New Flaw Search Strategies in CEGAR for Cartesian Abstractions in Optimal Classical Planning

MARTÍN POZO*, Computer Science and Engineering Department, Universidad Carlos III de Madrid, Spain
ÁLVARO TORRALBA, Aalborg University, Denmark
CARLOS LINARES LÓPEZ, Computer Science and Engineering Department, Universidad Carlos III de Madrid, Spain

Background: Counterexample-Guided Abstraction Refinement (CEGAR) is a prominent technique to generate Cartesian abstractions for guiding search in cost-optimal planning. The core idea is to start from the trivial abstraction—an abstract state representing all concrete states—and iteratively refine it in a loop by splitting one abstract state into two. Each iteration of the loop searches an optimal abstract plan and replicates it into the concrete state space until finding a flaw—and splitting such abstract state—or returning a plan for the task if no flaw is found. Previous work finds only a single flaw in progression, by executing the abstract plan from the initial state and stopping when execution cannot be continued.
Objectives: A theoretical framework for identifying different flaws of abstract plans, analysing their properties and using them to refine abstractions in different ways leading to higher-quality heuristics.
Methods: We show that alternative types of flaw can be defined. Specifically, we search flaws in regression from the goals—resulting in higher heuristic values—and identify sequence flaws along the whole abstract plan by searching them in a Cartesian relaxation of the task after the first flaw, which greatly increases the flexibility of refinements in CEGAR.
Results: Our experiments show that across existing benchmarks numerous sequence flaws exist in most abstract plans. We observe that the selected flaw has a high impact on the heuristic, and we propose several strategies that generate more informed abstractions. Overall, this greatly improves the performance of the baseline that only used the first progression flaw.
Conclusions: Guiding the CEGAR process by analysing multiple flaws of an abstract plan trace and selecting the best refinement can improve its performance as a tool to generate admissible heuristics in planning. Thus, using our new types of flaw and flaw selection strategies, we get stronger heuristics than the state-of-the-art planners based on Cartesian abstractions.

JAIR Associate Editor: Gabriele Röger

JAIR Reference Format:
Martín Pozo, Álvaro Torralba, and Carlos Linares López. 2026. New Flaw Search Strategies in CEGAR for Cartesian Abstractions in Optimal Classical Planning. *Journal of Artificial Intelligence Research* 86, Article 49 (August 2026), 60 pages. DOI: 10.1613/jair.1.21501

1 Introduction

Automated Planning is an area of Artificial Intelligence focused on, given a description of the world, determine what is the best course of action to achieve a given goal (Ghallab et al. 2004). Specifically, in classical planning,

*Corresponding Author.

Authors' Contact Information: Martín Pozo, ORCID: 0009-0003-7998-3715, marcosmartin.pozo@uc3m.es, Computer Science and Engineering Department, Universidad Carlos III de Madrid, Avenida de la Universidad, 28911, Leganés, Madrid, Spain; Álvaro Torralba, ORCID: 0000-0002-5352-2529, alto@cs.aau.dk, Aalborg University, Selma Lagerløfs Vej 300, 9220, Aalborg, Aalborg, Denmark; Carlos Linares López, ORCID: 0000-0003-1811-4754, carlos.linares@uc3m.es, Computer Science and Engineering Department, Universidad Carlos III de Madrid, Avenida de la Universidad, 28911, Leganés, Madrid, Spain.

DOI: 10.1613/jair.1.21501

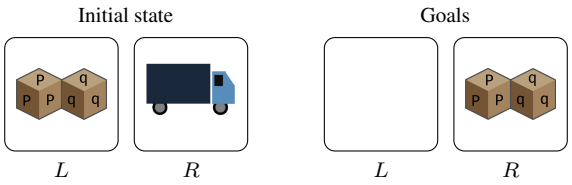


Fig. 1. Running example. In the initial state a truck is at R and there are two packages at L, which must be delivered to R.

we aim to find a sequence of actions of minimum cost that can be applied in the current state to reach a goal state (Bäckström and Nebel 1995; Fikes and Nilsson 1971). A common way is to perform heuristic search in the state space of the planning task (Bonet and Geffner 2001), using the A* algorithm (Hart et al. 1968). A* leverages an admissible heuristic function that lower bounds the goal distance from each state to the goal in order to guide the search. Thus, having informative heuristics is an essential component of many modern planning systems.

Abstractions are commonly employed in optimal planning to generate admissible heuristics, as they offer great flexibility to define well-informed heuristics for the planning task at hand (Edelkamp 2001; Helmert, Haslum, Hoffmann, and Nissim 2014; Sievers and Helmert 2021). The general idea is to map states of the planning task into an abstract state space that over-approximates the state space of the original problem where optimal goal distances can be computed in polynomial time. Such distances can then be used as lower-bound estimates of the goal distance from each state to the goal. This offers great flexibility, and there are numerous methods to automatically obtain abstractions given the definition of a planning task. However, such flexibility raises the question of how to efficiently compute the right abstraction for a given planning task. A very promising method is Counterexample-Guided Abstraction Refinement (CEGAR), originally introduced as a method for model checking (Clarke, Grumberg, Jha, et al. 2003). In the context of planning, CEGAR has successfully been used for different types of abstractions, such as Cartesian abstractions (Seipp and Helmert 2013, 2018), Pattern Databases (Culberson and Schaeffer 1998; Edelkamp 2001; Rovner et al. 2019) and domain abstractions (Hernádvölgyi and Holte 2000; Kreft et al. 2023). CEGAR starts from the trivial abstraction, where all states of the task are considered equivalent to each other. And then it iteratively refines the abstraction by choosing an optimal abstract plan, and executing it in the original state space to determine whether it is a valid plan. If the plan is valid, then it is guaranteed to be an optimal plan for the original problem, so the task is solved. Otherwise, the abstraction is refined, by identifying a *flaw* (some step in the plan that could not be performed), and refining the abstraction to distinguish states in which such a step is possible from those where it is not.

In this paper, we consider in detail the refinement step in the CEGAR algorithm for Cartesian abstractions, and how different choices influence the abstraction that is generated as well as the quality of the resulting heuristic. While the notion of *flaw* from prior work (Seipp and Helmert 2018)—which we call progression flaw—corresponded to the first step in the plan that cannot be executed in the original state space, we argue that there may be many other reasons why the abstract plan is not valid. We illustrate this by using our running example, shown in Figure 1, a Transport problem with two locations L and R, one truck t initially at R and two packages p and q at L that must be delivered to R. Consider now an abstraction where the position of the truck has been fully abstracted away (which can be thought of as if the truck was simultaneously at both locations), and whenever package q has been delivered, the abstraction does not distinguish whether the package p is at the goal R or in the truck (so, we can consider that it is simultaneously in t and at R). Then, the sequence of operators $\langle pickup(q, L), pickup(p, L), drop(q, R)\rangle$, corresponds to an optimal abstract plan. Figure 2 shows the plan with a graphical depiction of the traversed abstract states. Clearly, this is not a valid plan for the original problem for multiple reasons. Using progression flaws, as considered by prior work, we would find that $pickup(q, L)$ is not

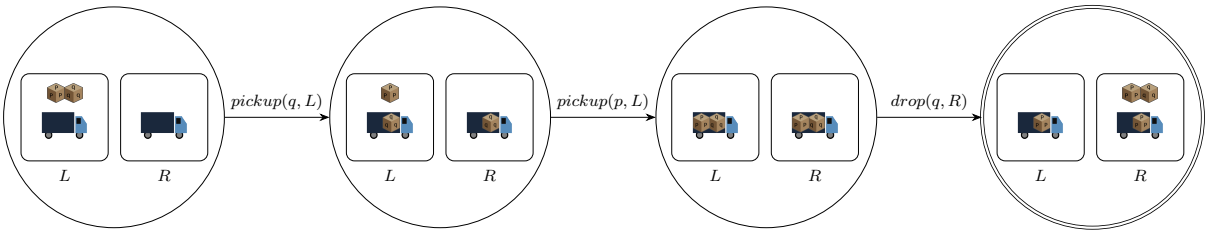


Fig. 2. An abstract plan trace for our running example, showing the selected actions and the abstract states that the plan traverses. In the abstraction, the position of the truck has been abstracted away. This is represented in the figure by the same truck being at the same time in L and R. Similarly, in the goal abstract state, the abstraction ignores whether package p is in the truck or at R.

applicable in the initial state due to t not being at L. So, to address this they would refine the abstraction by splitting the first abstract state into two abstract states that distinguish the position of the truck. This is indeed a reasonable choice that improves the abstraction, but, is it the best choice to get the most informative heuristic?

In this paper, we generalize the notion of flaw of an abstract plan trace, and introduce several new types of flaw that lead to different abstraction refinements, each with different strengths.

The first type that we introduce is **regression flaws**, where the abstract plan is not executed forwards but backwards from the goals. In the example of Figure 2, the regression flaw is that $pickup(p, L)$ does not achieve the goal of the problem, as the package p would end in the truck and not be delivered. This leads to a different abstraction refinement, distinguishing whether p is at R or in the truck in parts of the abstraction. Thus, using regression flaws results in different abstractions, and we will show that it often leads to more informed heuristics.

The second type of flaw we introduce are **sequence flaws**, with the objective of characterising issues in the middle of the abstract plan, as opposed to progression and regression flaws, which are exclusively focused on the prefix and the suffix of the plan, respectively. In our running example, q cannot be dropped in R because t is not at that location after it picked up the package. So, consecutive *pickup-drop* operators in different locations without a *drive* operator between them always corresponds to a flaw, independently of the initial state and the goals.

Considering different types of flaw enables us to study the question of what types of flaw are most suitable for guiding the CEGAR process. We introduce a number of selection strategies, and empirically analyse their performance, showing that carefully choosing what flaw to use greatly pays off to obtain better heuristics and, ultimately, find optimal solutions to more planning tasks.

This article is an extension of our work on regression flaws (Pozo, Torralba, et al. 2024b) and sequence flaws (Pozo, Torralba, et al. 2024a). Here, we provide an overview of the different types of flaws we consider, and frame them into a common framework that captures their common characteristics. Also, we significantly extend our analysis on how refining the abstraction based on different flaws affects the resulting heuristic. To that end, we introduce several new strategies to select which flaw to use (or even refining based on multiple flaws), based on different factors such as their goal distance, predefined variable orderings, methods assigning weight to the facts used in the split, and others. Finally, a comprehensive empirical evaluation of all strategies shows that no single strategy is the best in all domains, and that this often depends on the exact setting considered (e.g., using a single abstraction, multiple abstractions, and the state-of-the-art configuration of the Scorpion planner (Seipp 2023)).

The paper is structured as follows. Section 2 introduces the background. Section 3 gives a general definition of flaw as a framework to specify new types of flaw. Section 4 defines regression flaws and analyses their properties. Section 5 explains sequence flaws. Section 6 introduces flaw selection strategies to choose what flaw should be used to refine the abstraction. Section 7 extensively evaluates the behaviour of all types of flaws and strategies. Finally, Section 8 describes the related work and Section 9 summarises the main findings of this work.

2 Background

In this section we explain essential concepts to understand the rest of the paper: SAS$^+$ Planning, abstraction heuristics, Cartesian abstractions and the CEGAR algorithm.

2.1 SAS$^+$ Planning

We consider tasks in SAS$^+$ representation (Bäckström and Nebel 1995), where states are described in terms of a tuple of variables $V = \langle v_1, v_2, \ldots, v_n \rangle$ with domain $\mathcal{D} = \langle \mathcal{D}_{v_1}, \mathcal{D}_{v_2}, \ldots, \mathcal{D}_{v_n} \rangle$, such that each $v \in V$ has a finite domain $\mathcal{D}_v$ and each $\mathcal{D}_v \in \mathcal{D}$ is a set of values. A fact is a variable assignment $v \mapsto d$, where $d \in \mathcal{D}_v$.

Definition 1 (States). *Given a tuple of variables V with domain $\mathcal{D}$, a partial state $p = \{v_1 \mapsto d_1, v_2 \mapsto d_2, \ldots, v_n \mapsto d_n\}$ is a set of facts over some variables* $\text{vars}(p) = \{v_1, v_2, \ldots, v_n\} \subseteq V$.

A (concrete) state $s = \langle d_1, d_2, \ldots, d_n \rangle$ is a tuple of values where $d_i \in \mathcal{D}_{v_i}$ for all $d_i \in s$ and each value v_i represents a fact $v_i \mapsto d_i$. A state s can also be interpreted as a partial state defined for all variables $v \in V$.

We write $p[v]$ for the value assigned to the variable $v \in \text{vars}(p)$ in the partial state p. Two partial states p and c are consistent if $p[v] = c[v]$ for all $v \in \text{vars}(p) \cap \text{vars}(c)$. $[\![p]\!]$ is the set of concrete states consistent with p.

Definition 2 (SAS$^+$ Task). *A SAS$^+$ task Π is a tuple $\langle V, O, I, G \rangle$ where $V = \langle v_1, v_2, \ldots, v_n \rangle$ is a tuple of finite-domain variables, I is the initial state, G is a partial state that describes the goals, and O is a set of operators. An operator $o \in O$ has preconditions $pre(o)$ and effects $eff(o)$, both of which are partial states, and a non-negative cost,* $\text{cost}(o) \in \mathbb{R}_0^+$.

An operator o is applicable in progression in a state s if s is consistent with $pre(o)$. The result of applying o in s is another state $s' = progr(s, o)$ where $progr(s, o)[v] = eff(o)[v]$ if $v \in \text{vars}(eff(o))$ and $progr(s, o)[v] = s[v]$ otherwise. We write $s \xrightarrow{o} s'$ as a shorthand for $s' = progr(s, o)$. The post-conditions of an operator o in progression, $post(o)$, are defined for $v \in \text{vars}(pre(o)) \cup \text{vars}(eff(o))$ and $post(o)[v] = eff(o)[v]$ for $v \in \text{vars}(eff(o))$ and $post(o)[v] = pre(o)[v]$ otherwise.

Regression starts from a partial state p and derives from which states we can reach some state in $[\![p]\!]$ by applying an operator (Alcázar, Borrajo, et al. 2013; Rintanen 2008). An operator o is applicable in regression in p if and only if p is consistent with $post(o)$. The predecessor partial state $p' = regr(p, o)$ is defined for $(\text{vars}(p) \setminus \text{vars}(eff(o))) \cup \text{vars}(pre(o))$ and $regr(p, o)[v] = pre(o)[v]$ for $v \in \text{vars}(pre(o))$ and $regr(p, o)[v] = p[v]$ otherwise. We write $p \xleftarrow{o} p'$ as a shorthand for $p' = regr(p, o)$.

Example 1 (Running Example). *Figure 1 shows a typical transport problem with one truck that must deliver packages to destination locations. The SAS$^+$ task for this problem is $\Pi = \langle V, O, I, G \rangle$ where*

- *$V = \langle v_p, v_q, v_t \rangle$, with $\mathcal{D}_{v_p} = \{p_L, p_R, p_T\}$, $\mathcal{D}_{v_q} = \{q_L, q_R, q_T\}$ and $\mathcal{D}_{v_t} = \{t_L, t_R\}$.*
- *The operators are:*

$$O = \left\{ \begin{array}{r} drive(L,R),\ drive(R,L), \\ pickup(p,L),\ pickup(p,R),\ pickup(q,L),\ pickup(q,R), \\ drop(p,L),\ drop(p,R),\ drop(q,L),\ drop(q,R) \end{array} \right\},$$

  where v_p and v_q have been replaced by p and q respectively in the operators for the sake of readability. The preconditions and effects with generic parameters l and m for locations and x for a package are:
  - *$pre(drive(l,m)) = \{v_t \mapsto t_l\}$, $eff(drive(l,m)) = \{v_t \mapsto t_m\}$;*
  - *$pre(pickup(x,l)) = \{v_x \mapsto x_l, v_t \mapsto t_l\}$, $eff(pickup(x,l)) = \{v_x \mapsto x_T\}$;*
  - *$pre(drop(x,l)) = \{v_x \mapsto x_T, v_t \mapsto t_l\}$, $eff(drop(x,l)) = \{v_x \mapsto x_l\}$.*
- *$\text{cost}(o) = 1$ for all $o \in O$.*
- *$I = \langle p_L, q_L, t_R \rangle$, $G = \{v_p \mapsto p_R, v_q \mapsto q_R\}$.*

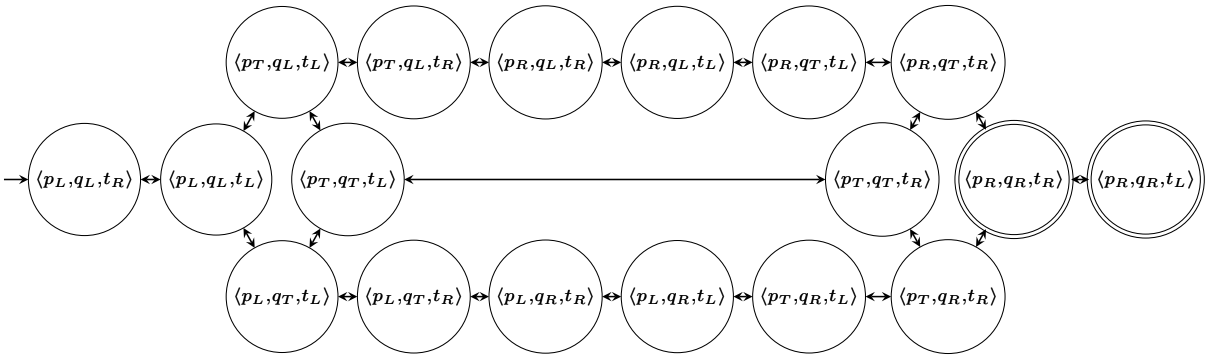


Fig. 3. State space of the task of our running example (Figure 1). Operators labels are omitted for the sake of clarity.

Thus, in our running example the application in progression of $o = pickup(v_p, L)$ in the state $s = \langle p_L, q_L, t_L\rangle$ is $progr(s, o) = \langle p_T, q_L, t_L\rangle$, and the regression of o in the state $s' = \langle p_T, q_L, t_L\rangle$ is $regr(s', o) = \langle p_L, q_L, t_L\rangle$.

Definition 3 (State Space). *The* state space *of a SAS$^+$ task $\Pi = \langle V, O, I, G\rangle$ is a transition system, $\Theta = \langle S, O, T, I, S_G\rangle$, where S is the set of states, O is the set of operators of Π, I is the initial state of Π, $S_G = [\![G]\!]$ is the set of goal states, and $T = \{\langle s, o, s'\rangle \mid s \xrightarrow{o} s'\}$ is the set of transitions. The size of a state space $|S|$ is the number of states in S.*

Figure 3 shows the state space of our running example. In this figure, the initial state is shown at the left, marked by a small right arrow at its left side. This state space has two goal states (since the position of the truck is irrelevant to the goals), marked by a double circle. Transitions are simplified in the figure for the sake of readability, but each bidirectional arrow represents two transitions with a different operator. For example, the operator of the transition starting at the initial state is $drive(R, L)$ and the operator of the transition in the opposite direction is $drive(L, R)$.

Definition 4 (Traces, Costs, Plans). *Let $\Theta = \langle S, O, T, I, S_G\rangle$ be the state space of a SAS$^+$ task $\Pi = \langle V, O, I, G\rangle$. A* trace *$\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} s^n$, where $s^i \in S$ for all $i \in \{0, 1, \dots, n\}$ and $o^i \in O$ for all $i \in \{1, 2, \dots, n\}$, is a sequence of states linked by transitions, where s^i is the state in the i^{th} position of the trace and o^i is the operator that connects s^{i-1} to s^i, in such a way that it defines a path from the first state s^0 to the last one s^n. We use superscripts (e.g., s^i) to denote the i^{th} element of a sequence and subscripts (e.g., o_2) for identifiers. Similarly, a backward trace $\tau^r = p^n \xrightarrow{o^n} p^{n-1} \xrightarrow{o^{n-1}} \dots \xrightarrow{o^1} p^0$, where p^i are partial states for all $i \in \{0, 1, \dots, n\}$ and $o^i \in O$ for all $i \in \{1, 2, \dots, n\}$, is a sequence of partial states linked by transitions in regression. A sub-trace $\tau[i, j]$ is a chunk of the full trace τ starting at s^i and ending at s^j. For instance, $\tau[1, 3] = s^1 \xrightarrow{o^2} s^2 \xrightarrow{o^3} s^3$ and $\tau^r[0, 1] = p^1 \xleftarrow{o^1} p^0$. Also, $\tau[i] = s^i$ and $\tau^r[i] = p^i$.*

A plan *π in Θ for s is a sequence of operators $\langle o^1, o^2, \dots, o^n\rangle$, where $\tau = s \xrightarrow{o^1} s^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} s^n$ reaches a goal state $s^n \in S_G$. The cost of π is the summed up cost of its operators. A plan for Π is a plan for I. A trace $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} s^n$ is a* plan trace *for Π if $s^0 = I$ and $s^n \in S_G$. A plan is optimal if its cost is minimal among all plans.*

The goal distance from a state s to the closest goal state, $h^*(s)$, is the cost of an optimal plan for s, or ∞ if no plan exists.

An optimal plan (with cost 6) for our running example task would be $\langle drive(t, L), pickup(p, t, L), pickup(q, t, L), drive(t, R), drop(p, t, R), drop(q, t, R)\rangle$. A common approach to find optimal plans is to use an optimal search algorithm like A* with an admissible heuristic (Hart et al. 1968).

Definition 5 (Heuristic). *A heuristic for a SAS$^+$ task $\Pi = \langle V, O, I, G\rangle$ with a state space $\Theta = \langle S, O, T, I, S_G\rangle$ is an estimate of the distance to the closest goal state defined as a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$. A heuristic h is called:*

- *Perfect if $h(s) = h^*(s)$ for all states $s \in S$.*
- *Goal-aware if $h(s) = 0$ for all states $s \in S_G$.*
- *Consistent if $h(s') \leq h(s) + \text{cost}(o)$ for all states s, s' such that $\langle s, o, s'\rangle \in T$. Consistent heuristics do not reopen nodes in A^*.*
- *Admissible if $h(s) \leq h^*(s)$ for all states $s \in S$. Heuristics that are both consistent and goal-aware, are also admissible.*

2.2 Abstractions

Definition 6 (Abstraction). *An abstraction α for a transition system $\Theta = \langle S, O, T, I, S_G\rangle$ is a surjective function $\alpha : S \to S^\alpha$, where S^α is a finite set of abstract states.*

Similarly to partial states, we denote by $[\![s^\alpha]\!]$ the set of concrete states consistent with an abstract state s^α, that is, the set of states that the function α maps to s^α, $[\![s^\alpha]\!] = \{s \mid s \in S, \alpha(s) = s^\alpha\}$. Each abstract state $s^\alpha \in S^\alpha$ is identified with the set of concrete states consistent with it. We say that a concrete state s is mapped to an abstract state s^α if s is consistent with s^α.

Definition 7 (Abstract State Space). *The abstract state space $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, I^\alpha, S_G^\alpha\rangle$ of an abstraction α for a concrete state space $\Theta = \langle S, O, T, I, S_G\rangle$ is a homomorphism of Θ, i.e., if a transition $s \xrightarrow{o} t$ exists in the concrete set of transitions T, then a transition $\alpha(s) \xrightarrow{o} \alpha(t)$ exists in the abstract set of transitions T^α. Also, S^α is the set of all abstract states mapped by α, $I^\alpha = \alpha(I)$, and $S_G^\alpha = \{\alpha(s) \mid s \in S_G\}$.*

In this paper, we only consider induced abstractions, whose state space is a *strict* homomorphism of the transition system, i.e., $T^\alpha = \{\alpha(s) \xrightarrow{o} \alpha(t) \mid s \xrightarrow{o} t \in T\}$. That is, each abstract state s^α is consistent with a set of concrete states $[\![s^\alpha]\!]$, and a transition $a_i \xrightarrow{o} a_j$ exists in the abstraction if o is applicable in any of the concrete states consistent with a_i, and reaches another concrete state consistent with a_j. Figure 4 represents an abstraction over the state space represented in Figure 3, where concrete states are grouped into abstract states (represented by coloured bubbles). This abstraction ignores the position of the truck in all states and, only in a_5, whether v_p is in the truck or R is also ignored. Because the position of the truck is not taken into account, no *drive* operator is needed to reach another abstract state in the abstraction, so all *drive* operators are self-loops. Another interesting observation is that the abstract initial state includes another concrete state, and that the abstract goal state includes non-goal concrete states in addition to the goal states. To better understand the homomorphism, we can focus on transitions between two abstract states, like a_0 and a_1: a_0 reaches a_1 with the operator $pickup(p, L)$ because the state $\langle p_L, q_L, t_L\rangle$ reaches a state consistent with a_1. The same happens in the opposite direction for the operator $drop(p, L)$.

We aim to get abstract state spaces several orders of magnitude smaller than the state space they represent, so the distance of each abstract state to the closest abstract goal state can be optimally computed. Each abstraction induces a heuristic function where $h^\alpha(s)$ is $h^*(\alpha(s))$. The homomorphism property of abstractions makes this heuristic consistent and goal-aware, and thus admissible (Helmert, Haslum, and Hoffmann 2008).

2.3 Cartesian Abstractions

Definition 8 (Cartesian Abstraction). *A Cartesian abstraction $\alpha : S \to S^\alpha$, for a task $\Pi = \langle V, O, I, G\rangle$ is a type of abstraction where the set of states $[\![s^\alpha]\!]$ is Cartesian for all abstract states $s^\alpha \in S^\alpha$ (Seipp and Helmert 2018).*

A set of states is Cartesian if it is of the form $A_1 \times A_2 \times \cdots \times A_n$, where $A_i \subseteq \mathcal{D}_{v_i}$ for all $v_i \in V$.

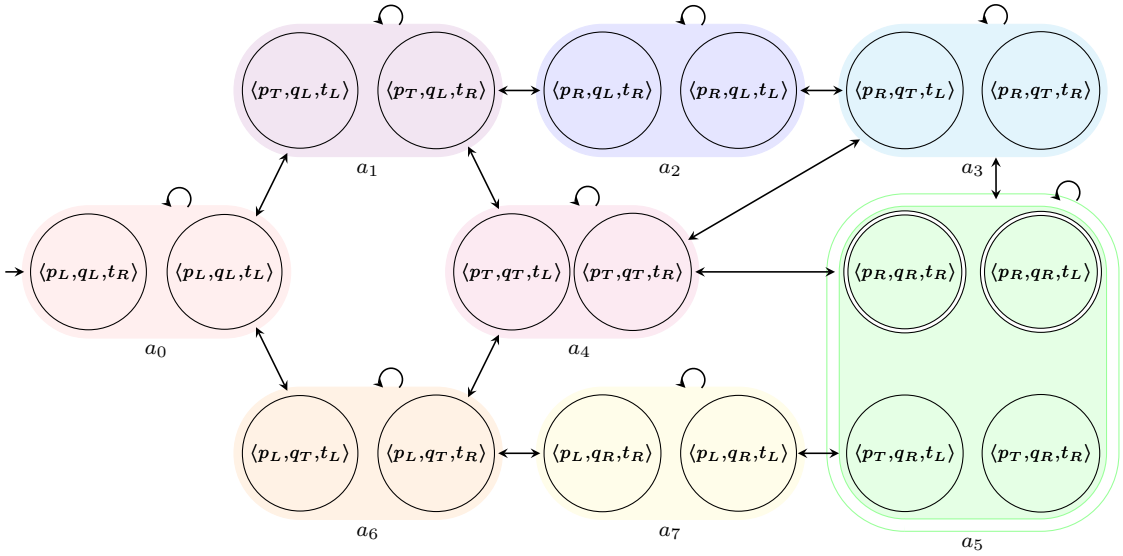


Fig. 4. Example of an abstraction for our running example. Operators labels are omitted for the sake of clarity.

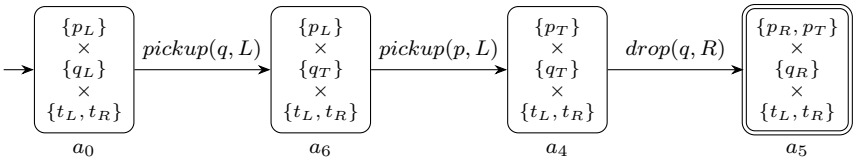


Fig. 5. Abstract plan over the abstraction of Figure 4 with flaws beyond the first flaw in progression and regression.

Definition 9 (Cartesian Representation). *A Cartesian representation for a task* $\Pi = \langle V, O, I, G\rangle$ *is a tuple of sets* $\langle A_1, A_2, \ldots, A_n\rangle$*, where* $A_i \subseteq \mathcal{D}_{v_i}$ *for all variables* $v_i \in V$.

A Cartesian representation a represents the Cartesian set $[\![a]\!] = A_1 \times A_2 \times \cdots \times A_n$. We say that a is empty if $[\![a]\!] = \emptyset$, i.e., iff $a[v] = \emptyset$ for any variable $v \in V$. We use the empty set ($\emptyset$) symbol also to represent empty Cartesian representations.

We use the notation s^α for the Cartesian representation of an abstract Cartesian state $s^\alpha \in S^\alpha$ and $[\![s^\alpha]\!]$ for the Cartesian set of states consistent with s^α.

The Cartesian representation allows us to store Cartesian sets with a linear space complexity in the number of facts even if they contain exponentially many states. However, in the figures we use a Cartesian sets notation for the sake of clarity. Given a Cartesian representation $a = \langle A_1, A_2, \ldots, A_n\rangle$, we denote by $a[v_i]$ the set of values that v_i can take in a, i.e., $a[v_i] = A_i \subseteq \mathcal{D}_{v_i}$. As an example, Figure 5 shows a plan for the Cartesian abstraction of Figure 4, where, for example, $a_0[v_p] = \{p_L\}$, $a_0[v_q] = \{q_L\}$ and $a_0[v_t] = \{t_L, t_R\}$. Also, for any (partial) state p, we can build a Cartesian representation $C(p)$ such that $[\![C(p)]\!] = [\![p]\!]$, by making $C(p)[v] = \{p[v]\}$ if $v \in \text{vars}(p)$ and $C(p)[v] = \mathcal{D}_v$ otherwise.

The intersection of two Cartesian representations c_1 and c_2 is another Cartesian representation $c' = c_1 \cap c_2$, where $c'[v] = c_1[v] \cap c_2[v]$ for all $v \in V$. The intersection of two Cartesian sets is Cartesian, and $[\![c_1]\!] \cap [\![c_2]\!] = [\![c_1 \cap c_2]\!]$.[1] Similarly, we define the subset ($\subseteq$) and strict subset ($\subset$) operations of non-empty Cartesian representations c_1 and c_2 as $c_1 \subseteq c_2$ iff $c_1[v] \subseteq c_2[v]$ for all $v \in V$, $c_1 \subset c_2$ iff $c_1 \subseteq c_2$ and $c_1[v] \subset c_2[v]$ for some $v \in V$. Also, $c_1 \subseteq c_2$ iff $[\![c_1]\!] \subseteq [\![c_2]\!]$ and $c_1 \subset c_2$ iff $[\![c_1]\!] \subset [\![c_2]\!]$.

An operator o is applicable in a Cartesian representation c if $c \cap C(pre(o)) \neq \emptyset$, and the result of applying o in c is another Cartesian representation $progr(c, o)$ where $progr(c, o)[v] = \{post(o)[v]\}$ if $v \in \text{vars}(post(o))$ and $progr(c, o)[v] = c[v]$ otherwise. In the resulting Cartesian representation, the variables of effects and preconditions of o have a single value. Note that this corresponds to all states reachable by applying o in any state in $[\![c]\!]$: $[\![progr(c, o)]\!] = \{s' \mid s \in [\![c]\!] \wedge s \xrightarrow{o} s'\}$. An operator o is applicable in regression in c if $c \cap C(post(o)) \neq \emptyset$, and the result of applying o in regression in c is another Cartesian representation $regr(c, o)$ where $regr(c, o)[v] = \{pre(o)[v]\}$ for $v \in \text{vars}(pre(o))$, $regr(c, o)[v] = \mathcal{D}_v$ for $v \in \text{vars}(\mathit{eff}(o)) \setminus \text{vars}(pre(o))$, and $regr(c, o)[v] = c[v]$ otherwise.

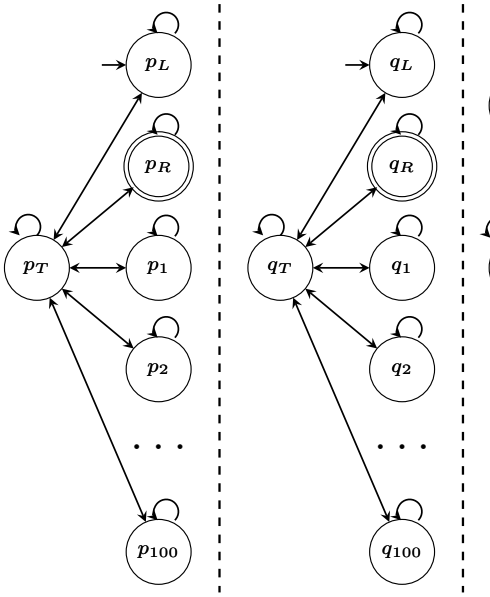
Cartesian abstractions are very flexible because they allow abstracting each variable with a different granularity at each abstract state. Indeed, they generalise domain abstractions (Hernádvölgyi and Holte 2000; Kreft et al. 2023), where the value of a variable is abstracted in all states or in none of them, which, at the same time, generalise Pattern Databases (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001), a heuristic that is based on the combination of abstractions where each variable can be fully abstracted or not abstracted at all (also called projections). Merge-and-shrink abstractions (Helmert, Haslum, and Hoffmann 2007; Helmert, Haslum, Hoffmann, and Nissim 2014; Torralba and Sievers 2019) is the other type of abstractions used for Classical Planning and generalise Cartesian abstractions, but Cartesian abstractions have achieved better results recently (Seipp 2018, 2023).

Consider our running example but with 100 additional locations not involved in the initial position of p, q or t and not mentioned in the goals. Figure 6a shows a PDB with a projection that abstracts all variables except one for this task, which are the atomic projections from which the merge-and-shrink procedure starts. Figure 6b shows a domain abstraction where the domain of v_t is fully abstracted and the domains of v_p and v_q are reduced to three values that differentiate L and R (each irrelevant location can be mapped to any of these values resulting in the same abstraction) and being in the truck. Finally, Figure 6c shows a Cartesian abstraction for this task. A PDB can only fully abstract variables or not abstract them at all, so they cannot ignore only the locations not interesting for this task. Domain abstractions can abstract some values of each variable along the full abstraction, so for this task they can ignore the locations not necessary to solve the task at the same level as Cartesian abstractions. However, domain abstractions cannot abstract a variable with different granularity for each abstract state, so the best domain abstractions completely ignore *drive* operators if the position of the truck is fully ignored (Figure 6b) or result in the complete transition system of our running example shown in Figure 3 on page 5, if the domain of v_t is reduced to two values that differentiate L from R, with each irrelevant location mapped to any of them, with ten more states than the Cartesian abstraction of Figure 6c, and two goal abstract states instead of only one.

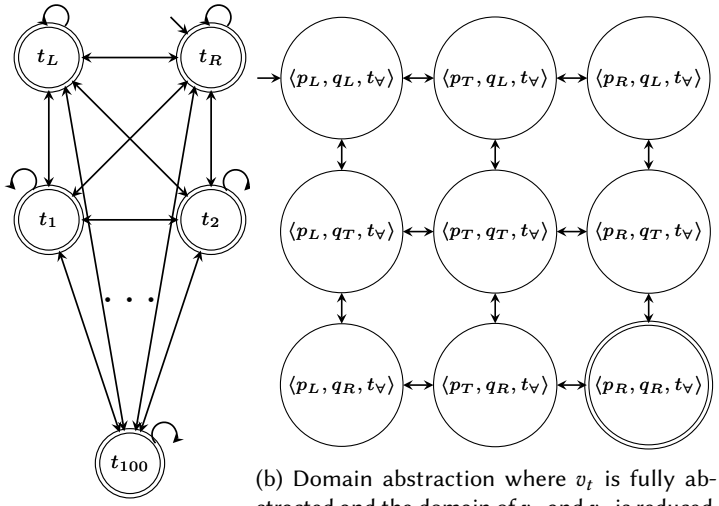
2.4 Counterexample-Guided Abstraction Refinement

The only technique to build Cartesian abstractions with any granularity to the best of our knowledge is CEGAR (Counterexample-Guided Abstraction Refinement) (Seipp and Helmert 2013, 2018), shown in Algorithm 1. It starts with the trivial abstraction, which consists of a single abstract state a where $a[v] = \mathcal{D}_v$ for all $v \in V$. Then, the abstraction is iteratively refined until solving the problem either by finding an optimal plan (line 7), or proving the task unsolvable (an abstract plan cannot be found, line 4). It can also be stopped by a termination condition (typically a time or memory limit), resulting in a Cartesian abstraction that induces a heuristic (lines 1 and 9).

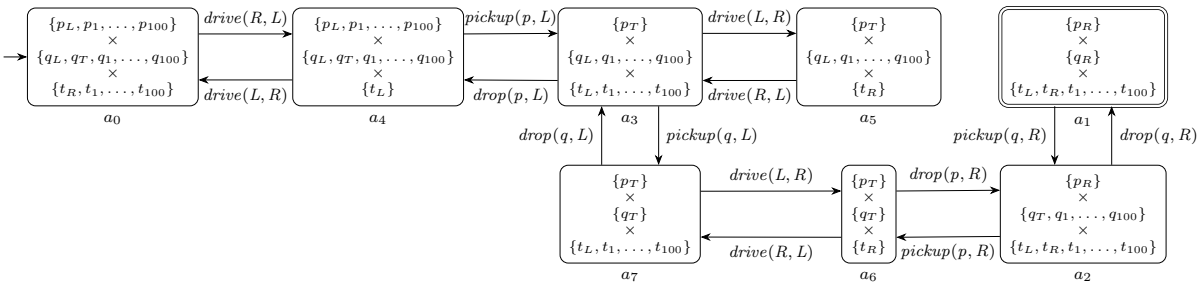
[1] Note that operations such as the union or complement are not defined over Cartesian representations, as the union of two Cartesian sets or the complement of a Cartesian set are not always Cartesian sets.



(a) PDB with three projections.



(b) Domain abstraction where v_t is fully abstracted and the domain of v_p and v_q is reduced to locations L and R, with irrelevant locations mapped to any of them.



(c) Cartesian abstraction.

Fig. 6. PDB, domain abstraction and Cartesian abstraction for our running example with 100 additional irrelevant locations.

Definition 10 (Abstract Traces and Abstract Plan Traces). *An* abstract trace *for an abstraction α with an abstract transition system $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, I^\alpha, S^\alpha_G \rangle$ is a trace $\tau^\alpha = a^0 \xrightarrow{o^1} \ldots \xrightarrow{o^n} a^n$ in Θ^α. It is an abstract plan trace if $a^0 = I^\alpha$ and $a^n \in S^\alpha_G$. It is an optimal abstract plan trace if the abstract plan $\langle o^1, o^2, \ldots, o^n \rangle$ is optimal.*

Similarly to traces, an abstract sub-trace $\tau^\alpha[i, j]$ is a chunk of the full abstract trace τ^α starting at a^i and ending at a^j. For instance, $\tau^\alpha[1, 3] = a^1 \xrightarrow{o^2} a^2 \xrightarrow{o^3} a^3$. Also, $\tau^\alpha[i]$ denotes the abstract state a^i.

Algorithm 1: CEGAR main algorithm (adapted from Seipp and Helmert (2018))

Data: Task $\Pi = \langle V, O, I, G \rangle$, initial abstraction $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, I^\alpha, S^\alpha_G \rangle$
Result: unsolvable, concrete plan, final abstraction

while *not TerminationCondition()* **do**
  $\tau^\alpha \leftarrow$ FindOptimalTrace(Θ^α)
  if τ^α = *"no trace"* **then**
    return *task is unsolvable*
  $\varphi \leftarrow$ FindFlaw(Π, τ^α)
  if φ = *"no flaw"* **then**
    return *plan extracted from* τ^α
  $\Theta^\alpha \leftarrow$ Refine(Θ^α, φ)
return Θ^α

Algorithm 2: FindFlaw (adapted from Seipp and Helmert (2018))

Data: Task $\Pi = \langle V, O, I, G \rangle$, abstract plan trace τ^α
Result: Progression Flaw ⟨abstract state, concrete state, Cartesian representation⟩ or "no flaw"

$s \leftarrow I$
$b \leftarrow I^\alpha$
forall $a \xrightarrow{o} b \in \tau^\alpha$ **do**
  if o *is not applicable in* s **then**
    return $\langle a, s, a \cap C(pre(o)) \rangle$
  $s' \leftarrow progr(s, o)$
  if $s' \notin [\![b]\!]$ **then**
    return $\langle a, s, a \cap C(regr(b, o)) \rangle$
  $s \leftarrow s'$
if $s \notin [\![G]\!]$ **then**
  return $\langle b, s, b \cap C(G) \rangle$
return "no flaw"

The loop shown in Algorithm 1 finds an optimal abstract plan trace $\tau^\alpha = a^0 \xrightarrow{o^1} \ldots \xrightarrow{o^n} a^n$ in the abstraction and tries to execute it in the concrete state space resulting in a trace $\tau = s^0 \xrightarrow{o^1} \ldots \xrightarrow{o^k} s^k$, where $s^0 = I$ and $k \in \{1, \ldots, n\}$ such that o^k is the last operator from τ^α that is applicable in s^{k-1}. Note that τ is unique because it starts in a well-defined state (I) and operators are deterministic in Classical Planning tasks.

Algorithm 2 executes each operator of the abstract plan trace over the concrete state space starting in I. This execution is stopped when an operator is not applicable or the target concrete state is not mapped to the successor abstract state. These are two of the conditions that serve to identify a flaw, as described in Definition 11. A third one happens when the full trace is executed but the final concrete state is not a goal state. We call them explicitly "progression flaws". Previous work defines a flaw as a tuple of a concrete state and a Cartesian set, while we identify them as a tuple of an abstract state, a concrete state and a Cartesian representation, because the flaw is actually a property of the abstract plan trace in a specific point of that trace.

Algorithm 3: Refine (adapted from Seipp and Helmert (2018))

Data: Abstraction $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, I^\alpha, S_G^\alpha \rangle$, flaw $\varphi = \langle a, s, c \rangle$
Result: Refined abstraction $\Theta^{\alpha'}$

```
1  S^α' ← S^α, T^α' ← T^α, I^α' ← I^α, S_G^α' ← S_G^α
2  ⟨d, e⟩ ← ChooseSplit(a, s, c)
3  S^α' ← (S^α' \ {a}) ∪ {d, e}
4  T^α' ← RewireTransitions(T^α', a, d, e)
5  if a ∈ S_G^α' then
6  |  S_G^α' ← (S_G^α' \ {a})
7  for child ∈ {d, e} do
8  |  if I ∈ [[child]] then
9  |  |  I^α' ← child
10 |  if child ∩ C(S_G) ≠ ∅ then
11 |  |  S_G^α' ← S_G^α' ∪ {child}
12 return Θ^α' = ⟨S^α', O, T^α', I^α', S_G^α'⟩
```

Definition 11 (Progression Flaw). *Let $\tau^\alpha = a^0 \xrightarrow{o^1} \dots \xrightarrow{o^n} a^n$ be an abstract plan trace of an abstraction α for a transition system $\Theta = \langle S, O, T, I, S_G \rangle$. Let $\tau = s^0 \xrightarrow{o^1} \dots \xrightarrow{o^k} s^k$ be a trace in Θ, where $s^0 = I$ and k is a value lower or equal than n such that $k = n$ or o^{k+1} is inapplicable in s^k and such that $s^i \in [\![a^i]\!]$ for all $0 \leq i \leq k$. A* progression flaw *in τ^α is a tuple $\langle a^k, s^k, c \rangle$ of an abstract state a^k in τ^α, a concrete state s^k, and a Cartesian representation $c \subset a^k$ for which one of the following cases apply:*

(1) $k < n$ and o^{k+1} is inapplicable in s^k. c is the Cartesian representation subset of a^k in which o^{k+1} is applicable, i.e., the intersection between a^k and the Cartesian representation of the preconditions of o^{k+1}. Formally, $c = a^k \cap C(pre(o^{k+1}))$.

(2) $k < n$ and $progr(s^k, o^{k+1})$ is not mapped to a^{k+1}. c is the Cartesian representation subset of a^k that reaches a^{k+1} after applying o^{k+1}, i.e., the intersection between a^k and the Cartesian representation of $regr(a^{k+1}, o^{k+1})$. Formally, $c = a^k \cap C(regr(a^{k+1}, o^{k+1}))$.

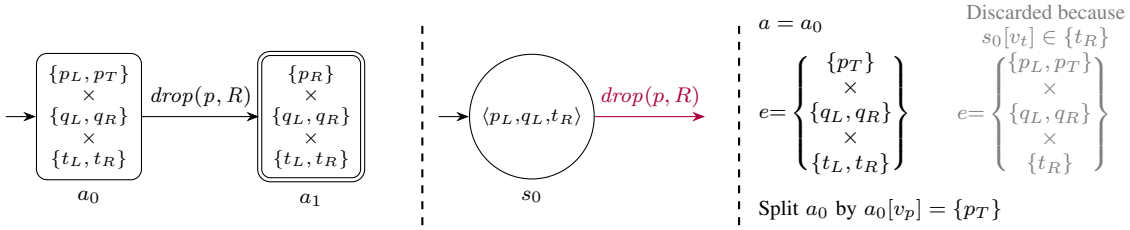
(3) $k = n$ and s^n is not a goal state. c is the intersection between a^n and the Cartesian representation of the goal partial state, i.e., $c = a^k \cap C(G)$.

Semantically, a progression flaw is a tuple of an abstract state a^i, a counterexample concrete state s^i where the abstract plan trace τ^α does not work as intended, and a Cartesian representation (subset of a^i) c in which the flaw does not exist (o^{i+1} is applicable, o^{i+1} only reaches states mapped to a^{i+1}, or a^i only contains goal states).

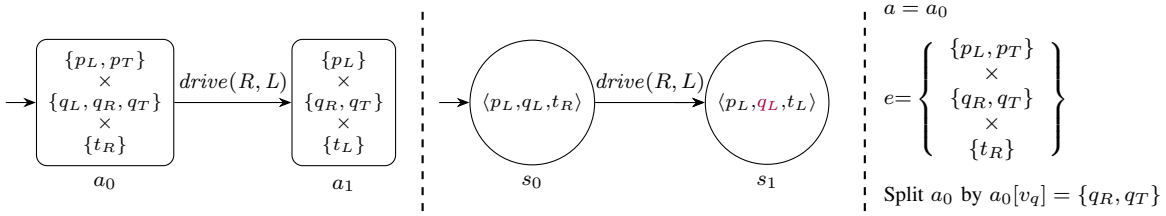
For example, in Figure 5 on page 7 $\tau^\alpha = a_0 \xrightarrow{pickup(q,L)} a_6 \xrightarrow{pickup(p,L)} a_4 \xrightarrow{drop(q,R)} a_5$. Trying to execute this trace in the state space (Figure 3 on page 5) shows that the first operator $pickup(q, L)$ is not applicable because $v_t \mapsto R$ instead of L, so the concrete plan trace in this case is simply I.

If this execution succeeds and $s^k \in S_G$, then it is an optimal plan for the task (Algorithm 1, line 7). If some flaw $\langle a, s, c \rangle$ is found, it is repaired by Algorithm 3, which first chooses a split of the flaw (line 2).

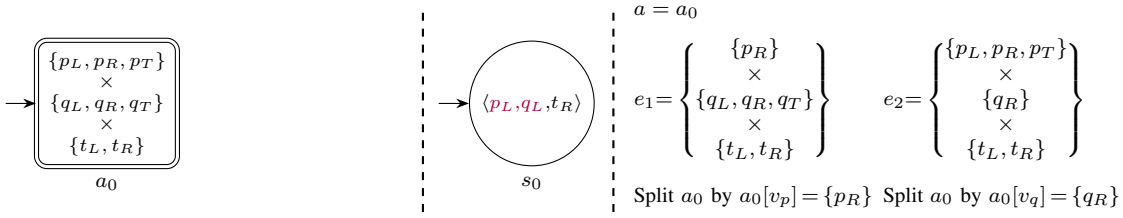
Definition 12 (Progression Flaw Split). *A split of a progression flaw $\langle a, s, c \rangle$, where a is an abstract state of a Cartesian abstraction α, s is a concrete state, and c is a Cartesian representation such that $c \subset a$, is a pair of Cartesian representations d and e such that $d \subset a$, $e \subset a$, $s \in d$, $c \subseteq e$, $[\![d]\!] \cup [\![e]\!] = [\![a]\!]$, and $d \cap e = \emptyset$. We call a the parent state and d and e the children states.*



(a) Operator is not applicable.



(b) Deviation.



(c) The final state is not a goal state.

Fig. 7. Example of each type of flaw for different abstractions in the running example. The left side of the figures is the abstract plan (the abstraction may have more states, like in the second one) and the centre part is the execution of the plan on the state space. The child abstract state e after each split is on the right side.

Given the nature of Cartesian representations, splits can only separate values from one variable at a time. We call this variable the split variable v_s. *Thus,* $d[v] = e[v] = a[v]$ *for all variables* $v \in V \setminus \{v_s\}$, $d[v_s] \cap e[v_s] = \emptyset$, *and* $a[v_s] = d[v_s] \cup e[v_s]$.

Note that multiple splits may exist for a split variable v_s, since there are values of $a[v_s]$ included in neither $c[v_s]$ nor $\{s[v_s]\}$. Also note that this definition of split only considers a split of a if it separates the counterexample s in a new abstract state (d) different from the new abstract state for c (e), so a split is never applied in a split variable where $s[v_s] \in c[v_s]$. A split selection strategy is a criterion to choose a split among all possible splits for a flaw in an abstract state a (Seipp and Helmert 2013, 2018). We discuss the strategies to split values for a split variable in Section 4.3 and the split selection strategies used in this paper in Sections 6.2 and 7. Thus, Algorithm 3 splits the abstract state of the plan trace where the flaw has been found into two abstract states d

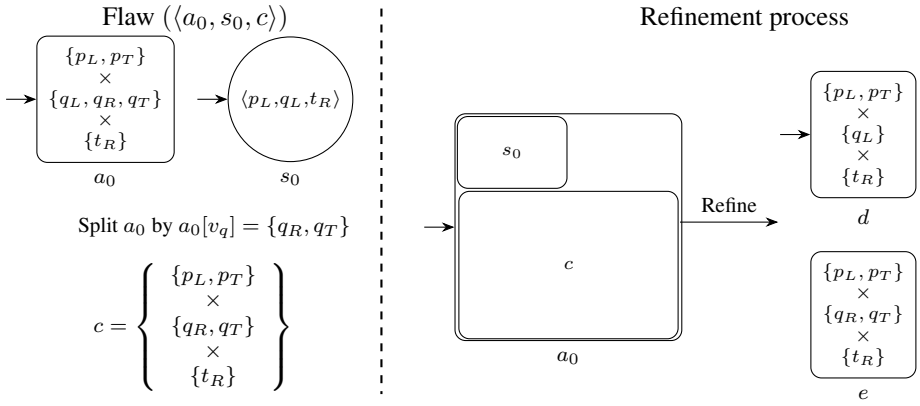


Fig. 8. Refinement of the flaw shown in Figure 7b (loops omitted for the sake of clarity).

and e with $s \in [\![d]\!]$ and $c \subseteq e$, in such a way that the condition of the flaw cannot happen again. Note, however, that the same condition for a flaw may remain in e when splits in different split variables exist, e.g., if multiple preconditions of an operator are not satisfied before the split. However, after this refinement it is possible that the state e is not involved in an abstract optimal plan any more, and so it is usually more efficient to repair it in later iterations of the loop if it is ever found again. Then, transitions are rewired in such a way that incoming transitions of the abstract state can be directed to d, e or both, outgoing transitions can be originated in d, e or both, and self-looping transitions can remain as self-looping transitions or be directed between d and e. Finally, the new abstract initial state is set as d if the parent state was I^α because then $s = I$ and $s \in [\![d]\!]$ and d and e are added to the set of abstract goal states if the Cartesian set that they represent contains a concrete goal state.

The three conditions for progression flaws are represented in Figure 7 for different example abstract plan traces. In Figure 7a, it is clear that the operator is not applicable because the package p is not in the truck, so c is in this case the subset of a_0 where $v_p \mapsto p_T$ and $v_t \mapsto t_R$ (the preconditions of $drop(p, R)$), but we identify a single split because $s_0[v_t] \in c[v_t]$, and so splitting by v_t would put t_R in the child e and then s would not be consistent with the child d, which is not compatible with the definition of a progression flaw split. In Figure 7b, the operator is applicable but it reaches a concrete state not mapped to a_1 because $q_L \notin \{q_R, q_T\}$, so c is the subset of a_0 where $v_q \mapsto q_R$ or $v_q \mapsto q_T$ for this flaw. In Figure 7c, the abstract trace contains no operators, so it is executable in the concrete state space, but goals are not satisfied in the initial state, so c is the subset of a_0 where the goals are fulfilled. Note that this flaw is always found in the trivial abstraction (the initial abstract state that contains all values for all variables) if goals are not satisfied in the initial state, but it may also exist in other abstractions if abstract goal states contain non-goal states, as shown in Figure 4.

The refinement process for the flaw of Figure 7b is illustrated in Figure 8. In this example, there is only one possible split, $q = \{q_R, q_T\}$. $I \notin [\![c]\!]$, as otherwise it would not be a flaw. The refinement splits a_0 in two states d and e, in such a way that $I \in [\![d]\!]$ and $c \subseteq e$. Note that d has become the new initial state of the abstraction. In this case, $c = e$, but the values of the split variable neither present in s_0 nor in c ($\emptyset$ in this example), can be assigned to any of the two states, so e is a superset of c when some of these values are assigned to e. However, the original implementation always puts them in d with the aim of getting larger heuristic values (Seipp and Helmert 2018). c is also a superset of e when multiple conditions for a flaw exist in different variables, since a split can be done only in one of them.

3 Flaws of Abstract Plans in a Cartesian Abstraction

This work focuses on identifying alternative definitions of flaws in Cartesian abstractions in order to improve the quality of the refinements applied in the CEGAR loop. That is, we keep the main structure of the CEGAR algorithm (Algorithm 1 on page 10), but change the subprocedure FindFlaw replacing Algorithm 2 (page 10) with other alternatives. Our new subprocedures may return multiple flaws, in which case the ChooseSplit method will have further possibilities to choose from, as discussed in Section 6.

First, we show the properties needed for a flaw to preserve completeness and soundness of Algorithm 1.

Theorem 1. *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, I^\alpha, S^\alpha_G\rangle$, and τ^α an abstract plan trace found in Θ^α. Algorithm 1 with no termination condition and unlimited time and memory is sound and complete if whenever the subprocedure FindFlaw returns "no flaw" for τ^α then $\langle o^1, \ldots, o^n\rangle$ is a plan for Θ and otherwise returns any flaw $\langle a, c_1, c_2\rangle$, where $a \in S^\alpha$ is an abstract state, and c_1 and c_2 are Cartesian representations s.t. $c_1 \neq \emptyset$, $c_2 \neq \emptyset$, $c_1 \subset a$, $c_2 \subset a$, $c_1 \cap c_2 = \emptyset$.*

Proof. Algorithm 1 is complete if it always returns an answer. Without termination condition, the loop of line 1 can only terminate with a return statement, so we prove that any of them is always reached. If no abstract plan can be found, Algorithm 1 returns that no plan exists at line 4. If FindFlaw returns a flaw $\langle a, c_1, c_2\rangle$, as c_1 and c_2 are disjoint and not empty, a can be split into two Cartesian representations d and e such that $d \cap e = \emptyset$, $d \subset a$, and $e \subset a$ by splitting from $a[v_s]$ the values $c_1[v_s]$ or $c_2[v_s]$ for some variable $v_s \in V$ s.t. $c_1[v_s] \subset a[v_s]$ or $c_2[v_s] \subset a[v_s]$ respectively, keeping the strict homomorphism property of the abstraction, and the execution continues with no issue. Such a variable v_s exists because c_1 and c_2 are strict subsets of a. Eventually, if no flaw is found, Algorithm 1 returns a solution at line 7.

Algorithm 1 is sound if whenever it returns an answer, it is correct. If no solution exists, an abstract plan trace cannot be found, which is guaranteed by the strict homomorphism property of the abstract transition system, so Algorithm 1 returns that no plan exists at line 4. Otherwise, a trace is found in the abstraction because it is a homomorphism of Θ. A plan is returned only when FindFlaw returns "no flaw", so soundness is guaranteed in line 7 if FindFlaw only returns "no flaw" when $\langle o^1, \ldots, o^n\rangle$ is an optimal plan for Θ, and any plan extracted from the sequence of the operators of an abstract plan trace returned by FindOptimalTrace is optimal because it is optimal in a homomorphism of Θ. Finally, as all splits keep the strict homomorphism property of the Cartesian abstraction, these arguments are valid for all iterations of the loop. □

Note that these conditions for c_1 and c_2 still allow returning a flaw even when the abstract plan is a plan for the task. However, that only defers the convergence of the algorithm, doing more refinements than necessary, but returning always an admissible heuristic induced by a Cartesian abstraction. We provide now a very general notion of flaw that generalises the previous definition but adds some requirements to the minimal properties of Theorem 1 to identify flaws in abstract states involved in the abstract plan trace $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ and return a flaw only if no concrete state s^i mapped to a^i exists for some $i \in \{0, 1, \ldots, n\}$, reducing the number of unnecessary refinements. To that end, we consider the notion of reification (Fan and Holte 2015) to identify when the abstract trace has a corresponding trace in the state space of the planning task.

Definition 13 (Reification of an Abstract Trace). *Let $\Theta = \langle S, O, T, I, S_G\rangle$ be a transition system, α a Cartesian abstraction for Θ with abstract transition system Θ^α, and $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract trace found in Θ^α. A trace $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} s^n$ in Θ **reifies** τ^α if $s^i \in [\![a^i]\!]$ for all $i \in \{0, 1, \ldots, n\}$. $traces(\tau^\alpha)$ is the set of all traces in Θ that reify τ^α. $traces_{init}(\tau^\alpha)$ is the subset of $traces(\tau^\alpha)$ starting in $s^0 = I$ and $traces_{goal}(\tau^\alpha)$ is the subset of $traces(\tau^\alpha)$ ending in some $s^n \in S_G$. Also, $S^{end}_{\tau_{init}}(\tau^\alpha) = \{s^n \mid \tau[n] = s^n, \tau \in traces_{init}(\tau^\alpha)\}$ is the set of states ending a trace in $traces_{init}(\tau^\alpha)$, and $S^{start}_{\tau_{goal}}(\tau^\alpha) = \{s^0 \mid \tau[0] = s^0, \tau \in traces_{goal}(\tau^\alpha)\}$ is the set of states starting a trace in $traces_{goal}(\tau^\alpha)$.*

We say that τ^α is:

- ***Init-reifiable*** *if $traces_{init}(\tau^\alpha) \neq \emptyset$, so there exists a trace that reifies it starting at $s^0 = I$.*
- ***Goal-reifiable*** *if $traces_{goal}(\tau^\alpha) \neq \emptyset$, so there exists a trace that reifies it ending at $s^n \in S_G$.*
- ***Mappable*** *if $traces_{init}(\tau^\alpha) \cap traces_{goal}(\tau^\alpha) \neq \emptyset$, so there exists a trace that reifies it starting at $s^0 = I$ and ending at $s^n \in S_G$.*

Note that, if τ^α is mappable, $\langle o^1, \ldots, o^n \rangle$ is a plan for the task. However, the opposite is not always true, as $\langle o^1, \ldots, o^n \rangle$ could be a plan for the task that passes through states not mapped into the abstract states in τ^α.

Definition 14 (Abstract Flaw). *Let Π be a planning task with transition system $\Theta=\langle S, O, T, I, S_G \rangle$, α a Cartesian abstraction for Θ that induces the transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract plan trace in Θ^α, and $i = \{0, 1, \ldots, n\}$ an index in τ^α. $\langle a^i, \overrightarrow{c}, \overleftarrow{c} \rangle$ is an abstract flaw of τ^α if $\overrightarrow{c}$ and $\overleftarrow{c}$ are Cartesian representations s.t.:*

(AF.1) $\overrightarrow{c} \neq \emptyset$
(AF.2) $\overleftarrow{c} \neq \emptyset$
(AF.3) $\overrightarrow{c} \subset a^i$
(AF.4) $\overleftarrow{c} \subset a^i$
(AF.5) $\overrightarrow{c} \cap \overleftarrow{c} = \emptyset$
(AF.6) $S^{end}_{\tau_{init}}(\tau^\alpha[0, i]) \subseteq [\![\overrightarrow{c}]\!]$
(AF.7) $S^{start}_{\tau_{goal}}(\tau^\alpha[i, n]) \subseteq [\![\overleftarrow{c}]\!]$.

In words, $\overrightarrow{c}$ and $\overleftarrow{c}$ are non-empty disjoint subsets of a^i such that $[\![\overrightarrow{c}]\!]$ contains all final states of init-reifiable traces in $\tau^\alpha[0, i]$ and $[\![\overleftarrow{c}]\!]$ contains all states goal-reifiable by the sub-trace $\tau^\alpha[i, n]$.

This definition is slightly generic, as any disjoint subsets of a^i that represent supersets of $S^{end}_{\tau_{init}}(\tau^\alpha[0, i])$ and $S^{start}_{\tau_{goal}}(\tau^\alpha[i, n])$ can be used to define a flaw. Nevertheless, Definition 14 offers a minimal definition, common to all our notions of flaw. This is sufficient to show that (1) it guarantees the correctness of the overall CEGAR algorithm, since every trace that does not correspond to a valid plan has at least a flaw that can be used to refine the abstraction by splitting an abstract state; and (2) ensures that if the abstract plan is mappable there is no flaw and the CEGAR algorithm terminates without further refining the abstraction.

Typically, it is also desirable that $\overleftarrow{c}$ and $\overrightarrow{c}$ only contain states consistent with a^i that are in $S^{start}_{\tau_{goal}}(\tau^\alpha[0, i])$ and $S^{end}_{\tau_{init}}(\tau^\alpha[i, n])$ either in the concrete state space or in some relaxation of it, since they are the states where the flaw does not exist. Throughout the paper, we consider more restrictive definitions of flaw that satisfy this definition.

Theorem 2. *Let Θ be a transition system, α a Cartesian abstraction for Θ with abstract transition system Θ^α, and $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α. τ^α is mappable iff there is no abstract flaw in τ^α.*

Proof. On the one hand, if τ^α is not mappable, some abstract flaw can be found. Since τ^α is not mappable, it follows that $traces_{init}(\tau^\alpha) \cap traces_{goal}(\tau^\alpha) = \emptyset$. Therefore, $S^{end}_{\tau_{init}}(\tau^\alpha[0, i]) \cap S^{start}_{\tau_{goal}}(\tau^\alpha[i, n]) = \emptyset$ for all $i \in \{0, 1, \ldots, n\}$ (Definition 13). Also, $S^{end}_{\tau_{init}}(\tau^\alpha[0, i])$ and $S^{start}_{\tau_{goal}}(\tau^\alpha[i, n])$ are always Cartesian for all $i \in \{0, 1, \ldots, n\}$. This directly follows because the progression and regression of a Cartesian set with respect to a SAS$^+$ operator, and the intersection of two Cartesian sets are also Cartesian (Seipp and Helmert 2018). So, by induction, $S^{end}_{\tau_{init}}(\tau^\alpha[0, 0]) = \{I\}$ and $S^{end}_{\tau_{init}}(\tau^\alpha[0, i{+}1])$ is the result of applying progression and intersecting with $[\![a^i]\!]$. Similarly, $S^{start}_{\tau_{goal}}(\tau^\alpha[n, n]) = [\![G]\!]$ is Cartesian, and $S^{start}_{\tau_{goal}}(\tau^\alpha[i, n])$ is the result of applying regression on $S^{start}_{\tau_{goal}}(\tau^\alpha[i + 1, n])$ and intersecting with $[\![a^i]\!]$.

Now, we prove by induction in the step i of τ^α that either $S^{end}_{\tau_{init}}(\tau^\alpha[0, i]) \neq \emptyset$ or a flaw can be found at step $j < i$ for all $i \in \{0, 1, \ldots, n\}$.

For the base case, $i = 0$ and $S^{end}_{\tau_{init}}(\tau^\alpha[0,0]) = \{I\}$.

For the inductive case, $i = \{1, 2, \ldots, n\}$, it suffices to show that if $S^{end}_{\tau_{init}}(\tau^\alpha[0,i]) = \emptyset$, then we can find a flaw in a previous step. By the induction hypothesis, either we can find a flaw in some step $j < i-1 < i$ (in which case the proof is trivial), or $S^{end}_{\tau_{init}}(\tau^\alpha[0,i-1]) \neq \emptyset$. Then, we can differentiate two different cases for $j = i-1$, depending on whether $S^{start}_{\tau_{goal}}(\tau^\alpha[0,j])$ is empty.

- (Non-empty case) If $S^{start}_{\tau_{goal}}(\tau^\alpha[0,j]) \neq \emptyset$, the tuple $\langle a^i, d, e\rangle$, where $[\![d]\!] = S^{end}_{\tau_{init}}(\tau^\alpha[0,i])$ and $[\![e]\!] = S^{start}_{\tau_{goal}}(\tau^\alpha[i,n])$, is a flaw because $S^{end}_{\tau_{init}}(\tau^\alpha[0,i])$ and $S^{start}_{\tau_{goal}}(\tau^\alpha[i,n])$ are non-empty Cartesian sets and subsets of $[\![a^i]\!]$ so (AF.1)–(AF.4) are satisfied; $d \cap e = \emptyset$ because τ^α is not mappable (AF.5); and (AF.6)–(AF.7) are trivial.
- (Empty case) If $S^{start}_{\tau_{goal}}(\tau^\alpha[0,j]) = \emptyset$, then we consider whether $[\![a^j]\!] = S^{end}_{\tau_{init}}(\tau^\alpha[0,j])$.
  - If $[\![a^j]\!] = S^{end}_{\tau_{init}}(\tau^\alpha[0,j])$ then $S^{end}_{\tau_{init}}(\tau^\alpha[0,i]) \neq \emptyset$. This holds due to homomorphism property of the abstraction, as the transition $a^j \xrightarrow{o^i} a^i$, which is part of the trace, implies the existence of some transition $s^j \xrightarrow{o^i} s^i$ s.t. $s^j \in [\![a^j]\!]$, $s^i \in [\![a^i]\!]$, so $s^i \in S^{end}_{\tau_{init}}(\tau^\alpha[0,i])$.
  - If $[\![a^j]\!] \neq S^{end}_{\tau_{init}}(\tau^\alpha[0,j])$, then the tuple $\langle a^i, d, e\rangle$, where $[\![d]\!] = S^{end}_{\tau_{init}}(\tau^\alpha[0,i])$ and $[\![e]\!] = \{s\}$ s.t. s is any state $s \in [\![a^i]\!] \setminus S^{end}_{\tau_{init}}(\tau^\alpha[0,i])$, is a flaw because a single state is Cartesian, so this satisfies (AF.1)–(AF.4); $d \cap e = \emptyset$ (AF.5); and (AF.6)–(AF.7) are trivial.

For $i = n$, if no flaw can be found in a previous step, the case is the same as (Non-empty case), where the tuple $\langle a^n, d, e\rangle$, where $[\![d]\!] = S^{end}_{\tau_{init}}(\tau^\alpha[0,n])$ and $[\![e]\!] = S^{start}_{\tau_{goal}}(\tau^\alpha[n,n]) = [\![G]\!]$ is a flaw.

On the other hand, if τ^α is mappable then we prove by contradiction that it is impossible to satisfy properties (AF.1)–(AF.7) for all $i \in \{0, 1, \ldots, n\}$. Assume there is a flaw $\langle a^i, \overrightarrow{c}, \overleftarrow{c}\rangle$ in a step i. Then, to satisfy (AF.6) $S^{end}_{\tau_{init}}(\tau^\alpha[0,i]) \subseteq [\![\overrightarrow{c}]\!]$, and to satisfy (AF.7) in i, $S^{start}_{\tau_{goal}}(\tau^\alpha[i,n]) \subseteq [\![\overleftarrow{c}]\!]$. To satisfy (AF.5) at the same time, $S^{end}_{\tau_{init}}(\tau^\alpha[0,i]) \cap S^{start}_{\tau_{goal}}(\tau^\alpha[i,n]) = \emptyset$, but then $traces_{init}(\tau^\alpha) \cap traces_{goal}(\tau^\alpha) = \emptyset$, reaching a contradiction. □

This new definition of flaw also requires a generalisation of the definition of a split.

Definition 15 (Abstract Split). *A split of an abstract flaw $\langle a, \overrightarrow{c}, \overleftarrow{c}\rangle$ is a pair of Cartesian representations d and e such that $d \subset a$, $e \subset a$, $\overrightarrow{c} \subseteq d$, $\overleftarrow{c} \subseteq e$, $[\![d]\!] \cup [\![e]\!] = [\![a]\!]$, and $d \cap e = \emptyset$. We call a the parent state and d and e the children states.*

As for progression flaw splits (Definition 12), given the nature of Cartesian representations, splits can only separate values from one variable at a time. We call this variable the split variable *v_s. Thus, $d[v] = e[v] = a[v]$ for all variables $v \in V \setminus \{v_s\}$, $d[v_s] \cap e[v_s] = \emptyset$, and $a[v_s] = d[v_s] \cup e[v_s]$.*

Note that there exists at least one split for each flaw, since $\overrightarrow{c}$ and $\overleftarrow{c}$ are non-intersecting Cartesian representations, which implies the existence of some v_s for which $\overrightarrow{c}[v_S] \cap \overleftarrow{c}[v_S] \neq \emptyset$. We show now that the traditional definition of flaw is also compatible with Definition 14 but more restricted.

Theorem 3. *Let $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ be an abstract plan trace found in an abstraction α with the induced transition system Θ^α for a transition system Θ. If $\langle a^i, s^i, c\rangle$ is a progression flaw in τ^α, then $\langle a^i, \overrightarrow{c}{=}C(s^i), \overleftarrow{c}{=}c\rangle$ is an abstract flaw. However, not all abstract flaws $\langle a^i, \overrightarrow{c}, \overleftarrow{c}\rangle$ in τ^α correspond to progression flaws in τ^α.*

Proof. On the one hand, we show that $\langle a^i, \overrightarrow{c}{=}C(s^i), \overleftarrow{c}{=}c\rangle$ satisfies Definition 14. First, we prove that it satisfies (AF.1), (AF.3), and (AF.6). Let $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \ldots \xrightarrow{o^k} s^k$ be a plan trace in Θ such that $s^0 = I$ and $0 \leq k \leq n$ s.t. $k = n$ or o^{k+1} is inapplicable in s^k and s.t. $s^i \in [\![a^i]\!]$ for all $0 \leq i \leq k$. Then, $S^{end}_{\tau_{init}}(\tau^\alpha[0,k]) = \{s^k\}$, so (AF.6) is satisfied. (AF.1), and (AF.3) are satisfied by $C(s^k)$ because s^k is mapped to a^k, since $S^{end}_{\tau_{init}}(\tau^\alpha[0,k]) = \{s^k\}$.

Next, we prove that it satisfies (AF.2), (AF.4), (AF.5) and (AF.7). We distinguish the three cases from the definition of progression flaw (Definition 11):

(1) $k < n$ and o^{k+1} is not applicable in a state $s^k \in [\![a^k]\!]$. Then, $c = a^k \cap C(pre(o^{k+1}))$. (AF.2) and (AF.4) are satisfied because o^{k+1} must be applicable in some state consistent with a^k to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise o^{k+1} would be applicable in s^k. (AF.7) is satisfied because o^{k+1} is not applicable in any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{start}_{\tau_{goal}}(\tau^\alpha[k,n])$.

(2) $k < n$ and o^{k+1} is applicable but $progr(s^k, o^{k+1})$ is not mapped to a^{k+1}. Then, $c = a^k \cap C(regr(a^{k+1}, o^{k+1}))$. (AF.2) and (AF.4) are satisfied because o^{k+1} must reach some state consistent with a^{k+1} to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise $progr(s^k, o^{k+1}) \in [\![a^{k+1}]\!]$. (AF.7) is satisfied because $progr(s, o^{k+1}) \notin a^{k+1}$ from any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{start}_{\tau_{goal}}(\tau^\alpha[k,n])$.

(3) $k = n$ but s^n is not a goal state. Then, $c = a^n \cap C(G)$. (AF.2) and (AF.4) are satisfied because some state $s \in S_G \cap [\![a^n]\!]$ exists by the definition of abstraction, as a^n would not be an abstract goal state otherwise. (AF.5) is satisfied because s^n would be a goal state otherwise. (AF.7) is satisfied because $[\![c]\!]$ is the set of goal states in $[\![a^n]\!]$.

On the other hand, we prove by counterexample that not all abstract flaws are progression flaws: in the abstract plan of Figure 5 on page 7 an abstract flaw exists in a_4 because $S^{end}_{\tau_{init}}(a_0 \xrightarrow{pickup(q,L)} a_6 \xrightarrow{pickup(p,L)} a_4) = \emptyset$ and $\langle p_T, q_T, t_L\rangle \notin S^{start}_{\tau_{goal}}(a_4 \xrightarrow{drop(q,R)} a_5)$, and so $\langle a_4, \langle\{p_T\},\{q_T\},\{t_L\}\rangle, \langle\{p_T\},\{q_T\},\{t_R\}\rangle\rangle$ is an abstract flaw, but it is not a progression flaw because only a progression flaw exists, the one in a_0. □

Theorem 4. *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ with abstract transition system Θ^α, τ^α an abstract plan trace in Θ^α. τ^α is mappable iff there is no progression flaw in τ^α.*

Proof. On the one hand, assume $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ has no progression flaw and $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} s^n$ is a reification of τ^α. If no precondition flaw exists, all operators o^i can be executed in s^{i-1} for all $i \in \{1, 2, \ldots, n\}$. If no deviation flaw exists, then $s^i \in [\![a^i]\!]$ for all $i \in \{0, 1, \ldots, n\}$. If no goal flaw exists, $s^n \in [\![G]\!]$. Therefore, τ^α is a plan for the task and $s^i \in [\![a^i]\!]$ for all $i \in \{0, 1, \ldots, n\}$, and therefore τ^α is mappable.

On the other hand, no progression flaw can be found in τ^α if it is mappable because all progression flaws are abstract flaws and Theorem 2 proves that no abstract flaw exists in mappable abstract plan traces. □

Definition 14 provides a broad definition of flaw that lays the foundations for new more specific definitions that satisfy its properties. In the following two sections we explore some novel more restricted definitions of flaw, along the CEGAR modifications necessary to find them, that result in very different heuristics, as we show in Section 7.

4 Regression Flaws

The CEGAR algorithm has proven highly effective in the state-of-the-art planner Scorpion (Seipp 2023), and it converges to an optimal plan when used over a single abstraction of the original problem. However, we noticed that it is very biased into incrementing the heuristic value around the initial state, since refinements happen in the first flaw of the abstract plan. Also, previous work showed that, when flaws are searched over all the optimal abstract plans instead of using only one plan, refining the flaws of the plans where flaws are closer to goals usually achieves a better heuristic (Speck and Seipp 2022). In this section, we consider how to find flaws using regression from the goal, so that they are less focused on the initial state and closer to the abstract goal states.

4.1 Definition and Properties of Regression Flaws

Given an abstract plan, we define a regression flaw as an explanation of why a given suffix of the plan is not a solution for the planning task at hand. The definition is similar to that of progression flaws (Seipp and Helmert 2018). However, as the goal is not a fully specified state, regression should be done on partial states, in an attempt to cover all possible ways of reaching the goal with such an abstract plan.

Definition 16 (Regression Flaw). *Let Π be a planning task, α a Cartesian abstraction of Π, and $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ an abstract plan trace. Let $p^n = G$ and $\tau^r = p^n \xleftarrow{o^n} p^{n-1} \xleftarrow{o^{n-1}} \dots \xleftarrow{o^{k+1}} p^k$ a backward trace in Θ where $0 \leq k \leq n$ such that $k = 0$ or o^k is inapplicable in regression in p^k and such that $p^i \cap a^i \neq \emptyset$ for all $k \leq i \leq n$. A regression flaw of τ^α is a tuple $\langle a^k, c, p^k \rangle$ of a Cartesian state a^k, a Cartesian representation c such that $c \subset a^k$ and $c \neq \emptyset$, and a partial state p^k for which one of the following cases apply:*

(1) $k > 0$ and o^k is inapplicable in regression in p^k (p^k is not consistent with $post(o^k)$). Then, $c = a^k \cap C(post(o^k))$ is the subset of a^k that is applicable in regression.
(2) $k > 0$ and o^k is applicable in regression in p^k but $[\![p^{k-1}]\!] \cap [\![a^{k-1}]\!] = \emptyset$, i.e., no state in the predecessor is mapped to a^{k-1}. In this case, $c = a^k \cap C(progr(a^{k-1}, o^k))$.
(3) $k = n$ but $I \notin [\![p^0]\!]$, and then $c = C(I)$.

Now, we show that, similar to progression flaws, regression flaws are also a more restricted definition of abstract flaws in the sense that all regression flaws correspond to an abstract flaw, but the opposite is not necessary true. Therefore, regression flaws keep the CEGAR algorithm complete and sound because any mappable plan has no regression flaw, as shown in Theorem 2.

Theorem 5. *Let $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ be an abstract plan trace found in an abstraction α with the induced transition system Θ^α for a transition system Θ. If $\langle a^i, c, p^i \rangle$ is a regression flaw in τ^α, then $\langle a^i, \overrightarrow{c} = c, \overleftarrow{c} = C(p^i) \cap a^i \rangle$ is an abstract flaw. However, not all abstract flaws $\langle a^i, \overrightarrow{c}, \overleftarrow{c} \rangle$ in τ^α correspond to regression flaws in τ^α.*

Proof. On the one hand, we show that $\langle a^i, c, C(p^i) \cap a^i \rangle$ satisfies Definition 14 on page 15.

First, we prove that it satisfies (AF.2), (AF.4), and (AF.7). Let $p^n = G$ and $\tau^r = p^n \xleftarrow{o^n} p^{n-1} \xleftarrow{o^{n-1}} \dots \xleftarrow{o^{k+1}} p^k$ a backward trace in Θ where $0 \leq k \leq n$ such that $k = 0$ or o^k is inapplicable in regression in p^k and such that $p^i \cap a^i \neq \emptyset$ for all $k \leq i \leq n$. Then, $S^{start}_{\tau_{goal}}(\tau^\alpha[i, n]) = [\![z]\!]$, where $z = C(p^i) \cap a^i \subseteq a^i$, for all $k \leq i \leq n$, so (AF.7) is satisfied. (AF.2), and (AF.4) are held because $\overleftarrow{c} = z \subseteq a^i$, for all $k \leq i \leq n$ and z is not empty.

Now we consider the three cases from Definition 16:

(1) $k > 0$ and o^k is inapplicable in regression in p^k, and $c = a^k \cap C(post(o^k))$. Then, (AF.1) is satisfied because o^k must be applicable in regression in some state in a^k to induce the transition in Θ^α by the definition of induced abstraction. (AF.3) directly follows from the properties of intersection. (AF.5) is satisfied because otherwise o^k would be applicable in regression in p^k. (AF.6) is satisfied because o^k is not applicable in regression in any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{end}_{\tau_{init}}(\tau^\alpha[0, k])$.
(2) $k > 0$ and o^k is applicable in regression but $regr(p^k, o^k)$ is not consistent with a^{k-1}, and then $c = a^k \cap C(progr(a^{k-1}, o^k))$. Then, (AF.1) and (AF.3) are satisfied because o^k applied in regression in a^k must reach some state in a^{k-1} to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise o^k applied in regression in a^k would reach some state $s \in [\![p^k]\!] \cap a^k$ (no deviation flaw). (AF.6) is satisfied because o^k applied in regression in a^k does not reach any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{end}_{\tau_{init}}(\tau^\alpha[0, k])$.
(3) $k = 0$ but $I \notin [\![p^0]\!]$, and $c = C(I)$. Then, (AF.1) and (AF.3) are satisfied because $c = C(I)$ and $I \in [\![a^0]\!]$. (AF.5) is satisfied because otherwise $I \in p^0$ (no flaw). (AF.6) is satisfied because $S^{end}_{\tau_{init}}(\tau^\alpha[0]) = \{I\} = [\![c]\!]$.

On the other hand, not all abstract flaws are regression flaws. For example, in the abstract plan of Figure 5 on page 7 an abstract flaw exists in a_0 because $S^{start}_{\tau_{goal}}(a_0 \xrightarrow{pickup(q,L)} a_6 \xrightarrow{pickup(p,L)} a_4 \xrightarrow{drop(q,R)} a_5) = \emptyset$ and $\langle p_L, q_L, t_L \rangle \notin S^{end}_{\tau_{init}}(a_0)$, and so $\langle a_0, \langle \{p_L\}, \{q_L\}, \{t_R\} \rangle, \langle \{p_L\}, \{q_L\}, \{t_L\} \rangle \rangle$ is an abstract flaw, but it is not a regression flaw because the only one is in a_5 because the package p is at R after applying the $drop(q, R)$ operator in regression. □

Algorithm 4: FindRegressionFlaw

Data: Task $\Pi = \langle V, O, I, G \rangle$, abstract plan trace τ^α
Result: Regression Flaw ⟨abstract state, Cartesian representation, partial state⟩

$p \leftarrow G$
forall $b \xrightarrow{o} a \in \mathit{reverse}(\tau^\alpha)$ **do**
  if o *is not applicable in regression in* p **then**
    return $\langle a, a \cap C(\mathit{post}(o)), p \rangle$
  $p' \leftarrow \mathit{regr}(p, o)$
  if $[\![p']\!] \cap b \neq \emptyset$ **then**
    return $\langle a, a \cap C(\mathit{progr}(b, o)), p \rangle$
  $p \leftarrow p'$
if $p \cap I \neq I$ **then**
  return $\langle b, C(I), p \rangle$
return "no flaw"

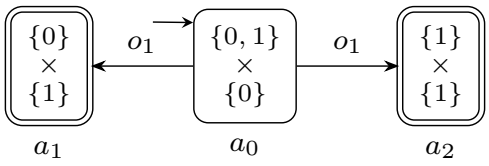


Fig. 9. Example of a Cartesian abstraction for a task with two binary variables, $I = \langle 0, 0 \rangle$, $G = \{v_2 \mapsto 1\}$, and a single operator o_1 with $\mathit{pre}(o_1) = \{v_2 \mapsto 0\}$ and $\mathit{eff}(o_1) = \{v_2 \mapsto 1\}$.

Algorithm 4 shows how to find regression flaws by generating the sequence of partial states in regression along the plan, checking the conditions of Definition 16. An important property for the correctness of the approach is whether returning "no flaw" means the abstract plan is valid for the original task. In progression flaws, this property is guaranteed, as any plan trace τ^α without progression flaws is always mappable and therefore it is a plan for the original task (see Theorem 4). However, partial states generated during regression may correspond to different abstract states. Therefore, for deviation flaws we check whether the intersection is empty instead of requiring the partial state to be included in the abstract state. This implies that abstract plans with no regression flaw could be not mappable.

Theorem 6. *There are abstract plan traces without regression flaws that are not mappable.*

Proof. In the Cartesian abstraction of Figure 9, the abstract plan trace $\tau^\alpha = a_0 \xrightarrow{o_1} a_2$ is not mappable because the state $s_1 = \mathit{progr}(I, o_1)$ is mapped to a_1 instead of the expected a_2. But, τ^α has no regression flaws because the goal partial state $\{v_2 \mapsto 1\}$ has a non-empty intersection with a_2 and the operator o_1 can be applied in the goal partial state $\{v_2 \mapsto 1\}$ reaching the state $\{v_2 \mapsto 0\}$, which has a non-empty intersection with both, I^α and I. □

Nevertheless, regression flaws do keep the most important property, which is that any abstract plan without a flaw corresponds to a plan for the original task.

Theorem 7. *Let* $\Pi = \langle V, O, I, G \rangle$ *be a planning task, and* α *a Cartesian abstraction for* Π. *If an abstract plan trace* $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ *has no regression flaws, then the sequence of operators* $\langle o^1, o^2, \ldots, o^n \rangle$ *is a plan for* Π.

Proof. If there is no regression flaw, then the backward trace $\tau^r = p^n \xleftarrow{o^n} p^{n-1} \xleftarrow{o^{n-1}} \dots \xleftarrow{o^1} p^0$ is well-defined, i.e., o^i is consistent with $post(o^i)$ and $p^{i-1} = regr(p^i, o^i)$ for all $i \in \{1, 2, \dots, n\}$. We show by induction that there exist $s^0, \dots, s^n$ such that $s^0 = I$, $s^i \in [\![p^i]\!]$ for all $i \in \{0, 1, \dots, n\}$, and $progr(s^i, o^{i+1}) = s^{i+1}$.

The base case, $I \in [\![p^0]\!]$, holds due to the absence of flaws of case 3.

For the inductive case, if $s^i \in [\![p^i]\!]$ and $p^i = regr(p^{i+1}, o^{i+1})$, then by the definition of regression there must exist $s^{i+1} \in [\![p^{i+1}]\!]$ such that $progr(s^i, o^{i+1}) = s^{i+1}$.

Finally, $s^n \in [\![p^n]\!] = S_G$, so the whole sequence is a plan for Π. □

4.2 Relation to Progression Flaws

After showing the correctness of CEGAR with regression flaws, we consider whether they imply a significant change compared to CEGAR with forward flaws. First, we consider the relationship between both types of flaws.

Theorem 8. *Let $\Pi = \langle V, O, I, G \rangle$ be a planning task, and α a Cartesian abstraction for Π. If an abstract plan trace has a regression flaw, then it has a progression flaw, but the opposite is not necessarily true.*

Proof. ($\Rightarrow$) Assume the opposite. Let $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ be an abstract plan trace such that τ^α has a regression flaw, but it is valid in progression (i.e., does not have a progression flaw). Then, there is a plan trace $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} s^n$ s.t. $\alpha(s^i) = a^i$ for all $i \in \{0, 1, \dots, n\}$ and $s^0 = I$. We show by induction that, given a backward trace $\tau^r = p^n \xleftarrow{o^n} p^{n-1} \dots \xleftarrow{o^{k+1}} p^k, k \in \{0, 1, \dots, n\}$, $s^i \in [\![p^i]\!]$ for all $i \in \{k, k+1, \dots, n\}$.

Base case (i = n): $s^n \in [\![G]\!] = [\![p^n]\!]$.

Recursive case: $s^i \in [\![a^i]\!]$ for all $i \in \{0, 1, \dots, n\}$ because there is no progression flaw. So, if $s^i \in [\![p^i]\!] \Rightarrow s^i \in [\![p^i \cap a^i]\!]$ and $s^{i-1} \in [\![p^{i-1} \cap a^{i-1}]\!]$, so $p^{i-1} \cap a^{i-1} \neq \emptyset$ because at least one state (s^{i-1}) is consistent with a^{i-1} and p^{i-1}.

$I \in p^0$ and $p^i \cap a^i \neq \emptyset$ for all $i \in \{0, 1, \dots, n\}$ because there are no progression flaws, so regression flaws are not possible and the theorem is proved by contradiction.

($\Leftarrow$) Figure 9 shows an abstract plan with a progression flaw but no regression flaw. This is possible because regression flaws do not require reification of an abstract plan trace but only that the trace is a plan for the task. □

The initial motivation for using regression flaws is that, intuitively, doing regression over the abstract plan will find flaws on abstract states closer to the goal. This is expected to be beneficial, as a refinement of an abstract state can never increase the goal distance of abstract states closer to the goal. So refining states closer to the goal has greater chances of propagating the increment of goal distance to other abstract states. Indeed, Speck and Seipp (2022) experimentally showed that strategies refining states closer to the goal result in better heuristics. However, regression flaws are not guaranteed to exist closer to the goal.

Proposition 1. *Let $\tau^\alpha = a^0 \xrightarrow{o^1} \dots \xrightarrow{o^n} a^n$ be an abstract trace with a progression flaw at step $a^i \xrightarrow{o^{i+1}} a^{i+1}$ and a regression flaw at step $a^j \xleftarrow{o^j} a^{j-1}$. Cases where $i \geq j$ exist.*

Proof. Let Π be a task with binary variables $V = \langle v_1, v_2, v_3, v_4 \rangle$, $I = \langle 0, 0, 0, 0 \rangle$, $G = \{v_2 \mapsto 1, v_3 \mapsto 1, v_4 \mapsto 1\}$, $O = \{o_1, o_2\}$ with $pre(o_1) = \{v_2 \mapsto 0\}$, $eff(o_1) = \{v_2 \mapsto 1, v_4 \mapsto 1\}$, $pre(o_2) = \{v_2 \mapsto 0, v_3 \mapsto 0\}$, and $eff(o_2) = \{v_2 \mapsto 1, v_3 \mapsto 1\}$.

Figure 10 shows the abstract states along an optimal abstract plan. The forward flaw is found at step 2 because o_2 is not applicable in s_1. The regression flaw is found at step 1 because o_1 is not applicable in regression in p_1. □

Note that this happens when concrete states reached in progression are not inside partial states reached in regression, like s_1 in Figure 10. In this example, both refinements are applied over a_1, but it could easily be extended so that the progression flaw is found on some a_i and the regression flaw is found on some a_j with $i > j$.

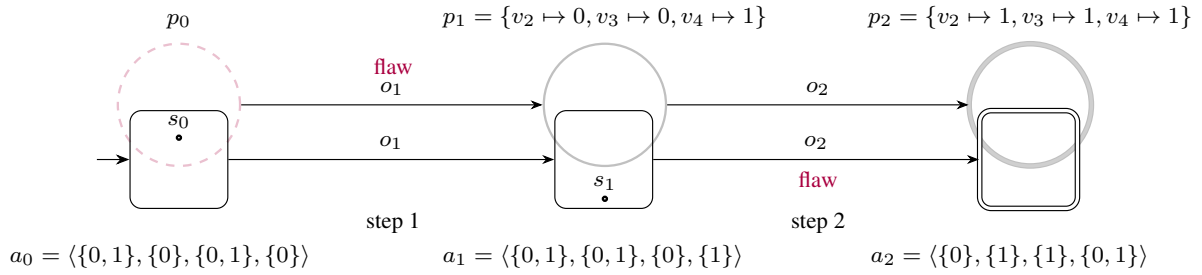


Fig. 10. States of an abstract optimal plan (other states of the abstraction and self-loops omitted) for the example of Proposition 1. Rectangles are abstract states, circles are partial states and dots are concrete states. The portions of partial states outside their mapped abstract state may be part of any other abstract state. If the operator o_1 were applicable in regression, I would be inside p_0.

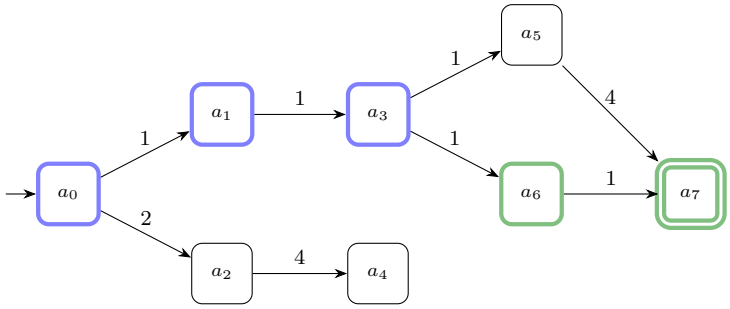


Fig. 11. Graphical representation of the goal distance propagation when a_3 is refined into a_3 and a_6. The states with increased goal distance are thicker and blue while the states with increased distance to the initial state are thicker and green.

Despite this theoretical result, in practice regression flaws are often closer to the goals than progression flaws, around 99% of the times in a single abstraction according to our experiments. This value decreases when using multiple abstractions, but regression flaws are still closer to the goals than progression flaws more often anyway.

Another difference is that progression flaws focus on plan prefixes, splitting an abstract state on which there is some reachable state. In contrast, regression flaws are found on abstract states where the heuristic value is correct for some state, to discriminate the ones where this is not the case with the current abstract plan trace.

Generally, regression flaws lead to improving heuristic values in more states, as each refinement may increase the goal distance of all abstract states whose optimal abstract plan passed through the refined state. This also happens with refinements from progression flaws, but regression flaws tend to be closer to the abstract goal state, potentially increasing the goal distance from a higher number of states on average. Figure 11 shows this intuition graphically. Furthermore, it is well known that improving low heuristic values is often more useful during search than improving heuristic values that are already high (Holte, Felner, et al. 2006). The reason is that this results in fewer states with a very low value. This explains why refining the abstraction using regression flaws tends to result in better abstraction heuristics than using progression flaws, as we will show in the experiments.

On the other hand, progression flaws increase the distance from the initial abstract state for a larger number of states, so the resulting abstractions better estimate the g value of concrete states. Note that progression flaws are also better at finding concrete plans because they refine the abstract states relative to the concrete states found from the initial state, so they focus on revealing the optimal path to the goal from the initial state.

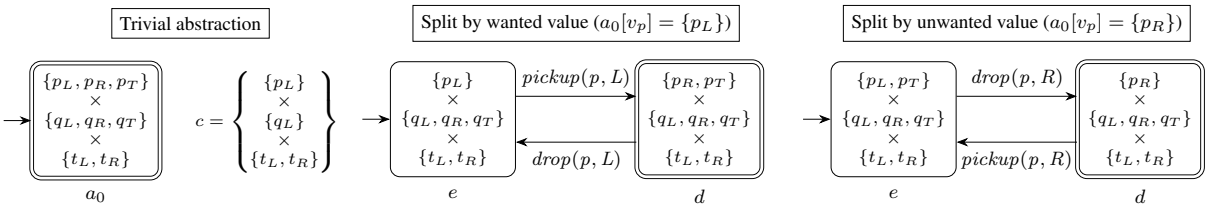


Fig. 12. Refinement of the trivial abstraction after a regression flaw. The regression flaw identifies that the goal disagrees with I and chooses to refine based on $v = v_p$. The value p_T for v_p can be assigned to either side, but "unwanted" is preferable.

4.3 Splitting Strategy

A split of an abstract flaw $\langle a, \overrightarrow{c}, \overleftarrow{c} \rangle$ is a pair of Cartesian representations d and e (Definition 15 on page 16). For regression flaws we pick a variable $v_s \in \text{vars}(p)$ such that $p[v_s] \not\subseteq c[v_s]$, but $p[v_s] \in d[v_s]$. The choice of v_s can be based on the same strategies defined for forward flaws (Speck and Seipp 2022). For regression flaws, the split is done in such a way that $p[v] \in d[v]$ and $c[v_s] \subseteq e[v_s]$, so d is consistent with p and e is consistent with c. All other values of $a[v]$ can be assigned either to d or e. The original algorithm assigns them to d but for regression flaws it makes more sense to assign them to e.

Figure 12 illustrates this, by showing two possible splits. The original CEGAR algorithm splits the wanted values (the values in $c[v]$) into e from the other values of $a[v]$. This achieves the desirable behaviour for progression flaws, i.e., to have as many values as far from the goals as possible. In regression flaws we reverse this, by splitting away the unwanted value of the partial state ($p[v]$) into d. This is more desirable because that way all states taking the remaining values ($\{p_T\}$ in our example) will get a larger heuristic value. Note that this corresponds in both cases to split the value in $\overleftarrow{c}$ of the equivalent abstract flaw. The difference is more clear in the trivial abstraction, where splitting the wanted values splits only the initial state fact, which provides low improvements to the heuristic. Note also that splitting the wanted value results in connecting children states by one of the first operators of the final plan, while splitting by the unwanted value connects them by the last operator of the final plan, what makes more sense when refining backwards.

5 Sequence Flaws

After revealing the benefits of regression flaws, in this section we explore a type of flaw that allows to find flaws in any part of the abstract plan, by finding multiple flaws either in progression or regression.

Consider our running example of a truck that must deliver two packages and the abstraction shown in Figure 4 on page 7, where the position of the truck is abstracted in all states and p may be in both p_R or p_T, only for the goal abstract state. Now let us observe the abstract plan of Figure 13. This plan has a progression flaw in a_0 because the package q cannot be picked up if the truck is at another location. This plan has also a regression flaw in a_5 because p would be at R instead of T after applying the first *drop* operator in regression, since this is its location in the goals. But, intuitively, we can detect other problems in this abstract plan. For example, between a *pickup* operator in a truck and a *drop* operator from the same truck at another location, a *drive* operator from the first location to the last one is always needed, and indeed this is true independently of the initial state or the goals. According to this logic, in this abstract plan we can detect the following sequence flaws (in addition to the first progression and regression flaws):

- In progression:
  - The last *drop* operator cannot be applied because $t \mapsto t_L$ after picking up the package at L.
  - The state reached after applying all operators does not meet the goals because $p \mapsto p_T$ instead of $p \mapsto p_R$.

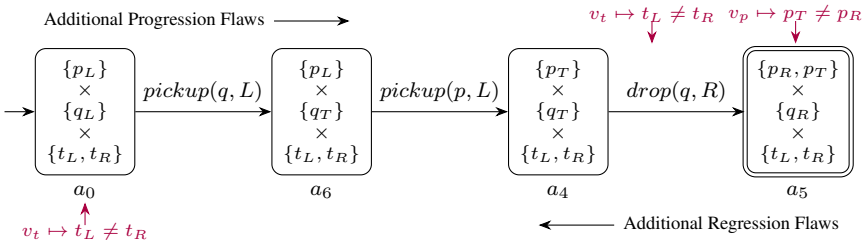


Fig. 13. Additional sequence flaws found in the optimal abstract plan of Figure 5.

- In regression:
  - After applying all operators in regression, the final state is not I because $t \mapsto t_L$ instead of $t \mapsto t_R$.

Note that the first progression and regression flaws do not find important issues of the abstract plan. Sequence flaws can capture these issues that would not be detected until repairing all the first-flaws detected before them.

In previous work, flaws are found by executing the abstract plan in the concrete state space, generating a sequence $s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \dots \xrightarrow{o^k} s^k, k \in \{0, 1, \dots, n\}$ (or partial states $p^n \xleftarrow{o^n} p^{n-1} \xleftarrow{o^{n-1}} \dots \xleftarrow{o^{k+1}} p^k, k \in \{0, 1, \dots, n\}$ in regression). Instead, we consider a relaxation of such approach, that results in a sequence of Cartesian representations. To this end, we define $progr^!(c, o)$ and $regr^!(c, o)$ as the application of an operator o on a Cartesian representation c in progression, and regression respectively even when o is not applicable. $progr^!(c, o)$ sets the post values of o in the successor state, i.e., $progr^!(c, o)[v] = post(o)[v]$ for all variables $v \in \text{vars}(post(o))$ and $progr^!(c, o)[v] = c[v]$ for all variables $v \in V \setminus \text{vars}(post(o))$. $regr^!(c, o)$ is the same as $regr(c, o)$ but applied even when c is not consistent with $post(o)$, since $regr^!(c, o)[v] = pre(o)[v]$ for all variables in $\text{vars}(pre(o))$, $regr^!(c, o)[v] = \mathcal{D}_v$ for all variables $v \in \text{vars}(post(o)) \setminus \text{vars}(pre(o))$ and $regr^!(c, o)[v] = c[v]$ for all variables $v \in V \setminus (\text{vars}(post(o)) \cup \text{vars}(pre(o)))$.

5.1 Progression Sequence Flaws

To define progression sequence flaws, we first introduce which conditions must be fulfilled by the relaxed execution of abstract plans.

Definition 17 (Relaxed Plan Execution). *Let $\Pi = \langle V, O, I, G \rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ with the induced transition system Θ^α, and $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α. A relaxed plan execution $r = r^0, r^1, \dots, r^n$ for τ^α is a sequence of Cartesian representations r^i s.t.:*
(A) $I \in [\![r^0]\!]$,
(B) if o^{i+1} is applicable in r^i, $progr(r^i, o^{i+1}) \cap a^{i+1} \subseteq r^{i+1}$,
(C) if o^{i+1} is not applicable in r^i, $progr^!(r^i, o^{i+1}) \cap a^{i+1} \subseteq r^{i+1}$, and
(D) $r^i \cap a^i \neq \emptyset$.

Each Cartesian representation r^i in a relaxed plan execution represents the states related to the corresponding abstract state a^i of the abstract plan that could be reached by applying the prefix plan. The execution is relaxed, meaning that at any step in the plan, more states can be added into r^i depending on the relaxation chosen. For example, this allows to execute a plan while ignoring some of the variables altogether to detect flaws on the remaining variables. These conditions keep the execution coherent with the application of the operators in the plan. Specifically, (A) and (B) ensure that if the abstract plan trace is executable and mappable in the concrete state space, then no flaw exists. Condition (C) aims to preserve some coherence in the execution even when an

operator is not applicable. Specifically, variables not affected by the operator must keep their values, and the resulting state must satisfy the post-conditions of the operator. The intuition is that the remaining part of the execution will check if the suffix of the plan could work if we were able to fix the prefix of the plan in such a way that the operator was applicable. If that is not the case, we consider that the suffix has a flaw. Lastly, condition (D) ensures that any flaw we find at any step can be used to refine the corresponding abstract state a^i.

Definition 18 (Progression Sequence Flaw). *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α, and $r = r^0, r^1, \dots, r^n$ a relaxed plan execution for τ^α. A progression sequence flaw in τ^α is a tuple $\langle a^i, r^i, c\rangle$ consisting of an abstract state a^i, a counterexample Cartesian representation r^i, and a non-empty Cartesian representation c for which one of the following cases apply:*

(1) o^{i+1} is not applicable in r^i,resulting in the flaw $\langle a^i, r^i, a^i \cap C(pre(o^{i+1}))\rangle$;
(2) o^{i+1} is applicable in r^i, but its successor does not intersect to a^{i+1}, $progr(r^i, o^{i+1}) \cap a^{i+1} = \emptyset$, and $c = a^i \cap regr(a^{i+1}, o^{i+1})$, resulting in the flaw $\langle a^i, r^i, c\rangle$;
(3) $i = n$, and $r^n \cap C(G) = \emptyset$, producing the flaw $\langle a^n, r^n, a^n \cap C(G)\rangle$.

Note that, to determine if there is a flaw at step i, only the prefix of the execution $r^0, r^1, \dots, r^i$ is relevant, as the relaxed plan execution can always be continued, e.g., by setting $r^j = a^j$ for all $j \in \{i+1, i+2, \dots, n\}$. Now, we analyse their relation to the general definition of flaw and the reification of plan traces and plans.

Theorem 9. *Let $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ be an abstract plan trace found in an abstraction α with the induced transition system Θ^α for a transition system Θ, and $r = r^0, r^1, \dots, r^n$ a relaxed plan execution for τ^α. If $\langle a^i, r^i, c\rangle$ is a progression sequence flaw in τ^α, then $\langle a^i, \overrightarrow{c} = r^i \cap a^i, \overleftarrow{c} = c\rangle$ is an abstract flaw. However, not all abstract flaws $\langle a^i, \overrightarrow{c}, \overleftarrow{c}\rangle$ in τ^α correspond to progression sequence flaws in τ^α.*

Proof. On the one hand, we show that the flaw $\langle a^i, r^i, c\rangle$ corresponds to an abstract flaw $\langle a^i, \overrightarrow{c} = r^i \cap a^i, \overleftarrow{c} = c\rangle$ that satisfies Definition 14 on page 15.

The flaw $\langle a^i, \overrightarrow{c} = r^i \cap a^i, \overleftarrow{c} = c\rangle$ satisfies (AF.1), (AF.3) and (AF.6) by the properties of Definition 17.

We show that it satisfies (AF.2), (AF.4), (AF.5) and (AF.7) for each case of Definition 18:

(1) o^{i+1} is not applicable in a relaxed Cartesian representation r^i, $r^i \cap a^i \neq \emptyset$, reached by applying $\langle o^1, o^2, \dots, o^i\rangle$ in I, and $c = a^i \cap C(pre(o^{i+1}))$. (AF.2) and (AF.4) are satisfied because o^{i+1} must be applicable in some state consistent with a^i to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise o^{i+1} would be applicable in r^i. (AF.7) is satisfied because o^{i+1} is not applicable in any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{start}_{\tau^\alpha_{goal}}(\tau^\alpha[i, n])$.

(2) o^{i+1} is applicable in r^i but $progr(r^i, o^{i+1}) \cap a^{i+1} = \emptyset$. Then, $c = a^i \cap regr(a^{i+1}, o^{i+1})$. (AF.2) and (AF.4) are satisfied because o^{i+1} must reach some state consistent with a^{i+1} when it is applied in a^i to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise $progr(r^i, o^{i+1}) \cap a^{i+1} \neq \emptyset$. (AF.7) is satisfied because $progr(s, o^{i+1}) \notin a^{i+1}$ from any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{start}_{\tau^\alpha_{goal}}(\tau^\alpha[i, n])$.

(3) The sequence can be executed but $r^n \cap C(G) = \emptyset$. Then, $c = a^n \cap C(G)$. (AF.2) and (AF.4) are satisfied because some state $s \in S_G \cap [\![a^n]\!]$ must exist by the definition of abstraction, since a^n would not be an abstract goal state otherwise. (AF.5) is satisfied because r^n would be a goal state otherwise. (AF.7) is satisfied because c is the Cartesian representation of all goal states consistent with a^n.

On the other hand, we prove by counterexample that not all abstract flaws are progression sequence flaws: in the abstract plan of Figure 13 on page 23 an abstract flaw exists in a_6 because $S^{end}_{\tau^\alpha_{init}}(a_0 \xrightarrow{pickup(q,L)} a_6) = \emptyset$ and $\langle p_L, q_T, t_R\rangle \notin S^{start}_{\tau^\alpha_{goal}}(a_6 \xrightarrow{pickup(q,L)} a_4 \xrightarrow{drop(q,R)} a_5)$, and so $\langle a_6, \langle\{p_L\}, \{q_T\}, \{t_L\}\rangle, \langle\{p_L\}, \{q_T\}, \{t_R\}\rangle\rangle$ is a flaw, but it

is not a progression sequence flaw because $t_L \in r^1[v_t]$ after applying $pickup(q, L)$ in r^0, so $progr(r^1, pickup(p, L)) = \langle p_T, q_T, t_L \rangle \in [\![a_4]\!]$. □

THEOREM 10. *Let $\Pi = \langle V, O, I, G \rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α, and $r = r^0, r^1, \ldots, r^n$ a relaxed plan execution for τ^α. Then, τ^α is mappable iff there is no progression sequence flaw in τ^α.*

PROOF. On the one hand, assume τ^α has no progression sequence flaw. Then, we show by induction that τ^α is mappable. If τ^α is mappable, $\tau = s^0 \xrightarrow{o^1} s^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} s^n$ is a reification of τ^α where $s^0 = I$, $s^n \in [\![G]\!]$, and $s^i \in [\![a^i]\!]$ for all $i \in \{0, 1, \ldots, n\}$. For the base case, r^0 is the Cartesian representation such that $[\![r^0]\!] = \{I\}$. For the inductive case, since there is no flaw of cases (1) and (2), o^i is applicable in r^{i-1}, resulting in some $r^i = C(s^i)$ for all $i \in \{0, 1, \ldots, n\}$. Finally, as there is no flaw of case (3), $r^n \cap C(G) \neq \emptyset$, so s^n is a goal state. Thus, τ^α is mappable in a trace τ where $s^i = [\![r^i]\!]$ for all $i \in \{0, 1, \ldots, n\}$.

On the other hand, no progression sequence flaw can be found in τ^α if it is mappable because all progression sequence flaws are abstract flaws and Theorem 2 on page 15 proves that no abstract flaw exists in mappable abstract plan traces. □

PROPOSITION 2. *Let τ^α be an abstract plan trace in a Cartesian abstraction α for a planning task Π and π the sequence of operators of τ^α. π may be a plan for Π even if τ^α has progression sequence flaws of case (2).*

PROOF. The example of Figure 9 on page 19 applies. The abstract plan trace $\tau^\alpha = a_0 \xrightarrow{o_1} a_2$ has a progression sequence flaw of case (2) in o_1 but $\pi = \langle o_1 \rangle$ is a plan, though the final state is mapped to a_1 instead of a_2. □

Even though our definition allows Cartesian representations arbitrarily big along the relaxed plan execution, doing so results in finding fewer flaws. In the extreme case, if r^i is completely relaxed for all $i \in \{0, 1, \ldots, n\}$, all operators would be applicable in each r^i and no flaws could be found. For instance, in Figure 13 on page 23, r^2 could be the Cartesian representation $\langle \{p_L, p_R, p_T\}, \{q_L, q_R, q_T\}, \{t_L, t_R\} \rangle$. But doing this no flaw would be found in $drop(q, R)$ due to $t \mapsto t_R$ is in r^2, while if we make $r^2 = \langle \{p_T\}, \{q_T\}, \{t_L\} \rangle$, $drop(q, R)$ is inapplicable due to $t \mapsto t_L$ after applying $pickup(p, L)$.

PROPOSITION 3. *Let $\Pi = \langle V, O, I, G \rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α, and $r = r^0, r^1, \ldots, r^n$ a relaxed plan execution for τ^α. Let z^i be a Cartesian representation such that $z^i \cap a^i \neq \emptyset$, $progr(r^{i-1}, o^i) \cap a^i \subseteq z^i$, and $[\![z^i]\!] \subset [\![r^i]\!]$. Then, there exists another relaxed plan execution $z = r^0, r^1, \ldots, r^{i-1}, z^i, z^{i+1}, \ldots, z^n$ such that $[\![z^j]\!] \subseteq [\![r^j]\!]$ for all $j \in \{i, i+1, \ldots, n\}$ and any progression sequence flaw of r is a progression sequence flaw of z.*

PROOF. Any flaw found at step i on r^i, is a flaw for z^i as well:

(1) If o^{i+1} is not applicable in r^i, then there is some precondition $v_i \mapsto x$ of o^{i+1} such that $x \notin r^i[v]$. As $[\![z^i]\!] \subset [\![r^i]\!]$, then $x \notin z^i[v]$, and the flaw is also found in z^i.
(2) If $progr(r^i, o^{i+1}) \cap a^{i+1} = \emptyset$, then there exist some v such that $a^{i+1}[v] \cap progr(r^i, o^{i+1}) = \emptyset$. As $[\![z^i]\!] \subset [\![r^i]\!]$, then $progr(z^i, o^{i+1})[v] \subseteq progr(r^i, o^{i+1})[v]$, and so $progr(z^i, o^{i+1})[v] \cap a^{i+1} = \emptyset$, so a flaw is found for z as well.
(3) If $i = n$ and $r^i \cap C(G) = \emptyset$, then $z^i \cap C(G) = \emptyset$.

For the rest of the execution, note that applying $z^j = r^j$ for all $j \in \{i+1, i+2, \ldots, n\}$ results in no additional flaws. However, it is worth noting that other continuations where $[\![z^j]\!] \subset [\![r^j]\!]$ may result in other flaws. □

Algorithm 5: Find Progression Sequence Flaws

Data: Task $\Pi = \langle V, O, I, G\rangle$, abstract plan traced τ^α
Parameters: $r = I, i = 0$ // r and i, with default values
Result: Progression sequence flaws for τ^α

1 $flaws \leftarrow \emptyset$
2 **while** $i < n$ **do**
3 **if** *not applicable*(r, o^{i+1}) **then**
4 $flaws \leftarrow flaws \cup \{\langle a^i, r^i, a^i \cap C(pre(o^{i+1}))\rangle\}$
5 $r \leftarrow progr^!(r, o^{i+1})$ // Apply o^{i+1} even if it is not applicable
6 **if** $r \cap a^{i+1} = \emptyset$ **then**
7 $flaws \leftarrow flaws \cup \{\langle a^i, r^i, a^i \cap regr(a^{i+1}, o^{i+1})\rangle\}$
  // Undeviate the Cartesian representation
8 **forall** $v \in V$ **do**
9 **if** $r[v] \cap a^{i+1}[v] = \emptyset$ **then**
10 $r[v] \leftarrow a^{i+1}[v]$
11 $i \leftarrow i + 1$
12 **if** $r \cap G = \emptyset$ **then**
13 $flaws \leftarrow flaws \cup \{\langle a^n, r^n, a^n \cap C(G)\rangle\}$
14 **return** *flaws*

5.2 Progression Sequence Flaws Collection

Algorithm 5 shows the proposed procedure to collect progression sequence flaws of an abstract plan. Contrary to the standard procedure that executes the abstract plan on the concrete state space, we consider the execution over Cartesian representations. Initially, the abstraction contains a single state, and therefore the first flaw found by Algorithm 5 will be the same flaw reported by the standard procedure. However, after a flaw has been previously found, Algorithm 5 continues looking for other flaws over a Cartesian space throughout the abstract plan.

The algorithm is always called with the default parameters except for special strategies discussed in Section 6.6, so it starts at I. We apply operators even when they are not applicable, and when flaws of the second case occur, we "undeviate" the resulting Cartesian representation by resetting $r^{i+1}[v]$ to all values compliant with the abstract state a^{i+1}. By doing this, when a flaw with respect to v has been found, we allow v to take any value consistent with the next abstract state to only detect subsequent flaws that would exist after fixing that flaw. Note that a second flaw involving the same variable v may be found when a different flaw exists in the same variable, e.g., if somewhere in the remaining abstract plan two operators are applied with contradicting preconditions over v.

Proposition 4. *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α, and $r = r^0, r^1, \ldots, r^n$ a relaxed plan execution for τ^α. All flaws returned by Algorithm 5 are progression sequence flaws.*

Proof. Flaws are accumulated in lines 4, 7 and 13. In line 4, when o^{i+1} is not applicable the flaw $\langle a^i, r^i, a^i \cap C(pre(o^{i+1}))\rangle$ is added, as flaw (1) in Definition 18 does. In line 7, the flaw $\langle a^i, r^i, a^i \cap C(regr(a^{i+1}, o^{i+1}))\rangle$ is added if $r \cap a^{i+1} = \emptyset$, exactly as flaw (2) does. In line 13, the flaw $\langle a^n, r^n, a^n \cap C(G)\rangle$ is added if $r^n \cap C(G) = \emptyset$, exactly as flaw (3) does. □

Algorithm 5 does not find all progression sequence flaws. As Proposition 3 shows, this would require always keeping each r^i as small as possible. Yet, there are two points in Algorithm 5 where r keeps values that could be

removed in an attempt of finding only flaws that are relevant. In line 5, we could replace r by $r \cap a^{i+1}$. However, this simply insists on keeping the relaxed plan execution fully aligned with the abstract plan trace, which could lead to finding flaws in cases where the plan is valid through other abstract states (as Proposition 2 shows). Also, in line 9, assigning a single value from $a^{i+1}[v]$ instead of all of them would suffice to keep property (D) of Definition 17. However, at that point, the algorithm has already found a flaw with respect to v so, by continuing the relaxed plan execution with all values in a^{i+1}, we seek to only find another flaw if the execution fails for all those values.

5.3 Regression Sequence Flaws

A backward relaxed plan execution can be defined analogously to Definition 17 but replacing I by G and progression semantics by regression semantics.

Definition 19 (Backward Relaxed Plan Execution). *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ with the induced transition system Θ^α, and $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α. A backward relaxed plan execution $r^r = r^n, r^{n-1}, \dots, r^0$ for τ^α is a sequence of Cartesian representations r^i such that:*

(A) $G \in [\![r^n]\!]$,
(B) if o^i is applicable in regression in r^i, $regr(r^i, o^i) \cap a^{i-1} \subseteq r^{i-1}$,
(C) if o^i is not applicable in regression in r^i, $regr^!(r^i, o^i) \cap a^{i-1} \subseteq r^{i-1}$, and
(D) $r^i \cap a^i \neq \emptyset$.

Definition 20 (Regression Sequence Flaw). *Let $\Pi = \langle V, O, I, G\rangle$ be a planning task with transition system Θ, α a Cartesian abstraction for Θ that induces the abstract transition system Θ^α, $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ an abstract plan trace found in Θ^α, and $r^r = r^n, r^{n-1}, \dots, r^0$ a backward relaxed plan execution for τ^α. A regression sequence flaw in τ^α is a tuple $\langle a^i, c, r^i\rangle$ consisting of an abstract state a^i and a non-empty Cartesian representation c, and a counterexample Cartesian representation r^i for which one of the following cases apply:*

(1) o^i is not applicable in regression in r^i,resulting in the flaw $\langle a^i, a^i \cap C(post(o^i)), r^i\rangle$;
(2) o^i is applicable in regression in r^i, but its successor does not intersect to a^{i-1}, i.e. $regr(r^i, o^i) \cap a^{i-1} = \emptyset$, and $c = a^i \cap progr(r^{i-1}, o^i)$, resulting in the flaw $\langle a^i, c, r^i\rangle$;
(3) $i = 0$, and $I \notin [\![r^0]\!]$, producing the flaw $\langle a^0, C(I), r^0\rangle$.

Theorem 11. *Let $\tau^\alpha = a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \dots \xrightarrow{o^n} a^n$ be an abstract plan trace found in an abstraction α with the induced transition system Θ^α for a transition system Θ, and $r^r = r^n, r^{n-1}, \dots, r^0$ a backward relaxed plan execution for τ^α. If $\langle a^i, c, r^i\rangle$ is a regression sequence flaw in τ^α, then $\langle a^i, \overrightarrow{c}{=}c, \overleftarrow{c}{=}r^i \cap a^i\rangle$ is an abstract flaw. However, not all abstract flaws $\langle a^i, \overrightarrow{c}, \overleftarrow{c}\rangle$ in τ^α correspond to regression sequence flaws in τ^α.*

Proof. On the one hand, we show that the flaw $\langle a^i, c, r^i\rangle$ corresponds to an abstract flaw $\langle a^i, \overrightarrow{c}{=}c, \overleftarrow{c}{=}r^i \cap a^i\rangle$ that satisfies Definition 14 on page 15.

The flaw $\langle a^i, \overrightarrow{c} = c, \overleftarrow{c} = r^i \cap a^i\rangle$ satisfies (AF.2), (AF.4) and (AF.7) by the properties of Definition 19.

We show that it satisfies (AF.1), (AF.3), (AF.5) and (AF.6) for each case of Definition 20:

(1) o^i is not applicable in regression in a relaxed Cartesian representation r^i, $r^i \cap a^i \neq \emptyset$, reached by applying $\langle o^n, o^{n-1}, \dots, o^i\rangle$ in regression in G, and $c = a^i \cap C(post(o^i))$. Then, (AF.1) and (AF.3) are satisfied because o^i must be applicable in regression in some state in a^i to induce the transition in Θ^α by the definition of induced abstraction. (AF.5) is satisfied because otherwise o^i would be applicable in regression in r^i. (AF.6) is satisfied because o^i is not applicable in regression in any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{end}_{\tau_{init}}(\tau^\alpha[0, i])$.

(2) o^i is applicable in regression in r^i but $regr(r^i, o^i) \cap a^{i-1} = \emptyset$. Then, (AF.1) and (AF.3) are satisfied because o^i must reach some state in a^{i-1} when it is applied in regression in a^i to induce the transition in Θ^α by the

Algorithm 6: Find Regression Sequence Flaws

Data: Task $\Pi = \langle V, O, I, G\rangle$, abstract plan trace τ^α
Parameters: $r = G, i = n$ // r and i, with default values
Result: Regression sequence flaws for τ^α

```
flaws ← ∅
while i > 0 do
    if not backward-applicable(r, o^i) then
        flaws ← flaws ∪ {⟨a^i, a^i ∩ C(post(o^i)), r^i⟩}
    r ← regr^!(r, o^i) // Apply o^i backwards even if it is not applicable in regression
    if r ∩ a^{i−1} = ∅ then
        flaws ← flaws ∪ {⟨a^i, a^i ∩ progr(a^{i−1}, o^i), r^i⟩}
        // Undeviate the Cartesian representation
        forall v ∈ V do
            if r[v] ∩ a^{i−1}[v] = ∅ then
                r[v] ← a^{i−1}[v]
    i ← i − 1
if I ⊄ r then
    flaws ← flaws ∪ {⟨a^0, C(I), r^0⟩}
return flaws
```

definition of induced abstraction. (AF.5) is satisfied because otherwise o^i would reach in regression some state $s \in [\![r^i]\!]$ (no deviation flaw). (AF.6) is satisfied because o^i does not reach in regression any state $s \notin [\![c]\!]$ and so they cannot be part of $S^{end}_{\tau_{init}}(\tau^\alpha[0, i])$.

(3) The sequence can be executed but $I \notin [\![r^0]\!]$, and $c = C(I)$. Then, (AF.1) and (AF.3) are satisfied because $c = C(I)$, which contains one concrete state mapped to a^0. (AF.5) is satisfied because otherwise $I \in r^0$ (no flaw). (AF.6) is satisfied because $S^{end}_{\tau_{init}}(\tau^\alpha[0]) = \{I\} = [\![c]\!]$.

On the other hand, we prove by counterexample that not all abstract flaws are regression sequence flaws: in the abstract plan of Figure 13 on page 23 an abstract flaw exists in a_6 because $S^{end}_{\tau_{init}}(a_0 \xrightarrow{pickup(q,L)} a_6) = \emptyset$ and $\langle p_L, q_T, t_R\rangle \notin S^{start}_{\tau_{goal}}(a_6 \xrightarrow{pickup(q,L)} a_4 \xrightarrow{drop(q,R)} a_5)$, and so $\langle a_6, \langle\{p_L\}, \{q_T\}, \{t_L\}\rangle, \langle\{p_L\}, \{q_T\}, \{t_R\}\rangle\rangle$ is a flaw, but no regression sequence flaw exists in this abstract state because $t_L \in r^1[v_t]$ after applying the $pickup(p, L)$ operator in regression from $r^2 = \langle\{p_T\}, \{q_T\}, \{t_R\}\rangle$, so $regr^!(r^2, pickup(q, L)) = \langle p_L, q_T, t_L\rangle \in [\![a_6]\!]$. □

Algorithm 6 is used to collect regression flaws. It is like Algorithm 5 but interchanging I and G and using regression semantics.

In any trace $a^0 \xrightarrow{o^1} a^1 \xrightarrow{o^2} \ldots \xrightarrow{o^n} a^n$ progression flaws in the state a^i are different to regression flaws in the state a^{i+1} for all $i \in \{0, 1, \ldots, n-1\}$, so identifying all flaws requires searching in both directions.

6 Flaw Selection Strategies

Sequence flaws allow identifying an arbitrarily large set of flaws for a single abstract plan. The next step is then to refine the abstraction, i.e., splitting an abstract state according to one of the flaws, so that the same abstract plan trace is no longer applicable in the refined abstraction. While it would be possible to refine the abstraction according to multiple flaws, it suffices to choose one of them, refining the abstraction, and finding

new flaws according to the new abstract plan, because other flaws might become irrelevant after the refinement. Still, repairing the right flaw at each step becomes paramount. Otherwise, finding many flaws in the abstract plan is even harmful if the chosen refinement is worse than repairing the first flaw.

These strategies receive as input a set of flaws (computed forwards, backwards or in both directions), and they return the split to apply to an abstract state. A split of an abstract flaw $\langle a, \overrightarrow{c}, \overleftarrow{c} \rangle$ divides the abstract state a into d and e according to a split variable v_s such that $a[v_s] = d[v_s] \cup e[v_s]$ and $a[v] = d[v] = e[v]$ for all $v \in V \setminus \{v_s\}$ (Definition 15). The strategies introduced in this section select a split by assuming that the values of the split variable v_s that are not involved in the flaw are assigned to a particular child. Consequently, their purpose is to choose a split variable within an abstract state, considering how the values will be distributed among the child abstract states based on this assumption. Section 4.3 explains the "wanted" splitting strategy (moving them to the child with the values that fix the flaw) and the "unwanted" splitting strategy (moving them to the child with the value that produces the flaw). The strategies introduced in this section can work together to choose a split and break ties. Thus, in the experiments we do not use only one of these criteria to choose a split but several layers of them (Section 7). For instance, we can use a strategy based in the position of the flaw to choose a split of an abstract state, then choose the most refined split in such a state, and then break ties by choosing the first split found.

We have designed many different strategies that can be grouped by the high-level criterion on which they are based, which are discussed in the following subsections. These criteria are based on previous work (splits properties and variable orderings) (Seipp and Helmert 2018; Speck and Seipp 2022), new criteria enabled by sequence flaws (position of the flaw, iterative strategy, refine all flaws), and new criteria based on an evaluation of the resulting abstraction after the split (metrics of the children states, splits filters based on the children states). Strategies based on the position of the flaw (Section 6.1) and iterative strategies (Section 6.6) first choose an abstract state with a flaw (what we also call a flawed state) so that the best split is computed only in this state, but the rest of strategies depend on the splits, so splits in all abstract states of the plan must be computed. To save the expensive computations of splits, we cache the best split for each abstract state. Unfortunately, the cached values are often invalidated, each time a connected abstract state is refined, so the computation impact of splits remains high.

6.1 Strategies Based on the Position of the Flaw

The relative position of the flawed state in the abstract plan trace is a relevant criterion to choose flaws, based on the previous findings that suggest that refining closer to the goal produces better heuristics. Also, this allows us to compare the differences between the last progression sequence flaw and the first regression flaw, as well as the differences between the flaw closest to the goal in bidirectional strategies and the first regression flaw.

These strategies only choose an abstract state with a flaw, so in the experiments we combine them with the best split selection strategies found by Speck and Seipp (2022): maximising the amount of covered flaws (the same split may be involved in different flaws), breaking ties in favour of the most refined split, with the first split found in case of a tie in both criteria.

The identified strategies that follow this criterion are:

Default: Choose the first flaw found in the abstract plan, which coincides with the original definition of progression and regression flaw. This strategy does not need to compute all flaws, but it stops at the first one as in the original CEGAR algorithm. In the bidirectional case, the first progression and regression flaw are selected, and only one of them is chosen by using the split selection strategy.

Last Flaw (`last`): Choose the last flaw found in the abstract plan. This is especially interesting for progression flaws, since the closest to the goal flaws are chosen.

Closest to Goal Flaw (`clo`): Choose the flaw closest to the goal. This is only relevant in the bidirectional case, because in the backward case this is equivalent to the default strategy, and in the forward case it is equivalent to `last`, while in the bidirectional case any of them may be closer to the goal.

6.2 Strategies Based on Split Properties

These strategies try to refine the best split among all the splits found in the abstract plan trace. To do it, the best split is computed on each abstract state, and then the best one among all flawed states is chosen and refined. Therefore, they impose a significant overhead when selecting a split. We have identified the following split criteria, some of them based on previous work (Speck and Seipp 2022) and others that are new criteria based on other Classical Planning heuristics, distance to goal and the cost of the operator involved in the flaw:

Random Split (`rand`): This strategy randomly chooses a split found in all the abstract states with flaws as a baseline to evaluate the rest of strategies. It was considered by Seipp and Helmert (2018).

First Split (`FIFO`) : This strategy chooses the first split found. Splits are found from the first abstract state of the trace for progression and bidirectional flaws and from the last abstract state of the trace for regression flaws. Inside an abstract state, splits about precondition flaws are found first in the topological order of the variables based on the causal graph (Helmert 2004, 2006), then deviation flaws are found in the same order of variables, and finally non-goal/initial state flaws are found.

Most/Least Refined Split (`ref` and `-ref`): The most refined split is the strategy that provided the best results in previous work (Seipp and Helmert 2018), where the least refined split was also considered. They choose the most/least refined split, i.e., the one with the lowest/highest ratio of values for the variable affected by the split. That is, given a flaw in an abstract state a, the variable that minimises/maximises the fraction $|a[v]|/|\mathcal{D}_v|$. The most refined split strategy deeps into states refined in previous steps, which results in more focused refinements.

Balance Closest to goal and Most Refined Split ($|^{\texttt{clo}}_{\texttt{ref}}$): The most refined split and the closest to goal split were the most useful strategies in previous experiments (Pozo, Torralba, et al. 2024a), and they are complementary, so this combination is a good candidate to achieve a better heuristic. This strategy tries to combine them to get the best of both worlds and to mitigate the performance impact of long plans obtained by the most refined splits. This combination is computed as follows:

$$loss_{|^{\texttt{clo}}_{\texttt{ref}}} = remaining\ values/all\ values + distance/max\ distance$$

The split with the minimum loss is selected, so small distances to the goal and few remaining values favour the selection of a split. The maximum goal distance is computed as the goal distance of the initial abstract state or 1.0 if it is 0 (in such a case the goal distance of all states of an abstract plan is 0).

Highest/Lowest Cost Operator (`cos` and `-cos`): Given a flaw in the trace $a^i \xrightarrow{o^{i+1}} a^{i+1}$, choose it if o^{i+1} is the operator with the highest/lowest cost among flaws. The aim of the highest cost is to refine at points where more cost is being spent, which could be especially relevant with cost partitioning, as the cost of many operators is low. The intuition for the lowest cost operator is that if the cost is low, it is easier to increase it after a refinement.

Maximum/Minimum $h^{\mathbf{add}}$ (`add` and `-add`): Given a flaw in an abstract state a that does not exist in a subset c of a, the split with a maximum/minimum h^{add} value (Bonet and Geffner 2001) from the initial state for any of the facts of c in the split variable v_s, i.e., $\max/\min_{x \in c[v_s]} h^{add}(v_s \mapsto x)$. Ties are broken in favour of the variable with the smallest index. The h^{add} heuristic is precomputed before creating the abstraction, so this score is computed fast. These criteria get an estimation of how expensive it is to reach the split facts from the initial state, which can be relevant to refine facts closer to either the goals or the initial state.

Most/Least Unwanted Values (`un` and `-un`): Not to be confused with *wanted* and *unwanted* splitting strategies (Section 4.3). The most unwanted strategy refines the split $\langle d, e \rangle$ in the split variable v_s with the largest $|d[v_s]|$ for the wanted splitting strategy, and with the largest $|e[v_s]|$ for the unwanted splitting strategy. That is, an abstract state with all values for the variable with the largest domain where only one value is left into e would get the maximum score for the wanted splitting strategy. For the least unwanted values, the behaviour is the opposite: choosing the split for which $|d[v_s]|/|e[v_s]|$ is the smallest, getting the maximum score for a split that leaves only one value in $d[v_s]$ for the wanted splitting strategy. Note that when the splitting strategy is *unwanted*, only one value is always split, so the maximum/minimum value only depends on the number of values of the abstract state in the split variable before the split.

Landmarks (`lm` and `lmr`): Fact landmarks are facts that are guaranteed to be reached in any plan (Hoffmann et al. 2004), so they are important for the task, and focusing splits on these facts is a manner of prioritizing the refinement of relevant parts of the abstraction. Specifically, we get causal fact landmarks (goals and facts consistent with the preconditions of some operator that appears in all plans) obtained from the delete relaxation of the planning task with the algorithm by Keyder et al. (2010) for finding h^m landmarks, with $m = 1$, which is the method used by Seipp and Helmert (2018) to get landmarks for different subtasks (see Section 7.3). Given a flaw in an abstract state a that does not exist in a subset c of a, these strategies choose the split where the value of the split variable v_s in c contains the fact in the lowest position in the priority list of facts, i.e., the split with the lowest $\min_{x \in c[v_s]} priorityList.pos(v_s \mapsto x)$. `lm` gets fact landmarks ordered according to their h^{add} value for the initial state in descending order as a priority list for splits. All facts not present in the list have the same lowest priority. `lmr` does the same but sorting the landmarks according to their h^{add} value in ascending order.

Highest/Lowest Potential (`pot` and `-pot`): Potential heuristics assign a potential to each fact after computing a linear programming task that guarantees a consistent and admissible heuristic (Pommerening, Röger, Helmert, and Bonet 2014). Thus, the potential of facts reveals useful information. We use the maximisation of the potential value for all syntactic states as the optimization function (Seipp, Pommerening, and Helmert 2015). Potentials are computed before creating the abstraction, so getting them is immediate during the splits selection. Given a flaw in an abstract state a that does not exist in a subset c of a, `pot` prefers splits where the value of the split variable v_s in c contains the fact with the highest potential, i.e., the split with the largest $\max_{x \in c[v_s]} pot(v_s \mapsto x)$, and `-pot` does the same for the lowest potential value.

6.3 Strategies Based on Variable Orderings

Using a fixed ordering to select splits helps to focus the refinement on specific aspects of the task. Variables are the common element to all abstract states, so they are a good attribute for a fixed ordering. The following strategies compute a fixed total order on V and then choose the split with a split variable v_s such that it is the earliest variable in the ordering:

Random Orderings ($<_{\texttt{rand}}$) This strategy creates a random order of the variables at the beginning of the refinement loop and uses it until the end. It is a good baseline to compare against. This strategy is similar to random splits, but the random decision is taken only once (before the refinement loop), so the impact of this random ordering is more significant since it is not compensated in next iterations of the loop.

Topological Order of the Causal Graph ($<_{\texttt{CG}}$ and $<^{\texttt{r}}_{\texttt{CG}}$): We use two orders based on topological order of the causal graph (Helmert 2004, 2006), which have been used before as merge strategies in merge-and-shrink (Helmert, Haslum, Hoffmann, and Nissim 2014). $<_{\texttt{CG}}$ selects first the variables with the largest indirect influence over the goals, whereas the reverse ordering, $<^{\texttt{r}}_{\texttt{CG}}$, attempts to select variables close to the goal first.

Landmarks Order ($<_{\texttt{lm}}$ and $<^{\texttt{r}}_{\texttt{lm}}$): As explained in the previous subsection, landmarks are a good criterion to focus the refinements in relevant aspects of the task. This strategy places goals first and then the rest of landmark facts computed in the h^m task (Keyder et al. 2010) with $m = 1$ in descending order of their h^{add} value for the initial state. Note that landmarks are facts instead of variables, so the first occurrence of each variable is used as the order. Also note that this technique only gets causal fact landmarks (goals and facts consistent with the preconditions of some operator that appears in all plans), and so some variables could not be implied in any landmark. The variables not implied in any landmarks are placed at the tail in the reverse topological order of the causal graph. The reverse strategy $<^{\texttt{r}}_{\texttt{lm}}$ sorts the landmark facts according to their h^{add} value but in ascending order, filling the non-landmarks variables at the tail in the (non-reverse) topological order of the causal graph.

Highest/Lowest Potential Order ($<_{\texttt{pt}}$ and $<^{\texttt{r}}_{\texttt{pt}}$): As discussed in the previous subsection, potentials reveal the importance of the fact about the heuristic value of states, so they are a good criterion to select splits. We use the maximisation of the potential value for all syntactic states as the optimization function (Seipp, Pommerening, and Helmert 2015). The potentials of facts are computed before creating the abstraction, and then, these strategies sort the variables according to the facts in that variable with the highest/lowest potential.

6.4 Metrics Based on the Resulting Children Abstract States

These strategies simulate the refinement in a copy of the transition system and then they use some criterion on the children abstract states to evaluate the quality of the refinement. That is, instead of using a criterion to determine if the split will be valuable, they measure the improvement of doing the split by evaluating the resulting abstraction. The biggest disadvantage of these strategies is the large overhead in simulating the refinement.

Refinement that Increases the Cost ($\texttt{plan}^{\uparrow}$): This strategy uses the cost of the optimal abstract plan after the refinement. A higher-cost optimal plan means the heuristic is more informed, but, as many optimal abstract plans may exist, it is not always a good measure because all refinements can tie at no cost increased.

Refinement that Increases the Goal Distance of the Split State ($\texttt{dist}^{\uparrow}$): This strategy uses the difference between the maximum goal distance of the children, and the goal distance of the parent as a measure of the quality of the refinement, since an increased distance provides more information. This strategy produces fewer ties than $\texttt{plan}^{\uparrow}$.

6.5 Filter Flaws before any Strategy

For some criteria that rate splits in absolute numbers like the new abstract plan cost after a refinement against the previous optimal plan cost, the amount increased is not very relevant, but the improvement in some quantity is a valuable indicator of the quality of the refinement. Thus, instead of choosing the split with the highest rating, we can use a secondary criterion to choose a split among the splits that satisfy the criterion, or among all splits if no split satisfies it.

This happens in criteria where the cost of operators is involved, and also in strategies that evaluate if the refinement is useful, since the actual relevant aspect is whether the cost of the operator is non-zero, or the refinement achieves an improvement respectively.

Consider only Flaws where a Non-Zero-Cost Operator is Involved: For an abstract transition $a^i \xrightarrow{o^{i+1}} a^{i+1}$, applying a refinement in a^i only if o^{i+1} has a non-zero cost can be a good strategy to focus on those parts of the plan where the cost is spent. This may seem a rare case only useful in domains with zero-cost operators, but in additive abstractions the cost of many operators is zero in the last abstractions, so this strategy may help to do refinements in non-fully-saturated costs.

Consider only Flaws that Increase the Plan Cost: This favours flaws where the cost of the optimal abstract plan is increased, which is a good measure of the usefulness of repairing the flaw. However, this has the same concerns as the $\texttt{plan}^{\uparrow}$ strategy: for most of the iterations all splits are filtered, because there are other optimal abstract plans, in such a way the secondary criterion is often used but with a high additional overhead.

Consider only Flaws that Increase the Goal Distance of Some Child: This strategy considers a split when the distance to the goal of some of the children abstract states is larger than the distance to the goal of the parent.

Consider only Flaws that Increase the Estimated Goal Distance: This uses a fast estimation of whether the new optimal abstract plan involving the same abstract states but replacing the parent by the children would have a new operator after the refinement:

- If the split state is the initial abstract state, the goal distance is increased if none of the optimal outgoing transitions in the parent exists in the new abstract initial state, since a new transition from the new abstract initial state (one of the children) to the other child is required to reach the second abstract state of the plan trace.
- If the split state is a goal abstract state, the goal distance is increased if none of the optimal incoming transitions reaches the child that becomes a goal abstract state, since in this case the second-last abstract state of the abstract plan trace has to reach the other (non-goal) child before reaching a goal state.
- If the split state is a middle state, the goal distance is increased if neither of the two children states contain both an optimal predecessor of the parent as a predecessor, and an optimal successor of the parent as a successor, since in this case both children states are needed to connect the prefix and the suffix of the abstract plan trace.

This estimation does not take into account the cost of the transition between the children, so it could incorrectly estimate a distance increase if the children are connected by a zero-cost operator. The advantage of this filter is that it is really fast compared to the other estimations of plan cost and distance increase. However, it can also be better than using the increase of the distance of some child in some cases, because the increase of the distance of a child may not be very relevant if the plan remains valid by using the other child.

6.6 Iterative Abstract Flaws (`it`)

This is a special strategy that searches flaws in a different manner. For that, it calls Algorithms 5–6 with different parameters to start in a different state.

The idea behind this strategy is that the flaws inherent to the abstraction, those that would happen in the task independently of the initial state and the goals, can be more critical that flaws very tied to the initial state and goals. Also, this strategy takes advantage of the insight that flaws closer to goals are used to produce better heuristics.

This works because starting the search of flaws in an intermediate abstract state allows finding prefixes (suffixes in regression) of the plan that are valid for no concrete state consistent with such an abstract state, instead of searching plan prefixes or suffixes not valid for the initial state.

In the forward direction, this strategy starts to search flaws at the end of the plan (Algorithm 5 parameters $r = a^n$, $i = n$) and it iteratively calls the algorithm from the previous step of the plan until finding a flaw (including the flaws where the final state is not a goal state). Finally, if no flaw is found from the initial abstract state, then it is searched from the concrete initial state. In the backward direction, this strategy is like the first flaw but starting from the goal abstract state instead of the goal partial state, returning the first flaw from the goal partial state if no flaw is found from the goal abstract state.

6.7 Refinement of All Flaws Found

These strategies refine all the flaws found before a new search, with a lower overhead due to less flawed states and split searches, but at the cost of refining all flaws without taking into account their quality.

This is possible because sequence flaws only find the flaws that would exist after refining the flaws found in the prefix (suffix in regression) of the plan, but the following situations can happen when more than one flaw is refined without searching again for additional flaws:

- The flaw to refine is part of an abstract plan, but this plan is not optimal because after refining the previous flaw the cost of the plan has increased and other optimal abstract plans exist with lower cost. This often happens, but it is not a major concern because not many refinements are done in a non-optimal plan, and also it will probably become again an optimal plan when the other optimal plans are refined.
- The flaw to refine is not part of an abstract plan. This only may happen if the children of the abstract states are not connected by any operator. This happens scarcely, and it is not either a major concern because few refinements in a non-plan sequence will be done until the next flaws search.

Two different strategies exist for this: prioritizing the flaws found in the abstraction or always taking into account all flaws, as explained below.

All Flaws (`|all|`): All flaws can be sequentially refined in the order they have been found before a new search, since sequence flaws are defined in such a way that a flaw is found in a subsequent step of the plan if and only if it exists after refining the previous flaws.
Note that this is only true when refining in only one direction, since progression and regression flaws can interfere among them.

All Flaws in the Abstraction (`|abs|`): This strategy also refines all flaws found but searching from the initial abstract state without taking into account the goals (from the goal abstract state without taking into account the concrete initial state for regression flaws).
However, in some cases, as in the trivial abstraction, no flaws are found in the abstraction, so when this happens the flaws are searched as usual from the initial concrete state/partial goal state. Notice that this only happens in few iterations, since after refining such flaws typically new flaws in the abstraction arise. The same concerns for bidirectional strategies as for `|all|` are applicable here.

7 Experiments

We analyse the performance of our strategies in three different settings: building a single abstraction, using additive abstractions over subtasks and in the state-of-the-art Scorpion planner (Seipp 2023). For the results in a single abstraction, we thoroughly analyse strategies at each high-level category and after that we compare the best strategies of each category. We perform this analysis only for a single abstraction because the effects of choosing flaws are more clear when there are no other factors like the subtasks used or the order in the saturated cost partitioning. We state the conclusions obtained from these deep analyses at the end of that subsections.

7.1 Experimental Setup

We implemented our flaw selection strategies within the Scorpion planner (Seipp 2018, 2023; Seipp, Keller, et al. 2020), a fork of Fast Downward 23.06 (Helmert 2006). For linear programs computed for potential heuristics we use CPLEX Studio 22.11.

Our experiments run on the Autoscale 21.11 benchmark set (Torralba, Seipp, et al. 2021), which contains 42 domains[2] of the International Planning Competitions (IPC) up to 2018 with 30 tasks scaled with the number of objects each, so for optimal planning runtime typically scales exponentially. This benchmark set uses for

[2]Domains not included are listed in https://github.com/AI-Planning/autoscale-benchmarks.

domains and tasks the standard specification based on first-order logic called PDDL (Planning Domain Definition Language) (Edelkamp and Hoffmann 2004; Fox and Long 2003; McDermott et al. 1998). Scorpion uses the Fast Downward algorithm to transform a PDDL representation into SAS^+ (Helmert 2009).

All experiments are limited to 30 minutes and 8 GB of RAM and run in a Debian 10.2 server with an AMD EPYC 7551 CPU at 2.5 GHz. Experiments are run with Downward Lab (Seipp, Pommerening, Sievers, et al. 2017).

For single abstraction experiments we set 10 million of non-looping transitions as the termination condition except when another limit is explicitly mentioned because the default value of 1 million is too low for a single abstraction, and for a better measurement of the performance penalty of computing sequence flaws. For additive abstractions we tried the default limit of 1 million of non-looping transitions in total for all abstractions but using 100 000 transitions achieves a better performance. For Scorpion experiments we use the default limit of 1 million of non-looping transitions or 100 seconds (what happens earlier) in total for all abstractions.

In Scorpion's implementation, goals are refined before starting the main CEGAR loop as an optimization, but it is disabled on all configurations to compute all flaws in all steps and keeping the implementation closer to the CEGAR algorithm. Another optimization, kept on all configurations, is refining all unreachable facts before goal on tasks with a single goal, which is needed for landmarks-based subtasks but does not hurt when used on the full task or goals-based subtasks. These landmarks are found with the algorithm to find h^m causal fact landmarks by Keyder et al. (2010) with $m = 1$. All experiments use incremental search to find optimal abstract plans (Seipp, von Allmen, et al. 2020), a technique much faster than A^*.

When multiple splits can be selected, we break ties according to the most refined split (Section 6.2), since it is the best strategy found in previous work (Seipp and Helmert 2018), and if multiple splits are tied after that, then we choose the first split found. However, for strategies based in the position of the flaw, the first criterion is the split that happens in the highest number of flaws, and then those criteria are used to break ties. When splitting an abstract state by a split variable v_s, the values of v_s not implied in the flaw can be assigned to any of the children. All experiments reported here use the "wanted" splitting strategy for progression flaws and the "unwanted" strategy for regression flaws, which result in moving them to the child farther from the goals in both cases, that is the child with the values where the flaw does not exist. This setting results in the most useful splits according to our theoretical findings and also got the best results in previous work (Pozo, Torralba, et al. 2024b). For more details, read Section 4.3.

Code and experimental data is available in Zenodo (Pozo, Torralba, et al. 2025).

7.2 Single Abstraction Experiments

In these experiments, the CEGAR algorithm is applied on a single abstraction. They converge (with infinite time and memory resources) to an optimal plan without searching, but additive abstractions over subtasks generally work better because the performance of abstractions decrease as their size is increased, and additive abstractions allow focusing each abstraction on a different part of the task.

Despite their lower performance, focusing on a single abstraction allows us to better understand the behaviour of each strategy without external factors like subtasks properties or the abstractions orders in the Saturated Cost Partitioning.

We first analyse each group of strategies to identify the best ones according to each criterion, and then we compare the best overall strategies at the end of the section.

7.2.1 Strategies Based on the Position of the Flaw. First, we analyse the behaviour of the first flaw, but applied also in regression and in a bidirectional strategy that selects the progression flaw or the regression flaw based on the split that fixes a greater number of flaws, breaking ties by the most refined split and after that by the first split found. Thus, we compare the first progression flaw (F), the first regression flaw (B), the first bidirectional

Table 1. Coverage of position strategies over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). The cell in row x and column y shows the number of domains where method x solved more tasks than method y. "Cov" indicates the total number of tasks solved.

	CEGAR								**A* + CEGAR**							
	1^{st} Flaw			Sequence Flaws					1^{st} Flaw			Sequence Flaws				
	F	B	D	$F_{\texttt{last}}$	$B_{\texttt{last}}$	$D_{\texttt{last}}$	$D_{\texttt{clo}}$	Cov	F	B	D	$F_{\texttt{last}}$	$B_{\texttt{last}}$	$D_{\texttt{last}}$	$D_{\texttt{clo}}$	Cov
F	–	**19**	**10**	**26**	**18**	**18**	**29**	**173**	–	0	0	7	**11**	**11**	7	416
B	4	–	4	**18**	9	9	**17**	133	**20**	–	**7**	**18**	**24**	**24**	**13**	**451**
D	5	**15**	–	**20**	**14**	**13**	**23**	151	**16**	4	–	**14**	**21**	**21**	**12**	446
$F_{\texttt{last}}$	2	5	2	–	3	1	**8**	110	**9**	3	2	–	**15**	**15**	4	420
$B_{\texttt{last}}$	2	**10**	9	**19**	–	2	**19**	146	2	0	0	2	–	**0**	1	400
$D_{\texttt{last}}$	2	**12**	9	**19**	**3**	–	**20**	148	2	0	0	2	**0**	–	1	400
$D_{\texttt{clo}}$	2	1	2	6	1	1	–	109	**13**	0	4	**8**	**17**	**17**	–	428

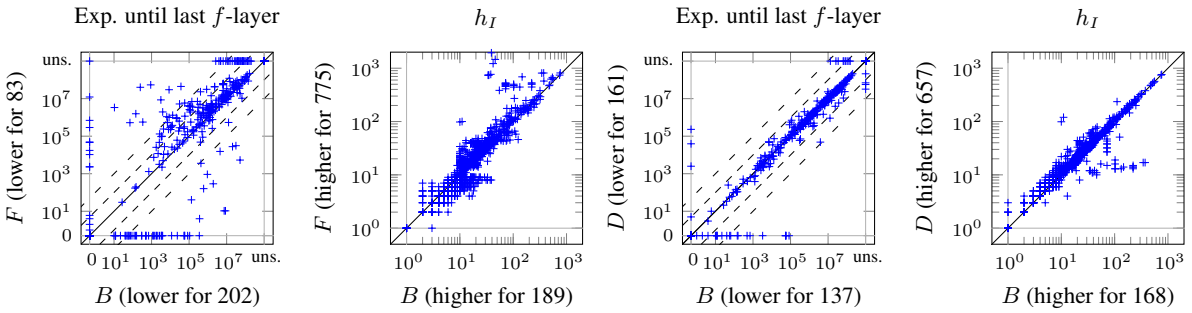


Fig. 14. Expansions and h_I of single abstractions: F vs. B (left) and B vs. D (right).

flaw (D)—which sets the same priority to the first progression flaw and the first regression flaw and uses tie-breaking criteria to choose one of them—, the last progression flaw ($F_{\texttt{last}}$), the last regression flaw ($B_{\texttt{last}}$), the last bidirectional flaw ($D_{\texttt{last}}$) and the bidirectional closest to the goal flaw ($D_{\texttt{clo}}$).

Table 1 compares these strategies in terms of coverage. We distinguish those tasks solved directly in the CEGAR refinement loop while building the abstraction from those solved by A* search using the resulting abstraction to compute the heuristic. Interestingly, the comparison in terms of tasks solved directly by the CEGAR algorithm is completely opposite to the tasks solved by an informed search with the resulting abstractions. Regression flaws are significantly worse than the baseline F in terms of tasks solved by the CEGAR algorithm, but they result in far better heuristics, solving more tasks. This can be observed in terms of total coverage, where F solves 40 more instances than B during the CEGAR loop but 35 fewer instances when combined with search. And the same happens on a per-domain basis, where in terms of tasks solved by CEGAR, F is superior to B in 19 domains and worse than it in only 4 of them. However, when combined with search, B solves more tasks than F in 20 domains and no task solved by F is not solved by B, which is a rare case in heuristics for Planning, where some are usually better in some domains and worse in others.

Figure 14 shows the expansions until the last f-layer and the heuristic value of the initial state for additional insights on the comparison of F and B. Expansions until the last f-layer are the number of concrete states expanded during A* search without taking into account the states with the optimal $f = g + h$ value, in such a way

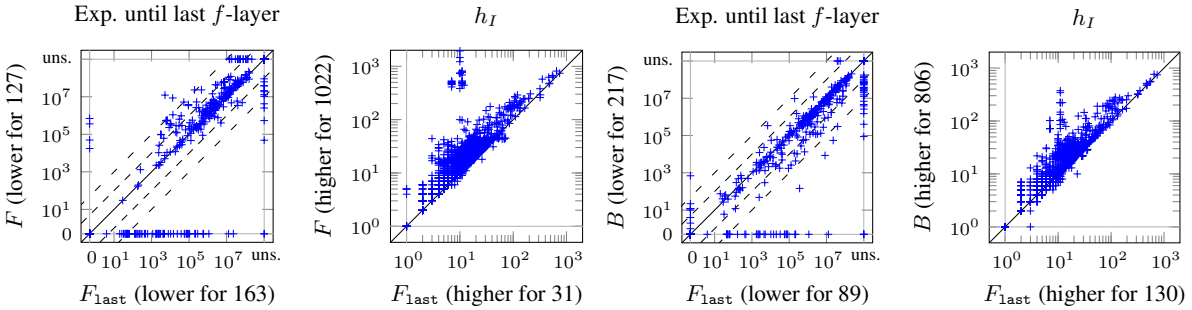


Fig. 15. Expansions and h_I: $F_{\texttt{last}}$ vs. F (left) and $F_{\texttt{last}}$ vs. B (right).

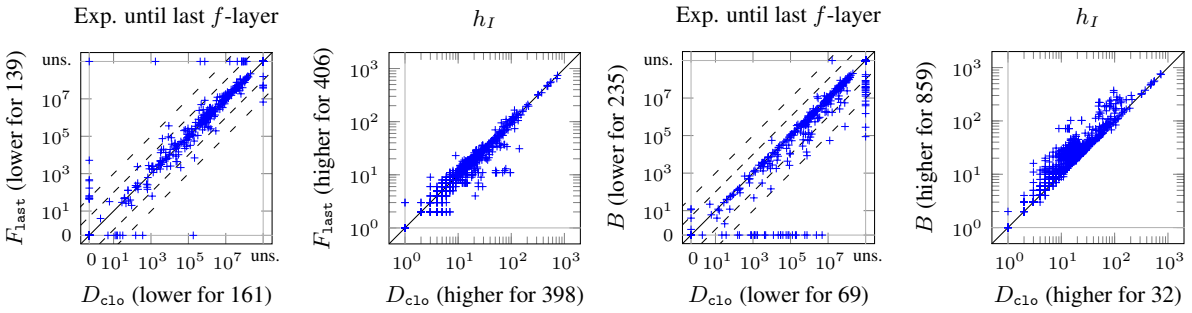


Fig. 16. Expansions and h_I: $D_{\texttt{clo}}$ vs. $F_{\texttt{last}}$ (left) and $D_{\texttt{clo}}$ vs. B (right).

that a perfect heuristic for the initial state gets 0 expansions with a consistent heuristic (such as abstractions). This confirms that F's abstractions obtain a higher heuristic value for the initial state (h_I) in most of the tasks,[3] and solve more tasks with no search (0 expansions), as the horizontal line of points in the 0 value of the leftmost plot shows. However, B's abstractions induce more informed heuristics in almost all cases, leading to fewer expansions. When comparing B and D, we see that they are very similar in terms of expansions and h_I, with most of the points around the diagonal and few tasks where also refining progression flaws is highly positive or highly detrimental. However, this depends on the task, and no heuristic consistently achieves fewer expansions in any domain.

For the last sequence flaw the behaviour is totally different, and the last progression flaw gets the best coverage when using A* (solving 4 more tasks than the first progression flaw) and solves fewer tasks during the refinement loop. This means that refining close to the goal is one of the main strengths of regression flaws, which solve even fewer tasks than F when the last regression flaw is refined instead of the first one. This is also reflected in $D_{\texttt{clo}}$, the best non-default sequence strategy based on the position of the flaw in spite of the overhead of computing flaws in both directions at each iteration of the refinement loop.

Figure 15 shows the number of expansions until the last f-layer and the initial heuristic value of $F_{\texttt{last}}$ against F and B. It is clear that $F_{\texttt{last}}$ has fewer expansions than F but a very low h_I in most tasks, while B is much better than $F_{\texttt{last}}$ under both criteria. Figure 16 shows the same comparisons for $D_{\texttt{clo}}$ against $F_{\texttt{last}}$ and B. In this

[3]In all h_I plots the maximum value has been bounded to 2000. This excludes the ParcPrinter domain, with a similar behaviour but orders of magnitude larger values, making visualisation difficult.

Table 2. h and f statistics of same-size abstractions in some selected instances.

	Logistics p_{04}				Transport p_{02}				DataNetwork p_{08}			
	h_I	$\overline{h}$	$\overline{f}$	Search (s)	h_I	$\overline{h}$	$\overline{f}$	Search (s)	h_I	$\overline{h}$	$\overline{f}$	Search (s)
F	**24**	9.9	**24.3**	**1.18**	**685**	133.8	687.2	2.12	**45**	5.8	**48.8**	104.74
B	21	**11.6**	21.3	4.30	637	220.9	688.8	**0.66**	30	2.2	30.1	183.12
D	22	11.2	23.2	2.23	651	**222.3**	**692.0**	0.71	32	2.2	32.1	180.06
$F_{\texttt{last}}$	18	11.4	18.3	4.87	600	205.3	652.8	0.99	17	**7.8**	20.5	**23.21**
$B_{\texttt{last}}$	16	5.7	16.6	107.28	637	183.2	643.9	1.63	**45**	2.1	45.0	181.80
$D_{\texttt{last}}$	16	6.2	16.9	88.90	636	182.5	642.3	1.24	**45**	2.1	45.0	181.97
$D_{\texttt{clo}}$	17	10.8	17.9	10.91	613	214.1	660.3	0.77	28	7.4	31.4	30.71

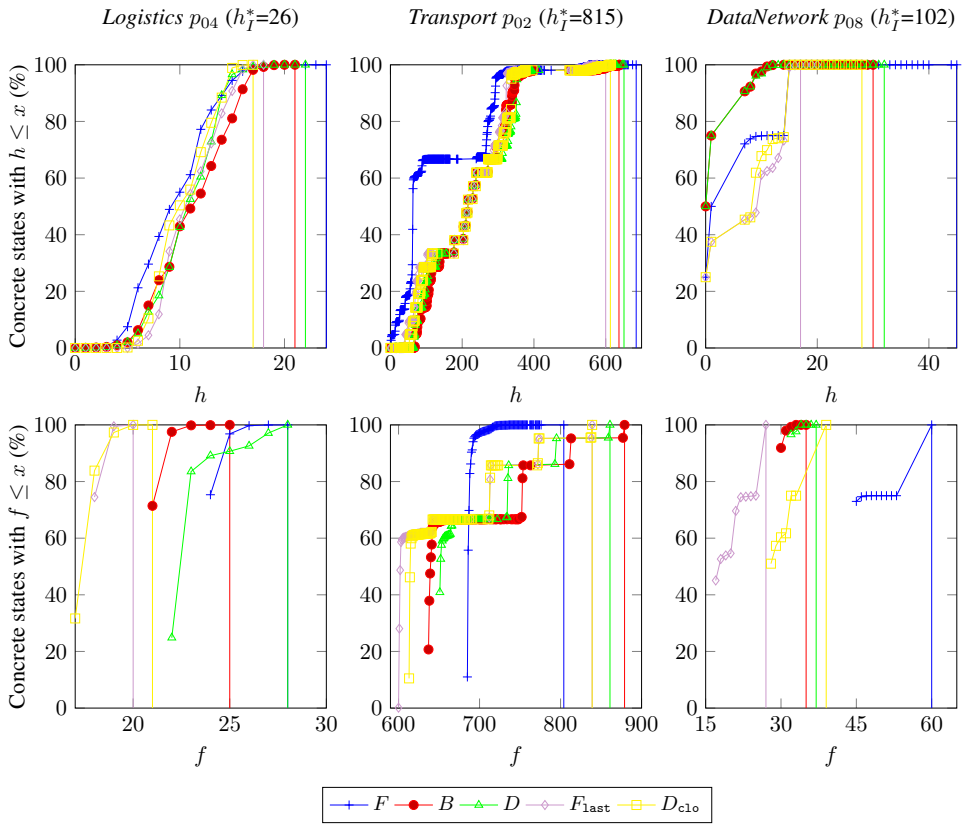


Fig. 17. Accumulated f distribution of same-size abstractions in some selected instances.

case, $D_{\texttt{clo}}$ is similar to $F_{\texttt{last}}$ in the number of expansions but stronger for some tasks, and clearly better in the initial heuristic value. Meanwhile, B is consistently better than $D_{\texttt{clo}}$ in both criteria, but this superiority is not so significant as against $F_{\texttt{last}}$.

To understand the counter-intuitive observation that the variants leading to a notably higher heuristic value for the initial state are overall less informed (A* performs more expansions), we take a closer look at the distribution of heuristic values on selected instances.

Table 2 shows the average h value, the average f value, h_I and search time of some representative instances using abstractions with a limit of 50 000 states for Logistics and Transport and 5 000 states for DataNetwork. We use the same number of states for all heuristics for a fair comparison, and the used number of states is similar to the greatest number obtained with 1 million of non-looping transitions. We use for this analysis smaller abstractions to avoid finding the solution during the CEGAR loop. Note that we can use the number of states as a fair limit for a reduced number of tasks but we cannot use it for the full benchmark because the number of states for a big enough abstraction that do not exhaust the available memory varies a lot among domains and tasks, so we use a fixed number of transitions as the general size limit. The value of g and h is obtained by using the distance to the abstract initial state and closest abstract goal state respectively for each abstract state, and then counting that value for the number of concrete states consistent with such abstract state. Therefore, the g value is the estimated distance to the initial state done by the abstraction and h is an estimation of how the abstraction estimates h^*, which is the h value used for concrete states. DataNetwork is one of the few domains where B performs worse than F (although F does not solve more problems with 10 million non-looping transitions abstractions), and the average heuristic value is also lower for B. However, in most instances the average heuristic value of B is higher despite having a smaller h_I, which explains why it gets a better coverage and lower search times. h_I is more influential in problems where only a few nodes are expanded, like the selected Logistics task. But in larger instances of the same and other domains, where more expansions are required, the average h value is essential. Works on potential heuristics have reported similar results, since optimizing the average heuristic value produces better heuristics than optimizing only the heuristic value of the initial state, with the best results obtained when both criteria are taken into account (Seipp, Pommerening, and Helmert 2015). Another factor is the distribution of the heuristic value (see Figure 17, where $B_{\texttt{last}}$ and $D_{\texttt{last}}$ have been omitted for the sake of readability), since having fewer states with low h values may be more helpful in the search than having a very high value on some states (Holte, Felner, et al. 2006). Indeed, we observe that this happens in some domains like Transport, where despite reaching a high h_I value, F has many states with very low h values. $F_{\texttt{last}}$ and $D_{\texttt{clo}}$ have mixed behaviours, being the best in DataNetwork (and overcoming the limitations of regression flaws in this domain) but working very poorly in Logistics, though they have a similar performance in Transport. As expected, $B_{\texttt{last}}$ and $D_{\texttt{last}}$ are the worst strategies in all domains, although they achieve a higher h_I in the DataNetwork domain. Also, it is noteworthy the disastrous average heuristic value and search time of $B_{\texttt{last}}$ and $D_{\texttt{last}}$ in Logistics, even 100 times higher than for B. The f value has the opposite behaviour, with the largest values for progression flaws, what explains why progression flaws usually find concrete plans quicker. This happens because progression flaws aim to increase the f value by repairing flaws in the optimal path from I, so they are better estimating the actual g value of states, as explained in Section 4. However, higher estimated average f values do not guarantee a better behaviour during the search, where higher h values especially close to goals are paramount.

As conclusions, **regression flaws generally get higher heuristic values and fewer states with very low heuristic values**, but **progression flaws are more focused on getting a concrete plan**. The **last regression sequence flaw is the worst one** because finding less accurate flaws with higher overhead close to the initial state is not useful. However, other strategies like the **last progression flaw and the closest to goal flaw at any direction work much better in some domains** like DataNetwork. **Bidirectional flaws are a trade-off between progression and regression flaws**.

7.2.2 Strategies Based on Split Properties. These strategies choose a split among all the splits in all abstract states of the optimal abstract plan, so they have a very high overhead. However, this can also be an advantage for the

Table 3. Coverage of different strategies based on split properties in forward, backward and bidirectional directions over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	CEGAR									**A* + CEGAR**								
	Forward			Backward			Bidir.			Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov
rand	–	–	104	–	–	**154**	–	–	121	–	–	395	–	–	**443**	–	–	412
Default	**27**	1	**173**	18	0	133	22	0	151	11	0	416	17	**2**	451	16	0	446
ref	21	0	159	21	0	145	21	0	142	11	0	410	16	0	429	15	0	415
-ref	11	0	97	15	0	106	12	0	89	9	0	395	11	0	426	10	0	418
cos	18	0	152	19	0	145	17	0	139	11	0	410	13	0	433	13	0	413
-cos	18	0	153	19	0	136	17	0	133	9	0	404	14	0	414	16	0	401
add	20	0	156	23	**4**	157	17	1	139	9	0	407	19	1	455	13	0	415
-add	14	0	108	14	0	103	12	0	97	7	0	399	9	0	413	7	0	392
un	19	0	148	16	0	135	16	0	127	9	0	405	13	1	429	11	0	402
-un	14	1	130	16	0	118	13	0	104	11	0	404	13	1	424	12	1	416
lm	20	0	149	19	0	146	19	1	142	11	0	403	16	0	427	11	0	404
lmr	17	0	144	19	0	147	16	0	133	8	0	399	15	0	428	10	0	398
pot	19	2	150	20	1	147	19	0	143	11	0	413	**22**	1	**461**	**22**	**2**	441
-pot	17	0	147	16	0	124	14	0	115	10	0	406	13	1	408	9	0	396
$*$	**42**	0	**189**	**42**	0	186	**42**	**1**	177	**42**	0	438	**42**	**5**	**509**	**42**	4	499
$**$			D: **42**			SD: **3**			Cov: **201**			D: **42**			SD: **7**			Cov: **513**

quality of the abstraction in case the selected split produces better refinements. These strategies choose the most and least refined split (ref and -ref), the split connected by an operator with the highest and lowest cost (cos and -cos), with the maximum and minimum h^{add} wanted value (add and -add), with the largest and smallest size on the split variable in the child that is farthest from the goals (un and -un), the split in causal landmarks ordered by their h^{add} value (lm and lmr), and the split on the wanted fact with the highest and lowest potential (pot and -pot). Also, we use a random split (rand) as a baseline.

Table 3 shows the coverage and the number of domains in which each strategy performs better (with the rest of strategies solving the same or fewer tasks) and strictly better (with the rest of strategies solving fewer tasks) for these strategies for progression flaws, regression flaws and bidirectional strategies. In the forward direction, pot is the best one, solving 3 fewer tasks than Default, and ref and cos solve 3 fewer tasks than pot. Meanwhile, for regression flaws the best strategy is also pot, which solves 461 tasks (10 more than Default), but add is also good, and it solves 4 more tasks than the first regression flaw, and 22 more tasks than the third-best sequence strategy, cos. When flaws in both directions are taken into account, pot is again the clear winner, but solving 5 fewer tasks than Default. The overhead of computing splits for all states is evident, and it is higher for bidirectional strategies because they must compute them in both directions. An interesting insight is that B_{add} is not only good at coverage but also solving tasks during the loop, being the second-best strategy, only worse than F. F_{un} and $F_{\text{-un}}$ are not very good overall but they are quite complementary to other strategies, being the most relevant strategies in Agricola, Blocksworld and Grid domains. pot and add are also very complementary, being B_{pot} better than B_{add} in 14 domains and B_{add} better than B_{pot} in 9 domains.

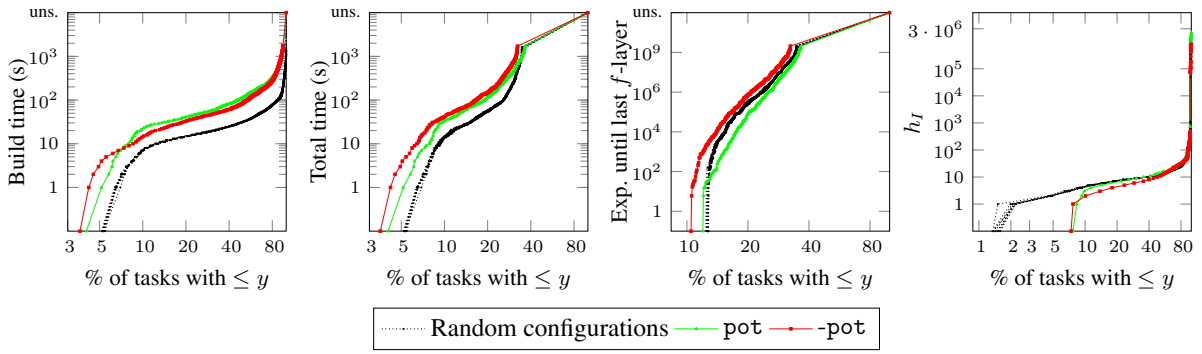


Fig. 18. Accumulated time to build the abstraction, total planner time, expansions until the last f-layer and h_I of random split configurations and the best (pot) and the worst (-pot) best split configurations for regression flaws.

In the two last rows of Table 3, the result of choosing the best strategy at each task is shown. The coverage during the refinement loop by choosing the best strategy in one of the directions is 189, very close to the coverage of F (173). But, more interestingly, choosing the best backward strategy gets very close, only 3 fewer tasks, and choosing the best strategy in any direction solves 12 more tasks than the best forward one, 201. Regarding the coverage by search, 48 more tasks could be solved by choosing always the best backward strategy, and the differences with respect to the forward direction are immense, 71 more solved tasks than choosing the best forward strategy for each task. This explains why choosing the best bidirectional strategy solves fewer problems, as regression flaws are often better and the overhead of computing progression flaws does not pay off. When the best strategy in any direction is used, 4 more tasks are solved during search, resulting in 513 solved tasks, and solving more tasks than any single strategy in 7 domains.

We executed a strategy that chooses a random split at each iteration with 10 different random seeds (1–10), and the first row of Table 3 shows the average coverage of these configurations. Choosing a random progression flaw resulted in a terrible performance: the average of random configurations for concrete solutions found is only better than -ref and finds 69 fewer solutions than the first flaw. The same happens in coverage during search, where the average of random configurations has the lowest performance, in a tie with -ref. Random regression flaws work much better, which means that regression flaws are useful not only because they are usually closer to goals but because they are more relevant to the heuristic. Bidirectional flaws have a behaviour in between the random progression and regression flaws, but they are still much worse than the best performers. Figure 18 shows the accumulated abstractions building time, total time, expansions until the last f-layer and h_I for regression flaws for all random configurations and the best (pot) and the worst (-pot) best-split strategies. Random configurations have a similar behaviour for all criteria, and almost in all domains the worst strategy is worse than them and the best strategy is significantly better. Regarding the building time of the abstractions, potential strategies are much worse because of the time to compute potentials, and even in some tasks of Agricola and Airport domains the abstraction cannot be built below the time limit. Interestingly, h_I is lower for all potential strategies, despite -pot is better in some tasks and pot is better in most of the tasks. Regarding total time and expansions, -pot is worse and pot is better than random strategies as expected, although if we conduct a comparison domain per domain, we can observe that -pot is not so bad in most of the domains, but only in Storage, Termes, Tetris and Transport.

We also run 100 random executions with different seeds (1–100) for one difficult but solved by the best strategies task of each domain (shown in Figure 19). Figure 19 shows random configurations as a box plot for each domain and compares them against the best and worst best-split-based strategies. In most domains, the best strategies

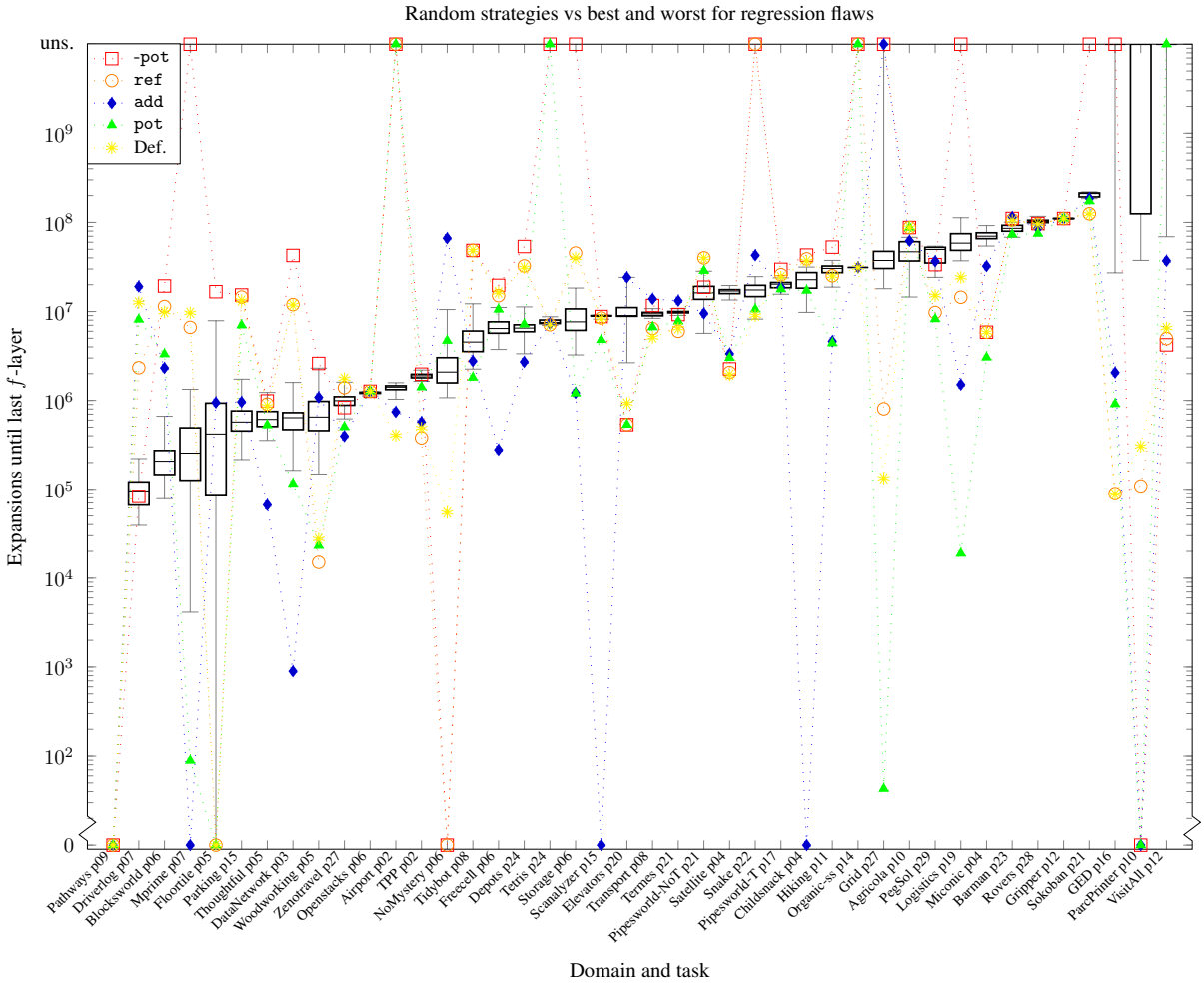


Fig. 19. Box plot of expansions until the last f-layer for 100 executions with different seeds for a random split against the best and worst best-split-based strategies. Tasks are sorted by the median of random executions and whiskers denote the minimum and maximum value.

get fewer expansions, with some notable examples like Mprime, ParcPrinter, GED or VisitAll. Also, the behaviour of pot and add are complementary, and one of them is the best in almost all domains. However, in a few domains our best strategies work worse than random splits, like in Driverlog and Blocksworld.

Summarising, **strategies based on potentials and $h^{\mathbf{add}}$ in regression work better than the first regression flaw**, but the overhead of computing all splits forwards do not pay off. **The best strategies are more informed and solve more tasks than selecting random flaws**, despite their higher building time. Also, **these strategies are very complementary**, and choosing the best one at each task would **solve 52 more tasks**.

7.2.3 Strategies Based on Variable Orderings. Strategies based on variable orderings require computing splits in all flawed states, as happens with the strategies based on the best split because any state could have a split on a

Table 4. Coverage of different strategies based on variable orderings for progression flaws over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	CEGAR									**A* + CEGAR**								
	Forward			Backward			Bidir.			Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov
$\overline{\texttt{rand}}$	–	–	104	–	–	**154**	–	–	121	–	–	395	–	–	**443**	–	–	412
$\overline{<_{\texttt{rand}}}$	–	–	145	–	–	148	–	–	142	–	–	409	–	–	433	–	–	414
Default	**26**	0	**173**	16	**2**	133	21	0	151	14	0	416	**22**	**2**	451	18	1	446
$<_{\texttt{CG}}$	16	0	146	13	0	139	15	1	152	14	0	411	12	0	412	12	0	407
$<^{\texttt{r}}_{\texttt{CG}}$	20	1	149	22	0	145	20	0	130	13	0	403	20	1	438	19	0	424
$<_{\texttt{lm}}$	19	0	166	**26**	**2**	168	23	0	163	11	0	415	**22**	1	**456**	21	**2**	436
$<^{\texttt{r}}_{\texttt{lm}}$	19	1	157	20	0	164	19	0	160	12	0	417	21	1	454	20	**2**	436
$<_{\texttt{pt}}$	16	0	150	18	1	156	17	0	149	14	0	410	19	0	442	17	0	416
$<^{\texttt{r}}_{\texttt{pt}}$	17	1	138	15	0	134	15	0	132	15	0	411	14	0	419	10	0	408
$*$	**42**	**3**	202	**42**	1	**210**	**42**	1	206	**42**	1	453	**42**	3	**494**	**42**	**4**	489
$**$			D: **42**			SD: **3**			Cov: **218**			D: **42**			SD: **5**			Cov: **500**

variable with a higher priority, so they also have a great overhead at each refinement step. Specifically, we order variables based on the topological order of the causal graph ($<_{\texttt{CG}}$ and $<^{\texttt{r}}_{\texttt{CG}}$), the variables with a causal landmark fact ordered by h^{add} ($<_{\texttt{lm}}$ and $<^{\texttt{r}}_{\texttt{lm}}$), and the variables with the highest and lowest potential fact ($<_{\texttt{pt}}$ and $<^{\texttt{r}}_{\texttt{pt}}$). We also compare them against random variable orderings as baselines.

Table 4 shows the coverage for each strategy in progression flaws, regression flaws and when both are used. Some strategies are slightly better than the first flaw. Also, these strategies are very complementary and each one solves more tasks in different domains. $B_{<^{\texttt{r}}_{\texttt{CG}}}$ solves only 13 fewer problems than B, and it dominates in 20 domains and strictly dominates in 1 domain. $B_{<_{\texttt{pt}}}$ solves only 9 fewer problems than B, and $B_{<_{\texttt{lm}}}$ solves 5 more tasks than B and it dominates in 22 domains and strictly dominates in another domain, while using the other landmarks order dominates in 21 domains and strictly dominates in another domain, solving only 2 fewer tasks than $B_{<_{\texttt{lm}}}$. This complementary behaviour can also be seen in the performance of the strategy that choose the best one at each task, that solves 43 more tasks than B, and even 6 more tasks when the best strategy of both directions is used, solving more tasks than any individual strategy in 5 domains. Specifically, landmark order strategies are good at Agricola, Barman, Blocksworld, Floortile, Logistics and Tetris domains, while potentials orders are good at Blocksworld, Tetris, Transport and VisitAll, and $<^{\texttt{r}}_{\texttt{CG}}$ is good at Blocksworld, Floortile, Hiking, ParcPrinter, Pathways, Pipesworld-Tankage and Snake.

We executed the best variable ordering strategy, $<_{\texttt{lm}}$, with random tie-breaking policies with 10 different random seeds (1–10) to determine if its good performance is due to chance or because it is actually a good strategy. All executions achieved the same number of expansions and heuristic value of the initial state in all solved instances, with small variations in the time needed to build the abstraction. Coverage is also the same (456 solved instances) except for one of them where in the task 19 of Tetris it lasts around 50 more seconds to build the abstraction resulting in a search timeout, because the rest of configurations solve it few seconds before the time limit. Therefore, we can conclude that the strategy is actually good and tie-breaks do not affect its performance.

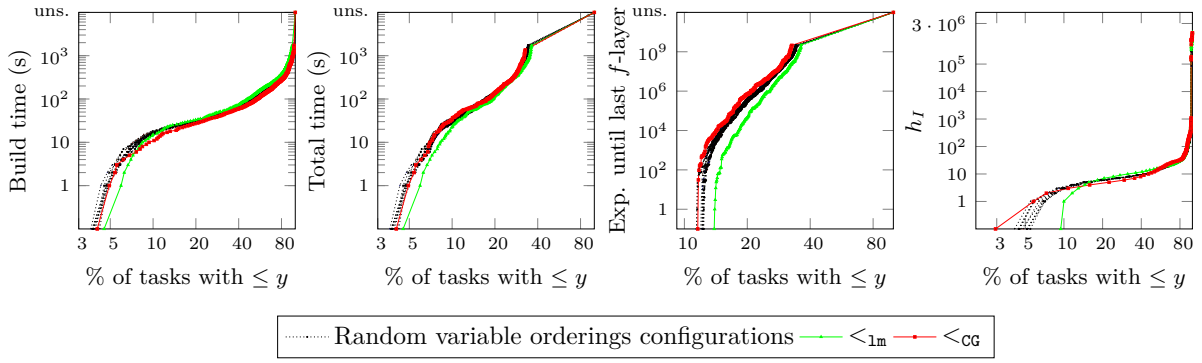


Fig. 20. Accumulated time to build the abstraction, total planner time, expansions until the last f-layer and h_I of random variables orders configurations and the best ($<_{\mathtt{lm}}$) and the worst ($<_{\mathtt{CG}}$) variable orders configurations for regression flaws.

We also executed 10 configurations with totally random variable orders and different random seeds (1–10). The average coverage of these executions is shown in the second row of Table 4. For concrete solutions found, the average of random variable ordering executions are worse than random splits and only better than $<^{\mathtt{r}}_{\mathtt{pt}}$ for progression flaws, and $<^{\mathtt{r}}_{\mathtt{CG}}$ for bidirectional flaws, and it gets better results for regression flaws but still far away from the best performers. Regarding coverage during search, we observe a similar behaviour, only improving $<^{\mathtt{r}}_{\mathtt{CG}}$ for progression flaws, and $<^{\mathtt{r}}_{\mathtt{pt}}$ and $<_{\mathtt{CG}}$ for regression and bidirectional flaws. Therefore, we can conclude that regression flaws are better for any strategy but still landmark and the maximum potential variable orderings improve significantly their effectiveness. We can also conclude that random variable orderings are worse than choosing a random flaw at each iteration of the refinement loop. Figure 20 shows the accumulated abstractions building time, total time, expansions until the last f-layer and h_I for regression flaws for all random configurations and the best ($<_{\mathtt{lm}}$) and the worst ($<_{\mathtt{CG}}$) variable orderings strategies. Random variable orderings configurations have a similar behaviour for the four criteria, although the abstractions building time is different. The building time is much higher for the landmarks strategy, but it is partially explained by the time needed to compute landmarks. Anyway, spending time in building a good abstraction usually pays off. As expected, the total time is lower for $<_{\mathtt{lm}}$ and higher for $<_{\mathtt{CG}}$, while random strategies are in the middle. These differences are more apparent in terms of expansions, where $<^{\mathtt{r}}_{\mathtt{lm}}$ clearly requires fewer expansions than the rest. Regarding h_I, $<_{\mathtt{CG}}$ gets the highest values despite being the worst strategy, but $<_{\mathtt{lm}}$ also gets better values than random configurations for some tasks, although it gets h_I equal to 0 and 1 for more than 10% of the tasks.

We also run 100 random executions with different seeds (1–100) for a task of each domain solved with many expansions by the best strategies of each domain, in such a way that this benchmark contains hard but solvable tasks. Figure 21 shows random variable orderings configurations as a box plot for each domain, and compares them against the best and worst variables-ordering-based strategies. In most of the domains, the best strategies get fewer expansions, with some notable examples like Mprime or Sokoban. Also, $<_{\mathtt{lm}}$ and $<^{\mathtt{r}}_{\mathtt{lm}}$ are quite complementary, and one of them gets the lowest number of expansions in almost all domains. The behaviour of $<_{\mathtt{CG}}$ and $<^{\mathtt{r}}_{\mathtt{CG}}$ is also very complementary but, although they are even the best in some domains, they need more expansions in most of them. In general, random variable orderings work slightly worse than random splits and the best split strategies, with a large variance.

Concluding, a **fixed variable ordering is not as useful as some split properties**. But **orderings based on landmarks get better results than the first regression flaw** and **much better than random orderings**. Also, these strategies are **very complementary**, and choosing the best one at each task **solves 44 more tasks**.

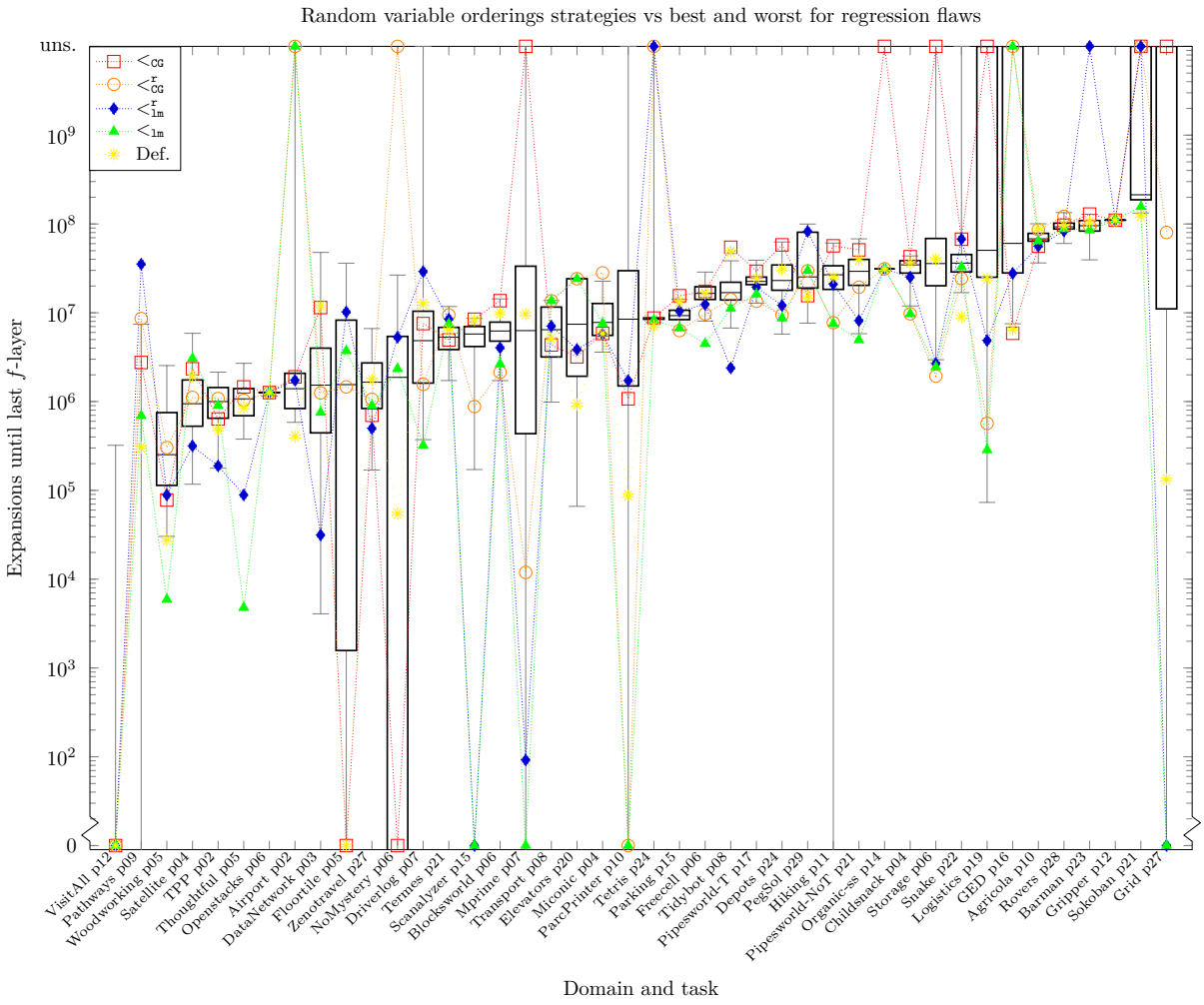


Fig. 21. Box plot of expansions until the last f-layer for 100 executions with different seeds for a random variable ordering against the best and worst variables-ordering-based strategies. Tasks are sorted by the median of random executions and whiskers denote the minimum and maximum value.

7.2.4 All Found Flaws. Refining all flaws generates the abstractions faster because all splits of every refinement step are refined, so fewer flaws and splits must be computed to get the same number of refinements. We evaluate the use of all flaws in progression and regression ($F_{|\mathtt{all}|}$ and $B_{|\mathtt{all}|}$) and all flaws found starting from the initial abstract state in progression ($F_{|\mathtt{abs}|}$) and from the goal abstract state in regression ($B_{|\mathtt{abs}|}$).

However, the quality of refinements is lower because some splits are not so good to improve the abstraction but are refined anyway. Also, after the first refinement the rest of splits may not be part of an optimal plan, but still refined. Note that the order of refinements is essential, and this procedure can also refine splits in a different order than other strategies, increasing the number of refinements to get an abstraction of the same quality.

Table 5. Coverage of refining all flaws over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). The cell in row x and column y shows the number of domains where method x solved more tasks than method y. "Cov" indicates the total number of tasks solved.

	CEGAR							**A* + CEGAR**						
	1^{st} Flaw		Sequence Flaws					1^{st} Flaw		Sequence Flaws				
	F	B	$F_{\|\texttt{all}\|}$	$B_{\|\texttt{all}\|}$	$F_{\|\texttt{abs}\|}$	$B_{\|\texttt{abs}\|}$	Cov	F	B	$F_{\|\texttt{all}\|}$	$B_{\|\texttt{all}\|}$	$F_{\|\texttt{abs}\|}$	$B_{\|\texttt{abs}\|}$	Cov
F	–	**19**	**36**	**36**	**36**	**36**	**173**	–	0	**35**	**32**	**34**	**31**	416
B	4	–	**35**	**35**	**35**	**35**	133	**20**	–	**38**	**35**	**37**	**35**	**451**
$F_{\|\texttt{all}\|}$	**8**	**17**	–	**17**	**6**	**16**	**178**	**11**	2	–	2	4	2	432
$B_{\|\texttt{all}\|}$	4	**0**	2	–	2	0	133	**19**	0	**13**	–	**12**	1	448
$F_{\|\texttt{abs}\|}$	6	**17**	3	**17**	–	**16**	174	**15**	6	**8**	6	–	6	442
$B_{\|\texttt{abs}\|}$	5	**3**	2	**3**	2	–	137	**20**	**2**	**15**	**3**	**13**	–	**451**

Table 5 shows the coverage of the first flaw, all flaws and all refinements in the abstraction in progression flaws and regression flaws. Note that it is not possible to refine all flaws in both directions simultaneously, unless some criterion is used to choose a direction at each step. In the forward direction, the coverage when all flaws are refined is surprisingly higher than the first flaw, and doing that only in the abstraction solves 10 more tasks, while refining all flaws gets more plans during the refinement loop. In the backward direction, the coverage when all flaws are taken into account is slightly lower but the same when only abstract flaws are taken into account. The heuristic is better in all directions if only abstract flaws are used, and this is one reason behind the good results of iterative strategies, especially forward. Note, however, that refining all flaws is not as effective as selecting a good flaw in each iteration, since the performance is better for some of the above strategies.

As conclusions, **splitting all flaws works better than the first flaw forwards** but **worse than the first flaw and other strategies backwards** because all flaws are refined in spite of their quality. **Refining only the flaws found in abstract states work better** than refining all flaws.

7.2.5 General Analysis. Table 6 compares the best strategies analysed in the previous subsections with the strategies not analysed yet and Table 7 shows the effects of filters over the best strategies. `it` is the iterative strategy that searches for flaws from the last abstract state of the abstract plan trace to the initial abstract state stopping in the first flaw found, and $|^{\texttt{clo}}_{\texttt{ref}}$ balances the distance to the goal and the refinedness of the split. $\texttt{plan}^{\uparrow}$ and $\texttt{dist}^{\uparrow}$ simulate a refinement for each split to evaluate which of them is the best one to apply. This is computationally expensive, so all abstractions that use them have a limit of 100 000 instead of 10 million non-looping transitions. The performance for these 100-times-smaller abstractions is not bad taking into account their size, so they are good criteria for refinements, and reducing their computation time would make them competitive. $\texttt{plan}^{\uparrow}$ gets slightly worse results than $\texttt{dist}^{\uparrow}$ because in some situations a refinement may improve the abstraction without increasing the plan cost due to other non-altered optimal abstract plans exist.

$|^{\texttt{clo}}_{\texttt{ref}}$ works bad for progression flaws, despite `last` and `ref` being two of the best non-default strategies for progression flaws and its motivation of combining their benefits. For regression flaws, however, it gets good results in comparison with other strategies, although worse than the best strategies compared here.

Finally, the iterative strategy provides very good results in the forward direction, only a bit worse than $|\texttt{abs}|$, and does not provide bad results in the backward direction, but worse than the first regression flaw.

The good results and the expensive computation of simulating refinements moved us to design more efficient ways of computing them in the form of filters (see Section 6.5 for more details). For plan cost and goal distance we have increased the limit to 1 million non-looping transitions, while the estimated goal distance and non-zero-cost

Table 6. Coverage of the best strategies in forward, backward and bidirectional directions over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	CEGAR									**A* + CEGAR**								
	Forward			Backward			Bidir.			Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov
Default	**26**	0	173	17	0	133	22	0	151	11	0	416	15	1	451	12	0	446
pot	20	1	150	19	1	147	19	0	143	9	0	413	17	1	**461**	**19**	**3**	441
add	19	0	156	23	**2**	157	16	1	139	7	0	407	17	2	455	11	0	415
$<_{\texttt{lm}}$	19	0	166	25	**2**	168	23	0	163	8	0	415	15	1	456	15	2	436
$<_{\texttt{pt}}$	15	0	150	17	1	156	16	0	149	10	0	410	14	0	442	11	0	416
$\lvert\texttt{all}\rvert$	**26**	**2**	**178**	17	0	133	–	–	–	12	0	432	14	0	448	–	–	–
$\lvert\texttt{abs}\rvert$	23	0	174	18	0	137	–	–	–	11	0	442	15	0	451	–	–	–
it	18	0	140	20	0	144	13	0	120	15	0	437	14	1	443	10	0	427
$\lvert^{\texttt{clo}}_{\texttt{ref}}$	10	0	106	21	0	151	10	0	108	8	0	400	13	0	439	10	0	407
$\texttt{plan}^{\uparrow}_{\texttt{100k}}$	7	0	58	6	0	48	8	0	51	6	0	397	7	0	394	7	0	395
$\texttt{dist}^{\uparrow}_{\texttt{100k}}$	7	0	60	7	0	54	6	0	53	6	0	403	7	0	401	8	1	410
$*$	**42**	2	201	**42**	2	**209**	**42**	**3**	199	**42**	0	465	**42**	**2**	**508**	**42**	1	501
$**$			D: **42**			SD: **5**			Cov: **215**			D: **42**			SD: **3**			Cov: **515**

operators are limited to 10 million non-looping transitions because their lower computation time. But applying these filters results in weak heuristics, as Table 7 shows. This happens because good strategies select a refinement that improves the heuristic most of the time, so the time spent in computing filters does not pay off, and also filters focus only in the next refinement, with a bias against the original long-term purpose of the main strategy.

7.3 Additive Abstractions Experiments

The use of additive abstractions created by using saturated cost partitioning improves the performance of CEGAR in most of the domains, as this technique uses smaller abstractions that combine their power by focusing on different parts of the task. It is paramount that abstractions are different from each other, and to achieve it the state of the art creates an abstraction for each subtask. Subtasks are generated by setting each goal as the only goal of the subtask and with a subtask for each landmark ℓ obtained via the algorithm by Keyder et al. (2010) for h^m causal fact landmarks with $m = 1$ where ℓ is set as the only goal in a domain abstraction of the task that reduces the domain of some variables by combining the landmark facts that must be achieved before ℓ, and removing all facts that cannot be achieved before reaching ℓ to keep the admissibility (Seipp and Helmert 2018). The saturated cost partitioning consists in subtracting the actual cost used by an abstraction (the maximum goal distance decreased in the successors of any abstract state by each operator) from the cost operators for the following abstractions (Seipp and Helmert 2018). In this way, the added cost of operators among all abstractions is always lower or equal than the actual cost.

Table 8 compares the results of the best strategies for additive abstractions. We also tried the default limit of 1M non-looping transitions added over all abstractions and 10M non-looping transitions, but they obtained a lower performance than using 100k non-looping transitions added over all abstractions as the limit, so this is the limit used in all the results reported in this section. Some of the best strategies are the same as for a

Table 7. Coverage of filtered strategies and their non-filtered counterparts in forward, backward and bidirectional directions over a single abstraction when considering only the tasks solved with the CEGAR algorithm (left side), and search with the resulting abstraction (right side). "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	CEGAR									**A* + CEGAR**								
	Forward			Backward			Bidir.			Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov	D	SD	Cov
Default	**31**	**1**	**173**	18	0	133	23	0	151	13	0	416	**21**	**2**	**451**	19	1	446
it	18	**0**	140	**21**	**0**	**144**	13	**0**	120	18	0	437	**21**	**1**	**443**	15	0	427
it$^{\|\sim d}$	12	**0**	114	17	**0**	132	8	**0**	41	15	0	419	12	0	388	4	0	233
it$^{\|dist}_{lm}$	7	**0**	76	8	**0**	67	7	**0**	31	8	0	370	5	0	229	1	0	256
it$^{\|plan}_{lm}$	7	**0**	74	8	**0**	67	7	**0**	29	7	0	362	4	0	235	0	0	269
it$^{\|0cos}$	17	**0**	136	**21**	**0**	**144**	8	**0**	50	17	0	431	20	0	442	5	0	251
ref	**24**	**0**	**159**	21	**0**	145	21	**0**	142	13	**0**	410	19	**0**	**429**	**20**	**0**	415
ref$^{\|\sim d}$	22	**0**	154	21	**0**	138	21	**0**	139	11	**0**	394	12	**0**	352	10	**0**	325
ref$^{\|dist}_{lm}$	9	**0**	92	7	**0**	71	7	**0**	64	4	**0**	256	2	**0**	169	0	**0**	132
ref$^{\|plan}_{lm}$	9	**0**	90	7	**0**	66	7	**0**	62	2	**0**	266	1	**0**	163	1	**0**	128
ref$^{\|0cos}$	23	**0**	158	22	**0**	146	21	**0**	143	13	**0**	409	19	**0**	**429**	18	**0**	404
$\|^{clo}_{ref}$	10	**0**	106	**23**	**0**	**151**	10	**0**	108	11	**0**	400	**19**	**0**	**439**	15	**0**	407
$\|^{clo\|\sim d}_{ref}$	12	**0**	110	21	**0**	142	10	**0**	106	12	**0**	395	12	**0**	356	9	**0**	345
$\|^{clo\|dist}_{ref\ lm}$	7	**0**	58	7	**0**	71	7	**0**	54	4	**0**	251	2	**0**	168	2	**0**	172
$\|^{clo\|plan}_{ref\ lm}$	7	**0**	57	7	**0**	69	7	**0**	52	3	**0**	251	1	**0**	163	2	**0**	156
$\|^{clo\|0cos}_{ref}$	11	**0**	108	**23**	**0**	150	9	**0**	106	11	**0**	398	**19**	**0**	434	14	**0**	403
add	23	**0**	156	**27**	**0**	157	19	**0**	139	11	0	407	**26**	**1**	**455**	14	0	415
add$^{\|\sim d}$	19	**0**	143	23	**0**	151	19	**0**	138	9	0	393	8	0	302	5	0	288
add$^{\|dist}_{lm}$	7	**0**	85	8	**0**	87	8	**0**	73	6	0	275	0	0	145	0	0	140
add$^{\|plan}_{lm}$	8	**0**	86	8	**0**	77	8	**0**	69	4	0	281	0	0	141	0	0	131
add$^{\|0cos}$	22	**0**	155	**27**	**0**	**158**	19	**0**	142	9	0	400	24	0	453	13	0	406
$*$	**42**	**1**	182	**42**	**1**	**183**	**42**	0	173	**42**	0	452	**42**	1	**489**	**42**	**2**	471
$**$			D: **42**			SD: **2**			Cov: **195**			D: **42**			SD: **3**			Cov: **492**

single abstraction, but others behave worse for additive abstractions, like add and all strategies based on variable orderings. Interestingly, simulating a refinement to check if the cost of the optimal abstract plan has been increased is the best strategy, but using the increased distance of some of the children works slightly worse. The main reason is that increasing the plan cost is the most difficult task with saturated costs, since the cost of most of the operators is 0, and so using a strategy that enforces increasing it addresses this problem to some extent. Filters were executed with the same limits as the other strategies but they work as badly as for a single abstraction, with the non-zero-cost operators filter somewhat decreasing the coverage, while other filters are much more detrimental. A remarkable aspect of potential-based and landmark-based strategies is the computation of potentials and landmarks for each subtask, so the time needed to build abstractions is significantly higher for these strategies, although their coverage is among the best (except for the minimum potential) because the

Table 8. Coverage of the best strategies in forward, backward and bidirectional directions for additive abstractions for search with the resulting abstraction. "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov
Default	24	0	484	29	0	496	28	**1**	492
clo	23	0	489	29	0	496	29	0	494
ref	22	0	479	29	0	496	29	0	495
cos	25	0	485	29	0	495	28	0	496
add	23	0	478	24	0	476	26	0	479
lm	23	0	480	29	0	496	30	0	497
lmr	23	0	480	29	0	496	30	0	497
pot	22	0	478	29	0	494	29	0	493
-pot	22	0	463	25	0	471	27	0	473
$<^{r}_{CG}$	22	0	462	23	0	469	25	0	467
$<_{lm}$	23	0	463	25	0	476	25	0	467
$<^{r}_{lm}$	22	0	466	25	0	473	26	0	473
$<_{pt}$	16	0	448	21	0	463	21	0	454
$<^{r}_{pt}$	19	0	450	20	0	458	21	0	456
|all|	26	0	489	29	0	496	–	–	–
|abs|	26	0	494	29	0	496	–	–	–
it	26	**1**	496	29	0	496	29	0	495
$\|^{clo}_{ref}$	29	0	496	29	0	497	28	0	494
plan$^{\uparrow}$	26	0	487	30	0	**500**	30	**1**	**500**
dist$^{\uparrow}$	30	0	490	**31**	**1**	493	30	0	488
$*$	**42**	**5**	511	**42**	2	507	**42**	4	**512**
$**$			D: **42**			SD: **7**			Cov: **516**

subtasks are smaller and require lower times to compute them. The results are very similar for all strategies because abstractions and tasks are small and also because getting useful abstractions when the costs of operators are very saturated is really hard. The exception are the strategies based on variable orderings, which perform much worse because in landmark-based subtasks the facts are combined and thus almost all variables have a similar score, that are practically blind. add also gets a low coverage but this is mainly because it works very poorly in Mprime and Sokoban domains.

7.4 Scorpion Experiments

Scorpion is a state-of-the-art optimal planner (Seipp 2018, 2023) that has achieved impressive results in the last International Planning Competitions. It uses additive Cartesian abstractions created by CEGAR and PDBs combined by online saturated cost partitioning (Seipp 2021), with the additions of the h^2 preprocessor (Alcázar and Torralba 2015) and atom-centric stubborn sets pruning (Röger et al. 2020; Valmari and Hansen 2010).

Table 9 shows the coverage of Scorpion for the best strategies applied on the Cartesian abstraction. In Scorpion, significant differences exist among strategies. Those trying to improve the plan cost or the children distance are too slow in some domains like Mprime, getting up to 11 fewer solved instances than the best strategy in the same direction. Landmarks also work badly, because they do not get a strong heuristic even if they do many

Table 9. Coverage of the best strategies in forward, backward and bidirectional directions for Scorpion for search with the resulting abstraction. "D" means the strategy is the best or solves as many problems as the best in such domains and "SD" is the same but only for the domains where the strategy is the single best. "Cov" indicates the total number of tasks solved. $*$ is the result of using the best strategy of each direction on each task and $**$ is the same for all directions.

	Forward			Backward			Bidir.		
	D	SD	Cov	D	SD	Cov	D	SD	Cov
Default	26	**1**	727	28	0	739	27	0	738
`clo`	26	0	734	28	0	739	23	0	735
`ref`	24	0	737	27	0	**749**	22	0	743
`lm`	21	0	727	22	0	729	23	0	727
`pot`	26	0	739	24	0	743	27	**1**	745
`|all|`	27	0	734	28	0	740	–	–	–
`|abs|`	26	0	742	29	0	741	–	–	–
`it`	28	0	742	**30**	0	740	23	0	738
$\|^{\texttt{clo}}_{\texttt{ref}}$	27	0	742	26	0	747	27	0	742
`dist`$^{\uparrow}$	24	0	738	26	0	742	24	0	741
`plan`$^{\uparrow}$	22	0	731	26	**1**	740	25	0	737
$*$	**42**	**2**	762	**42**	**2**	**766**	**42**	**2**	764
$**$			D: **42**			SD: **5**			Cov: **770**

refinements. The best strategy for progression flaws is the balance between the most refined flaw and the closest to goal flaw, but for regression flaws refining directly the most refined flaw is better, solving 749 tasks, 10 more than the first regression flaw. $|^{\texttt{clo}}_{\texttt{ref}}$ solves only 7 fewer tasks for progression flaws, solving up to 15 more tasks than the first progression flaw, while the best bidirectional strategy, `pot`, solves only 4 fewer tasks than `ref` for regression flaws. These strategies are also very complementary to each other, with 17 more tasks being solved when the best regression strategy is used for each task.

In conclusion, Scorpion with regression sequence flaws **solve up to 22 more tasks** than the original Scorpion.

8 Related Work

In this section we explain the relation of our work to similar approaches for model-checking—the research area where the CEGAR algorithm was originated—and other classical planning techniques.

8.1 CEGAR in Model-Checking

Counterexample-Guided Abstraction Refinement (CEGAR) was initially conceived more than two decades ago as a technique for program verification via symbolic model checking (Clarke, Grumberg, Jha, et al. 2000, 2003), and is a well-established technique in the verification of software and hardware systems (Ball et al. 2001; Beyer and Löwe 2013; Leucker et al. 2015). Thus, there has been extensive research about how to guide the CEGAR process in that context (Albarghouthi 2015; Hajdu and Micskei 2020; Hajdu, Tóth, et al. 2016; Löwe 2017). This line of research was the inspiration for the use of CEGAR in classical planning (Seipp and Helmert 2013), and some of the extensions in model-checking and verification are related to our new methods for finding flaws. To clarify the relationship to our work, we first briefly describe the area and its connection to planning.

A common problem in model-checking is proving *safety properties*, where the objective is to check if a set of unsafe states is never reached by the program flow (Clarke, Grumberg, and Peled 1999). So, as in classical planning, we are interested in whether there exists some sequence of transitions that goes from some initial

state to a goal (error) state or proving its absence. Furthermore, the state space is also represented via a compact representation such as a Symbolic Transition System $S = (V, R, T, S_0)$, where V is a set of variables with arbitrary domains, R is an invariant formula over V that must hold for every state, T is the transition formula over $V \cup V'$, and S_0 is a formula over V, which defines the set of initial states (Hajdu, Tóth, et al. 2016).

Despite the similarities, there are also some significant differences to the problem of optimal planning with the SAS$^+$ planning representation we consider. Firstly, in some cases state variables may have infinite domains (e.g., numeric variables) resulting in an infinite state space, though oftentimes this can be restricted by the invariant formulas. Secondly, the representation of the transition relation is often more expressive than SAS$^+$ operators whose preconditions and effects are simply partial states. In some cases, the transition relation is represented as an arbitrary formula. In others, there may be some restrictions (e.g., using control-flow automata (Beyer, Henzinger, et al. 2007; Hajdu and Micskei 2020)) but even then transitions that increment a variable by 1 cannot be encoded as a SAS$^+$ operator without conditional effects. Finally, in model checking the focus is on determining whether an error trace (plan) exists; consequently, techniques are primarily tailored toward proving the absence of such traces. In that context, abstractions should aim to separate states that belong to a solution (regardless of the cost) from those that do not. In contrast, in planning our focus is to find optimal solutions on solvable tasks. Therefore, admissible heuristics giving good lower bounds on solution cost are very useful to reduce search effort.

The CEGAR framework in model checking works very similarly as we consider here. As in planning, different types of abstractions can be considered: such as predicate abstraction (Graf and Saïdi 1997), and explicit-value abstraction (Clarke, Gupta, et al. 2004), which similarly to Pattern Databases, partitions variables between visible or not. The closest representation to ours is Cartesian predicate abstraction (Ball et al. 2001) that aims to address the overhead on determining the abstract transitions, even though in their case this is still worst-case exponential.

Several approaches have been used to refine abstractions within the CEGAR framework in model-checking, including Craig interpolation and sequence interpolation. Consider an abstract error trace with n steps, passing through abstract states represented as arbitrary formulas $S_0, \ldots, S_n$ that describe the set of concrete states mapped to them, with transitions that have transition relations $T_1, \ldots, T_{n-1}$. To simplify the discussion, we assume that all concrete states mapped to S_1 are valid initial states and all concrete states mapped to S_n are error states. Then the abstract trace corresponds to a real error trace if and only if the formula $\phi = S_1 \wedge T_1 \wedge S_2 \wedge T_2 \wedge \cdots \wedge T_{n-1} \wedge S_n$ is satisfiable. If the formula is unsatisfiable, we need to decide how to refine some of the abstract states, which can be done by using interpolation. Given a pair of formulas $\langle A, B \rangle$, such that $A \wedge B$ is unsatisfiable, an interpolant for $\langle A, B \rangle$ is a formula I such that A implies I, $I \wedge B$ is unsatisfiable, and I only refers to common symbols of A and B (McMillan 2005). Craig proved that for unsatisfiable first-order formulas, an interpolant always exists (Craig 1957), and many SMT solvers can derive a quantifier-free interpolant from a refutation of a quantifier-free formula of the form $A \wedge B$ (McMillan 2005). A binary interpolant for an infeasible abstract trace can then be computed by defining $A = S_1 \wedge T_1 \wedge S_2 \wedge \ldots S_i$ and $B = T_i \wedge S_{i+1}$. In that case, the only common symbols by A and B are those that describe states after the ith step of the plan. Thus, the interpolation I can be used to split S_i into two abstract states. Note that this is very similar to progression flaws in that it identifies the reasons that make a prefix of the abstract trace infeasible.

However, there are several significant differences between how CEGAR is applied in model-checking and planning. First of all, the abstraction is constructed with a different overall objective. In model-checking, CEGAR is used until the abstraction is refined enough to prove that no error state can be reached from any initial state. In our case, as we analysed in the experiments, it is rare that the process finishes with the abstraction proving the optimal plan correct. Instead, following prior work, we use CEGAR to generate abstraction heuristics to be used in A* search. As we analysed in the experimental results section, this difference is significant and different refinement strategies are superior when the objective is to solve the problem by the CEGAR process (where progression flaws are often superior) as opposed as to generate abstraction heuristics (where regression and sequence flaws are often the best choice).

Second, in planning, all the algorithms to find flaws (including the ones we propose in this paper) run in polynomial time. This is because they leverage the simplest setting: Cartesian abstractions where all abstract states correspond to a Cartesian set of states in the original task, the planning task is expressed in SAS^+, and we find flaws only over a single abstract plan trace. All of this ensures that all operations used (e.g. intersection of Cartesian representations, or computing the image of a Cartesian representation and a SAS^+ action) can be efficiently computed in polynomial time. Interpolation techniques, on the other hand, derive the I_i formulas from an unsatisfiable logic formula, which is more general and harder to compute.

Other types of interpolation have also been considered in model-checking. For example, Hajdu and Micskei (2020) uses backward binary interpolation, where the trace is explored backwards. Thus, this resembles regression flaws, similarly as binary interpolation resembles progression flaws, and the differences outlined above still apply.

Another common technique is sequence interpolation. Given a sequence of formulas $\Gamma = A_1, \ldots, A_n$, $I_0, \ldots, I_n$ is an interpolant for Γ when (i) $I_0 = \top$ and $I_n = \bot$, and (ii) $I_{i-1} \wedge A_i$ implies I_i for all $1 \leq i \leq n$, and (iii) I_i only contains symbols in common in $A_1, \ldots A_i$ and $A_{i+1}, \ldots, A_n$ for all $1 \leq i < n$. In other words, the i-th element of the interpolant is a formula over the common vocabulary of the prefix $A_0 \ldots A_i$ and the suffix $A_{i+1} \ldots A_n$, and each interpolant implies the next one. Sequence interpolants can be computed by setting $A_1 = S_0$ and $A_i = T_i \wedge S_{i+1}$ for $2 \leq i \leq n$. In that case, the interpolants I_i correspond to a refinement over S_i such that altogether make the path from the initial location to the error location infeasible (McMillan 2006; Vizel and Grumberg 2009).

Despite having similar names, sequence interpolation and sequence flaws are very different. The main similarity is that each r^i/I_i is an over-approximation for the set of states reachable from the initial state with the first i steps of the trace. And these over-approximations are then used to refine the abstraction accordingly. In sequence interpolation, the main motivation is to perform multiple refinements at the same time to reduce the number of iterations of the algorithm as well as keeping the formulas simpler (decomposing the reason for the conflict in multiple steps rather than in a single step). With sequence flaws, however, we typically refine a single flaw even when many flaws are found. This is sufficient because we consider abstract traces that correspond to a specific sequence of operators and all abstract states correspond to Cartesian sets. Furthermore, performing multiple refinements in a single iteration is not critical in our case, because the process of finding a new trace and checking for flaws is very efficient so it's worth repeating it in order to find more relevant abstraction refinements. We have also evaluated fixing all found flaws before a new flaw search in Section 7.2.4, but results show that the performance gains do not pay off the reduced relevance of the repaired flaws, especially for regression flaws.

8.2 CEGAR methods in Planning

Our work directly builds on top of the work by Seipp and Helmert (2018), which introduced the CEGAR framework for planning, as a method to construct Cartesian abstractions. They used the notion of progression flaws, which is our baseline in this paper. But there are other extensions that are orthogonal to the improvements that we consider here. For example, Seipp, von Allmen, et al. (2020) consider how to recompute the optimal abstract plan incrementally after each refinement. This greatly speeds up the process, allowing to construct abstractions faster. Their approach is independent of the refinement chosen, so we use the same enhancements in our experiments. Other existing enhancements to the framework include refining the abstraction in an online fashion during the A^* search (Eifler and Fickert 2018; Eifler, Fickert, et al. 2019), computing abstract transitions lazily on demand to reduce the memory overhead (Seipp 2024), and using mutex invariants to disambiguate abstract states and transitions (Pozo, Torralba, et al. 2026). We did not consider these enhancements in the experiments as they were introduced only very recently, but they are independent of the strategy for finding flaws.

The closest related work is the refinement strategies by Speck and Seipp (2022). Like us, they also aim at improving the process of refining Cartesian abstractions in CEGAR by finding multiple flaws in each iteration. However, in their case, they still find a single flaw per abstract plan trace, but consider multiple abstract plans

instead of only one. They do this by searching flaws in progression but expanding paths from the initial state that correspond to a valid prefix of some optimal abstract plan. This may be positive because the CEGAR loop can finish as soon as a valid plan is found. However, this is rarely the case and the search process can sometimes become very expensive. Yet, they show that this can still pay off.

After finding a set of flaws, they can choose one refinement with some flaw selection strategy. The best strategy for them was to choose the split that happens the most among all paths. However, such a strategy is limited in our case, as we consider a single abstract plan, and so we can only count different flaws in the same abstract plan, used for strategies based on the position of the flaw (Section 7.2.1). They also consider a batch refinement that refines the abstraction according to multiple flaws in each iteration of the CEGAR loop instead. To do so, they iteratively refine the flaw in the abstract state with the lowest h value, discarding the flaws in which the h value has been modified after refining the previous flaws because they are not part of an optimal abstract plan any longer. Interestingly, while refining multiple flaws did not pay off in our case (see Section 7.2.4), this is the best configuration for them. A key difference is that, by considering a single plan, our process of finding many sequence flaws is still cheap, so the main advantage of refining a batch (speeding up the process due to having less iteration in the CEGAR loop) is diminished in our case.

Whether the two approaches can be combined, for example by computing flaws of all optimal abstract plan traces using regression and/or sequence flaws, remains an open question. On the one hand, this would allow finding even more candidate flaws to refine the abstraction. On the other hand, the process of searching all optimal abstract paths is already very expensive in the simpler setting of progression flaws (as there may be exponentially-many paths to consider). So, it is unclear whether the overhead of considering partial states in regression, or continuing after the first flaw like in sequence flaws could pay off.

Some research has been focused on extending the applicability of the framework of CEGAR for Cartesian abstractions (among others) to more expressive formalisms than SAS^+. This is challenging because the success of CEGAR for Cartesian abstractions is partly due to the ability to find flaws in low-order polynomial time, and this is based on the fact that the progression and regression of SAS^+ operators over Cartesian representations results in another Cartesian representation that can be easily computed. When considering other planning formalisms, this is no longer guaranteed. However, Büchner et al. (2024) showed that the same CEGAR algorithm can still be performed efficiently in factored planning tasks (Sievers and Helmert 2021; Torralba and Sievers 2019) as well as any formalism where the applicability of operators, as well as progression and regression of an operator can be represented as Cartesian representations (Büchner et al. 2024). Another very common formalism is that of SAS^+ with conditional effects (Pednault 1989). It was recently shown that, even though the computation of the progression of an operator with conditional effects over a Cartesian representation may be really computationally expensive, one can over-approximate this resulting in non-induced abstractions (Pozo and Seipp 2025). This work also applied regression and sequence flaws for planning tasks with conditional effects. Interestingly, sequence flaws showed to be very useful in that setting due to regression being non-Cartesian with conditional effects, and therefore being over-approximated to make it Cartesian. Finally, Klösner et al. (2023) applied the framework to probabilistic planning in the form of stochastic shortest paths (Bertsekas and Tsitsiklis 1991). There the process of finding flaws also involves a search on the space of possible outcomes of the optimal abstract policy. Thus, it remains an open question whether regression and sequence flaws can be transferred to that setting.

CEGAR has also been applied to construct other types of abstractions in planning, such as pattern databases (Rovner et al. 2019) and domain abstractions (Kreft et al. 2023). In both cases, they find flaws by executing the abstract plan forward, similarly to progression flaws. However, they also consider some extensions in the form of wildcard plans and blacklisted variables. The idea behind blacklist variables is to ignore such variables while finding flaws, so that operator's preconditions on those variables are ignored while executing the plan to find the first progression flaw. This is similar to the notion of our sequence flaws in that they continue the execution of the abstract plan after the first flaw (if it occurs on a blacklisted variable). However, the motivation is different: the

main objective is to exclude variables whose refinement would blow-up the abstraction size over the limit (since refinements in PDBs and domain abstractions can multiply the size of the abstraction, unlike in the Cartesian abstractions we considered) and diversify abstractions when creating multiple ones.

8.3 Other Related Techniques in Planning

Cartesian abstractions are primarily constructed by using CEGAR. However, there are also alternative algorithms to automatically construct other types of abstractions.

For Pattern Databases, there are multiple approaches that search the space of possible abstractions, by using different types of search strategies, such as evolutionary algorithms (Edelkamp 2006) and hill climbing (Haslum, Botea, et al. 2007). Compared to CEGAR, these approaches lack guidance in terms of trying to forbid specific abstract plans. But, in exchange, they are guided by a performance measure, e.g., how much the average heuristic value has increased, or how well they perform in an ensemble of abstractions (Franco et al. 2017; Seipp 2019). This type of approach has not been tried yet for Cartesian abstractions, as the space of possible abstractions is huge.

The merge-and-shrink framework constructs abstractions by applying a sequence of transformations (Helmert, Haslum, Hoffmann, and Nissim 2014; Sievers and Helmert 2021). They start from atomic projections for each variable, where all variables except one are abstracted, and then abstractions are shrunk and merged in an iterative process. In each iteration, two abstractions are chosen from the pool of abstractions, and then they are shrunk until the product of the number of their states is lower than the bound. Then, they are merged by the synchronised product. In this step a shrink can be applied on the merged abstraction to reduce its size without altering their informativeness. Then, the merged abstraction replaces them in the pool. Also, label reduction and pruning can be applied at the end of each iteration to reduce the size of the abstraction without losing information. Different strategies exist for merge, shrink, label reduction and pruning (Sievers and Helmert 2021).

Another technique that considers spurious paths in abstractions is SPECO (Fan and Holte 2015), which removes spurious paths in a multi-level abstraction framework (Holte, Grajkowski, et al. 2005; Holte, Perez, et al. 1996). However, they do not refine the abstraction iteratively, but instead avoid such spurious paths during the computation of abstract distances to achieve more informed heuristics. A similar idea is to use mutex information. This has been considered in multiple contexts, including pattern databases (Haslum, Bonet, et al. 2005), symbolic perimeter abstractions (Torralba and Alcázar 2013; Torralba, Alcázar, et al. 2017; Torralba, Linares López, et al. 2018), and recently in Cartesian abstractions as well (Pozo, Torralba, et al. 2026).

Furthermore, instead of building a single abstraction, different heuristics can be combined to achieve a better estimation of the distance to the closest goal during search. However, this combination must preserve the admissibility necessary to guarantee optimal plans. This can always be ensured by taking the maximum of several admissible heuristics (Holte, Felner, et al. 2006). But much stronger heuristics can be obtained by considering conditions under which the sum of multiple heuristics is still admissible (Felner et al. 2004; Korf and Felner 2002; Yang et al. 2008). A general and powerful way to do so is to consider cost-partitioning (Katz and Domshlak 2008, 2010), which distributes the cost of each operator among the combined heuristics, in such a way that the sum of the cost of the operator for all heuristics is always lower or equal than the actual cost, thus preserving admissibility. There are different techniques to automatically compute a cost-partitioning for a given ensemble of abstractions. It is well-known that the optimal cost-partitioning can be computed in polynomial time by encoding this as a linear programming task (Katz and Domshlak 2008, 2010), but the cost of computing this is often too high in practice. Therefore, there are multiple approximations such as post-hoc optimization (Pommerening, Röger, and Helmert 2013), zero-one cost partitioning (Edelkamp 2006; Haslum, Bonet, et al. 2005), and saturated cost partitioning (Seipp and Helmert 2014; Seipp, Keller, et al. 2017, 2020). Saturated cost partitioning distributes costs greedily to each heuristic in a fixed order assigning to each heuristic the cost that is actually used. Online

Saturated Cost Partitioning improves it by computing a good order of heuristics during search, resulting in a non-consistent but admissible heuristic (Seipp 2021, 2023).

Another classical planning technique based on abstraction refinement is Property Directed Reachability (PDR) (Suda 2014), also known as IC3, and originally introduced for model-checking (Bradley 2011; Eén et al. 2011). The strength of this method is proving reachability without unrolling, that is, by calling a SAT solver for each small step instead of the full task, resulting in much faster calls. Suda (2014) represents planning tasks as SAT formulas by using classical compilations (Kautz, McAllester, et al. 1996; Kautz and Selman 1992, 1996), and then uses the PDR algorithm to solve the task. This algorithm iteratively attempts to get a plan in k steps in an over-approximation of the planning task into k layers, such that $L_0 = G$, and each layer L_{i+i} over-approximates the preimage of L_i for all $i \geq 0$. Each layer represents a formula that must be satisfied to solve the task at each step. In each iteration of the loop, the algorithm tries to build a path from k to 0, which is a plan for the task if succeeded. The algorithm also includes some optimizations and the final planner built by Suda (2014) is adapted to planning tasks to avoid the expensive SAT calls. Compared to our work, both are based on refining an abstraction. But they consider a different representation (a sequence of set of clauses instead of a Cartesian abstraction), do not consider action cost, and attempt to solve the planning task directly instead of building heuristics.

The abstraction heuristics we consider in this paper, fit well into theoretical frameworks that analyse the properties of abstractions of planning tasks (Bäckström and Jonsson 2022; Sievers and Helmert 2021). The framework by Bäckström and Jonsson (2022) categorise state abstractions according to their properties. All the Cartesian abstractions that we use in this work are enclosed in strong homomorphisms created by state refinement. So, our work can be understood as an automatic method to find this type of abstraction heuristics.

Partial-Order Causal-Link (POCL) (Bercher, Geier, and Biundo 2013; Penberthy and Weld 1992; Younes and Simmons 2003) is another planning technique that refines partial plans by addressing their flaws. However, despite the similar terminology, flaws in POCL are very different to the flaws we consider in the context of Cartesian abstractions. POCL flaws directly refer to the structure of SAS^+ operators, e.g., they correspond to an open precondition not protected by a causal link or an operator that can delete a precondition before it is needed. Similarities are that multiple flaws exist at each step of the refinement loop and that the flaw selection is critical to reduce the steps required to get a correct plan (Bercher 2021; Bercher, Geier, and Biundo 2013; Bercher, Geier, Richter, et al. 2013). But the way that flaws are dealt with is completely different: we perform refinements that eventually result in an abstraction with a plan without flaws, whereas flaws in POCL require modifying the partial plan and backtracking is usually necessary.

9 Conclusions

CEGAR is a method to iteratively refine abstractions by identifying flaws in optimal abstract plans, and it is the only method to build Cartesian abstractions with any granularity for Planning so far. Cartesian abstractions built by CEGAR have shown an outstanding performance for optimal classical planning, and they are an essential part of the state-of-the-art Scorpion planner. However, previous work finds a single flaw per plan in progression, the first one along its execution, and no other work has researched the concept of flaw and other manners to detect flaws in an abstract plan. We generalise the concept of flaw and present two new definitions that allow searching flaws in regression from the goals (regression flaws) and searching flaws after the first one in any direction (sequence flaws).

Regression flaws increase the heuristic value of a larger number of states, usually obtaining a higher average heuristic value and also fewer states with a very low heuristic value. Thus, abstractions built with regression flaws generally induce more useful heuristics for searching optimal plans with A^* search in progression than progression flaws. Nevertheless, progression flaws are more focused on getting plans for the task, so abstractions

built with progression flaws obtain higher estimations of the g value, and they also find plans during the CEGAR process quicker.

Sequence flaws are found in a relaxed Cartesian space by relaxing the execution of operators in two ways: applying operators although they are not applicable and undeviating the relaxed states that do not intersect with the successor abstract state by assigning the value of that abstract state to the non-intersecting variables of the relaxed state. Sequence flaws typically identify many flaws in an abstract plan trace, and so some criterion to choose a flaw is required. We have designed many flaw selection strategies to choose a flaw, although all flaws can also be refined before a new search of additional flaws, with results that are not as good as the ones obtained by the best flaw selection strategies.

We have experimentally shown that regression flaws are better than progression flaws in most of the tasks for a single abstraction, additive abstractions and the Scorpion planner. Bidirectional strategies are a tradeoff between progression and regression flaws with a little worse performance than regression flaws overall but a better behaviour on domains where regression flaws are not so useful. Also, different flaw selection strategies result in very different behaviour, and A* search with the abstraction generated by a good strategy solves more tasks than with the abstraction generated by the first flaw even in regression, with very complementary behaviour among the best strategies. Simulating flaws to choose one that increases the plan cost the most is very powerful but also very computationally expensive, so it only pays off for small additive abstractions. The maximum potential and the highest h^{add} value are the best strategies for a single abstraction. However, the behaviour is very different for additive abstractions, for which the best results are achieved by the flaw that increases the most the cost of the optimal abstract plan after the refinement. For Scorpion this strategy does not work bad, but some less computationally expensive strategies work even better and the most refined split gets the strongest heuristics, resulting in a significant increase of the number of tasks solved with respect to the state-of-the-art Scorpion planner in the Autoscale benchmark.

Future work will apply these new types of flaw to all optimal abstract plans of the abstraction, instead of only to a random optimal abstract plan. This is very computationally expensive, but it has the potential to better characterise the most important flaws, and a number of flaws can be refined for each flaw search. Also, on-demand computation of transitions can be applied to reduce the memory usage of abstractions (Seipp 2024). Another line of research is using different flaw search strategies to build more complementary additive abstractions, in the aim to get higher heuristic values for most of the states. This line of research can also be combined with the latest improvements in cost partitioning for Cartesian abstractions (Salerno et al. 2025) and merge-and-shrink (Sievers, Pommerening, et al. 2020). Finally, another interesting line of research is applying the insights of this work to CEGAR for domain abstractions (Kreft et al. 2023) and PDBs (Rovner et al. 2019).

References

A. Albarghouthi. 2015. "Software Verification with Program-Graph Interpolation and Abstraction." Ph.D. Dissertation. University of Toronto, Canada. http://hdl.handle.net/1807/69199.

V. Alcázar, D. Borrajo, S. Fernández, and R. Fuentetaja. 2013. "Revisiting Regression in Planning." In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. Ed. by F. Rossi. AAAI Press, 2254–2260.

V. Alcázar and Á. Torralba. 2015. "A Reminder about the Importance of Computing and Exploiting Invariants in Planning." In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by R. Brafman, C. Domshlak, P. Haslum, and S. Zilberstein. AAAI Press, 2–6.

C. Bäckström and P. Jonsson. 2022. "A framework for analysing state-abstraction methods." *AIJ*, 302, 103608.

C. Bäckström and B. Nebel. 1995. "Complexity Results for SAS$^+$ Planning." *Computational Intelligence*, 11, 4, 625–655.

T. Ball, A. Podelski, and S. K. Rajamani. 2001. "Boolean and Cartesian Abstraction for Model Checking C Programs." In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)* (LNCS). Ed. by T. Margaria and W. Yi. Vol. 2031. Springer-Verlag, 268–283.

P. Bercher. 2021. "A Closer Look at Causal Links: Complexity Results for Delete-Relaxation in Partial Order Causal Link (POCL) Planning." In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. Ed. by R. P. Goldman, S. Biundo, and M. Katz. AAAI Press, 36–45.

P. Bercher, T. Geier, and S. Biundo. 2013. "Using State-Based Planning Heuristics for Partial-Order Causal-Link Planning." In: *KI 2013: Advances in Artificial Intelligence - 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings* (Lecture Notes in Computer Science). Ed. by I. J. Timm and M. Thimm. Vol. 8077. Springer, 1–12. doi:10.1007/978-3-642-40942-4_1.

P. Bercher, T. Geier, F. Richter, and S. Biundo. 2013. "On Delete Relaxation in Partial-Order Causal-Link Planning." In: *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*. IEEE Computer Society, 674–681. doi:10.1109/ICTAI.2013.105.

S. Bernardini and C. Muise, (Eds.) . 2024. *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. AAAI Press.

D. P. Bertsekas and J. N. Tsitsiklis. 1991. "An Analysis of Stochastic Shortest Path Problems." *Mathematics of Operations Research*, 16, 580–595.

D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)* (LNCS). Ed. by A. Gupta and S. Malik. Vol. 5123. Springer-Verlag, 504–518.

D. Beyer and S. Löwe. 2013. "Explicit-State Software Model Checking Based on CEGAR and Interpolation." In: *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (Lecture Notes in Computer Science). Ed. by V. Cortellessa and D. Varró. Vol. 7793. Springer, 146–162.

B. Bonet and H. Geffner. 2001. "Planning as Heuristic Search." *AIJ*, 129, 1, 5–33.

A. R. Bradley. 2011. "SAT-Based Model Checking without Unrolling." In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*. Ed. by R. Jhala and D. Schmidt, 70–87.

R. Brafman, C. Domshlak, P. Haslum, and S. Zilberstein, (Eds.) . 2015. *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. AAAI Press.

C. Büchner, P. Ferber, J. Seipp, and M. Helmert. 2024. "Abstraction Heuristics for Factored Tasks." In: *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. Ed. by S. Bernardini and C. Muise. AAAI Press, 40–49.

S. Chien, A. Fern, W. Ruml, and M. Do, (Eds.) . 2014. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. AAAI Press.

E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. "Counterexample-Guided Abstraction Refinement." In: *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*. Ed. by E. A. Emerson and A. P. Sistla, 154–169.

E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking." *JACM*, 50, 5, 752–794.

E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. The MIT Press.

E. M. Clarke, A. Gupta, and O. Strichman. 2004. "SAT-based counterexample-guided abstraction refinement." *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 23, 7, 1113–1123. doi:10.1109/TCAD.2004.829807.

W. Craig. 1957. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory." *J. Symb. Log.*, 22, 3, 269–285. doi:10.2307/2963594.

J. C. Culberson and J. Schaeffer. 1998. "Pattern Databases." *Computational Intelligence*, 14, 3, 318–334.

S. Edelkamp. 2006. "Automated Creation of Pattern Database Search Heuristics." In: *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*. Ed. by S. Edelkamp and A. Lomuscio, 35–50.

S. Edelkamp. 2001. "Planning with Pattern Databases." In: *Proceedings of the Sixth European Conference on Planning (ECP 2001)*. Ed. by A. Cesta and D. Borrajo. AAAI Press, 84–90.

S. Edelkamp and J. Hoffmann. 2004. *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition*. Tech. rep. 195. University of Freiburg, Department of Computer Science.

N. Eén, A. Mishchenko, and R. Brayton. 2011. "Efficient implementation of property directed reachability." In: *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011)*. Ed. by P. Bjesse and A. Slobodova, 125–134.

R. Eifler and M. Fickert. 2018. "Online Refinement of Cartesian Abstraction Heuristics." In: *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*. Ed. by V. Bulitko and S. Storandt. AAAI Press, 46–54.

R. Eifler, M. Fickert, J. Hoffmann, and W. Ruml. 2019. "Refining Abstraction Heuristics during Real-Time Planning." In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*. AAAI Press, 7578–7585.

G. Fan and R. C. Holte. 2015. "The Spurious Path Problem in Abstraction." In: *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*. Ed. by L. Lelis and R. Stern. AAAI Press, 18–27.

A. Felner, R. Korf, and S. Hanan. 2004. "Additive Pattern Database Heuristics." *JAIR*, 22, 279–318.

R. E. Fikes and N. J. Nilsson. 1971. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *AIJ*, 2, 189–208.

M. Fox and D. Long. 2003. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains." *JAIR*, 20, 61–124.

S. Franco, Á. Torralba, L. H. S. Lelis, and M. Barley. 2017. "On Creating Complementary Pattern Databases." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*. Ed. by C. Sierra. IJCAI, 4302–4309.

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

R. P. Goldman, S. Biundo, and M. Katz, (Eds.) . 2021. *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. AAAI Press.

S. Graf and H. Saïdi. 1997. "Construction of Abstract State Graphs with PVS." In: *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*. Ed. by C. Courcoubetis, 72–83.

Á. Hajdu and Z. Micskei. 2020. "Efficient Strategies for CEGAR-Based Model Checking." *Journal of Automated Reasoning*, 64, 6, 1051–1091.

Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik. 2016. "A Configurable CEGAR Framework with Interpolation-Based Refinements." In: *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings* (Lecture Notes in Computer Science). Ed. by E. Albert and I. Lanese. Vol. 9688. Springer, 158–174. doi:10.1007/978-3-319-39570-8_11.

P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, 4, 2, 100–107.

P. Haslum, B. Bonet, and H. Geffner. 2005. "New Admissible Heuristics for Domain-Independent Planning." In: *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*. AAAI Press, 1163–1168.

P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. 2007. "Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning." In: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, 1007–1012.

M. Helmert. 2004. "A Planning Heuristic Based on Causal Graph Analysis." In: *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*. Ed. by S. Zilberstein, J. Koehler, and S. Koenig. AAAI Press, 161–170.

M. Helmert. 2009. "Concise Finite-Domain Representations for PDDL Planning Tasks." *AIJ*, 173, 503–535.

M. Helmert. 2006. "The Fast Downward Planning System." *JAIR*, 26, 191–246.

M. Helmert, P. Haslum, and J. Hoffmann. 2008. "Explicit-State Abstraction: A New Method for Generating Heuristic Functions." In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*. AAAI Press, 1547–1550.

M. Helmert, P. Haslum, and J. Hoffmann. 2007. "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Ed. by M. Boddy, M. Fox, and S. Thiébaux. AAAI Press, 176–183.

M. Helmert, P. Haslum, J. Hoffmann, and R. Nissim. 2014. "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces." *JACM*, 61, 3, 16:1–63.

I. T. Hernádvölgyi and R. C. Holte. 2000. "Experiments with Automatically Created Memory-Based Heuristics." In: *Proceedings of the 4th International Symposium on Abstraction, Reformulation and Approximation (SARA 2000)* (LNAI). Ed. by B. Y. Choueiry and T. Walsh. Vol. 1864. Springer-Verlag, 281–290.

J. Hoffmann, J. Porteous, and L. Sebastia. 2004. "Ordered Landmarks in Planning." *JAIR*, 22, 215–278.

R. C. Holte, A. Felner, J. Newton, R. Meshulam, and D. Furcy. 2006. "Maximizing over Multiple Pattern Databases Speeds up Heuristic Search." *AIJ*, 170, 16–17, 1123–1136.

R. C. Holte, J. Grajkowski, and B. Tanner. 2005. "Hierarchical Heuristic Search Revisited." In: *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)* (LNAI). Ed. by J.-D. Zucker and L. Saitta. Vol. 3607. Springer-Verlag, 121–133.

R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. 1996. "Hierarchical A*: Searching Abstraction Hierarchies Efficiently." In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*. AAAI Press, 530–535.

M. Katz and C. Domshlak. 2008. "Optimal Additive Composition of Abstraction-based Admissible Heuristics." In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*. Ed. by J. Rintanen, B. Nebel, J. C. Beck, and E. Hansen. AAAI Press, 174–181.

M. Katz and C. Domshlak. 2010. "Optimal admissible composition of abstraction heuristics." *AIJ*, 174, 12–13, 767–798.

H. Kautz, D. McAllester, and B. Selman. 1996. "Encoding Plans in Propositional Logic." In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996)*. Ed. by L. C. Aiello, J. Doyle, and S. C. Shapiro. Morgan Kaufmann, 374–384.

H. Kautz and B. Selman. 1992. "Planning as Satisfiability." In: *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*. Ed. by B. Neumann. John Wiley and Sons, 359–363.

H. Kautz and B. Selman. 1996. "Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search." In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*. AAAI Press, 1194–1201.

E. Keyder, S. Richter, and M. Helmert. 2010. "Sound and Complete Landmarks for And/Or Graphs." In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*. Ed. by H. Coelho, R. Studer, and M. Wooldridge. IOS Press, 335–340.

T. Klösner, J. Seipp, and M. Steinmetz. 2023. "Cartesian Abstractions and Saturated Cost Partitioning in Probabilistic Planning." In: *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*. Ed. by K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein, and R. Rădulescu. IOS Press, 1272–1279.

R. E. Korf and A. Felner. 2002. "Disjoint Pattern Database Heuristics." *AIJ*, 134, 1–2, 9–22.

S. Kraus, (Ed.) . 2019. *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. IJCAI.

R. Kreft, C. Büchner, S. Sievers, and M. Helmert. 2023. "Computing Domain Abstractions for Optimal Classical Planning with Counterexample-Guided Abstraction Refinement." In: *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*. Ed. by S. Koenig, R. Stern, and M. Vallati. AAAI Press, 221–226.

M. Leucker, G. Markin, and M. R. Neuhäuser. 2015. "A New Refinement Strategy for CEGAR-Based Industrial Model Checking." In: *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings* (Lecture Notes in Computer Science). Ed. by N. Piterman. Vol. 9434, 155–170.

S. Löwe. 2017. "Effective Approaches to Abstraction Refinement for Automatic Software Verification." Ph.D. Dissertation. University of Passau, Germany. https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/481.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. 1998. *PDDL – The Planning Domain Definition Language – Version 1.2*. Tech. rep. CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control, Yale University.

K. L. McMillan. 2005. "Applications of Craig Interpolants in Model Checking." In: *Lecture notes in computer science*. Lecture Notes in Computer Science. Vol. 3440. Ed. by N. Halbwachs and L. D. Zuck. Springer, 1–12. doi:10.1007/978-3-540-31980-1_1.

K. L. McMillan. 2006. "Lazy Abstraction with Interpolants." In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by T. Ball and R. B. Jones. Vol. 4144. Springer, 123–136. doi:10.1007/11817963_14.

E. P. D. Pednault. 1989. "ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus." In: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR 1989)*. Ed. by R. J. Brachman, H. J. Levesque, and R. Reiter. Morgan Kaufmann, 324–332.

J. S. Penberthy and D. S. Weld. 1992. "UCPOP: A Sound, Complete, Partial Order Planner for ADL." In: *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992)*. Ed. by B. Nebel, C. Rich, and W. Swartout. Morgan Kaufmann, 103–114.

F. Pommerening, G. Röger, and M. Helmert. 2013. "Getting the Most Out of Pattern Databases for Classical Planning." In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. Ed. by F. Rossi. AAAI Press, 2357–2364.

F. Pommerening, G. Röger, M. Helmert, and B. Bonet. 2014. "LP-based Heuristics for Cost-optimal Planning." In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. Ed. by S. Chien, A. Fern, W. Ruml, and M. Do. AAAI Press, 226–234.

M. Pozo and J. Seipp. 2025. "Abstraction Heuristics for Classical Planning Tasks with Conditional Effects." In: *Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI 2025)*. Ed. by J. Kwok. IJCAI, 8608–8616.

M. Pozo, Á. Torralba, and C. Linares López. 2025. *Code, Experimental Results and Scripts for the paper New Flaw Search Strategies in CEGAR for Optimal Classical Planning*. 10.5281/zenodo.14905468. (2025). doi:{10.5281/zenodo.14905468}.

M. Pozo, Á. Torralba, and C. Linares López. 2024a. "Gotta Catch 'Em All! Sequence Flaws in CEGAR for Classical Planning." In: *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI 2024)*. Ed. by U. Endriss, F. S. Melo, K. Bach, A. J. B. Diz, J. M. Alonso-Moral, S. Barro, and F. Heintz. IOS Press, 4287–4294. doi:{10.3233/FAIA241003}.

M. Pozo, Á. Torralba, and C. Linares López. 2026. "Not Everything is Permitted: Constrained Cartesian Abstractions for Optimal Classical Planning." In: *Proceedings of the Forty AAAI Conference on Artificial Intelligence (AAAI 2026)*. AAAI Press.

M. Pozo, Á. Torralba, and C. Linares López. 2024b. "When CEGAR Meets Regression: A Love Story in Optimal Classical Planning." In: *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*. AAAI Press, 20238–20246.

1996. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*. AAAI Press.

J. Rintanen. 2008. "Regression for Classical and Nondeterministic Planning." In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*. Ed. by M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. Avouris. IOS Press, 568–572.

G. Röger, M. Helmert, J. Seipp, and S. Sievers. 2020. "An Atom-Centric Perspective on Stubborn Sets." In: *Proceedings of the 13th Annual Symposium on Combinatorial Search (SoCS 2020)*. Ed. by D. Harabor and M. Vallati. AAAI Press, 57–65.

F. Rossi, (Ed.) . 2013. *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*. AAAI Press.

A. Rovner, S. Sievers, and M. Helmert. 2019. "Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning." In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. Ed. by N. Lipovetzky, E. Onaindia, and D. E. Smith. AAAI Press, 362–367.

M. Salerno, R. Fuentetaja, D. Speck, and J. Seipp. 2025. "Merging Cartesian Abstractions for Classical Planning." In: *Proceedings of the 28th European Conference on Artificial Intelligence (ECAI 2025)*. Ed. by I. Lynce and A. Murano. IOS Press, 4766–4773.

J. Seipp. 2024. "Efficiently Computing Transitions in Cartesian Abstractions." In: *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. Ed. by S. Bernardini and C. Muise. AAAI Press, 509–513.

J. Seipp. 2018. "Fast Downward Scorpion." In: *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 77–79.

J. Seipp. 2021. "Online Saturated Cost Partitioning for Classical Planning." In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. Ed. by R. P. Goldman, S. Biundo, and M. Katz. AAAI Press, 317–321.

J. Seipp. 2019. "Pattern Selection for Optimal Classical Planning with Saturated Cost Partitioning." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. Ed. by S. Kraus. IJCAI, 5621–5627.
J. Seipp. 2023. "Scorpion 2023." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts*, 1–2. https://ipc2023-classical.github.io/abstracts/planner25_scorpion_2023.pdf.
J. Seipp and M. Helmert. 2013. "Counterexample-guided Cartesian Abstraction Refinement." In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*. Ed. by D. Borrajo, S. Kambhampati, A. Oddi, and S. Fratini. AAAI Press, 347–351.
J. Seipp and M. Helmert. 2018. "Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning." *JAIR*, 62, 535–577.
J. Seipp and M. Helmert. 2014. "Diverse and Additive Cartesian Abstraction Heuristics." In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. Ed. by S. Chien, A. Fern, W. Ruml, and M. Do. AAAI Press, 289–297.
J. Seipp, T. Keller, and M. Helmert. 2017. "A Comparison of Cost Partitioning Algorithms for Optimal Classical Planning." In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. Ed. by L. Barbulescu, J. Frank, Mausam, and S. F. Smith. AAAI Press, 259–268.
J. Seipp, T. Keller, and M. Helmert. 2020. "Saturated Cost Partitioning for Optimal Classical Planning." *JAIR*, 67, 129–167.
J. Seipp, F. Pommerening, and M. Helmert. 2015. "New Optimization Functions for Potential Heuristics." In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*. Ed. by R. Brafman, C. Domshlak, P. Haslum, and S. Zilberstein. AAAI Press, 193–201.
J. Seipp, F. Pommerening, S. Sievers, and M. Helmert. 2017. *Downward Lab*. https://doi.org/10.5281/zenodo.790461. (2017).
J. Seipp, S. von Allmen, and M. Helmert. 2020. "Incremental Search for Counterexample-Guided Cartesian Abstraction Refinement." In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*. Ed. by J. C. Beck, E. Karpas, and S. Sohrabi. AAAI Press, 244–248.
S. Sievers and M. Helmert. 2021. "Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems." *JAIR*, 71, 781–883.
S. Sievers, F. Pommerening, T. Keller, and M. Helmert. 2020. "Cost-Partitioned Merge-and-Shrink Heuristics for Optimal Classical Planning." In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020)*. IJCAI, 4152–4160.
D. Speck and J. Seipp. 2022. "New Refinement Strategies for Cartesian Abstractions." In: *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*. Ed. by S. Thiébaux and W. Yeoh. AAAI Press, 348–352.
M. Suda. 2014. "Property Directed Reachability for Automated Planning." *JAIR*, 50, 265–319.
Á. Torralba and V. Alcázar. 2013. "Constrained Symbolic Search: On Mutexes, BDD Minimization and More." In: *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS 2013)*. Ed. by M. Helmert and G. Röger. AAAI Press, 175–183.
Á. Torralba, V. Alcázar, P. Kissmann, and S. Edelkamp. 2017. "Efficient Symbolic Search for Cost-optimal Planning." *AIJ*, 242, 52–79.
Á. Torralba, C. Linares López, and D. Borrajo. 2018. "Symbolic perimeter abstraction heuristics for cost-optimal planning." *AIJ*, 259, 1–31.
Á. Torralba, J. Seipp, and S. Sievers. 2021. "Automatic Instance Generation for Classical Planning." In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. Ed. by R. P. Goldman, S. Biundo, and M. Katz. AAAI Press, 376–384.
Á. Torralba and S. Sievers. 2019. "Merge-and-Shrink Task Reformulation for Classical Planning." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*. Ed. by S. Kraus. IJCAI, 5644–5652.
A. Valmari and H. Hansen. 2010. "Can Stubborn Sets Be Optimal?" In: *Proceedings of the 31st International Conference on Applications and Theory of Petri Nets (Petri Nets 2010)* (LNCS). Ed. by J. Lilius and W. Penczek. Vol. 6128. Springer-Verlag, 43–62.
Y. Vizel and O. Grumberg. 2009. "Interpolation-sequence based model checking." In: *2009 Formal Methods in Computer-Aided Design*, 1–8. doi:10.1109/FMCAD.2009.5351148.
F. Yang, J. Culberson, R. C. Holte, U. Zahavi, and A. Felner. 2008. "A General Theory of Additive State Space Abstractions." *JAIR*, 32, 631–662.
H. L. S. Younes and R. G. Simmons. 2003. "VHPOP: Versatile Heuristic Partial Order Planner." *JAIR*, 20, 405–430.

Received 30 December 2025; accepted 23 July 2026