University of Basel

Safe Abstraction in Fast Downward

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Remo Christen

Joan Moser
Gian.Moser@unibas.ch
21-050-372

15.12.2024

Acknowledgments

I would like to thank my supervisor Remo Christen for his consistent patience, insightful guidance, and invaluable feedback throughout the course of my thesis. His support has been instrumental in helping me navigate the many pitfalls along the way and made the experience of writing my thesis a fun one. I would also like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my Bachelor's thesis in the Artificial Intelligence Research Group.

Abstract

Classical planning is the discipline of finding a sequence of operators that reach a goal from a given initial state. Search algorithms are used to explore the state space of a problem to find a sequence of operators that solve it. When a problem is encoded, accidental complexity may be added. This accidental complexity makes the problem harder than it fundamentally is. One approach to reduce accidental complexity is safe abstraction. We have implemented safe abstraction in the Fast Downward planning system to simplify the problems before a search algorithm is used on them. Our findings show that safe abstraction can significantly reduce the complexity of some problems.

Table of Contents

1 Introduction

Classical planning is a subfield of artificial intelligence studies. The goal of planning is to find a sequence of actions that reaches a specified goal from some initial state. There are two approaches to this. There is optimal planning, which aims to find the optimal sequence of actions, and there is satisfying planning, which is interested in finding any valid solution as quickly as possible.

Search algorithms are used to explore the state space of a given problem and find the sequences of actions that, when applied, produce a goal state. In general, however, the state space of so-called planning problems can grow exponential in the size of the problem description. For a problem with a larger state space, the search algorithm requires more time and memory to find such a sequence. There are two main approaches to improve the performance of the search. The first approach is to create more sophisticated search algorithms, and the second is to improve the performance of heuristics. Modern algorithms use heuristics to guide them to be as efficient as possible. Heuristics estimate the distance of a certain path from the goal. With heuristics, a search algorithm can make estimations about which paths are more promising, hopefully allowing us to find a solution while having to explore as little of the state space as possible.

Another approach is to simplify the problem. Whenever a problem is described, its complexity can be grouped into two classes: *essential complexity* and *accidental complexity*[1]. Essential complexity is the complexity inherent in the problem. This complexity can not be simplified without changing the semantics of the problem. Accidental complexity is the complexity added to a problem when its encoding or form is ill-suited for the tool we want to use. This complexity can be reduced by reformulating the problem into another form more suited for the approach used to solve it.

Reformulating a problem to reduce its accidental complexity without changing the semantics of the problem is not an easy task. One approach introduced by Helmert [8] and later

[1] The terms essential complexity and accidental complexity were first introduced by Brooks [3].

expanded by Haslum [7] is *safe abstraction.* Safe abstraction simplifies a problem in such a way that a solution in the simplified problem can be efficiently expanded into a solution for the unmodified problem. Safe abstraction reduces the size of the state space the search algorithms have to search through and in some cases fully solve problems, without needing to search the state space at all. Safe abstraction guarantees that a solution of a simplified problem can be extended to be valid in the concrete problem, it doesn't guarantee that these solutions are optimal. Because of this, safe abstractions are only useful in satisfying planning.

In Chapter 2, we will first introduce the background knowledge needed for the paper. Next, in Chapter 3, we will present the theory of safe abstraction and then, in Chapter 4, discuss our implementation of safe abstraction in the Fast Downward planner. In Chapter 5 we test our implementation over various problem domains and finally, in Chapter 6 we will summarize the main points and discuss open questions.

2 Background

2.1 Planning Tasks

Planning involves finding a sequence of actions that transitions a system from a given initial state to a desired goal. These problems are referred to as *planning tasks*, and the sequences of actions that solve them are called *plans*.
There are many ways to describe a planning task formally. The Fast Downward planner introduced by Helmert [9] uses a *finite domain representation* formalism called SAS^+ proposed by Bäckström and Nebel [2]. The following definitions are based off their work.

Definition 1. *(SAS^+ planning task) A SAS^+ planning task is defined by a four tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ where $\mathcal{V}$ is a finite set of variables, $\mathcal{O}$ is a finite set of operators, s_0 is the initial state and s_* is the goal.*

Planning tasks are sometimes also called *planning problems* or just *problems.*
One simple SAS^+ planning task used by Haslum [7][2] is a simple transportation problem we will call "trucks". It has two locations, A and B, a truck that can drive between them and a package p. The package starts in location A and needs to be transported to B by loading and unloading it from the truck. The variables and operators of the problem are listed in Fig. 2.1. We will return to this example throughout the paper.

Definition 2. *(Variable) A variable $v \in \mathcal{V}$ has a finite domain D_v of values $d \in D_v$. A variable can be assigned one value from its domain $v \mapsto d$.*

When a variable is assigned a value, we call it a value assignment or atom $v \mapsto d$.
In the trucks example, we have the following variables *truck* with domain $D_{truck} = \{at\text{-}A, at\text{-}B\}$, p with domain $D_p \mapsto \{at\text{-}A, at\text{-}B,\ in\text{-}truck\}$ and *cargo* with domain $D_{cargo} \mapsto \{empty,$ $contains\text{-}p\}$.

[2] Their example has two packages: $pkg1$ and $pkg2$, but is otherwise the same.

```
Variables:
    truck
        Atom at-A
        Atom at-B
    p
        Atom at-A
        Atom at-B
        Atom in-truck
    cargo
        Atom empty
        Atom contains-p

Operators:
    drive-truck-a-b
        pre:  truck := at-A
        eff:  truck := at-B
    drive-truck-b-a
        pre:  truck := at-B
        eff:  truck := at-A
    drop-truck-a-p
        pre:  truck := at-A, p := in-truck, cargo := contains-p
        eff:  p := at-A, cargo := empty
    drop-truck-b-p
        pre:  truck := at-B, p := in-truck, cargo := contains-p
        eff:  p := at-B, cargo := empty
    pick-up-truck-a-p
        pre:  truck := at-A, p := at-A, cargo := empty
        eff:  p := in-truck, cargo := contains-p
    pick-up-truck-b-p
        pre:  truck := at-B, p := at-B, cargo := empty
        eff:  p := in-truck, cargo := contains-p
```

Figure 2.1: The variables and operators of the trucks problem

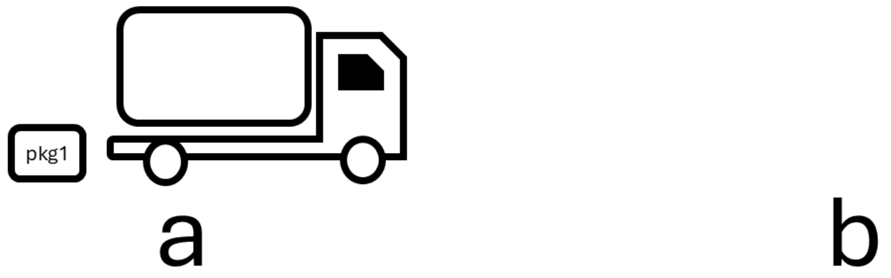

Figure 2.2: The trucks problem in its initial state

Definition 3. *(State) A state s is a value assignment over all variables $s = \{v \mapsto d \mid v \in \mathcal{V}, d \in D_v\}$.*

A partial state p is only defined over a subset of all variables $\mathcal{V}$. We denote the set of variables over which p is defined as $\mathcal{V}_p$.
We say partial state p *holds* in or is *consistent* with s if each variable $v_p \in \mathcal{V}_p$ is assigned the same value as in state s.
The set of all possible states is called the state space of the planning task.
For two partial states p and q, we say they are disjoint if no variable defined in p is defined in q and vice versa: $\mathcal{V}_p \cap \mathcal{V}_q = \emptyset$.

In the trucks example, $s = \{truck \mapsto at\text{-}A,\ p \mapsto at\text{-}A,\ cargo \mapsto empty\}$ is a valid state shown in Fig. 2.2, and $p = \{truck \mapsto at\text{-}A\}$ is a valid partial state that holds in s.

Definition 4. *(Initial State) The initial state s_0 is the starting state of the planning task before any operators are applied.*

The initial state of the trucks problem is the state $s_0 = \{truck \mapsto at\text{-}A, p \mapsto at - A, cargo \mapsto empty(truck)\}$ as seen in Fig. 2.2

Definition 5. *(Operator) An operator $o \in \mathcal{O}$ is a tuple, $\langle p, e\rangle$ where p are its preconditions and e are its effects. Both p and e are partial states.*

We say that operator o is applicable in state s if the preconditions of o hold in s
Applying o to s yields a successor state $s[\![o]\!] = s'$. If a variable v is defined in the effects e of o, then it will have that value assignment in s'. For all other variables, s' will have the same value assignment as s.
Applying a sequence O of operators $\langle o_1, o_2, ...\rangle$ to s will apply each operator o_i in sequence: $s[\![O]\!] = ((s[\![o_1]\!])[\![o_2]\!])...$

Operators have an associated cost, and the cost of an operator sequence is equal to the sum of operator costs. Since we are not concerned with finding an optimal solution to a planning task, we assume the cost of each operator to be one.

In the trucks example, one such operator is *drive-truck-a-b* with preconditions $p = \{truck \mapsto at\text{-}A\}$ and effects $e = \{truck \mapsto at\text{-}B\}$. Applying this operator to the state shown in Fig. 2.2 will yield a new state as shown in Fig. 2.3.

Figure 2.3: The effect of applying the operator *drive-truck-a-b* to the initial state of the trucks problem.

Definition 6. *(Goal) The goal s_* is a partial state. If we reach a state s where s_* holds from s_0, then the planning task is solved successfully.*

The only goal condition in the trucks problem is $s_* = \{p = at\text{-}B\}$. All states that fulfil this condition are goal states.

Definition 7. *(Plan) A plan π is a sequence of operators $\langle o_1, o_2, ..., o_n \rangle$ that solves the planning task. If the plan is valid, the state reached after applying the sequence to the initial state $s_0[\![\pi]\!]$ is consistent with the goal s_*.*

In our running example, a valid plan would be $\langle$*pick-up-truck-a-pkg*1, *drive-truck-a-b*, *drop-truck-b-pkg*1$\rangle$. In this case this would be the optimal path but as already mentioned in the introduction, in this paper we do not aim to generate optimal plans. Instead, we are interested in finding any plan that solves the planning task, so another longer path that results in a goal state would also be valid.

With the definition of a plan, we have all definitions needed to describe a SAS^+ planning task. Next we will look at the domain transition graphs of variables. They are useful to find out which values of a variable are reachable from a given value. The following definition of a domain transition graph is based on a definition of DTGs from Huang et al. [10].

Definition 8. *(Domain Transition Graph (DTG)) Given a variable, $v \in \mathcal{V}$, it's DTG is a directed graph DTG_v with a vertex set $\mathcal{N}_v$ and an arc set $\mathcal{A}_v$. For each value, d in the domain D_v of variable v, there is a vertex n_d in $\mathcal{N}_v$. An arc $a = (n_i, n_j)$ belongs to $\mathcal{N}_v$ only if there is an operator with precondition $v = i$ and effect $v = j$*

The domain transition graph for p is shown in Fig. 2.4. In later chapters, we will use the domain transition graph of a variable to determine which variables can be safely abstracted.

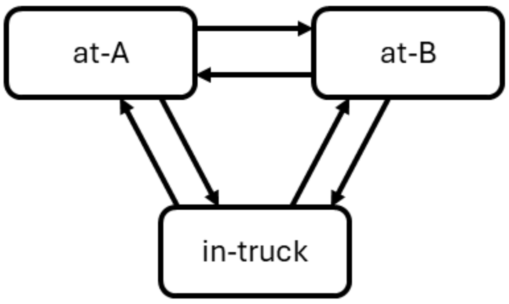

Figure 2.4: The domain transition graph of variable p of the trucks problem.

2.2 Abstractions

Because the state space of a planning task grows exponential in the size of the problem description[2], the number of states can quickly grow out of control. Since the search for a plan has to operate on the state space, we should try to reduce the size of the state space as much as possible. The idea of abstractions is to reduce the size of the state space. One possible use case for abstraction is the calculation of heuristics. Edelkamp [4] shows how abstractions can be used to greatly simplify a problem and solve it much quicker. The solutions in this abstracted problem are generally not solutions in the concrete problem, but can be used as a heuristic to guide the search in the concrete problem.

Abstractions are defined on the state space of a planning task. The transition system is a graph that represents this state space.

Definition 9. *(Transition Systems) A planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ *induces a transition system* $\mathcal{T}(\Pi) = \langle S, L, c, T, s_0, S_* \rangle$ *where* S *is the set of all states over variables* $\mathcal{V}$*,* L *is the set of operators* $\mathcal{O}$*,* $c(o)$ *is the cost of operators* $o \in \mathcal{O}$*,* T *is the set of transitions defined by tuples* $\langle s, o, s' \rangle$ *where operator* o *is applicable in state* s *and yields the state* s'*,* s_0 *is the initial state and* S_* *is the set of all states that are consistent with the goal* s_**.*

A planning task's transition system can be represented as a graph, where each vertex corresponds to a state of the planning task, and the arcs from one vertex to another represent all possible state transitions achievable by applying an operator to the state represented by the source vertex.

To reduce the number of states we have to consider while creating our plan, we want to create a function α that remaps the set of states S of $\mathcal{T}$ to another, usually smaller, set of states. There are many such mappings, but a common approach is to ignore some subset of the variables or combining groups of states into a single state. With the vertices of new abstract state, inheriting all the ingoing and outgoing arcs of its component state's vertices.

Definition 10. *(Abstractions) Given a transition system* $\mathcal{T}$*, an abstraction is a function* $\alpha : S \to S^\alpha$ *which maps the set of states* S *of the transition system to another set of states* S^α*.*

When a transition system $\mathcal{T}$ is abstracted with α, written as $\alpha(\mathcal{T})$, a new abstract transition system is created $\alpha(\mathcal{T}) = \langle S^\alpha, L, c, T^\alpha, s_0^\alpha, S_*^\alpha \rangle$ where S^α is the new set of states defined by α, L and c are unchanged, T^α is the set of transitions over the new states: $T^\alpha = \{\langle \alpha(s), o, \alpha(s') \rangle \mid \langle s, o, s' \rangle \in T\}$ and $s_0^\alpha = \alpha(s_0)$ is the new initial state and $S_*^\alpha = \{\alpha(s) \mid s \in S_*\}$ the new goal states.

We can always revert the changed of an abstraction α by applying its inverse α^{-1} to the abstracted transition system: $\alpha^{-1}(\alpha(\mathcal{T})) = \mathcal{T}$. Note that in general, abstractions are not commutative, so the transition system resulting from $\beta(\alpha(\mathcal{T}))$ is not necessarily the same as when applying $\alpha(\beta(\mathcal{T}))$. In the same way, applying the inverse of α on a transition system where α is not the most recent abstraction will not necessarily undo the changes of α: $\alpha^{-1}(\beta(\alpha(\mathcal{T}))) \neq \beta(\mathcal{T})$.

We can see this clearly by imagining a transition system with three states that describe the colour of a box. The box can be blue, red, or green, but the colours can only be changed by first colouring the box red. Now we construct two abstractions: $\alpha =$"Group the state where the box is red and the state where the box is blue together" and $\beta =$"Group the state where the box is red and the state where the box is green together". See Fig. 2.5 for an illustration of the transition system and the abstractions. If our original transition system is $\mathcal{T}$, then by first applying α, we group the states where the box is red and blue together, creating $\alpha(\mathcal{T})$. This new transition system has only two remaining states: one for the state where the box is blue or red and one for the state where the box is green. Since neither is the state where the box is red, we can not apply β to $\alpha(\mathcal{T})$. If we instead apply β to $\mathcal{T}$ first, we can not apply α since, once again, the state where the box is red no longer exists. Since applying an abstraction can make the application of another abstraction impossible, the order in which they are applied matters.

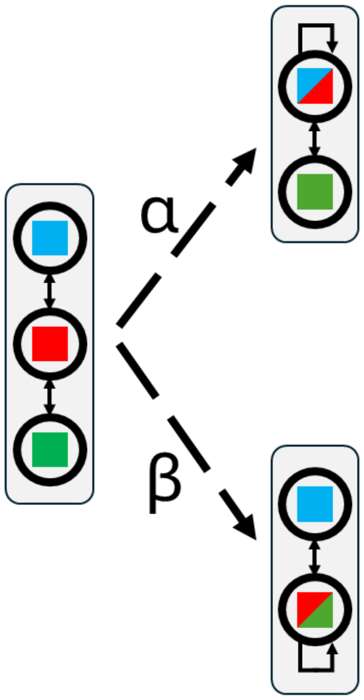

Figure 2.5: Effect of applying abstractions α and β.

One useful abstraction used by Edelkamp [4] is described by the projection. A projection is a function p_V that is applied to a planning task Π. Given some subset of the variables $V \subseteq \mathcal{V}$, the projection removes the variables $\mathcal{V} \setminus V$ from the planning task $p_V(\Pi) = \langle \mathcal{V}^{p_V}, \mathcal{O}^{p_V}, s_0^{p_V}, s_*^{p_V} \rangle$. The new variable set $\mathcal{V}^{p_V}$ contains only variables that are in V. The new operator set $\mathcal{O}^{p_V}$ contains all the operators from $\mathcal{O}$ of the Π, but all preconditions and effects over variables not in V have been removed. The new initial state $s_0^{p_V}$ is the same as, s_0 except that all variable assignments over variables not in V are removed. Similarly, the goal is the same, except that all requirements over variables not in V are removed.

If $\mathcal{T}$ is the transition system induced by a planning task Π and p_V is a projection that when applied to Π creates a new planning task $p_V(\Pi)$ which induces a transition system $\mathcal{T}^{\sim}$, then

p_V describes an abstraction α such that $\alpha(\mathcal{T}) = \mathcal{T}^{\sim}$.

2.2.1 Abstraction Hierarchy

It is possible that a transition system is not just abstracted one time. Since abstractions are not generally commutative, the order in which they are applied matters. When multiple abstractions are applied to a transition system $\mathcal{T}$, their sequence builds an abstraction hierarchy $\mathcal{H} = \langle \mathcal{T}, \alpha(\mathcal{T}), \beta(\alpha(\mathcal{T})), \dots, \gamma(\dots\beta(\alpha(\mathcal{T}))\dots) \rangle$. This hierarchy begins with the concrete, non-abstracted, transition system $\mathcal{T}$ at the bottom and each higher step is a more abstracted version of the transition system in the next lower step. We call this sequence of abstractions $\mathcal{Q} = \langle \alpha, \beta, \dots, \gamma \rangle$.

3 Safe Abstraction

While abstractions are a useful tool, in general they come with significant drawbacks. As described in Haslum [7], because some details are ignored, solutions in an abstract problem may be computed cheaper, but they are generally not valid in a more concrete version of the problem. These plans are often used as the basis for heuristics. However, with *safe abstractions*, we can guarantee that such plans can be efficiently refined into valid solutions for the concrete version of the problem. An abstraction is *safe* if in each abstraction step, we can guarantee that a valid solution in the more abstract problem can be efficiently refined into a valid solution in the more concrete version of the problem.

Haslum [7] introduces two theorems regarding safe abstraction: (safe) variable abstraction and operator composition. Both are functions that manipulate a planning task. We call such functions *simplifications*. Variable abstraction is a projection as described in Section 2.2. It reduces the size of the state space of a problem by removing variables from it. Despite its name, variable abstractions are not actual abstractions. They are projections that describe abstractions. This is an important nuance because abstractions as defined in Definition 10 can not be applied to planning tasks. In contrast, operator composition is a function that does not reduce the state space of the problem, instead it adds and removes operators from the planning task, hopefully enabling more safe variable abstraction in later steps. We call safe variable abstraction α_s and operator composition β_c.

To organise these simplifications, will loosen our definition of abstraction hierarchies from Section 2.2.1. A *simplification hierarchy* is a tuple of planning tasks, $\mathcal{S} = \langle \Pi, \sigma_1(\Pi), \sigma_2(\sigma_1(\Pi)), ...\sigma_n(...\sigma_2(\sigma_1(\Pi))...) \rangle$ where Π is the concrete planning task and σ_i are simplifications. We call the sequence of simplifications $\mathcal{Q}^{\sim} = \langle \sigma_1, \sigma_2, ..., \sigma_n \rangle$ the simplification sequence.

The idea is to repeatedly apply α_s and β_c in any order, until no further simplifications can be made. Their order is our simplification sequence $\mathcal{Q}^{\sim}$ and the problems they create, build our simplification hierarchy $\mathcal{S}$. If the sequence $\mathcal{Q}^{\sim}$ only consists of variable abstractions, $\mathcal{Q}^{\sim} = \langle \alpha_{s1}, \alpha_{s2}, ... \rangle$ then the abstractions they describe form an abstraction hierarchy.

Once we finished applying simplifications, we initiate a search on the most simplified version of our problem. If a plan was found, we undo the simplifications by going through $\mathcal{Q}^{\sim}$ in reverse and applying the inverse of each simplification. This is called *refinement*. As we do so, we also refine our plan and once we have arrived back at the concrete problem, the plan is a valid plan for that problem too.

3.1 Safe Variable Abstraction

Safe variable abstraction revolves around the identification and removal of "safe variables". If only such safe variables are removed in a simplification step, we have the guarantee that a plan in the more simplified problem can be efficiently extended to a valid plan in the more refined problem. Removing a variable this way simplifies a problem by reducing the amount of states in its state space. In the following chapters, when we say we abstract a variable, we mean that save variable abstraction is applied.

We can safely remove a variable v if it can, independently of other variables, take those values required to change other variables, from values it was assigned to in the initial state or by operators that changed other variables. The free domain transition graph, a subgraph of a variable's domain transition graph, allows us to determine which values of v can be reached without changing or requiring anything from other variables.

Definition 11. *(Free Domain Transition Graph (free DTG)) The free DTG of a variable v is a subgraph of the variable's DTG. The vertex set of the free DTG is the same as the DTG. An arc $a = (n_i, n_j)$ in the arc set of the DTG only belongs in the arc set of the free DTG if there is an operator whose preconditions only contains $v \mapsto i$ and whose effects only contains $v \mapsto j$.*

In other words, an arc from the DTG is included in the free DTG only if there is an operator that does not require or affect another variable as part of its precondition or effect. A value d_j is said to be *free reachable* from a value d_i if there is a path from d_i to d_j in the free DTG. We can compare the free DTG of p shown in Fig. 3.1 against its DTG from Fig. 2.4. We see that though each of the values are reachable from the others in general, all these transitions require operators who contain other variables in their preconditions or effects.

For the variable *truck* we see that its free DTG, shown in Fig. 3.2 is strongly connected since the *drive-truck-a-b* and *drive-truck-b-a* have no other variables in their preconditions or effects.

Definition 12. *(Externally Required) A value d_j of a variable v is externally required if it appears in the preconditions of an operator whose effects include changes over other variables than v.*

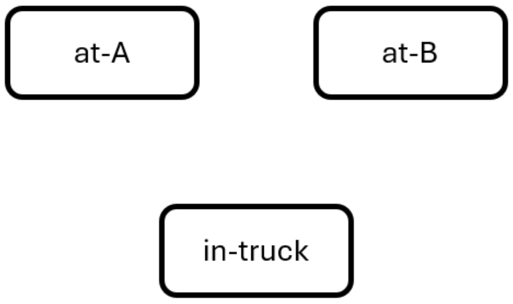

Figure 3.1: The free domain transition graph of p.

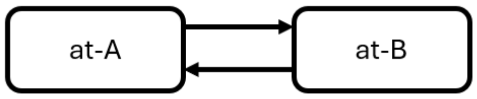

Figure 3.2: The free domain transition graph of *truck*.

Definition 13. *(Externally Caused) A value d_j of a variable v is externally caused if it either is the value of v in the initial state or if it appears in the effects of an operator that also includes changes over other variables than v in its effects.*

With this information, we can now apply the safe variable theorem introduced in Haslum [7]:

Theorem 1. *(Safe Variables) If all externally required values of v are strongly connected in the 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, then abstracting v is safe (i.e., preserves downward refinement[3])*

We can see why these conditions need to be met by considering only a variable v and its free DTG. We do not know what operators are being applied in the problem overall. All we know is what value the variable is and what value the variable should be, and we can only apply operators that do not require a value of another variable, or cause a change in another variable. When we look at such a case, our variable may be assigned any value that is marked as externally caused, either because it is assigned that value in the initial state or because some operator o_c may be needed to change the value of some other variable and as a side effect it changes the value of our variable. Our variable may be required to take any value that is marked as externally required, since those values may be needed to apply some operator o_r that changes another variable. Similarly, our variable may be required to take its goal value if it has one to complete the problem. Finally, we can only transition on arcs in the free DTG, since those represent the operators that do not change or require a value from any other variable.

[3] If the *downward refinement* property[1] holds, a solution to a more abstract problem can be extended to a solution for the next lower level in the abstraction hierarchy just by inserting missing operators into the plan.

We can safely abstract a variable if the operators that are required to change its values to the externally required ones do not interfere with other variables. Imagine we have two kinds of operators o and o^s. Assume the operators with the superscript s are operators that do not require nor change any other variable than some safe variable v^s. They are represented by the arcs on the safe variable's free DTG. If we know that we can cause v^s to have any externally required value (or the goal value) from any externally required or caused value by applying such operators o^s, then the operators o^s in the plan only enable the applicability of other operators o by assigning v^s to a required value or the reachability of a goal state by assigning v^s its goal value. If we remove v^s from the goal condition and ignored the preconditions and effects over v^s for all operators o , then we could also remove all operators o^s from the plan since they no longer contribute to the plan's validity. We can then undo this by reinstating v^s to the goal condition and to all preconditions and effects from which it was removed. Now the plan may not be valid since an operator o_i may have as part of its effects $v^s \mapsto a$ and the next operator in the plan o_{i+1} may have as part of its preconditions $v^s \mapsto b$. However, we know there is a sequence of o^s operators that changes the assignment of v^s from a to b without changing or requiring another variable. By inserting such sequences between all operators o_i and o_j with such a conflict, and at the end to assign v^s its goal value if it has one, we can restore the plan to full validity without fundamentally changing it. Since we can transform v^s from any externally caused value to any externally required value by just inserting sequences of operators o^s, this validity restoration is possible for all valid plans. Therefore, we can remove the variable v^s from the problem entirely before we start the search, and any plan found during the search can be extended to include v^s.

The search for these sequences of o^s can be performed in the free DTG of v which grows polynomial in the size of the problem description. Compared to that size of the state space, which can grow exponential in the size of the problem description, it is easy to see that inserting a sequence o^s into a plan is more efficient than having to search the full state space for it.

Since a safe variable is independent, meaning it doesn't require other variables to take on its externally required values, if we find multiple safe variables in the same problem, they can easily be removed by repeating the procedure above for each safe variable in any order. Removing all safe variable V_s of a problem is a projection on $\mathcal{V} \setminus V_s$ as described in Section 2.2

For an example, let's think back to the trucks problem. If we apply Theorem 1 to the problem shown in Fig. 2.1, we will find that the variable *truck* is a safe variable.
It has only two values; the truck is either in location A or in location B. Both values are externally required from operators like *drop-truck-a-p* and *drop-truck-b-p* but neither value is externally caused, since the only operators that change the location are the drive operators, both of which only have *truck* in its effects. Since the drive operators does not contain another variable than *truck* in its preconditions or effects, they both form an arc on the free DTG of *truck* (See Fig. 3.2). Since there is neither a goal value for the variable nor any externally caused values, and the externally required values are strongly connected, we can abstract it by applying a variable abstraction α_{truck} that removes the variable from

the problem. The package can now be loaded in one location and immediately unloaded in another without having to apply the drive operator in-between. We can imagine this by thinking of the truck being in both locations at once as shown in Fig. 3.4. The variables and operators of the simplified trucks problem are shown in Fig. 3.3.

```
Variables
    p
        Atom at-A
        Atom at-B
        in-truck
    cargo
        Atom empty
        Atom contains-p

Operators
    drop-truck-a-p
        pre:  p := in-truck, cargo := contains-p
        eff:  p := at-A, cargo := empty
    drop-truck-b-p
        pre:  p := in-truck, cargo := contains-p
        eff:  p := Atom at-B, cargo := empty
    pick-up-truck-a-p
        pre:  p := at-A, cargo := empty
        eff:  p := in-truck, cargo  := contains-p
    pick-up-truck-b-p
        pre:  p := Atom at-B, cargo := empty
        eff:  p := in-truck, cargo  := contains-p
```

Figure 3.3: The variables and operators of the trucks problem after the location of the truck has been abstracted away.

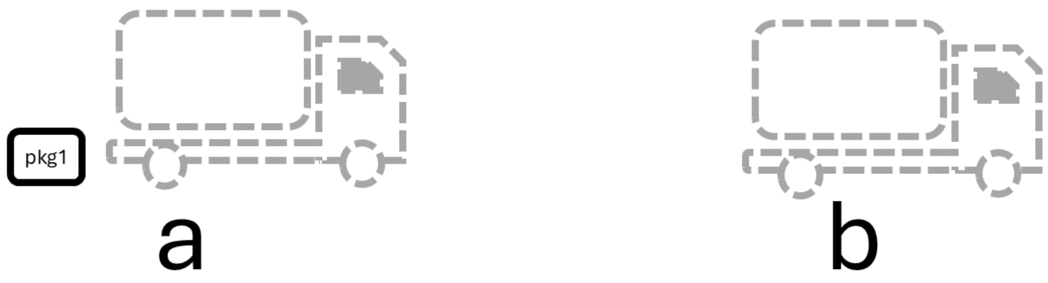

Figure 3.4: The initial state of the trucks problem after the location of the truck was abstracted away.

3.2 Operator Composition

Operator Composition combines sequences of operators into one composite operator. Operator Composition itself does not necessarily simplify a problem, but may causally decouple variables, allowing them to be abstracted with variable abstraction.

Remember the trucks example. We have already removed the variable *truck* with safe variable abstraction. If we tried to immediately run the safe variable abstraction again, we would not find any more safe variables because the remaining operators change both p and *cargo*.

However, the states where the package is in the truck are not really "interesting states". Our goal is to transport the package from one place to another, not to leave it in the truck. So in the current problem where the location of the truck has been abstracted away, every time we load the package into the truck, the next action is always going to be unloading it from the truck. We say the variable *cargo* is *transient*. Since we really only care about loading a package somewhere and unloading it somewhere, we don't care what happens to the package in-between (remember, the location of the truck has been abstracted). We can see that for all possible pairs of "pick-up" and "drop" operators, the *cargo* variable starts and ends with *empty*. The only variable that changes is p. If we were to "hide" the intermediate states, we could abstract *cargo* away. This can be done via operator composition, replacing all the "pick-up" and "drop" operators by composite operators of the form "pick-up $\rightarrow$ drop".

To find such chains of operators we can composite without affecting solvability, we used a theorem also introduced in Haslum [7]:

Theorem 2. *(Safe Sequencing Theorem) Let c be a condition on two or more variables, let A be the set of all actions whose effect includes c, and let B be the set of actions whose precondition includes c. If c does not hold in the initial state; c is either inconsistent with or disjoint from the effects of every action not in A; the postcondition of every action in A is either inconsistent with or disjoint from the goal condition; and every action not in B whose precondition is consistent with c is commutative with every action in $A \cup B$, then replacing the actions in $A \cup B$ with one composite action for each executable sequence $a, b_1, ..., b_k$, where $k \geq 1, a \in A$ and each $b_i \in B$ – excluding sequences with no effect — is safe (i.e., preserves solution existence).*

Since Theorem 2 is more complex than Theorem 1, we can not explain it in the same depth but if we look at each of the four conditions in the theorem separately, we can see what case each condition covers. The conditions are:

1. "c does not hold in the initial state"
2. "c is inconsistent with or disjoint from the effects of every action, not in A"
3. "The postcondition of every action in A is either inconsistent or disjoint from the goal condition"
4. "Every action not in B whose precondition is consistent with c is commutative with every action in $A \cup B$"

Condition 1 is important to preserve the ability to leave the initial state. Since c (partially) describes the states between the application of the operators in A and B, if we allow such

a c, we could end up in a situation where the only operator that leaves the initial state is replaced by a composite operator.
Say we have a solvable planning task where the initial state is not a goal state. Let B only consist of one operator b and it is the only operator whose preconditions hold in the initial state and A is not empty. Assuming the other conditions hold, but we ignore this condition, following the rest of the theorem, we would replace the operators in A and operator b with composite operators $\langle a_i, b\rangle$. Since the preconditions of operators a_i do not hold in the initial state, the precondition of composite operators $\langle a_i, b\rangle$ also do not hold in the initial state. In this case, there is no operator that allows us to leave the initial state and the solution's existence would not be preserved.

Condition 2 is important to preserve the ability to apply all valid operator sequences. If we do not check for this condition, there could be sequences of operators o_1, o_2 which are needed to reach a goal but no longer exist after composition.
Say we have a solvable planning task. Let o be an operator whose effects are consistent and not disjoint from c but is not in A. Further, o is the only operator that changes the value assignment of a variable v_1 ($v_1 \mapsto 0$ in initial state) to be $v_1 \mapsto 1$. Let b be an operator in B and the only operator that changes the value assignment of a variable v_2 ($v_2 \mapsto 0$ in initial state) to be $v_2 \mapsto 1$. Let A be not empty and S_A be the states in which the operators in A are applicable. Let none of the states in S_A be reachable from the states yielded by o. Finally, there is an operator g which takes as its preconditions $v_1 \mapsto 1$ and $v_2 \mapsto 1$ and is the only operator to yield a goal state. Assuming all other conditions hold, but we ignore this condition, we would replace the operators in A and operator b with composite operators $\langle a_i, b\rangle$. Since the states S_A are not reachable by the sates yielded by o, none of the composite operators $\langle a_i, b\rangle$ can ever be applied after applying o. Since we can not establish both $v_1 \mapsto 1$ and $v_2 \mapsto 1$ we can not apply g, meaning it is now impossible to reach a goal state and the solution's existence is not preserved.

Condition 3 is important to preserve the ability to set variables to their goal value. It could be the case that an operator in a is the only operator that sets a variable's value to its goal value, and if we replace it, we might not be able to reach a goal state anymore.
Say we have a solvable planning task. Let v be a variable and its goal value is $v \mapsto g$. Let A consist of just one operator a. a is the only operator that assigns $v \mapsto g$ in its effects. Let B be filled with operators whose effects contain the assignment $v \mapsto 0$. Assuming all other conditions hold, but we ignore this condition, we would replace a and the operators in B with composite operators $\langle a, b_i\rangle$. Since all operators in B assign $v \mapsto 0$ as part of their effects, all composite operators $\langle a, b_i\rangle$ also assign $v \mapsto 0$ in their effects. In this case, there is no operator that allows us to establish the assignment $v \mapsto g$, making it impossible to reach a goal state and the solution's existence is not preserved.

Conditions 4 is important to preserve the ability to use all operators not in B. It could be the case that an operator o not in B is only applicable in the state between the application of an operator in A and an operator of B. If we replace them with composite, operators o

would no longer be applicable in any reachable state.
Say we have a solvable planning task. o is an operator whose preconditions are consistent with c but o is not in B. Further, the only reachable state from the initial state in which o is applicable is s and the effects of o yield the only goal state s_g. s is the state in which c holds and all operators that yield it are in A. B is not empty. Assuming all other conditions hold, but we ignore this condition, we would replace the operators in A and the operators in B with composite operators $\langle a_i, b_i \rangle$. In this case, it is no longer possible to reach state s and since o is only applicable in s and is the only operator that yields the goal state s_g, we can no longer reach the goal state. The solution's existence is not preserved in this case.

Given some set of composite operators O_c and the sets of operators A and B then β is the simplification that adds an operator o to $\mathcal{O}^\beta$ only if o is in O_c or if it is in $\mathcal{O}$ but not in A or B. In other words, applying β to a planning task removes all the operators in $A \cup B$ from the planning task and adds the composite operators from O_c to the planning task. Applying β to a planning task Π creates a more simplified planning task $\beta(\Pi) = \langle \mathcal{V}, \mathcal{O}^\beta, s_0, s_* \rangle$.

For an example, let's carry on with the abstracted trucks problem shown in Fig. 3.3. If we apply Theorem 2 to the problem with a $c = \langle p \mapsto \textit{in-truck}, \textit{cargo} \mapsto \textit{contains-p} \rangle$, we will get the sets $A = \{\textit{pick-up-truck-a-p}, \textit{pick-up-truck-b-p}\}$ and $B = \{\textit{drop-truck-a-p}, \textit{drop-truck-b-p}\}$. Condition 1 holds, since c does not hold in the initial state. Operators *drop-truck-a-p*, *drop-truck-b-p* are not in A and their effects are inconsistent with c because they assign $p \mapsto \textit{at-A}$ and $p \mapsto \textit{at-B}$ respectively, so condition 2 also holds. Condition 3, holds because the effects of both *pick-up-truck-a-p* and *pick-up-truck-b-p* are inconsistent with the goal condition because they assign $p \mapsto \textit{in-truck}$. Finally, since none of the operators not in B have preconditions consistent with c, condition 4 holds in a trivial sense.
Since all conditions hold, we create the composite operators and replace the original operators in A and B. The new composite operators are listed in Fig. 3.5.

Theorem 2 only considers what replacements are safe, but not what replacements are useful, to further simplify the problem. If we remind ourselves of the trucks problem: operator composition was useful because there was a chain of operators that changed only one variable p permanently. The other variable $in(truck, pkg1)$ changes only in an intermediate, uninteresting state before being immediately reverted.
So a secondary condition is added. Consider a pair of variables v_1 and v_2. If for all possible c involving those two variables, the safe sequencing theorem holds and each of the new composite operators changes at most one of the two variables, then replacing the involved operators with these composite operators will remove the causal coupling between the two variables and potentially making them into safe variables to be abstracted. We call this condition the *decoupling condition.*

The simplification with the decoupling condition is similar to the simplification without considering the decoupling condition. Instead of the set of composite operators O_c being those composite operators generated from a specific c, it is $O_{\langle v_i, v_j \rangle}$, which contains all composite

```
Variables
    p
        Atom at-A
        Atom at-B
        in-truck
    cargo
        Atom empty
        Atom contains-p

Operators
    [pick-up-truck-a-p -> drop-truck-b-p]
        pre:  p := at-A, cargo := empty
        eff:  p := at-B, cargo := empty
    [pick-up-truck-b-p -> drop-truck-a-p]
        pre:  p := at-B, cargo := empty
        eff:  p := at-A, cargo := empty
```

Figure 3.5: The variables and operators of the trucks problem after the location of the truck has been abstracted away, and the remaining operators have been composited. The operators [*pick-up-truck-a-p* $\rightarrow$ *drop-truck-a-p*] and [*pick-up-truck-b-p* $\rightarrow$ *drop-truck-b-p*] are not included since they have no effect (preconditions and effects are identical).

operators generated from all valid c of a variable pair $\langle v_i, v_j \rangle$. And instead of replacing the operators in the set $A \cup B$ of a specific c, we similarly pool the operators of sets A and B from all valid c into a single set $O^{\sim}_{\langle v_i, v_j \rangle}$. With the decoupling condition, β is the simplification that adds an operator o to $\mathcal{O}^{\beta}$ only if o is in $O_{\langle v_i, v_j \rangle}$ or if it is in $\mathcal{O}$ but not in $O^{\sim}_{\langle v_i, v_j \rangle}$. Applying β to a planning task Π creates a more simplified planning task $\beta(\Pi) = \langle \mathcal{V}, \mathcal{O}^{\beta}, s_0, s_* \rangle$.

4 Implementation

We have implemented the findings of Haslum [7] in the Fast Downwards planner introduced by Helmert [9]. The planner consists of two phases: translation and the search. The translator transforms the planning tasks encoded in PDDL[6][5] and replaces propositional variables with multi-value variables where possible. During the search phase, Fast Downwards uses a given search algorithm to solve the problem. Safe Abstraction has to simplify the problem after translation but before the search is performed. It also has to refine the plan after the search. Fast Downward currently does not neatly support problem manipulation of this kind. This means that the integration of Safe Abstraction into Fast Downward is not clean.

We will build a simplification hierarchy by alternatively applying a safe variable abstraction α_s followed by an operator composition β_c. A single simplification step will transform some planning task Π into $\beta_c(\alpha_s(\Pi))$.
After we finished our simplification step, we will pass the simplified planning task $\beta_c(\alpha_s(\Pi))$ to the next simplification step and save the pair $\langle \alpha_s, \beta_c \rangle$. Once a plan is found, we can apply the inversion of the simplifications in reverse order to refine the task $\beta_c(\alpha_s(\Pi))$ into a more concrete task Π.

Three major components are used to perform this simplification and refinement. The *(variable) abstractor*[4], *compositor* and *refiner*. The abstractor uses Theorem 1 to find the currently safe variable in the problem, the compositor uses Theorem 2 to find a set of safely composable operators and the refiner will apply the inverse of the simplifications to refine the problem (and its plan) back to the concrete version.

The abstractor and the compositor are applied in a loop. In each step, we first pass the abstractor a planning task Π. The abstractor performs a safe variable abstraction α_s and returns a simplified task $\alpha_s(\Pi)$. This simplified task is then passed to the compositor, which applies operator composition and returns a further simplified task $\beta_c(\alpha_s(\Pi))$. We then use

[4] Note that, despite its name, the abstractor performs a projection which describes the save variable abstraction α_s. It should not be mistaken for abstractions in a more general sense.

$\beta_c(\alpha_s(\Pi))$ as our starting point in the next iteration of the loop. We exit the loop if, after finishing the last step, the problem was solved, meaning the set of variables $\mathcal{V}$ of Π is empty or if both the abstractor and the compositor made no further simplification the last time they were called. Once the loop has terminated, if the problem wasn't fully solved, the most simplified task Π_s is passed to the search component of Fast Downwards. If the search finds a solution, it will return a plan. That plan is then passed to the refiner. If the problem was solved by safe abstraction and no search was needed, an empty plan containing no operators is passed to the refiner instead. The refiner will extend the plan and, once finished, return a plan that is valid in the original, unmodified planning task.

Let us first have a look at the abstractor.

4.1 Abstractor

The abstractor takes as input only the planning task Π. It iterates through all operators in the problem and, using Definition 11 of the free domain transition graph, builds the free DTG for each variable. Simultaneously, the abstractor will identify and mark the externally required and externally caused values of the variables the using Definition 12 and Definition 13. Once this information has been collected, the abstractor will, for each variable, apply Theorem 1 and mark all safe variables. Then the abstractor removes all safe variables from Π as described in Section 3.1. As already mentioned in that section, the order in which safe variables which were found in the same step are processed does not matter.

The Theorem 1 introduces by Haslum [7] does not account for a specific edge case where there are no externally required values. In this edge case, a variable could be marked as safe even if it was not.

4.1.1 Uncovered edge case

The theorem states that if the externally required values are strongly connected, they are reachable from every externally cause value and that the goal value (if any) is free reachable from every externally required value, then the variable is safe. If there is at least one externally required value, this is sufficient, as we are guaranteed to reach the goal value from the externally caused values via the externally required values.
However, in the case where we have a goal value, and some externally caused values but no externally required values, the given theorem is not well defined. All the requirements are fulfilled in the sense that an empty graph (the graph of the externally required values) is strongly connected and since there are no externally required values, the requirements "all externally required values of V are (. . .) free reachable from every externally caused value" and "the goal value of V (if any) is free reachable from each externally required value" are true in a trivial sense.

In this case, the theorem would tell us that the variable is safely abstractable. This however is not necessarily true. Imagine the simple case of a variable v with domain $D_v = \{0, 1\}$. $v \mapsto 1$ the variable's goal value and $v \mapsto 0$ the variable value in the initial state. v has no externally required values. There is an operator that changes the value of v from 0 to 1, but it also changes some other variable. In this case, the free DTG of v would consist of two vertices with no arcs. So the goal value is not free reachable from the initial value. As discussed above, however, since v has no externally required values, the theorem would still mark it as a safe variable.

To cover this edge case, we inserted an additional check. If there are no externally required values, we instead require the goal value (if any) to be free reachable from all externally caused values.

Next, let us have a look at the compositor.

4.2 Compositor

The compositor takes as input the planning task Π. To begin with, it generates a variable pair $\langle v_i, v_j \rangle$ and from them generates all possible value assignment pairs $\langle v_i \mapsto d_k, v_j \mapsto d_l \rangle$ which do not hold in the initial state s_0 as is required by Condition 1 of Theorem 2.
Next, we use each of these assignment pairs as a $c_{\langle v_i, v_j \rangle}$ in Theorem 2 to generate the sets A and B for that c and check them for the conditions 2, 3 and 4 of Theorem 2. If any of the conditions do not hold for any $c_{\langle v_i, v_j \rangle}$ of the variable pair, we throw out any composite operators already generated and start over with the next variable pair.

If the safe sequencing condition holds for all c of the variable pair, we recursively build the operator sequences that will make up our composite operators. Each sequence begins with one operator from A, then, in each subsequent function call, the compositor adds one operator from B and checks if the new sequence is executable.

To determine if a sequence is executable, we iterate through the sequence with a simulated partial state s^{sim}. We initialize it with the preconditions of the first operator in the sequence o_0. Then, starting with o_0, we check if the preconditions of the current operator o_i are met in s^{sim}. For each precondition $v \mapsto d$, there are three cases: either the precondition is explicitly met, meaning v is assigned a value in s^{sim} and that value is d, it is implicitly met, meaning v is not assigned a value in s^{sim}, or it is not met, meaning v is assigned a different value than d in s^{sim}. If the precondition is explicitly or implicitly met, no further effort is required. If the precondition is not met, the sequence is not executable and we can stop the simulation. If all preconditions of o_i are met (explicitly or implicitly), we update s^{sim} with the effects of o_i and move on to the next operator o_{i+1}. If all operators in the sequence were checked and all preconditions either explicitly or implicitly met, then the sequence is executable.

If the new sequence is executable, a composite operator is generated from it and the procedure is continued by appending another operator from B to the sequence. If the new sequence isn't executable, the compositor will try to append another operator from B instead.

Once all executable and unique composite operator have been constructed, they are checked with the *decoupling condition* to determine whether the composition removes the causal coupling between the variables v_i and v_j. If they do, the compositor replaces all those operators which were used to create the composite operators with the composite operators, then terminates. Otherwise, the results are discarded, and the next variable pair is generated until either a pair can be found where their causal coupling is removed or until all variable pairs have been tested.

While Theorem 2 establishes which chains could be composited, it isn't entirely clear how these chains should be constructed. The most straightforward idea would be to create all possible combinations of chains from the actions in A and B. This approach fails, however, if there is a cycle of applicable actions within B, in which case there would be an infinite number of executable sequences.

4.2.1 Avoiding infinite loops

Imagine a set $A = \{a\}$ and a set $B = \{b_1, b_2\}$. The effects of a and b_2 establish a partial state s_1 and the effects of b_1 establishes a partial state s_2. b_1 is applicable in partial state s_1 and b_2 is applicable in both partial states.
Since there are cycles in B ($b_1 \leftrightarrow b_2$ and $b_2 \leftrightarrow b_2$), we can create an infinite number of executable sequences by repeatedly applying the cycles. Since we have to replace each executable sequence with a composite operator, the algorithm would never terminate.

To avoid this, whenever, we add a new operator to the end of an operator sequence, after we check if the sequence is executable, we check if the composite operator generated by the sequence is unique. For an operator to be unique, its preconditions or effects have to be different from each of the already created composite operator.

To find the preconditions and effects of a composite operator o_c generated by a sequence, we iterate through the sequence with a simulated partial state s^{sim}. We also keep track of a partial state, which will record the sequence's preconditions s^{pre}. We initialize both partial states with the preconditions of the first operator in the sequence o_0. Then, starting with o_0, we check the preconditions of the current operator o_i. Since we already know the sequence is executable, for each precondition $v \mapsto d$, there are two cases: either the precondition is explicitly met, meaning v is assigned a value in s^{sim} and that value is d or it is implicitly met, meaning v is not assigned a value in s^{sim}. If the precondition is explicitly met, no further effort is required. If the precondition is implicitly met, we record the precondition in s^{pre}. Next, we update s^{sim} with the effects of o_i and move on to the next operator o_{i+1}. Once the effects of the last operator have been applied to s^{sim}, it records the effects of o_c. Then, to determine if o_c is unique, its preconditions s^{pre} and effects s^{sim} are compared against all other composite operators already generated and if, for any other composite operator o_c both are identical, then the sequence is not unique.

If the new composite operator is not unique, it is discarded and the last operator of the sequence is replaced with another operator from B that wasn't tried yet. If it is unique, it is added to the list of composite operators and the recursion continues by appending an operator from B to the sequence.

4.2.2 Soft Composition

There is a lack of clarity in the description of the composition procedure used by Haslum [7]. Theorem 2 as described by them, talks about one specific condition c. If the safe sequencing conditions hold for that c and its sets of operators A and B, then we should replace the executable sequences $a, b_1, ..., b_K$ with composite operators.
However, the decoupling condition, that this replacement actually removes the causal coupling, talks about a set of such c, those generated from the variable pair $\langle v_1, v_2 \rangle$. If the safe sequencing condition holds for all c generated this way, and the composite operators remove the causal coupling due to simultaneous change between them, then all involved operators across all c are replaced by all composite operators in one step.
Unfortunately, with the assumptions we made as described in Section 4.2, we were unable to reproduce the composition of the trucks problem as described in their paper.

To see why, let's go back to the trucks example, from Fig. 3.3 where the location of the truck has just been abstracted. Now, if we want to perform a composition, the only choice we have for a variable pair is $\langle p, cargo \rangle$. As described in, Section 4.2 this is what we want to remove the causal coupling between them. There are five[5] valid c we can build from this. The one c we are really interested in is $\langle p \mapsto \textit{in-truck}, cargo \mapsto \textit{contains-p} \rangle$ since that will put the *pick-up* operators into A and the *drop* operators into B. Luckily, the safe sequencing condition holds in this case. However, we can only perform the composition if it holds for *all c*. However, the c where $\langle p \mapsto \textit{at-B}, cargo \mapsto \textit{empty} \rangle$ also needs to be considered. In this case, A contains only the operator *drop-truck-b-p* and B the *pick-up* operators. However, the effects of *drop-truck-b-p* are not disjoint nor inconsistent with the goal condition, since $p = \textit{at-B}$ is a goal condition. This violates the condition 3 of Theorem 2. In this case, we have to throw out the composite operators computed so far, and since there are no further variable pairs, the compositor terminates without replacing any operators. Since

[5] $\langle p \mapsto \textit{at-A}, cargo \mapsto \textit{empty} \rangle$ is not a valid c since it holds in the initial state

the abstractor can not find any more safe variables in this state either, safe abstraction ends.

Since we do not have access to their encodings, we have tried to reconstruct the trucks example from Haslum [7] as best as we can, but can not guarantee that our encoding is identical with theirs. In another encoding, it is possible that composition would happen without further change.

In response to this and in hopes of reproducing the composition illustrated in Haslum [7], we implemented an option for the compositor to be "soft". When the compositor is soft, if a c violates the safe sequencing condition, instead of discarding the entire variable pair, only that specific c is rejected. Meaning that no composite operators are generated using the sets of operators A and B associated with that c. In the case above, $c = \langle p \mapsto \textit{in-truck}, cargo \mapsto \textit{contains-p}\rangle$ fulfils all the conditions and is passed on, whereas $c = \langle p \mapsto \textit{at-B}, cargo \mapsto \textit{empty}\rangle$ does not and is rejected. After this step, if at least one c was passed on, the operator sequences are generated and checked as described in Section 4.2 and, if they remove causal coupling due to simultaneous change, the compositor replaces all operators from the $A \cup B$ associated with the c that were passed on, with the set of composite operators o_c. Otherwise, we move on to the next variable pair.

With soft composition, we create new *pick-up* $\rightarrow$ *drop* composite operators, replacing the current operators and the variables p and $cargo$ become safe for abstraction. The final output of soft composition is the same as shown in Fig. 3.5.

Finally, let us have a look at the refiner.

4.3 Refiner

The refiner is called after either the problem was fully solved by safe abstraction or a plan was found by a search algorithm. The refiner takes as input the simplified plan and the simplification hierarchy, in our case a list of pairs $\langle \alpha_i, \beta_j \rangle$. It will iteratively work backwards through the simplification hierarchy. In each step, it first *decomposes* the composition by applying, β_j^{-1} then it *refines* the plan by applying α_i^{-1}.

To decompose the problem, the refiner loops through the composite operators of β_j and replaces them with the chain of operators that made them up. Since the composite operator and the chain of operators making it up are identical, decomposing a composite operator in the plan does not change its validity and requires no further changes.

To refine the problem, the refiner reinserts the safe variables of α_i into the problem. Since variable abstraction changes the way operators can be applied with respect to each other, refinement may make the simplified plan not valid in the more concrete problem. We can see this when considering an simplified plan with just two operators $\pi = \langle o_i^\alpha, o_j^\alpha \rangle$. To apply

o_j^α to the partial state created by o_i^α, the effects of o_i^α must either be disjoint from the preconditions of o_j^α (assuming the preconditions of o_j^α hold in the initial state) or at least be consistent with the preconditions of o_j^α. If we now refine the problem by applying, α^{-1} we will reinsert variables into the preconditions and effects of the operators of the problem. We are now no longer guaranteed that the effects of o_i are disjoint or consistent with the preconditions of, o_j and the plan may no longer be valid. If the variable abstraction α removed a variable v from the problem and the effects of o_i now contains $v \mapsto 1$ and the precondition of o_j now contains $v \mapsto 2$, we must now find an applicable sequence of operators O who's overall effect is to change the assignment of v from $v \mapsto 1$ to $v \mapsto 2$ without violating some other precondition of o_j.

As already discussed in Section 3.1, since v is a safe variable, we can find this sequence by searching the free DTG of v. We perform a simple breath first search, which takes as arguments the source value ($v \mapsto 1$ this case) and the target value ($v \mapsto 2$). Once such a path has been found, it can be inserted between o_i and o_j to resolve this conflict. Once all such conflicts have been resolved and the plan expanded to be fully valid again, the refiner continues with the next refinement step.

5 Experimental Evaluation

To compare the theory and implementation presented in this paper against the baseline with no such simplifications, we have conducted a series of experiments. We will briefly introduce the setup we used for our experiments, then talk about the challenges and results of the experiments.

5.1 Setup

As stated in previous chapters, we implemented our safe abstraction algorithm in the Fast Downward system. We use the Python package Downward Lab by Seipp et al. [12] to run our experiments on the SciCORE scientific computing centre at the University of Basel. The experiments were run on Core Intel Xeon Silver 4114 2.2 GHz processors. We set the time limit for each run to 30 minutes, and the memory limit to 3.5 GiB. We tested the implementation on a collection of IPC benchmark instances.[6] It total we consider 1836 problems across 65 domains.
As our search algorithms, we used the first component of the Lama planning system introduced by Richter and Westphal [11], a greedy best first search algorithms using the FF heuristic and a simple breadth first search. We will call these algorithms Lama-First, FF and Blind. Since we are not interested in generating optimal plans, our search algorithms assume each operator to have a cost of one.

We ran all planning tasks for four configurations of Fast Downward; first, we ran it while not doing any simplifications. We use this as the baseline to compare our changes to. Then we ran it with just variable abstraction, performing no operator composition. Finally, we ran it with both variable abstraction and operator composition. Once as the normal "harsh" version and once in the "soft" version mentioned in Section 4.2.2. We will call these versions NONE when no safe abstraction was made, ABSTRACTION if only variable abstraction was performed, ALL for full safe abstraction with harsh operator composition and ALL_SOFT for full safe abstraction with soft operator composition.

[6] https://github.com/aibasel/downward-benchmarks

	NONE	ABSTRACTION	ALL	ALL_SOFT
Atoms Abstracted	0.0%	6.58%	6.58%	6.58%
Abstraction Steps	0	570	570	570
# Abstracted Variables	0	3141	3141	3141
# Composite Operators	0	0	0	0

Table 5.1: Abstraction result of experiments. The atoms abstracted is calculated by an arithmetic mean over all problems, while the abstraction steps and the number of abstracted variables and composite operators is the sum over all problems.

5.2 Results

Since the choice of search algorithm will not influence the success of safe abstraction, we have split the results of the experiments into two parts: the abstraction results and the search results.
The abstraction results show to what degree the benchmark problems could be simplified. The search results show the impact these simplifications had on the time and memory requirements of the search.

5.2.1 Abstraction Results

Let's start with the abstraction results. Here, we will be primarily looking at the following metrics:

Atoms Abstracted:[7] The percentage of atoms that were abstracted.

Abstraction Steps: The number of $\langle \alpha_s, \beta_c \rangle$ pairs that were generated.

Num. abstracted variables: The number of variables that were abstracted.

Num. composite operators: The number of composite operators that were created.

From the abstraction results in Table 5.1 we can see that on average, across all benchmark problems, a relatively small number of atoms is abstracted. The operator compositions provide no benefit. In fact, neither harsh nor soft operator composition creates even a single composite operator across all problem domains.

As discussed in Section 4.2.2 we know the soft operator composition works in our trucks toy example. The benchmark encodings of similar problems are different. Since operator composition is quite sensitive[8] to the encoding of a problem, those differences might explain why we don't see any operator composition in the benchmark problems. We can examine

[7] We can't abstract individual atoms. Instead, whenever a variable is abstracted, we known we abstracted a number of atoms equal to that variable's domain size.
[8] meaning that slight changes in the encoding of a problem can greatly affect how many composite operators can be generated.

one simple problem from the gripper domain, for which Haslum [7] has managed to produce operator compositions. The problem is similar to the trucks problem but here we have two rooms, two balls and a robot that can move between the rooms. The robot has two arms, each of which can pick up a ball if it isn't holding one or drop the ball it is holding it. Just like Fig. 3.3, the location of the robot has already been abstracted away. Now we would like to create the operator sequences $\langle \textit{pick-ball-roomA-arm}, \textit{drop-ball-roomB-arm} \rangle$. Looking at a concrete operator: *pick-ball*1*-roomA-armLeft* we find its effects to be $\{\textit{ball}1 \mapsto \textit{carried-by-armLeft}, \textit{armLeft} \mapsto \textit{not-free}\}$. So for it to be in A to start the composite operator, our c has to be $c = \{\textit{ball}1 \mapsto \textit{carried-by-armLeft}, \textit{armLeft} \mapsto \textit{not-free}\}$. If we now look at the preconditions of *drop-ball*1*-roomB-armLeft* we find $\{\textit{ball}1 \mapsto \textit{carried-by-armLeft}\}$. So we can't include it in B. In fact, there are no other operators whose preconditions contain c. Because our B is empty, we are unable to produce any operator sequences and no operator composition can be done for this c. The same goes for all other related operators over a different ball.

Of course, $\textit{ball}1 \mapsto \textit{carried-by-armLeft}$ implies that $\textit{armLeft} \mapsto \textit{not-free}$. We can change our encoding to capture this implication explicitly by adding $\textit{armLeft} \mapsto \textit{not-free}$ as a precondition to the operators *drop-ball-room-arm*. This doesn't change the semantics of the problem but allows the drop operators to be included in B. If we continue where we left off above, for our $c = \{\textit{ball}1 \mapsto \textit{carried-by-armLeft}, \textit{armLeft} \mapsto \textit{not-free}\}$ we now have *pick-ball*1*-roomA-armLeft* in A and *drop-ball*1*-roomB-armLeft* in our B. We can now check the conditions of Theorem 2. We see that Condition 2 is being violated. The operator *pick-ball*2*-roomA-armLeft* has the following effects: $\{\textit{ball}2 \mapsto \textit{carried-by-armLeft}, \textit{armLeft} \mapsto \textit{not-free}\}$. This means, since its effect do not contain c, it is not in A, but because of $\textit{armLeft} \mapsto \textit{not-free}$ its effects are not disjoint nor inconsistent with c. This issue is mirrored with the other operators over *ball*2 and the operators using *armRight*. So this simple reformulation is not enough and the encoding would have to be changed further to allow for composition in this problem.

The encoding of a problem is not the only relevant property that determines the success of variable abstraction and operator composition. The domain of a problem too greatly affects how much of the problem can be simplified. Table 5.2 shows, how some domains allowed for no simplification with our implementation while others like the logistics domain could be fully solved by safe abstraction. We chose these domains to match a table of results included in Haslum [7]. We compare our results against their results in Table 5.3. In that table, We show the range of simplification achieved per domain, instead of the arithmetic mean. We can see that Haslum generally achieve better ranges of simplification for most domains listed. In particular, Haslum manages to fully solve the gripper, movie, and satellite domains, whereas our implementation is only able to achieve very minor to moderate simplification degrees in those domains. We achieve similar results to Haslum in the depot, driverlogs and logistics domains. Particularly interesting is the logistics domain, which is fully solvable with just variable abstraction.

	NONE	ABSTRACTION	ALL	ALL_SOFT
gripper(20)	0%	2.43%	2.43%	2.43%
logistics00(28)	0.0%	100%	100%	100%
logistics98(24)	0.0%	100%	100%	100%
movie(30)	0%	0%	0%	0%
mystery(27)	0%	0%	0%	0%
mprime(28)	0%	0%	0%	0%
grid(5)	0%	0%	0%	0%
freecell(80)	0%	0%	0%	0%
depot(21)	0.0%	4.47%	4.47%	4.47%
driverlog(19)	0.0%	9.4%	9.4%	9.4%
rovers(35)	0%	60.57%	60.57%	60.57%
satellite(27)	0.0%	55.32%	55.32%	55.32%
airport(27)	0%	0%	0%	0%

Table 5.2: Percentage (arithmetic mean) of atoms abstracted from a problem domain. The number in parentheses denotes the number of problem instances in the domain.

	Haslum [7]	Our results
gripper(20)	100%	1–8%
logistics(28+24)	100%	100%
movie(30)	100%	0%
mystery(27)	0%	0%
mprime(28)	0%	0%
grid(5)	~50%	0%
freecell(80)	0%	0%
depot(21)	1–10%	1–12%
driverlog(19)	0–25%	0–35%
rovers(35)	60–90%	37–78%
satellite(27)	100%	32–83%
airport(27)	40–60%	0%

Table 5.3: The range of atoms abstracted from the problems of a domain achieved by Haslum [7] and our implementation. The number in parentheses denotes the number of problem instances in the domain that our implementation ran in. We don't know how many problems Haslum ran their implementation on.

There are several ways to explain this discrepancy. Firstly, as already mentioned in Section 4.2.2, we might not have the same domain and problem encodings as them. In general, the same planning task can be encoded in very different ways and safe variable abstraction and in particular safe operator composition are highly sensitive to the problem's encoding. On top of this, we have Fast Downward's translator module, mentioned at the start of Chapter 4, which takes the problems encoded in PDDL[5][6] and transforms them into a finite domain representation[9]. During this transformation, the translator is already performing some simplifications to produce an encoding that is as simple as possible. It is possible that even if the problem could be well abstracted with safe abstraction, the output of the translator is either already as simplified as it can be or that the new encoding doesn't allow safe abstraction. Finally, also mentioned in Section 4.2.2, we had to make a number of assumptions while implementing safe abstractions. Some of these assumptions might be wrong, or we have made other logical errors in the process. Unfortunately, it is also possible

	NONE	ABSTRACTION	ALL	ALL_SOFT
Blind	482	**560**	557	541
FF	1219	**1306**	1255	1101
Lama-First	1624	**1648**	1553	1347

Table 5.4: Coverage achieved by the search algorithms in the different configurations. The best results are highlighted in bold.

that undiscovered bugs are changing the behaviour of our implementation in unintended ways.

Let's now look at how our simplifications influence the time and memory behaviour of the search algorithms.

5.2.2 Search Results

In this section, we will mainly look at the following metrics and how they change as the degree of simplification changes:

Coverage: The number of problems for which a solution was found within the time and memory limits.

Search time: The time the search algorithm took to find a solution to the potentially simplified task.

Abstraction time: The time the abstractor took to perform the safe variable abstractions.

Composition time: The time the compositor took to perform the operator compositions.

Refinement time: The time the refiner took to refine the simplified plan.

Total time: The total time the planner took to solve the problem (including search and safe abstraction but excluding the translator).

Memory: The peak amount of memory used by the planner to solve the problem (including search, safe abstraction and the translator).

The easiest way to see if the search has benefitted from simplification is to look at the coverage. The coverage counts the number of problems each search algorithm was able to successfully solve. Excluding unsolvable cases, there are two reasons why a problem was not solved, either it ran out of memory or it ran out of time. Therefore, if we see an improvement in the number of problems solved, then we can expect our simplifications to have saved time or memory or both during the search. Table 5.4 shows the coverage of our experiments.

We see that when using Blind, ABSTRACTION gives us significantly more coverage than in NONE. When we compare the more sophisticated search algorithms, the gap in coverages between the unmodified problems and the problems simplified with safe variable abstraction begin to narrow. In the most sophisticated search algorithm tested, Lama-First, the unmodified problems are almost as well covered as those on which variable abstraction was performed.

One explanation of why variable abstraction offers a greater runtime benefit in Blind than in a more sophisticated search like FF or Lama-First could be that FF and Lama-First are already exploring much less of the state space of the problem. Since they use heuristics to guide them to a goal, they need only need to consider operators that lead towards the goal. If we abstract a variable and it reduces the size of the state space, Blind will benefit more from that reduction than algorithms like FF and Lama-First because they will have to explore less of the state space to begin with. We can compare the search time in the simplified and unmodified problems directly.

Fig. 5.1 shows a comparison of the search times of Lama-First in the NONE and ABSTRACTION configuration. In such scatter plots, if a data point lies on the upper or right bound of the plot, it means that that problem could not be solved within the time limit by the respective configuration or search algorithm. We can see that while some problems are solved significantly faster when safe variable abstraction is used, a large portion of the problems seem to gain very little or no benefit from the simplification. Fig. 5.2 shows the same figure but includes only those domains for which at least one variable was abstracted. We see that rovers and satellite gain no significant benefit from variable abstraction despite having a majority of their atoms abstracted (See Table 5.1), while with other domains like transport-sat08-strips (24.42% of atoms abstracted) and transport-sat11-strips (21.28% of atoms abstracted) we observe a significant reduction in the search time. When we compare this to, Fig. 5.3 which shows the same domains but with a search using Blind, we see that the search of both rovers and satellite is much faster in the ABSTRACTION configuration than NONE. The search over transport-sat08-strips too is running faster than in NONE, while transport-sat11-strips can't be shown because the planner exceeds the memory constraints for both configurations in all runs.

As previously discussed, Lama-First uses heuristics to guide them to the goal and has to explore much less of the state space than Blind. One explanation for the results above is that domains like satellite and rovers are more easily navigable. Meaning that the heuristics give strong guidance and only a minimal amount of that problem's state space has to be explored, while domains like transport-sat08-strips and transport-sat11-strips are less navigable, meaning more of the problem's state space has to be explored. If Lama-First has to explore more of the state spaces of the problems in the transport-sat domains, we would expect a greater benefit there than in satellite and rover. Blind of course has no heuristics and is not guided and benefits in all cases. This however only really explains the problems for which we see no benefit nor any detriment to the search time. One explanation of why some problems, like some satellite problems, were solved slower in ABSTRACTION than in NONE when using Lama-First is that the abstracted variables happened to be variables that were important to some heuristic, and without that variable the heuristic becomes much less informative. In this case, even though the simplified problem has a smaller state space, the heuristics in the unmodified problem are so much better informed that the benefit of these better heuristics to the search outweigh the benefit of having to search a smaller smaller state space.

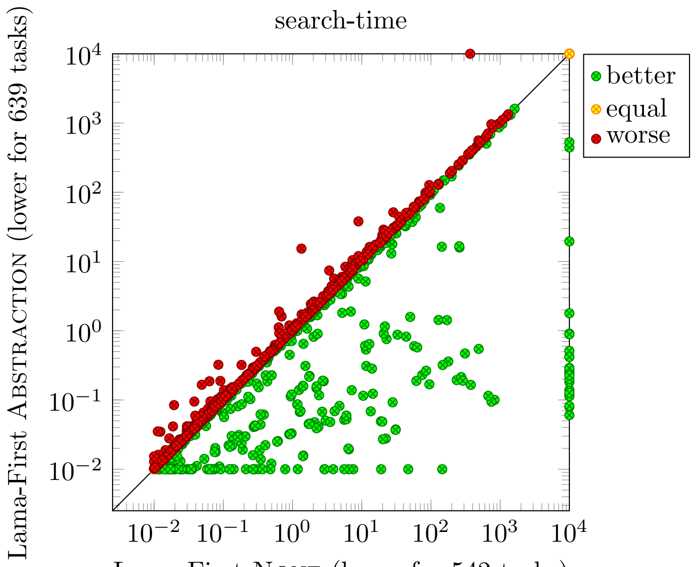

Figure 5.1: Comparison of the search time of Lama-First when run on problems simplified with variable abstraction versus problems with no simplifications performed.

Going back to our discussion on coverage, we see that safe abstraction with soft operator composition performs significantly worse than with harsh operator composition, especially when more sophisticated algorithms are used. We saw in Table 5.1 that ALL_SOFT does not produce a higher level of simplification than ALL, nor does ALL compared to ABSTRACTION. Despite this, ALL_SOFT requires significantly more effort. We can see that in Fig. 5.4. We see that in almost all problems solved, ALL_SOFT required more time to perform operator composition and generate the composite operators.

The difference between the time required by the compositions can be explained by the different conditions to abort operator composition in a variable pair. As discussed in Section 3.2, the harsh operator composition used in ALL aborts operator composition for a specific variable pair if any of the conditions in Theorem 2 do not hold for any valid[9] c. The soft operator composition used in ALL_SOFT has to try each valid c before moving on with the next variable pair. We can see the significance of this by imagining some variable pair where none of their c hold in all conditions of Theorem 2. ALL only has to try the first c. Once that c violated one of the conditions, ALL moves on with the next variable pair. ALL_SOFT

[9] c that violate condition 1 are not generated

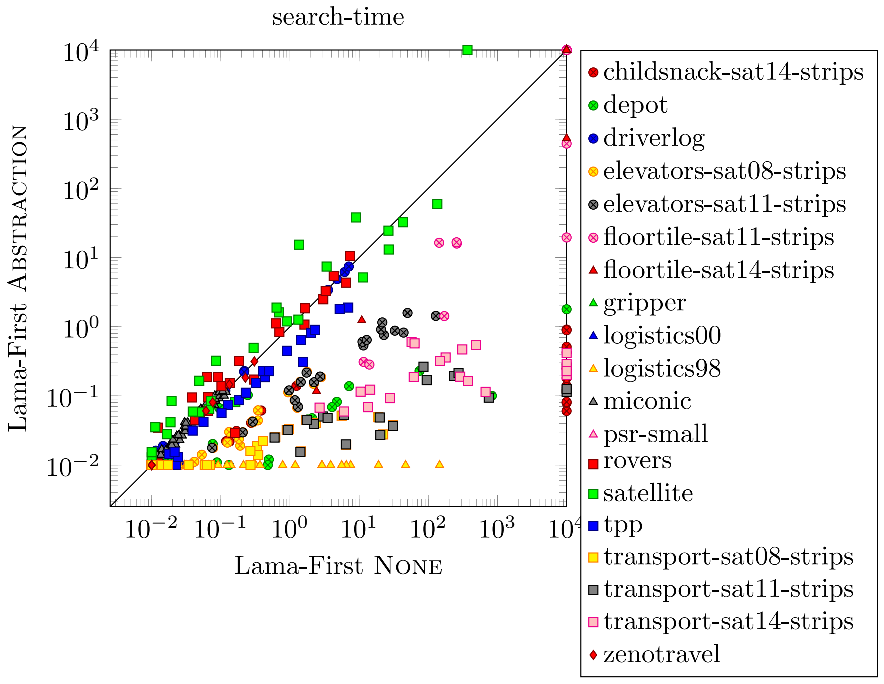

Figure 5.2: Comparison of the search time of Lama-First when run on problems simplified with variable abstraction versus problems with no simplifications performed. Depicted are only those domains for which at least one variable was abstracted.

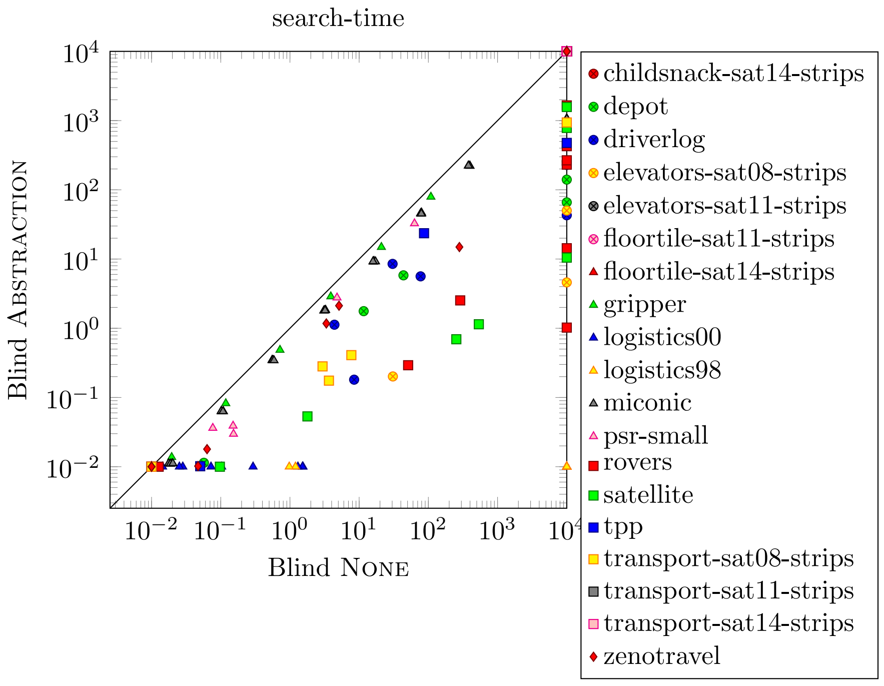

Figure 5.3: Comparison of the search time of Blind when run on problems simplified with variable abstraction versus problems with no simplifications performed. Depicted are only those domains for which at least one variable was abstracted.

	ABSTRACTION	ALL	ALL_SOFT
abstraction time	0.05s	0.05s	0.05s
composition time	0s	38.73s	240.52s
combined time	0.05s	38.78s	240.57s

Table 5.5: Abstraction and composition time (arithmetic mean) of the different configurations.

however, has to try every c of the variable pair before being able to move on to the next one.

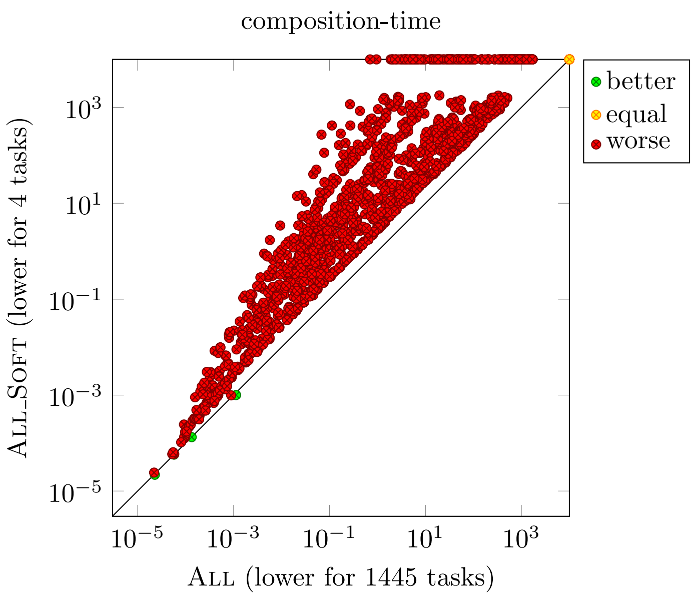

Figure 5.4: Comparison of the composition time of ALL_SOFT againstALL.

Not only is the overhead of ALL_SOFT much greater than ALL, but both harsh and soft operator compositions are much more expensive than variable abstractions. Table 5.5 compares the overhead of variable abstraction and operator composition in the configurations. We see that composition is orders of magnitude more expensive to compute than variable abstraction. Since, as we've seen in Table 5.1 ALL and ALL_SOFT, achieve no greater level of simplification than ABSTRACTION, we expect these configurations to do worse than ABSTRACTION and NONE in the overall time.

The final overhead of safe abstraction is the refinement time. Compared to the overhead of variable abstraction, plan refinement time is almost negligible. For an example, we look at a problem within the logistics98 domain. The abstractor took 2.64787 seconds to abstract all 116 variables from the problem, solving it entirely. Since the problem was solved, we skipped the search. Then, the refiner extended an empty plan to a valid plan in the concrete problem in just 0.0128406 seconds. For other problems with fewer safe variables found, the refinement was even faster.

	NONE	ABSTRACTION	ALL	ALL_SOFT
Blind	0.68s	**0.41s**	0.60s	1.02s
FF	**0.04s**	**0.04s**	0.11s	0.35s
Lama-First	**0.04s**	**0.04s**	0.11s	0.33s

Table 5.6: Total time (geometric mean) of the search algorithms in the different configurations. The best results are highlighted in bold.

Now that we have discussed all the overhead of the different components, let us look at how the different configurations of safe abstraction change the total time required.

In Table 5.6 we show the geometric mean of the total time the planner took. Since it includes both the search and the abstraction and composition times, it is a useful measure to weight the benefit of a configuration against its overhead. We see that variable abstraction can significantly reduce the total time of the planner when a breadth first search algorithm is used, but when more sophisticated search algorithms are used, ABSTRACTION no longer offers a benefit over NONE, but is able to keep pace with it. As discussed with Fig. 5.4 and Table 5.5, ALL and ALL_SOFT suffer from the overhead of operator composition, without benefiting the search more than ABSTRACTION, which results in significantly worse runtime.

Now that we have discussed the time requirements of the planner with various configurations and search algorithms, let us have a look at its memory usage.

Shown in Table 5.7 is the total amount of memory the planner required for each configuration. We see that, all configurations of safe abstraction have a comparable memory footprint. For Blind, ABSTRACTION reduces the memory required to solve the problems. With the more sophisticated search algorithms, ABSTRACTION requires marginally more memory than NONE.

An intuitive explanation may be that, as a search algorithm has to explore more of a problem's state space, it needs to track more information about what states have already been explored and which can be explored next. Since, as discussed above, Blind has to explore a greater fraction of the state space of a problem, the simplifications created by safe abstraction help to significantly cut down on the number of states it has to explore and thus how much memory it uses. In the cases of FF and Lama-First, since they already explore much less of the state space, the simplifications of safe abstraction do not reduce the memory load of the search enough to compensate for the memory usage of keeping track of the simplification steps.

Finally, though we aren't interested in finding optimal paths, we can have a look at the final plan length found by the search algorithms with ABSTRACTION and NONE.

Fig. 5.5 and Fig. 5.6 show the length of the plan for a problem in the two configurations.

	NONE	ABSTRACTION	ALL	ALL_SOFT
Blind	75.81 MB	**60.01 MB**	60.11 MB	60.11 MB
FF	**22.64 MB**	23.47 MB	23.49 MB	23.50 MB
Lama-First	**22.56 MB**	23.18 MB	23.19 MB	23.19 MB

Table 5.7: Memory (geometric mean) needed by the search algorithms and planner. The best results are highlighted in bold.

In the case of NONE this is just the normal plan length as of course no variable abstraction nor refinement takes place. In the case of ABSTRACTION it is the length of the plan after it has been fully refined. We can see from Fig. 5.5 that the plans found by Blind in ABSTRACTION are at best as good as those it found in Fig. 5.6 but in many cases slightly worse.

This makes sense because, since Blind is a breath first search, it will always find the shortest path in a problem. Since a simplified plan can never become shorter during refinement, the best case scenario in the problems simplified by ABSTRACTION is that Blind finds a plan with the same length, and it is not extended during refinement. We do not have that guarantee however and as we can see, the plans can be worse because the shortest simplified plan isn't necessarily the simplified plan that is the shortest after it was refined.

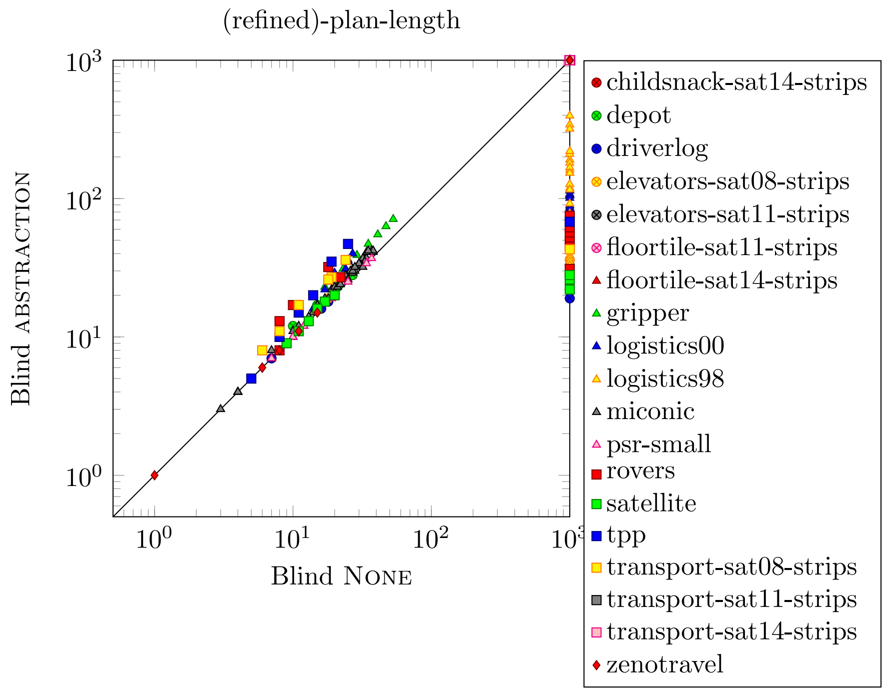

Figure 5.5: Comparison of length of the plan found by Blind. In the case of ABSTRACTION, after refinement took place. Depicted are only those domains for which at least one variable was abstracted.

We see in Fig. 5.6 that the same argument does not hold when Lama-First is used as the search algorithm. In this case, we see that Lama-First may find shorter plans in the simplified problems, though in many of the problems, the plans found are longer than when no variable abstraction is performed.

Lama-First, unlike Blind, doesn't necessarily find the shortest plan possible. One possible explanation is that, as mentioned before, variable abstraction messes with the heuristics used by Lama-First. In this case, the heuristics might guide the search algorithm differently and might find a plan that, even when refined, is shorter than the plan found by Lama-First in NONE.

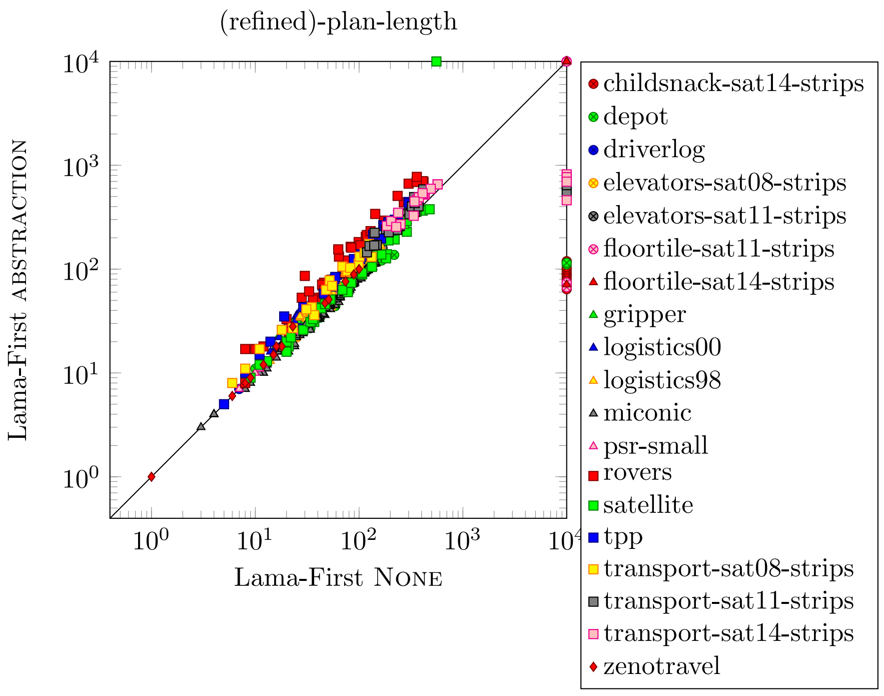

Figure 5.6: Comparison of length of the plan found by Lama-First after refinement took place. In the case of ABSTRACTION, after refinement took place. Depicted are only those domains for which at least one variable was abstracted.

6 Conclusion

We have implemented the safe abstraction as described by Haslum [7] in the Fast Downward planner[9]. We have developed separate components to handle safe variable abstraction, operator composition and problem refinement. We have shown that safe variable abstraction can significantly reduce the accidental complexity of a problem and in some cases outright solve them. Unfortunately, we were unable to reproduce the operator compositions described in Haslum [7] with our implementation.

Our experiments have shown that even with just safe variable abstraction, we can make significant improvements in the time and memory requirements of finding a plan, especially when less sophisticated search algorithms are used. We have also seen that some problems and encodings are better suited to be simplified by safe abstraction, while in others, no simplification can be achieved by applying safe abstraction.

An interesting direction for a future project is to investigate the differences in our results between our implementation and the implementation of Haslum [7]. There are several explanations that could at least partially explain this. Are the problem encodings different, and if so, what properties do our encodings have that makes operator composition so hard? Alternatively, our implementation of operator composition could be wrong or too harsh. In this case, it would be interesting to see how Haslum handled the case discussed in Section 4.2.2. Finally, Haslum mentioned some other techniques and restrictions in their implementation section. Another interesting project is to look at these techniques and restrictions and analyse their interplay with safe abstraction to see how beneficial they are and if they can explain the difference in results. A final interesting observation that might warrant further exploration is the search time and plan length of problems in the satellite and rover domain. With safe abstraction we would expect our search time to be reduced or at worst stay the same while the plan length grow longer or at best remain the same length. However, when Lama-First was used on those domains, we saw the inverse happen. We saw an increase in the search time and a decrease in the plan length when compared to the unsimplified problems. It would be interesting to see what properties of those domains or of Lama-First cause or enable this inversion of our expectation.

Bibliography

[1] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[2] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[3] Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[4] Stefan Edelkamp. Planning with pattern databases. In *Sixth European Conference on Planning*, 2014.

[5] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th International Planning Competition. Technical report, 195, University of Freiburg, 2004.

[6] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[7] Patrik Haslum. Reducing Accidental Complexity in Planning Problems. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1898–1903, 2007.

[8] Malte Helmert. Fast (diagonally) Downward. *5th International Planning Competition Booklet*, 2006.

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

[10] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. DTG-Plan: Fast Planning by Search in Domain Transition Graphs. *AAAI, Conference on Artificial Intelligence*, page 886–891, 2008.

[11] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[12] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.399255, 2017.

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:	Safe Abstraction in Fast Downward
Name Assessor:	Prof. Dr. Malte Helmert
Name Student:	Joan Moser
Matriculation No.:	21-050-372

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Basel, 15.12.2024 Student: [illegible]

Will this work, or parts of it, be published?

[✓ No]

☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: ______________________

Place, Date: ______________________ Student: ______________________

Place, Date: ______________________ Assessor: ______________________

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

September 2023