ECAI 2024
U. Endriss et al. (Eds.)

© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA241003

Gotta Catch 'Em All! Sequence Flaws in CEGAR for Classical Planning

Martín Pozo[a], Álvaro Torralba[b] and Carlos Linares López[a]

[a]Computer Science and Engineering Department – Universidad Carlos III de Madrid, Madrid, Spain
[b]Aalborg University, Aalborg, Denmark

Abstract. 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 iteratively refine the abstraction, by finding a flaw in the current optimal abstract plan. Previous works find only a single flaw, by executing the abstract plan in the concrete state space and stopping when such execution cannot be continued.

We show, however, that many flaws can be identified on a single plan. To that end, we introduce sequence flaws, which execute the plan in a Cartesian relaxation of the task to characterize issues beyond the first flaw found along its execution. This greatly increases the flexibility of CEGAR regarding how to refine the abstraction.

Our experiments show that a high number of sequence flaws exist in most abstract plans across existing benchmarks. We observe that the selected flaw has a high impact on the resulting heuristic, opening new research opportunities for better selection strategies.

1 Introduction

Abstractions are commonly employed in optimal planning to generate domain-independent admissible heuristics, as they offer great flexibility to define well-informed heuristics for the planning task at hand [8, 13, 28]. However, such flexibility raises the question of how to efficiently compute the right abstraction. A very promising method is Counterexample-Guided Abstraction Refinement (CEGAR), successfully used for Cartesian abstractions [25], PDBs [22] and domain abstractions [14]. CEGAR starts with a trivial abstraction, where all states in the problem are considered equivalent to each other. Then, it iteratively refines the abstraction trying to improve the heuristic value of the initial state. To do so, it selects an optimal abstract plan and executes it on the original state space. If the plan works, then an optimal plan has been found and the task is solved. If the plan fails, a *flaw* is identified at the point where the plan execution could not continue. Then, the abstraction is refined to distinguish states in which such a step is possible from those where it is not [24].

Recent work introduced regression flaws [19]. Instead of executing the abstract plan forwards, it proceeds backwards from the goals. This flaw is often very different from its forward counterpart. Indeed, using regression flaws greatly improves the quality of the resulting heuristics. This shows the importance of considering new ways of computing flaws, and brings up the question whether there are other flaws that could be identified.

Indeed, CEGAR was originally introduced in the context of program verification [1, 4, 10, 15, 29], where the notion of refinement is based on *sequence interpolation*, used to find flaws in several steps of the sequence. Inspired by this, we consider whether the same is true in the planning setting.

In this paper, we show that multiple flaws can be identified in a single abstract plan, opening multiple alternative ways for refining the abstraction. Consider a problem where a counter c must be increased from 1 to 6 and an abstract plan $\langle inc(c,2,3), inc(c,4,5)\rangle$. Clearly, there are three separate issues with this plan: (1) c "jumps" from 1 to 2 before the first action; (2) c "jumps" from 3 to 4 before the last action; and (3) the last state is not a goal state (5 instead of 6). But current methods will only find flaw (1) forwards and flaw (3) backwards, as they are the first one in each direction. Furthermore, some flaws are inherent to the abstraction and independent of the initial state and the goals. For example, if two consecutive actions $\langle inc(c,1,2), inc(c,3,4)\rangle$ are somewhere in the middle of an abstract plan, a flaw should be detected in any direction.

We introduce *sequence flaws*, a new type of flaw that allows the identification of multiple issues in the same abstract plan. Our experiments show that abstract plans have many different sequence flaws that can be repaired. As a single flaw is refined, strategies to determine the flaw to select are paramount. The results support previous findings, and refining closer to the goal is typically best. But there are also promising results, and we observe that different selection strategies can sometimes lead to better heuristic functions.

2 Background

We consider tasks in SAS$^+$ representation [3], where states are described in terms of a tuple of variables $V = \langle v_0, \ldots, v_n\rangle$, and each $v \in V$ has a finite domain, $\mathcal{D}_v$. A *partial state* p is a partial variable assignment over some variables $\mathrm{vars}(p) \subseteq V$. A (concrete) state s is a full assignment, $\mathrm{vars}(s) = V$. We write $p[v]$ for the value assigned to the variable $v \in \mathrm{vars}(p)$ in the partial state p. Two partial states p and c are consistent if $p[v] = c[v]$ for all $v \in \mathrm{vars}(p) \cap \mathrm{vars}(c)$. We denote by $[p] \subseteq S$ the set of states consistent with p.

A SAS$^+$ task Π is a tuple $\langle V, O, s_0, G\rangle$ where s_0 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 $\mathsf{cost}(o) \in \mathbb{R}_0^+$. An operator o is applicable in progression in a state s if s is consistent with $pre(o)$. The result of applying o to s is a state $s[\![o]\!]$ where $s[\![o]\!][v] = eff(o)[v]$ if $v \in \mathrm{vars}(eff(o))$ and $s[\![o]\!][v] = s[v]$ otherwise. We write $s \xrightarrow{o} s'$ as a shorthand whenever o is applicable on s and $s' = s[\![o]\!]$. The post-conditions of an operator o in progression are defined for $v \in \text{vars}(\mathit{pre}(o)) \cup \text{vars}(\mathit{eff}(o))$ and $\mathit{post}(o)[v] = \mathit{eff}(o)[v]$ for $v \in \text{vars}(\mathit{eff}(o))$ and $\mathit{post}(o)[v] = \mathit{pre}(o)[v]$ otherwise. The *state space* of a task Π is a transition system, $\Theta = \langle S, O, T, s_0, S_G \rangle$, where S is the set of all states, $S_G = \{s \in S \mid s \text{ is consistent with } G\}$ is the set of goal states, and $T = \{(s, o, s') \mid s \in S, o \text{ applicable in } s, s' = s[\![o]\!]\}$ is the set of transitions. A *plan* π for s is a sequence of operators $\langle o_1, o_2, \ldots, o_n \rangle$, s.t. the trace $s \xrightarrow{o_1} \ldots \xrightarrow{o_n} s_n$ reaches a goal state $s_n \in S_G$. The cost of π is the summed up cost of its operators. The goal distance from s to the goal $h^*(s)$ is the minimum cost of any plan for s, or ∞ if no plan exists. A plan for Π is a plan for s_0.

A common approach to find optimal plans is to use A* search with an admissible heuristic [7]. A *heuristic* is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. The heuristic is admissible if $h(s) \leq h^*(s)$ for all $s \in S$.

Regression starts from a partial state p and gets from which states we can reach some $s \in [p]$ via an operator o [2, 21]. An operator o is applicable in regression in p if p is consistent with $\mathit{pre}^r(o) = \mathit{post}(o)$. The successor partial state p' is defined for $(\text{vars}(p) \setminus \text{vars}(\mathit{eff}(o))) \cup \text{vars}(\mathit{pre}(o))$ and $\mathit{regr}(p, o)[v] = \mathit{pre}(o)[v]$ for $v \in \text{vars}(\mathit{pre}(o))$ and $\mathit{regr}(p, o)[v] = p[v]$ otherwise. We write $p \overset{o}{\leftarrow} p'$ as a shorthand.

An abstraction α for a transition system $\Theta = \langle S, O, T, s_0, S_G \rangle$ is a function $\alpha : S \mapsto S^\alpha$, where S^α is a finite set of abstract states. The abstract state space $\Theta^\alpha = \langle S^\alpha, O, T^\alpha, s_0^\alpha, S_G^\alpha \rangle$ is a homomorphism of the state space, i.e., $T^\alpha = \{(\alpha(s) \xrightarrow{o} \alpha(t) \mid s \xrightarrow{o} t \in T)\}$, $s_0^\alpha = \alpha(s_0)$, $S_G^\alpha = \{\alpha(s) \mid s \in S_G\}$. Each abstraction induces a heuristic function where $h^\alpha(s)$ is the distance from $\alpha(s)$ to the goal in Θ^α. Each abstract state $s^\alpha \in S^\alpha$ is identified with the set of states mapped to it, $[s^\alpha] = \{s \mid s \in S, \alpha(s) = s^\alpha\}$.

Cartesian abstractions are a type of abstractions where the set of states $[s^\alpha]$ is Cartesian $\forall s^\alpha \in S^\alpha$ [25]. A set of states is Cartesian if it is of the form $A_1 \times A_2 \times \cdots \times A_n$, where $A_i \subseteq \mathcal{D}_{v_i} \forall v_i \in V$. Given a Cartesian set a, we denote by $a[v_i]$ the set of values that v_i can take in a, i.e., $a[v_i] = A_i \subseteq \mathcal{D}_{v_i}$. The intersection of two Cartesian sets is a Cartesian set, where $a'[v] = a_1[v] \cap a_2[v] \; \forall v \in V$. Figure 1 shows a Cartesian plan, where for example, in a_1, $v_1 = \{1\}$ and $v_2 = \{2, 3\}$. Also, for any (partial) state p, we can build a Cartesian set $C(p)$ such that $[C(p)] = [p]$, by making $C(p)[v] = \{C(p)[v]\}$ if $v \in \text{vars}(p)$ and $C(p)[v] = \mathcal{D}_v$ otherwise. We will use this conversion of (partial) states into Cartesian sets implicitly, so with a slight abuse of notation we define operations such as the intersection of a partial state p and a Cartesian set a as the Cartesian set $p \cap a := C(p) \cap a$.

The most successful technique to obtain Cartesian abstractions is CEGAR [24, 25]. It starts with the trivial abstraction, which consists of a single abstract state a s.t. $a[v] = \mathcal{D}_v \; \forall v \in \text{vars}(v)$. Then, it is iteratively refined until reaching a termination condition or finding a concrete plan. The refinement loop finds an optimal abstract plan trace $\tau^\alpha = a_0 \xrightarrow{o_1} \ldots \xrightarrow{o_n} a_n$, and it is executed in the concrete space, resulting in a trace $s_0 \xrightarrow{o_1} \ldots \xrightarrow{o_n} s_n$. If this execution succeeds and $s_n \in S_G$, then it is an optimal plan for the task. Furthermore, we say that τ^α is *mappable* if each concrete state is included in the corresponding abstract state, i.e. $s_i \in [a_i] \; \forall i \in [0, n]$.

If the abstract plan trace is not mappable, a flaw is reported and the abstraction is refined by splitting an abstract state of the plan into two, in such a way that the same flaw cannot happen again. A *flaw* is a tuple $\langle s_i, c \rangle$ of a state $s_i \in S$ and a Cartesian set c. We can distinguish a different type of flaw for each reason that can cause the execution of τ^α to fail at step i: (1) s_i is the first state in which o_{i+1} is inapplicable and c is the set of states in a_i in which o_{i+1} is applicable, i.e. $c = a_i \cap \mathit{pre}(o_{i+1})$. (2) s_i is the first state where o_{i+1} is applicable but $s_i[\![o_{i+1}]\!]$ is not mapped to a_{i+1}, and c is the set of states in a_i from which a_{i+1} is reached when applying o_i, i.e., $a_i \cap \mathit{regr}(a_{i+1}, o_i)$. (3) The trace can be executed but $s_n \notin S_G$, resulting in the flaw $\langle s_n, s_n \cap G \rangle$.

A flaw $\langle s, c \rangle$ is repaired by splitting $\alpha(s)$ into two abstract states d and e with $s \in d$ and $c \subseteq e$. Usually, multiple possible splits exist in different variables to fix the flaw. A split selection strategy is a criterion to choose one of them [24, 25]. The process refines the abstraction until solving the problem either by finding an optimal plan or proving the task unsolvable (an abstract plan cannot be found). It can be stopped by some termination condition (usually a time or size limit), resulting in a Cartesian abstraction that induces a heuristic.

Recent work has introduced regression flaws, found by executing the abstract plan in regression from the goals, with better results than progression flaws [19]. They are similar to progression flaws, with the difference of not requiring that the Cartesian state of the partial state is included in the abstract state, but their intersection is not empty, since the former would be too restrictive. So, there are three types of flaws: (1) p_i is the first partial state in regression in which o_i is not backward applicable, and c is the set of states in a_i where o_i is backward applicable. (2) p_i is the first partial state in regression where o_i is backward applicable but the intersection of its successor and a_{i-1} is empty; then, c is the set of states in a_i reached from a_{i-1} when the o_i is applied in progression, i.e., $a_i \cap a_{i-1}[\![o_i]\!]$. (3) The sequence can be executed but $s_0 \notin p_0$, and c is the Cartesian set of s_0. The strength of this technique is maximizing h for states closer to the goals, increasing the average h despite getting lower heuristic values for s_0 and requiring more iterations to find a plan during the refinement loop.

Another concept introduced by this work is splitting strategies: for progression flaws the split value is the one inside c (the Cartesian set in which the flaw does not happen), but for regression flaws splitting the value in the partial state (the value producing the flaw) gets better results. Hence, this work defines two strategies: *wanted* for splitting the value in c and *unwanted* for splitting the value in the state [19]. This corresponds to put the values of the variable not contained in c nor the state into the child state d for *wanted* splits and the child state e for *unwanted* splits.

3 Sequence Flaws

Our main contribution is the definition of sequence flaws. This allows us to find more flaws in the abstract plan.

Consider a planning task in which a worker must get a package out of a building, passing through three rooms separated by doors, all open or closed. A button carried by the worker opens all doors, and all doors are automatically closed when somebody leaves the building. Initially, the worker is in the first room with doors closed, and the goal is to bring the package out of the building leaving all doors open. Let v_1 denote the state of the doors (1 means open, 0 means closed), and v_2 the room where the package is located $(1, 2, 3$ for the rooms and 4 for the street). o_1 opens doors, and o_2, o_3, o_4 move the package, with o_i moving it from $(i-1)$ to i. Formally,

$V = \{v_1, v_2\}$ with $\mathcal{D}_{v_1} = \{0, 1\}$ and $\mathcal{D}_{v_2} = \{1, 2, 3, 4\}$,
$O = \{o_1, o_2, o_3, o_4\}$ with
$\mathit{pre}(o_1) = \{v_1 \mapsto 0\}$, $\mathit{eff}(o_1) = \{v_1 \mapsto 1\}$,
$\mathit{pre}(o_2) = \{v_1 \mapsto 1, v_2 \mapsto 1\}$, $\mathit{eff}(o_2) = \{v_2 \mapsto 2\}$,
$\mathit{pre}(o_3) = \{v_1 \mapsto 1, v_2 \mapsto 2\}$, $\mathit{eff}(o_3) = \{v_2 \mapsto 3\}$,
$\mathit{pre}(o_4) = \{v_1 \mapsto 1, v_2 \mapsto 3\}$, $\mathit{eff}(o_4) = \{v_1 \mapsto 0, v_2 \mapsto 4\}$.
$s_0 = \{v_1 \mapsto 0, v_2 \mapsto 1\}$, $G = \{v_1 \mapsto 1, v_2 \mapsto 4\}$.

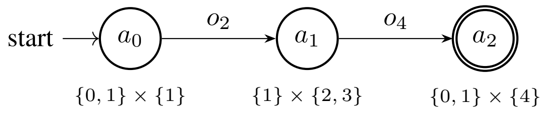

Figure 1: Abstract plan with sequence flaws in v_2 detected neither by forward nor backward first-flaws.

In the abstract plan shown in Figure 1, the first progression flaw is that o_2 is inapplicable in s_0 because $v_1 \mapsto 0$ (the doors are closed) and the first regression flaw is that o_4 cannot be applied in regression because $v_1 \mapsto 1$ in the goals but o_4 closes them. However, this plan also has two additional progression sequence flaws: in v_2 because o_4 is not applicable in the state reached after applying o_2 (mapped to a_1), since the worker is not in the correct room (o_3 must be applied before o_4), and in v_1 in the final state because doors are not open as required in the goal. It also has two additional regression sequence flaws: in v_2 because o_2 is not backward applicable in the state reached after applying o_4 in regression (mapped to a_1), as the worker is not in the correct room, and in v_1 in the initial state because doors are left open but in s_0 they are closed. So, among other problems, stopping at the first flaw completely ignores the existence of problems in the other goal ($v_2 \mapsto 4$).

Sequence flaws can capture issues not detected until repairing all the first-flaws happening before them. In previous work, flaws are found by executing the operators of the abstract plan on the concrete (partial in regression) state space, generating a trace $s_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} s_n$ ($p_n \xleftarrow{o_n} \dots \xleftarrow{o_0} p_0$ in regression). Instead, we consider a relaxation of this approach, that results in a sequence of Cartesian sets.

An operator is applicable in a Cartesian set c if $pre(o) \cap c \neq \varnothing$, and applying o to c results in another Cartesian set $c[\![o]\!]$ where $c[\![o]\!][v] = post(o)[v]$ if $v \in \text{vars}(post(o))$ and $c[\![o]\!][v] = c[v]$ otherwise. In the resulting Cartesian set, the variables of effects and preconditions of o have a single value. This corresponds to all states reachable by applying o from any state in c: $[c[\![o]\!]] = \{s' | s \in [c] \wedge s \xrightarrow{o} s'\}$. An operator o is applicable in regression in c if $pre^r(o) \cap c \neq \varnothing$, and applying o in regression to c produces a Cartesian set $regr(c, o)$ where $regr(c, o)[v] = pre(o)[v]$ for $v \in \text{vars}(pre(o))$, $\{\mathcal{D}_v\}$ for $v \in \text{vars}(eff(o)) \setminus \text{vars}(pre(o))$ and $c[v]$ otherwise. We define $c[\![o]\!]^!$ and $regr^!(c, o)$ as the application of an operator o on a Cartesian set c in progression and regression even when o is inapplicable in c.

3.1 Progression Sequence Flaws

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

Definition 1 (Relaxed Plan Execution). *A relaxed plan execution $r = r_0, r_1, \dots, r_n$, for an abstract plan $\tau^\alpha = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_2} \dots \xrightarrow{o_n} a_n$ is a sequence of Cartesian sets r_i so that:*

(A) $s_0 \in [r_0]$,
(B) if o_{i+1} is applicable on r_i, then $r_i[\![o_{i+1}]\!] \cap a_{i+1} \subseteq r_{i+1}$,
(C) if o_{i+1} is not applicable on r_i, then $r_i[\![o_{i+1}]\!]^! \cap a_{i+1} \subseteq r_{i+1}$,
(D) $r_i \cap a_i \neq \varnothing$.

Each Cartesian set r_i in a relaxed plan execution represents the states 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 can be found. Condition (C) aims to keep some coherence in the execution even when an operator is not applicable: 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 the prefix were fixed, reporting flaws in the suffix otherwise. Condition (D) ensures that any flaw we find at any step can be used to refine the corresponding abstract state a_i.

Definition 2 (Progression Sequence Flaw). *Let $\tau^\alpha = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_2} \dots \xrightarrow{o_n} a_n$ be an abstract plan 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 $\langle r_i, c \rangle$ consisting of two Cartesian sets r_i and c such that (one of):*

(1) o_{i+1} is not applicable from r_i, and c is the set of states in a_i in which o_{i+1} is applicable, i.e. $c = a_i \cap pre(o_{i+1})$;
(2) o_{i+1} is applicable from r_i, but its successor does not intersect to a_{i+1}, i.e. $r_i[\![o_{i+1}]\!] \cap a_{i+1} = \varnothing$, and c is the states in a_i from which a_{i+1} is reached by applying o_{i+1};
(3) $i = n$, and $r_n \cap G = \varnothing$, producing the flaw $\langle r_n, a_n \cap G \rangle$.

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 execution can always be continued, e.g., by setting $r_j = a_j$ for $j \in [i+1, \dots, n]$.

Theorem 1. *Let $\tau^\alpha = a_0 \xrightarrow{o_1} a_1 \dots \xrightarrow{o_n} a_n$ be an abstract plan trace. Then, τ^α is mappable iff τ^α has no progression sequence flaw.*

Proof. On the one hand, assume τ^α has no progression sequence flaw. Then, we show by induction that the plan is mappable. Consider the relaxed execution $r_0, \dots, r_n$ where $[r_i] = \{s_i\}$. As there is no flaw of type (1) o_i is applicable on s_{i-1}, resulting in some s_i. As there is no flaw of type (2), the intersection of r_i with a_i is not empty, so $s_i \in [a_i]$. Finally, as there is no flaw of type (3), $r_n \cap G \neq \varnothing$, so s_n is a goal state. Then, $s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \dots \xrightarrow{o_n} s_n$ is a plan, where $s_i \in [r_i]\ \forall i \in [0, n-1]$ and $s_n \in [G]$.

On the other hand, assume that τ^α is mappable. Then $s_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} s_n$ is a valid plan such that $s_i \in [a_i]$ for all $i \in [0, n]$. Consider any arbitrary relaxed plan execution $r_0, r_1, \dots, r_n$. By condition (A), $s_0 \in [r_0]$. By induction, o_i is applicable in r_{i-1} because it is applicable in s_{i-1}, so no flaw of type (1) exists. By condition (B) $r_i[\![o_{i+1}]\!] \cap a_{i+1} \subseteq r_{i+1}$, so $s_i \in [r_i]$ and $r_i \cap a_i \neq \varnothing : \forall i \in [0, n-1]$, so no flaw of type (2) exists. Finally, no flaw of type (3) exists because s_n is a goal state and $s_n \in [r_n]$. □

Theorem 2. *Let $\tau^\alpha = a_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} a_n$ be an abstract plan trace. Then, $\langle o_1, \dots, o_n \rangle$ may be a plan even if τ^α has progression sequence flaws of type (2).*

Proof. The example used in [19] applies. Consider a task with binary variables $V = \{v_1, v_2\}$, $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0\}$, $G = \{v_2 \mapsto 1\}$ and operators $O = \{o_1\}$, $pre(o_1) = \{v_2 \mapsto 0\}$, $eff(o_1) = \{v_2 \mapsto 1\}$, and an abstraction with states $S^\alpha = \{a_0 = \langle \{0,1\} \times \{0\} \rangle, a_1 = \langle \{0\} \times \{1\} \rangle, a_2 = \langle \{1\} \times \{1\} \rangle\}$. $\tau^\alpha = a_0 \xrightarrow{o_1} a_2$ has a progression sequence flaw of type (2) in o_1 but $\langle o_1 \rangle$ is a plan in the trace $\tau^\alpha = a_0 \xrightarrow{o_1} a_1$ □

Though our definition allows arbitrarily big Cartesian sets along the relaxed execution, doing so results in fewer flaws. In the extreme case, if all r_i are fully relaxed, all operators are applicable on r_i and no flaws are found. In Figure 1, r_1 could be the Cartesian set $\langle \{0,1\} \times \{1,2,3,4\} \rangle$, but then no flaw is found in o_4 due to $v_2 \mapsto 3 \in r_1$, while if $r_1 = \langle \{1\} \times \{2\} \rangle$, o_4 is inapplicable due to $v_2 \mapsto 2$.

Algorithm 1: Find Progression Sequence Flaws

```
   Data: $\Pi{=}\langle V, O, s_0, G\rangle, \tau^\alpha$ ;          // task, abstract plan trace
   Parameters: $r{=}s_0, i{=}0$ ;                  // $r$ and $i$, with default values
   Result: $\mathit{flaws}$ ;              // Progression sequence flaws for $\tau^\alpha$
1  $\mathit{flaws} \leftarrow \varnothing$
2  while $i < n-1$ do
3      if not applicable($r, o_i$) then
4          $\mathit{flaws} \leftarrow \mathit{flaws} \cup \{\langle r, a_i \cap \mathit{pre}(o_i)\rangle\}$
5      $r \leftarrow r[\![o_i]\!]^!$ ;              // Apply $o_i$ even if it is not applicable
6      if $r \cap a_{i+1} = \varnothing$ then
7          $\mathit{flaws} \leftarrow \mathit{flaws} \cup \{\langle r, a_i \cap \mathit{regr}(a_{i+1}, o_i)\rangle\}$
           // Undeviate the Cartesian set
8          forall $v \in V$ do
9              if $r[v] \cap a_{i+1}[v] = \varnothing$ then
10                 $r[v] \leftarrow a_{i+1}[v]$
11     $i \leftarrow i+1$
12 if $r \cap G = \varnothing$ then
13     $\mathit{flaws} \leftarrow \mathit{flaws} \cup \{\langle r, a \cap G\rangle\}$
14 return $\mathit{flaws}$
```

Theorem 3. *Let $r{=}r_0, r_1, \dots, r_n$ be a relaxed execution for $\tau^\alpha = a_0 \xrightarrow{o_1} a_1 \dots \xrightarrow{o_n} a_n$. Let z_i be a Cartesian set such that $z_i \cap a_i \neq \varnothing$, $r_{i-1}[\![o_i]\!] \cap a_i \subseteq z_i$, and $[z_i] \subset [r_i]$. Then, there exists another relaxed execution $z{=}r_0, r_1, \dots, r_{i-1}, z_i, z_{i+1}, \dots, z_n$ such that $[z_j] \subseteq [r_j]$ for $j \in [i, n]$ 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 on 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 $r_i[\![o_{i+1}]\!] \cap a_{i+1}{=}\varnothing$, then there exist some v such that $a_{i+1}[v] \cap r_i[\![o_{i+1}]\!] = \varnothing$. As $[z_i] \subset [r_i]$, then $z_i[\![o_{i+1}]\!][v] \subseteq r_i[\![o_{i+1}]\!][v] = \varnothing$, so a flaw is found for z as well.
(3) If $i = n$ and $r_i \cap G = \varnothing$, then $z_i \cap G = \varnothing$.

For the rest of the execution, note that applying $z_j = r_j$ for $j \in [i+1, n]$ results in a valid execution. However, it is worth noting that other continuations where $[z_j] \subset [r_j]$ may result in more flaws. □

3.2 Progression Sequence Flaws Collection

Algorithm 1 shows the proposed procedure to collect a set of sequence forward 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 sets. The algorithm is always called with the default parameters, $r = s_0$ and $i = 0$, except for special strategies explained in the next section, so it starts at s_0. Initially, the Cartesian set r contains a single state, and therefore the first flaw found by Algorithm 1 will be the same flaw reported by the standard procedure. However, instead of returning the flaw immediately, Algorithm 1 continues looking for more flaws until the end of the abstract plan trace.

We apply operators even when they are not applicable, and when flaws of the second type occur, we "undeviate" the resulting Cartesian set by resetting $r_{i+1}[v]$ to all values compliant with the abstract state a_{i+1}. When a flaw with respect to v has been found, we allow v to take any value consistent with the next abstract state.

Theorem 4. *All flaws returned by Algorithm 1 are progression sequence flaws.*

Proof. Flaws are accumulated in lines 4, 7 and 13. In line 4, the flaw $\langle r_i, a_i \cap \mathit{pre}(o_{i+1})\rangle$ is added if the operator is not applicable, as flaw (1) in Definition 2 does. In line 7, the flaw $\langle r_i, a_i \cap \mathit{regr}(b_i, o_{i+1})\rangle$ is added if $r \cap b = \varnothing$, exactly as flaw (2) does. In line 13, the flaw $\langle r_n, a \cap G\rangle$ is added if $r_n \cap G = \varnothing$, exactly as flaw (3) does. □

Algorithm 1 does not find all forward sequence flaws. As Theorem 3 shows, this would require always keeping each r_i as small as possible. Yet, there are two points in Algorithm 1 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 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 Theorem 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). However, at that point, the algorithm has already found a flaw with respect to v so, by continuing the relaxed execution with all values in a_{i+1}, we seek to only report another flaw if the execution fails from all those values. Note that a second flaw involving the same variable v may be reported, e.g., if somewhere in the remaining abstract plan two operators are applied with contradicting preconditions over v.

3.3 Regression Sequence Flaws

A relaxed plan backward execution can be defined analogously to Definition 1 but replacing s_0 by G and progression by regression.

Definition 3 (Relaxed Plan Backward Execution)**.** *A relaxed plan backward execution $r{=}r_n, r_{n-1}, \dots, r_0$ for an abstract plan $\tau^\alpha{=}a_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} a_n$ is a sequence of Cartesian sets r_i so that:*

(A) $G \subseteq [r_n]$,
(B) if o_i is regressable on r_i, $\mathit{regr}(r_i, o_i) \cap a_{i-1} \subseteq r_{i-1}$,
(C) if o_i is not regressable on r_i, $\mathit{regr}^!(r_i, o_i) \cap a_{i-1} \subseteq r_{i-1}$,
(D) $r_i \cap a_i \neq \varnothing$.

Definition 4 (Regression Sequence Flaw)**.** *Let $\tau^\alpha = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_2} \dots \xrightarrow{o_n} a_n$ be an abstract plan and $r = r_n, r_{n-1}, \dots, r_0$ a relaxed plan backward execution for τ^α. A regression sequence flaw in τ^α is a tuple $\langle r_i, c\rangle$ consisting of two Cartesian sets r_i and c such that (one of):*

(1) o_i is not regressable from r_i, and c is the set of states in a_i in which o_i is applicable, i.e. $c = a_i \cap \mathit{pre}^r(o_i)$;
(2) o_i is regressable from r_i, but its successor does not intersect to a_{i-1}, i.e. $\mathit{regr}(r_i, o_i) \cap a_{i-1}{=}\varnothing$, and c is the states in a_i from which a_{i-1} is reached by regressing o_i;
(3) $i = 0$, and $s_0 \notin [r_0]$, producing the flaw $\langle r_0, s_0\rangle$.

Theorem 5. *Let $\tau^\alpha = a_0 \xrightarrow{o_1} \dots \xrightarrow{o_n} a_n$ be an abstract plan. Then, $\langle o_1, \dots, o_n\rangle$ is a plan if τ^α has no regression sequence flaw.*

Proof. If no regression sequence flaw exists in the abstract plan, there is no flaw of type (3), so that $s_0 \in [r_0]$. For the inductive case, if $s_i \in [r_i]$ and $r_i = \mathit{regr}(r_{i+1}, o_{i+1})$, by the definition of regression there must exist $s_{i+1} \in [r_{i+1}]$ such that $s_i[\![o_i]\!] = s_{i+1}$. Finally, $s_n \in [G]$ due to condition (A), so the sequence is a plan. □

The algorithm to collect regression flaws is like Algorithm 1 but interchanging s_0 and G and using regression semantics.

Progression flaws in the state a_k are different to regression flaws in the state a_{k+1} [19]. Therefore, identifying all flaws requires searching in both directions.

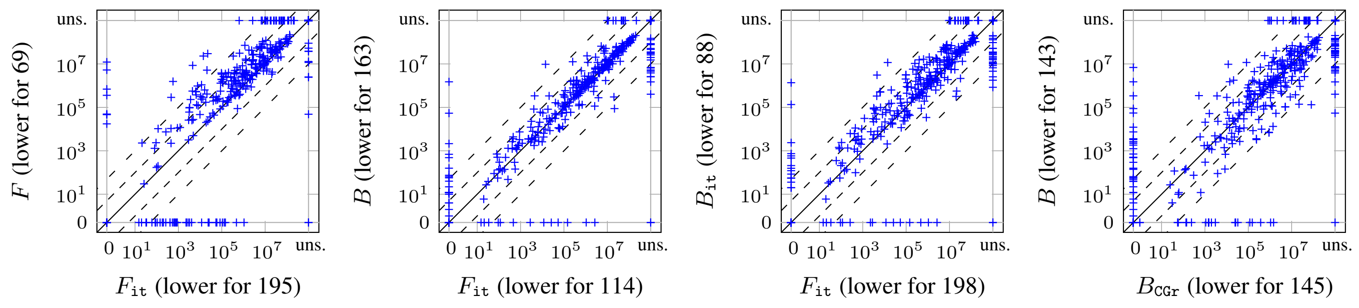

Figure 2: Expansions until last f-layer of selected single-abstraction strategies.

4 Flaw Selection Strategies

Using sequence flaws allows us to identify a possibly large set of flaws for a single abstract plan. The next step is then to refine the abstraction, splitting an abstract state according to one of the flaws, so that the same abstract plan is no longer applicable in the refined abstraction. While it would be possible to refine the abstraction according to multiple flaws, that results in less fine-grained refinements, since after refining a flaw the rest of the flaws may not be part of an optimal abstract plan, so the time to build the abstraction is lower but at the cost of a lower heuristic quality. Still, repairing the right flaw at each step becomes paramount, since finding many flaws in the abstract plan is even harmful if the chosen refinement is worse than repairing the first flaw.

We collect all flaws either in the forward (progression flaws), backward (regression flaws), or both directions (bidirectional strategies). Next, we pick one flaw according to one of the following strategies:

Default Choose the first flaw found along the abstract plan, which coincides with the previous definition of flaw used in previous work, since sequence flaws are a generalization of the first flaw.

Last flaw (`last`) Choose the last flaw found along the abstract plan. This is specially interesting for forward refinements, as flaws are found closer to the goal.

Most refined flaw (`ref`) Choose the most refined state, i.e., the one with the lowest number of values for the flawed variable respect the size of its domain. It deeps into states refined in previous steps, which results in more focused refinements. This criterion was used to choose splits among abstract plans with good results [30].

Highest cost operator (`cost`) Choose the flaw at step i if o_{i+1} is the operator with the highest cost among flaws. The aim is to refine at points where more cost is being spent, which could be specially relevant with cost partitioning, as the cost of many operators is low.

Causal Graph Variable Ordering (`CG` **and** `CGr`) Given a fixed ordering on V, choose the flaw related to the lowest variable in the ordering. We consider two orders based on topological order of the causal graph [11, 12], which have been used before as merge strategies in merge-and-shrink heuristics [13]. `CG` selects first the variables with the most indirect influence over the goals, whereas the reverse ordering, `CGr`, attempts to select variables close to the goal first.

Iterative abstract flaws (`it`) In the forward direction, this strategy starts to search flaws at the end of the plan (Algorithm 1 parameters $r = a_n, i = n$) and it iteratively calls the algorithm from the previous step of the plan until finding a flaw. 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.

Closest to goal flaw (`clo`) Choose the flaw closer to the goal, only relevant in the bidirectional case, as it is equivalent to the default strategy backwards and equivalent to `last` in the forward direction.

On all strategies, whenever more than one flaw could be selected, we break ties according to `ref`.

After selecting the flawed state a_i, one must decide how to split it to refine the abstraction. Typically, several possible splits exist, that divide a_i into a'_i and a''_i according to a variable so that $r_i \cap a'_i = \varnothing$ and $c \cap a''_i = \varnothing$.

The best split at each flawed state is chosen by using split selection strategies [30]. The default strategy maximizes the amount of flaws covered, breaking ties in favor of the most refined split.

Some flaw selection strategies depend on the splits to be accurate, so the split selection strategy must use the same criterion. These strategies are `ref`, `CG`, `CGr` and `cost`. In these cases, ties are broken in favour of the most refined split. For these strategies, splits must be computed in all the flawed states before choosing one of them. Otherwise, the best split is computed only in the chosen flawed state.

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.

5 Experiments

We implemented the sequence refinement within the Scorpion planner [26]. Our experiments run on the Autoscale 21.11 benchmark set [31], which contains the 42 domains 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. 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.

We enable the Scorpion's optimizations for the CEGAR procedure, e.g., using incremental search for finding the optimal abstract plans [27]. In Scorpion's implementation, goals are refined before starting the main CEGAR loop as an optimization. We keep this only for forward refinements, where it improves the results, but it is disabled on all configurations using sequence flaws to compute all flaws in all steps. Another optimization, kept on all configurations, is refining all unreachable facts before goal on tasks with a single goal. They are found using the relaxed planning graph [6]. All experiments reported here use the "wanted" splitting strategy for progression flaws and the "unwanted" strategy for regression flaws, since this setting got the best results in previous work [19].

For single abstraction experiments, we stop the refinement loop when the abstraction has 10 million non-looping transitions 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. Then, we use the resulting heuristic in an A* search.

Code and experiments data are available in Zenodo [20].

Table 1: Coverage of forward, backward and bidirectional strategies for a single abstraction. The $_*$ variant is the best strategy at each domain. The cell in row x and column y is the number of domains where x solved more tasks than y. "Cov" is the total number of tasks solved.

	F	$F_{\texttt{last}}$	$F_{\texttt{ref}}$	$F_{\texttt{cost}}$	$F_{\texttt{CG}}$	$F_{\texttt{CGr}}$	$F_{\texttt{it}}$	Cov
F_*	**23**	**23**	**24**	**25**	**24**	**18**	**11**	**456**
F	–	7	**2**	**5**	**8**	**11**	5	416
$F_{\texttt{last}}$	**9**	–	**8**	**9**	**10**	11	4	421
$F_{\texttt{ref}}$	**2**	6	–	**5**	**6**	**11**	4	413
$F_{\texttt{cost}}$	1	6	2	–	**4**	11	4	410
$F_{\texttt{CG}}$	4	8	4	**4**	–	12	5	411
$F_{\texttt{CGr}}$	**11**	**13**	**11**	**12**	**13**	–	7	405
$F_{\texttt{it}}$	**19**	**16**	**9**	**21**	**20**	**16**	–	**437**

	B	$B_{\texttt{last}}$	$B_{\texttt{ref}}$	$B_{\texttt{cost}}$	$B_{\texttt{CG}}$	$B_{\texttt{CGr}}$	$B_{\texttt{it}}$	Cov
B_*	**15**	**29**	**18**	**21**	**28**	**15**	**15**	**478**
B	–	**24**	**5**	**9**	**23**	**14**	**6**	**451**
$B_{\texttt{last}}$	0	–	4	3	9	5	0	400
$B_{\texttt{ref}}$	2	**21**	–	**6**	**20**	**12**	5	432
$B_{\texttt{cost}}$	0	**21**	1	–	**18**	9	4	433
$B_{\texttt{CG}}$	1	**10**	4	4	–	7	1	412
$B_{\texttt{CGr}}$	11	**22**	**12**	**13**	**20**	–	**12**	438
$B_{\texttt{it}}$	5	**23**	**8**	**9**	**21**	**12**	–	445

	D	$D_{\texttt{last}}$	$D_{\texttt{ref}}$	$D_{\texttt{cost}}$	$D_{\texttt{CG}}$	$D_{\texttt{CGr}}$	$D_{\texttt{it}}$	$D_{\texttt{clo}}$	Cov
D_*	**28**	**30**	**18**	**23**	**29**	**15**	**22**	**23**	**470**
D	–	**24**	**6**	**13**	**24**	**13**	**13**	**9**	**452**
$D_{\texttt{last}}$	0	–	4	4	9	5	2	2	401
$D_{\texttt{ref}}$	2	**21**	–	**12**	**21**	13	**9**	**11**	422
$D_{\texttt{cost}}$	1	**17**	2	–	**16**	9	8	9	413
$D_{\texttt{CG}}$	1	**10**	4	6	–	8	6	8	407
$D_{\texttt{CGr}}$	11	**23**	**14**	**15**	**20**	–	**14**	**15**	425
$D_{\texttt{it}}$	0	**20**	3	**9**	**18**	8	–	**7**	428
$D_{\texttt{clo}}$	0	**17**	4	**10**	**19**	8	3	–	427

Table 2: Statistics for heuristics with a single abstraction. 'Sol. in loop' is the number of tasks solved in the CEGAR loop, 'Abs. time (h)' is the time in hours to build all abstractions. 'Ref.' is the number of refinements, F/B. Flaws are the flawed states found for all tasks, and the percentage respect the states of the abstract plan. 'F/B. Pos.' is the relative position of the selected flawed state respect to the plan length.

	Forward Sequence Flaws							Backward Sequence Flaws							Bidirectional Sequence Flaws							
	F	$F_{\texttt{last}}$	$F_{\texttt{ref}}$	$F_{\texttt{cost}}$	$F_{\texttt{CG}}$	$F_{\texttt{CGr}}$	$F_{\texttt{it}}$	B	$B_{\texttt{last}}$	$B_{\texttt{ref}}$	$B_{\texttt{cost}}$	$B_{\texttt{CG}}$	$B_{\texttt{CGr}}$	$B_{\texttt{it}}$	D	$D_{\texttt{last}}$	$D_{\texttt{ref}}$	$D_{\texttt{cost}}$	$D_{\texttt{CG}}$	$D_{\texttt{CGr}}$	$D_{\texttt{it}}$	$D_{\texttt{clo}}$
Sol. in loop	**173**	110	**173**	152	146	149	140	133	**146**	145	145	139	145	144	142	145	144	139	**152**	130	120	109
Abs. time (h)	20.3	**18.1**	59.0	52.9	47.2	37.7	25.3	26.8	**26.3**	57.5	62.8	58.5	60.5	28.3	46.5	**31.5**	97.7	94.9	86.6	90.2	43.5	39.6
Cost (M)	0.19	0.14	0.16	0.08	0.19	0.06	**0.20**	**0.30**	0.17	0.24	0.29	0.26	0.06	0.30	**0.31**	0.17	0.24	0.26	0.18	0.05	0.11	0.13
Ref. (G)	**0.30**	0.17	0.28	0.28	0.28	0.16	0.21	0.30	0.14	0.27	0.27	0.26	0.16	**0.32**	0.29	0.14	0.26	0.27	**0.28**	0.14	0.19	0.18
F. Flaw (G)	0.3	1.3	**1.5**	1.4	1.5	1.2	0.2	–	–	–	–	–	–	–	0.3	1.0	1.8	**1.8**	1.5	1.6	0.2	1.3
B. Flaw (G)	–	–	–	–	–	–	–	0.3	0.8	**1.6**	1.5	1.5	1.4	1.4	0.8	0.8	1.5	1.5	**1.6**	1.4	0.6	0.6
F. Flaw (%)	8.0	49.0	38.0	40.8	47.6	**50.8**	14.2	–	–	–	–	–	–	–	9.3	50.5	47.9	47.6	49.4	**72.2**	14.4	49.2
B. Flaw (%)	–	–	–	–	–	–	–	9.6	39.7	36.4	34.8	42.3	**51.4**	9.59	9.3	39.7	36.3	35.9	44.2	**56.8**	11.4	27.5
F. Pos. (%)	45.7	51.2	46.4	45.6	45.9	49.0	**51.8**	–	–	–	–	–	–	–	7.2	3.7	34.4	13.0	41.0	42.1	48.3	**49.6**
B. Pos. (%)	–	–	–	–	–	–	–	45.9	39.9	46.6	**46.7**	43.9	43.4	45.9	**47.2**	39.9	46.9	45.5	44.2	45.8	45.4	46.4

5.1 Single Abstraction Experiments

Table 1 shows a comparison of the total coverage (and number of domains with more tasks being solved) of forward, backward and bidirectional strategies. Per-domain results are shown in the supplementary material [20]. The default strategy represents the previous definition of flaw, the state-of-the-art baseline. B is the best strategy, and the best non-default strategy in all directions is `it`, which in the forward direction solves 21 more problems than F and only 6 fewer problems backwards. $F_{\texttt{last}}$ also solves 5 more tasks than F, but its performance is far from $F_{\texttt{it}}$. On the other hand, $B_{\texttt{CGr}}$ favors the variables more causally related to goals, which solves only 7 fewer problems than $B_{\texttt{it}}$ and has a very complementary behavior to the other strategies, being the best in *blocksworld*, *data-network*, *depots*, *pathways*, *pipesworld*, *scanalyzer*, *snake* and *storage* domains. In fact, although the best non-default backward strategy solves 6 fewer tasks, some strategies perform better in some domains, and choosing the best sequence strategy for each domain in any direction would solve 483 problems (32 more problems than B). So better criteria to choose flaws could solve more problems than the state of the art.

The impact of computing sequence flaws is huge, as shown in Table 2 (much higher build time). It is larger in domains with long plans like *airport* and *agricola*, since more splits must be computed in each refinement step for those. It is also higher in strategies that enlarge the abstract plan, like `ref`, and lower in strategies that do not need to compute the splits in all states: `last` and `clo`, with the drawback for $D_{\texttt{clo}}$ of computing and comparing flaws in both directions.

Expansions until the last f-layer are a good indicator of how good the heuristic is during the search, and Figure 2 shows the most representative comparisons. $F_{\texttt{it}}$ is better than F and a bit worse than B, since most of the points are above the diagonal in the first plot and below in the second one. A surprising result is that $F_{\texttt{it}}$ is much better in expansions than $B_{\texttt{it}}$, despite solving one fewer task. $B_{\texttt{CGr}}$, the second-best backward non-default strategy, is similar in expansions to B, so the coverage gap is mostly by the abstractions build time penalty, as total time plots show in the supplementary material [20].

Table 2 shows statistics to analyze the behavior of each strategy. Per-domain details are omitted for space reasons.

One interesting observation is how many times the cost of the abstract plan has been increased, since this describes the preferences of each strategy for refining states: strategies that increase the cost of the abstract plan many times are more focused on getting the actual plan, while strategies with few increments are more focused in increasing the h value of other states. `ref` improves the cost more often in almost all domains, although it gets a lower total improvement due to *parcprinter*. Results vary on the domain, but `it` has the highest number of increments for progression flaws, and `CG` and `cost` get more improvements than `last`, `clo` and `CGr`. So `ref` is very focused in increasing the plan length while `last`, `clo` and `CGr` perform more width-like refinements. `it` refines close to the goal but enlarging the plan, so its refinements are very useful. A similar behavior is observed in the tasks solved during the loop, where F, $F_{\texttt{ref}}$, $B_{\texttt{last}}$ and $D_{\texttt{CG}}$ are the best strategies, but it is not equivalent because refining closer to s_0 is more relevant for this than increasing the cost.

Another interesting point is the total number of refinements, an indicator of the number of states of the abstraction and a proxy for the density of transitions because the loop ends when it has 10M non-looping transitions. The results vary on the domain, but `cost`, `ref` and `CG` are the strategies with the highest number of refinements, while `CGr`, `last` and `clo` are the strategies with fewest refinements.

Two related features are the total number of flaws and the percentage of states with a flaw in an abstract plan. The first one is correlated with the length of the plan, as the more states in the abstract plan, the more flawed states can exist. The second feature is inversely correlated to the first one, as the shorter the plan, the more likely it is to have flaws in a higher percentage of the states. `ref` is the strategy

Table 3: Per-domain coverage for additive abstractions. The $_*$ variant is the best strategy at each domain.

	F	$F_{\texttt{last}}$	$F_{\texttt{ref}}$	$F_{\texttt{cost}}$	$F_{\texttt{CG}}$	$F_{\texttt{CGr}}$	$F_{\texttt{it}}$	Cov
F_*	**13**	**12**	**12**	**23**	**19**	**14**	**14**	**507**
F	–	5	1	4	**11**	**7**	5	479
$F_{\texttt{last}}$	**6**	–	**5**	**8**	**14**	**11**	3	487
$F_{\texttt{ref}}$	**5**	**5**	–	**4**	**12**	**10**	**7**	483
$F_{\texttt{cost}}$	**8**	7	**4**	–	**9**	**10**	**9**	478
$F_{\texttt{CG}}$	6	6	4	4	–	7	5	456
$F_{\texttt{CGr}}$	5	6	4	5	**9**	–	7	466
$F_{\texttt{it}}$	**6**	**6**	**7**	**9**	**14**	**10**	–	**491**

	B	$B_{\texttt{last}}$	$B_{\texttt{ref}}$	$B_{\texttt{cost}}$	$B_{\texttt{CG}}$	$B_{\texttt{CGr}}$	$B_{\texttt{CGr}}$	Cov
B_*	**8**	**13**	**7**	**9**	**16**	**10**	**10**	**500**
B	–	**7**	1	**3**	**12**	**7**	**0**	489
$B_{\texttt{last}}$	1	–	0	3	**10**	4	1	477
$B_{\texttt{ref}}$	**2**	**6**	–	**4**	**13**	**8**	**2**	**492**
$B_{\texttt{cost}}$	2	**7**	1	–	**11**	**8**	2	487
$B_{\texttt{CG}}$	1	5	1	1	–	5	1	458
$B_{\texttt{CGr}}$	4	**6**	5	5	**9**	–	4	474
$B_{\texttt{it}}$	**0**	**7**	1	**3**	**12**	**7**	–	489

	D	$D_{\texttt{last}}$	$D_{\texttt{ref}}$	$D_{\texttt{cost}}$	$D_{\texttt{CG}}$	$D_{\texttt{CGr}}$	$D_{\texttt{it}}$	$D_{\texttt{clo}}$	Cov
D_*	**15**	**15**	**8**	**8**	**18**	**11**	**11**	**8**	**502**
D	–	**7**	0	**5**	**12**	**10**	1	1	492
$D_{\texttt{last}}$	0	–	0	3	**9**	**7**	0	0	476
$D_{\texttt{ref}}$	**1**	**8**	–	**5**	**12**	**10**	**2**	**2**	**493**
$D_{\texttt{cost}}$	**5**	**9**	4	–	**12**	**10**	**5**	**5**	491
$D_{\texttt{CG}}$	3	6	3	3	–	8	2	2	458
$D_{\texttt{CGr}}$	6	**7**	5	5	**10**	–	6	6	466
$D_{\texttt{it}}$	**2**	**9**	**2**	**5**	**13**	**12**	–	**0**	492
$D_{\texttt{clo}}$	**9**	**9**	**2**	**5**	**13**	**12**	**0**	–	492

with the highest number of flawed states and the lowest percentage of flawed states because its plans are the longest ones. But the behavior per domain is very diverse, and `CGr` has a low number of flawed states overall but many more flaws than the rest in *elevators*, *hiking*, *micomic* and *rovers*. `CGr`, `last` and `clo` have the highest percentage of flawed states because their plans are shorter.

The last analysis is the relative position of the refined state with respect to the plan length. That is, 0% means the initial state and 100% means the goal state, while in a plan of 3 states the state of the middle would be the 50%, and so on. The average in the total of domains is dominated by $F_{\texttt{it}}$, though $F_{\texttt{last}}$ and $D_{\texttt{clo}}$ are very close and they win in some domains. It may seem strange that $D_{\texttt{clo}}$ refines states less close to the goal, but the abstract plans found in each iteration depend on previous refinements, so refining a state closest to the goal can lead to not finding flaws as close in next iterations. The values seem low but, as their plans are short, not finding flaws in the last states decreases the average by a high amount.

5.2 *Additive Abstraction Experiments*

Table 3 shows results for additive abstractions via saturated cost partitioning [26]. Per-domain coverage is shown in the supplementary material [20].

The best strategy is $D^{\text{add}}_{\texttt{ref}}$, which solves 493 tasks (4 more tasks than the state of the art) and 14 more problems than F^{add}. $D^{\text{add}}_{\texttt{clo}}$ solves 492 tasks, and it is better than other strategies in more domains. $B^{\text{add}}_{\texttt{ref}}$ also solves 492 tasks, and it is better than B^{add} in 2 domains and worse in only one. $F^{\text{add}}_{\texttt{it}}$ is the best forward strategy, solving 12 more problems than F^{add} and even 2 more problems than B^{add}.

507 problems can be solved by choosing the best forward strategy in each domain, interesting because they are worse separately.

Sequence flaws are more useful in additive abstractions than in a single abstraction, as regression flaws turned out to be not so good in partitioned problems, and better strategies could get better results.

No solution is found in the loop because the problem is partitioned, but all other statistics can be compared [20]. Abstractions build time is very low for all strategies because problems and transitions limits are smaller, but it is lower for some strategies than for the first regression flaw. Generally, all features are lower. The behavior among strategies is like for a single abstraction with small differences.

6 Related Work

Model Checking is a research area that aims to automatically verify the correctness of hardware and software programs. In symbolic model checking, the transition relation of a system is represented with a first-order logic formula. A program is incorrect if the error location is reachable [16, 32]. CEGAR is one of the most successful techniques in Model Checking, and it was the inspiration for the use of CEGAR in Classical Planning [1, 10, 15, 24].

Our work is inspired by sequence interpolation approaches in the context of model checking [1, 16]. Sequence interpolants are a well-known technique to prove the unreachability of error states in symbolic model checking [17, 32]. Given a sequence of formulas $\Gamma = A_1, \dots, A_n$, $\hat{A}_0, \dots, \hat{A}_n$ is an interpolant for Γ when (i) $\hat{A}_0 = \top$ and $\hat{A}_n = \bot$ and (ii) for all $1 \le i \le n$, $\hat{A}_{i-1} \wedge A_i$ implies $\hat{A}_i$ and (iii) for all $1 \le i < n$, $\hat{A}_i \in (\mathcal{L}(A_1, \dots A_i) \cap \mathcal{L}(A_{i+1} \dots A_n))$, where $\mathcal{L}(\Sigma)$ denotes the set of well-formed formulas of first-order logic over a vocabulary Σ. There are direct similarities between sequence interpolants and our sequence flaws. In particular, our relaxed execution states r_i, as well as their $\hat{A}_{i+1}$ formulas, correspond to sets of states that include all states that could be reached by a prefix of the execution of the abstract plan. However, our setting on Cartesian abstractions of SAS$^+$ tasks is a lot simpler. This allows us to use Cartesian sets instead of arbitrary first-order logic formulas. We exploit this to extract multiple sequence flaws efficiently, without having to compute the interpolants from unsatisfiability proofs of logic formulas.

In the context of planning, there are multiple works that explore how to conduct CEGAR in different ways [9, 25], or how to combine multiple abstractions using cost partitioning [23]. The closest work to ours is the refinement strategies by [30], which also aims at identifying multiple flaws, but focusing on finding flaws from multiple abstract plans instead of only one.

Another topic in planning with similarities to our work is Partial-Order Causal-Link [5, 18, 33], which refines partial plans by detecting flaws in them. Flaws can be an open precondition not protected by a causal link or an operator that can delete a precondition before it is needed. Open preconditions are resolved adding a causal link and threats are resolved moving the operator before the variable is produced or after the operator that needs it. 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, but the flaws and the refinements are completely different.

7 Conclusions

CEGAR is a method to iteratively refine abstractions by identifying flaws in optimal abstract plans. But previous work find a single flaw per plan, the first one along its execution. Our main contribution is a new type of flaw that allows to search flaws after the first one. This enables identifying flaws that could not be found before, and it opens research opportunities for new refinement strategies.

We have experimentally shown that different selection flaw strategies result in very different behaviour, and that each strategy is better than the others in some domains. We have also shown that iterative strategies and strategies based on the most refined state can get better heuristics than regression flaws, especially in additive abstractions, opening research opportunities for smarter flaw selection strategies.

Acknowledgements

This work was partially funded by grant PID2021-127647NB-C21 from MCIN/AEI/10.13039/501100011033, by the ERDF "A way of making Europe", and by the Madrid Government under the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M17) in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

References

[1] A. Albarghouthi. *Software Verification with Program-Graph Interpolation and Abstraction*. PhD thesis, University of Toronto, Canada, 2015. URL http://hdl.handle.net/1807/69199.

[2] V. Alcázar, D. Borrajo, S. Fernández, and R. Fuentetaja. Revisiting regression in planning. In F. Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 2254–2260. AAAI Press, 2013.

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

[4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2001.

[5] P. Bercher. A closer look at causal links: Complexity results for delete-relaxation in partial order causal link (POCL) planning. In R. P. Goldman, S. Biundo, and M. Katz, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pages 36–45. AAAI Press, 2021.

[6] A. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

[7] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[8] S. Edelkamp. Planning with pattern databases. In A. Cesta and D. Borrajo, editors, *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 84–90. AAAI Press, 2001.

[9] R. Eifler and M. Fickert. Online refinement of Cartesian abstraction heuristics. In V. Bulitko and S. Storandt, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*, pages 46–54. AAAI Press, 2018.

[10] Á. Hajdu and Z. Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020.

[11] M. Helmert. A planning heuristic based on causal graph analysis. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 161–170. AAAI Press, 2004.

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

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

[14] R. Kreft, C. Büchner, S. Sievers, and M. Helmert. Computing domain abstractions for optimal classical planning with counterexample-guided abstraction refinement. In S. Koenig, R. Stern, and M. Vallati, editors, *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*. AAAI Press, 2023.

[15] S. Löwe. *Effective Approaches to Abstraction Refinement for Automatic Software Verification*. PhD thesis, University of Passau, Germany, 2017. URL https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/481.

[16] K. L. McMillan. Applications of craig interpolants in model checking. In N. Halbwachs and L. D. Zuck, editors, *Lecture notes in computer science*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005. doi: 10.1007/978-3-540-31980-1_1. URL https://doi.org/10.1007/978-3-540-31980-1_1.

[17] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006. doi: 10.1007/11817963_14. URL https://doi.org/10.1007/11817963_14.

[18] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992)*, pages 103–114. Morgan Kaufmann, 1992.

[19] M. Pozo, Á. Torralba, and C. Linares López. When CEGAR meets regression: A love story in optimal classical planning. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, pages 20238–20246. AAAI Press, 2024.

[20] M. Pozo, Á. Torralba, and C. Linares López. Gotta catch 'em all! sequence flaws in CEGAR for classical planning. supplementary material, code, experimental results and scripts. 10.5281/zenodo.13378665, 2024.

[21] J. Rintanen. Regression for classical and nondeterministic planning. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, pages 568–572. IOS Press, 2008.

[22] A. Rovner, S. Sievers, and M. Helmert. Counterexample-guided abstraction refinement for pattern selection in optimal classical planning. In N. Lipovetzky, E. Onaindia, and D. E. Smith, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pages 362–367. AAAI Press, 2019.

[23] J. Seipp. Better orders for saturated cost partitioning in optimal classical planning. In A. Fukunaga and A. Kishimoto, editors, *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, pages 149–153. AAAI Press, 2017.

[24] J. Seipp and M. Helmert. Counterexample-guided Cartesian abstraction refinement. In D. Borrajo, S. Kambhampati, A. Oddi, and S. Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pages 347–351. AAAI Press, 2013.

[25] J. Seipp and M. Helmert. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

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

[27] J. Seipp, S. von Allmen, and M. Helmert. Incremental search for counterexample-guided Cartesian abstraction refinement. In J. C. Beck, E. Karpas, and S. Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pages 244–248. AAAI Press, 2020.

[28] S. Sievers and M. Helmert. Merge-and-shrink: A compositional theory of transformations of factored transition systems. *Journal of Artificial Intelligence Research*, 71:781–883, 2021.

[29] J.-G. Smaus and J. Hoffmann. Relaxation refinement: A new method to generate heuristic functions. In D. A. Peled and M. J. Wooldridge, editors, *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MoChArt 2008)*, pages 147–165, 2009.

[30] D. Speck and J. Seipp. New refinement strategies for Cartesian abstractions. In S. Thiébaux and W. Yeoh, editors, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*, pages 348–352. AAAI Press, 2022.

[31] Á. Torralba, J. Seipp, and S. Sievers. Automatic instance generation for classical planning. In R. P. Goldman, S. Biundo, and M. Katz, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pages 376–384. AAAI Press, 2021.

[32] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 1–8. IEEE, 2009. ISBN 978-1-4244-4966-8. doi: 10.1109/FMCAD.2009.5351148.

[33] H. L. S. Younes and R. G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.