opt11-strips
◀	scanalyzer-sat11-strips
+	sokoban-opt08-strips
+	sokoban-opt11-strips
+	sokoban-sat08-strips
+	sokoban-sat11-strips
◆	storage
▼	tidybot-opt11-strips
▼	tidybot-sat11-strips
◀	tpp
●	transport-opt08-strips
●	transport-opt11-strips
●	transport-sat08-strips
●	transport-sat11-strips
×	trucks-strips
▲	visitall-opt11-strips
▲	visitall-sat11-strips
◆	woodworking-opt08-strips
◆	woodworking-opt11-strips
◆	woodworking-sat08-strips
◆	woodworking-sat11-strips
▶	zenotravel

B. Bibliography

Alkhazraji, Y., Wehrle, M., Mattmüller, R., and Helmert, M. (2012). A stubborn set algorithm for optimal planning. *Proc. 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 891 – 892.

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

Crawford, J., Ginzberg, M., Luks, E., and Roy, A. (1996). Symmetry-breaking predicates for search problems. *KR*, 96:148 – 159.

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

Domshlak, C., Katz, M., and Shleyfman, A. (2012). Enhanced symmetry breaking in cost-optimal planning as forward search. *Proc. ICAPS 2012*.

Domshlak, C., Katz, M., and Shleyfman, A. (2013). Symmetry breaking: Satisficing planning and landmark heuristics. *Proc. ICAPS 2013*, pages 298 – 302.

Fox, M. and Long, D. (1999). The detection and exploitation of symmetry in planning problems. *Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 956 – 961.

Godefroid, P. (1996). Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem. *Springer-Verlag*, 1032 of LNCS.

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

Haslum, P. (2007). Reducing accidental complexity in planning problems. *Proc. 20th International Joint Conference on Artificial Intelligence (ICJAI-07)*, pages 1898 – 1903.

Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. *Proc. AAAI 2007*, pages 1007 – 1012.

Haslum, P. and Jonsson, P. (2000). Planning with reduced operator sets. *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 150 – 158.

Helmert, M. (2006). Fast (diagonally) downward. *5th International Planning Competition Booklet*. Available at: http://zeus.ing.unibs.it/ipc-5/ (Accessed: 2016-05-06).

Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway? *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169.

Helmert, M. and Geffner, H. (2008). Unifying the causal graph and additive heuristics. *Proc. 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 544 – 550.

Hoffmann, J., Kissmann, P., and Torralba, A. (2014). "distance"? who cares? tailoring merge-and-shrink heuristics to detect unsolvability. *Proc. 21st European Conference on Artificial Intelligence (ECAI 14)*, pages 441 – 446.

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

Holte, R. C., Alkhazraji, Y., and Wehrle, M. (2015). A generalization of sleep sets based on operator sequence redundancy. *Proc. 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 3291 – 3297.

Junghanns, A. and Schaeffer, J. (2001). Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129:219 – 251.

Kissmann, P. and Torralba, A. (2015). Focusing on what really matters: Irrelevance pruning in merge-and-shrink. *Proc. 8th International Symposium on Combinatorial Search*, pages 122 – 130.

Nissim, R., Apsel, U., and Brafman, R. (2012). Tunneling and decomposition-based state reduction for optimal planning. *ECAI*, pages 624 – 629.

Pochter, N., Zohar, A., and Rosenschein, J. S. (2011). Exploiting problem symmetries in state-based planners. *Proc. AAAI 2011*, pages 1004 – 1009.

Pommerening, F., Röger, G., Helmert, M., and Bonet, B. (2014). LP-based heuristics for cost-optimal planning. *Proc. 24th International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pages 226 – 234.

Sievers, S., Wehrle, M., and Helmert, M. (2014). Generalized label reduction for merge-and-shrink heuristics. *Proc. 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2358 – 2366.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146 – 160.

Torralba, A. and Hoffmann, J. (2015). Simulation-based admissible dominance pruning. *Proc. 24th International Joint Conference on Artificial Intelligence (IJCAI'15).*

Valmari, A. (1989). Stubborn sets for reduced state space generation. *Proc. 10th International Conference on Applications and Theory of Petri Nets (APN 1989)*, pages 1 – 22.

Wehrle, M. and Helmert, M. (2009). The causal graph revisited for directed model checking. *Proc. 16th International Static Analysis Symposium (SAS 2009)*, pages 86 – 101.

Wehrle, M. and Helmert, M. (2012). About partial order reduction in planning and computer aided verification. *Proc. 22th ICAPS 2012*, pages 297 – 305.

Wehrle, M. and Helmert, M. (2014). Efficient stubborn sets: Generalized algorithms and selection strategies. *Proc. 24th ICAPS 2014*, pages 323 – 331.

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)

~~Bachelor's~~ / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:

Analysing and Combining Static Pruning Techniques for Classical Planning Tasks

First Name, Surname *(Please print in capital letters)*: Pressmar, Michaja

Matriculation No.: 11-059-011

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

□ Yes ☒ No

Place, Date: D-Lörrach, 14.05.2016

Signature: Michaja Preßmar

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI
BASEL