Configuring Lifted Initial States for Classical Planning

Alba Gragera, Raquel Fuentetaja, Ángel García-Olaya

Computer Science and Engineering Department, Universidad Carlos III de Madrid, Spain
{agragera, rfuentet, agolaya}@inf.uc3m.es

Abstract

In classical planning, the initial state is usually fully specified, which limits expressiveness and the discovery of better plans through alternative configurations. In this work, we consider lifted initial states, where some atoms may contain variables, and we seek a grounded initial state that optimizes the resulting plan. We solve this novel problem by compiling it into classical planning, allowing the planner to choose variable assignments. We introduce two compilations: minimum commitment instantiation, where variable assignments can be deferred until needed in the plan; and early instantiation, where variables are forced to be grounded at the start of the search. We empirically compare these strategies across different planning domains to evaluate their effectiveness.

1 Introduction

Classical planners generate sequences of actions, called plans, that achieve goals from specific initial states. To construct such plans, planning systems typically rely on complete and accurate models expressed in PDDL (McDermott et al. 1998), which separates the planning task into a domain description, specifying all available actions and the predicates used to describe states; and a problem description that contains the initial state and the goals to achieve. Traditionally, first-order logic has been confined to the domain definition, while the problem definition is expressed in a propositional form. Specifically, the initial state is encoded as a set of ground atoms that fully defines object properties and relations. This representation limits expressiveness, as it cannot naturally capture missing or flexible information. As a consequence, modelers must commit *a priori* to a single concrete configuration. This increases modeling effort, which is often challenging and time-consuming (McCluskey, Vaquero, and Vallati 2017), and restricts the solution space explored by the planner, possibly excluding better plans.

In general, there have been efforts in the literature to increase the expressiveness of planning task definitions, although most of the work has focused primarily on enhancing the capabilities of the domain itself (Pednault 1989; Lindsay 2023; Haslum and Corrêa 2024). With respect to the problem specification, Corrêa et al. (2024) relaxed the assumption that the set of objects is fixed at the start and cannot change during plan execution, and introduced an extension to the classical planning formalism in which action effects can create and remove objects. They motivate their work with a logistics scenario in which Bob, a classical planning researcher, opens a new logistics company. He needs to decide in advance how many trucks to buy for deliveries. Estimating this number is challenging: too few may prevent a solution, while too many dramatically slows down planning. In this work, we consider a follow-up to this scenario. Bob's logistics company has grown, now operating multiple depots in different cities and managing a fleet of trucks and vans, with vans having lower capacity than trucks. Given the set of deliveries, Bob tries to find the best initial location for each vehicle to obtain low-cost plans. However, the large number of possible combinations quickly becomes overwhelming. What if some vans are assigned to one depot, but a truck would have been a better choice to reduce travel time?

Typical Initial State	**Lifted Initial State**
(at truck1 depot1)	(at truck1 *?placeA* - location)
(at truck2 depot1)	(at truck2 *?placeA* - location)
(at van1 depot1)	(at *?vehicle1* - vehicle *?placeB* - location)
(at van2 depot2)	(at *?vehicle2* - vehicle *?placeC* - location)
(at van3 headquarter1)	(at van3 headquarter1)
...	...

Figure 1: Example of a classical initial state versus a lifted one (variables are denoted with ?).

In this paper, we introduce a novel problem in planning in which the initial state can represent flexible information, modeled by using variables. Figure 1 shows an example of a *lifted* initial state, specifying that the two trucks must be at the same location and that there are two additional vehicles at any location. To handle such lifted initial states, we propose a planning compilation that enables the planner to determine variable instantiations during the search process. The benefits of this approach are twofold. First, it increases expressiveness and reduces the effort to create the problem when there is a set of available objects, but we do not know, or do not care, their exact configuration. Second, it allows the planner to determine the best assignment of certain configurable parts of the initial state, potentially yielding lower-cost plans. Here, we assume a fixed set of objects. The planner must solve the problem using only the trucks and vans

already available.

Lifted initial states are related to conformant planning (Smith and Weld 1998; Hoffmann and Brafman 2006; Palacios and Geffner 2007), which is the problem of finding a sequence of actions for achieving a goal in the presence of uncertainty in the initial state or action effects. However, conformant planning focuses on finding (conformant) plans that succeed in all possible worlds consistent with that uncertainty. In contrast, we consider lifted initial states in a deterministic setting, in which the planner determines the variable assignments that optimize the resulting plan. The reconfiguration of initial states is also related to counterfactual reasoning (Lewis 1973). In planning, counterfactuals have been used in the context of initial states, mainly to render unsolvable planning tasks solvable (Göbelbecker et al. 2010; Sreedharan et al. 2019), rather than to optimize its configuration.

In the rest of the paper we first introduce lifted initial-state planning tasks, and then present two planning compilations to solve them. The experimental evaluation shows that the proposed approach can optimize the initial-state configuration to obtain lower-cost plans.

2 Background

We use the first-order (lifted) planning formalism (Fikes and Nilsson 1971), where a classical planning task is a pair $\Pi = \langle D, P \rangle$, where D is the *planning domain* and P defines a *planning problem*. A planning domain is a tuple $D = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A} \rangle$; where $\mathcal{H}$ is a *type hierarchy*; $\mathcal{C}$ is a set of (domain) *constants*; $\mathcal{P}$ is a set of *predicates* defined by the predicate name and the types of its arguments; and $\mathcal{A}$ is a set of *action schemas*. Each $p(t) \in \mathcal{P}$ is an *atom*, where p is an n-ary predicate, and $t = t_1, \ldots, t_n$ are either typed constants or typed variables. An atom is *grounded* if its arguments do not contain variables, and *lifted* otherwise. Action schemas $a \in \mathcal{A}$ are tuples $a = \langle \mathit{name}(a), \mathit{par}(a), \mathit{pre}(a), \mathit{add}(a), \mathit{del}(a), \mathit{cost}(a) \rangle$, defining the action *name*; the action *parameters* (a finite set of variables); the *precondition*; the *add* and *delete* effects; and the action *cost*. $\mathit{pre}(a)$ is a set of literals representing what must be true or false in a state to apply the action. $\mathit{add}(a)$ and $\mathit{del}(a)$ represent the changes produced in a state by the application of the action (added and deleted atoms, respectively). Finally, $\mathit{cost}(a)$ is the cost of the action. A planning problem is a tuple $P = \langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{O}$ is a set of typed constants representing problem-specific *objects*; $\mathcal{I}$ is the set of ground atoms in the *initial state*; and $\mathcal{G}$ is the set of ground atoms defining the *goal*.

Grounded actions are obtained from action schemas by substituting the variables in the parameters by constants in $\mathcal{O}$. A grounded action $\underline{a}$ is *applicable* in a state s if $\mathit{pre}(\underline{a}) \subseteq s$. When a grounded action is applied to s we obtain a successor state s', defined as $s' = (s \setminus \mathit{del}(\underline{a})) \cup \mathit{add}(\underline{a})$. A *plan* π is a sequence of grounded actions $\underline{a}_1, \ldots, \underline{a}_n$ such that each $\underline{a}_i$ is applicable to the state s_{i-1} generated by applying $\underline{a}_1, \ldots, \underline{a}_{i-1}$ to $\mathcal{I}$; $\underline{a}_1$ is applicable in $\mathcal{I}$; and the consecutive application of all actions in the plan generates a state s_n containing the goals, $\mathcal{G} \subseteq s_n$. The cost of a plan is defined as $\mathit{cost}(\pi) = \sum_{\underline{a}_i \in \pi} \mathit{cost}(\underline{a}_i)$.

3 Lifted Initial State Planning Tasks

In this work, we allow the initial state to contain lifted atoms, in which some terms may be typed variables, and denote it as $\overline{\mathcal{I}}$. The set of variables in the initial state is denoted as $\mathit{vars}(\overline{\mathcal{I}})$. Each variable $v \in \mathit{vars}(\overline{\mathcal{I}})$ is associated with a type from the type hierarchy $\mathcal{H}$, denoted as $\mathit{type}(v)$. In the example of Figure 1, $\mathit{vars}(\overline{\mathcal{I}}) = \{\mathit{placeA}, \mathit{placeB}, \mathit{placeC}, \mathit{vehicle1}, \mathit{vehicle2}\}$, with $\mathit{type}(\mathit{placeA}) = \mathit{type}(\mathit{placeB}) = \mathit{type}(\mathit{placeC}) = \mathit{location}$ and $\mathit{type}(\mathit{vehicle1}) = \mathit{type}(\mathit{vehicle2}) = \mathit{vehicle}$. Then, we define *lifted initial state planning tasks* as follows.

Definition 1 (Lifted Initial State Planning Task). *A lifted initial state planning task is a pair $\overline{\Pi} = \langle D, P \rangle$, where D is a planning domain and $P = \langle \mathcal{O}, \overline{\mathcal{I}}, \mathcal{G} \rangle$ is a planning problem such that the initial state is lifted, meaning that it contains variables, i.e., $|\mathit{vars}(\overline{\mathcal{I}})| \geq 1$.*

Although the initial state contains variables, we remain within classical planning, in which any solution requires fully instantiating the initial state into a concrete configuration. However, this introduces a key difference from standard classical planning: we must select among multiple possible instantiations, which we formalize as valid assignments. We denote by $\mathit{type}^{\downarrow}(v, \mathcal{H})$ the type of v and all its subtypes in $\mathcal{H}$, which allows us to ensure type consistency in variable assignments.

Definition 2 (Valid Assignment). *A* valid assignment *for a lifted initial state planning task $\overline{\Pi} = \langle D, \langle \mathcal{O}, \overline{\mathcal{I}}, \mathcal{G} \rangle \rangle$, is a mapping $\sigma : \mathit{vars}(\overline{\mathcal{I}}) \rightarrow \mathcal{O}$ such that:*

1. *For each variable $v \in \mathit{vars}(\overline{\mathcal{I}})$, the assigned object, $\sigma(v)$, satisfies type consistency: $\mathit{type}(\sigma(v)) \in \mathit{type}^{\downarrow}(v, \mathcal{H})$; and*
2. *The classical planning task $\overline{\Pi}[\sigma] = \langle D, \langle \mathcal{O}, \sigma(\overline{\mathcal{I}}), \mathcal{G} \rangle \rangle$ is solvable, where $\sigma(\overline{\mathcal{I}})$ is obtained by replacing each variable $v \in \mathit{vars}(\overline{\mathcal{I}})$ with $\sigma(v) \in \mathcal{O}$.*

Given a valid assignment σ for a lifted initial state planning task $\overline{\Pi}$, we say that $\overline{\Pi}[\sigma]$ is the classical planning task induced by σ. We define the set of plans for $\overline{\Pi}$ as the union of the sets of plans for all the classical planning tasks induced by valid assignments for $\overline{\Pi}$. Then, a lifted initial state planning task $\overline{\Pi}$ is *solvable* if there exists at least one valid assignment for it. A valid assignment σ for $\overline{\Pi}$ is *optimal* if there is no other valid assignment for $\overline{\Pi}$ for which the induced classical planning task has a plan with lower cost than an optimal plan for $\overline{\Pi}[\sigma]$. A plan for $\overline{\Pi}$ is *optimal* if it is obtained from an optimal valid assignment.

4 Solving Lifted Initial State Planning Tasks

We solve lifted initial state planning tasks by encoding the selection of a valid assignment (see Definition 2) directly into the planning task. This enables the planner to determine both the initial state instantiation and the plan. To this end, we compile $\overline{\Pi}$ into a classical planning task augmented with *assignment actions*, which explicitly instantiate variables during planning.

We propose two alternative compilations, each corresponding to a different variable instantiation strategy. In the first compilation, we adopt a minimum-commitment, or

lazy, strategy, in which variables are assigned values only when required to make progress toward the goals. In the second compilation, we adopt an *early*-commitment strategy, where variable assignments are forced to be applied first, and only after that the domain-specific actions are considered to search for a solution.

Given a lifted planning task $\overline{\Pi} = \langle D, P \rangle$, with $D = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A} \rangle$ and $P = \langle \mathcal{O}, \overline{\mathcal{I}}, \mathcal{G} \rangle$, we generate a classical planning task, denoted as follows depending on the strategy:

- **Lazy variable instantiation**: the compiled classical planning task is $\Pi_L = \langle D_L, P' \rangle$, where $D_L = \langle \mathcal{H}', \mathcal{C}', \mathcal{P}_L, \mathcal{A}_L \rangle$ and $P' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G} \rangle$.
- **Early variable instantiation**: the compiled classical planning task is $\Pi_E = \langle D_E, P' \rangle$, where $D_E = \langle \mathcal{H}', \mathcal{C}', \mathcal{P}_E, \mathcal{A}_E \rangle$ and $P' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G} \rangle$.

As can be seen, both compilations share the same type hierarchy ($\mathcal{H}'$), domain constants ($\mathcal{C}'$), problem specific constants ($\mathcal{O}$), and initial state ($\mathcal{I}'$), but differ in their predicates and actions. The original objects and goals are preserved from the lifted planning task. The only changes concern how the variables in the initial state are instantiated with those objects.

The next section describes the lazy variable instantiation compilation, and then the early variable instantiation compilation is presented in the following section as an extension of the former.

Lazy Variable Instantiation

In this compilation, variable assignments are interleaved with domain-specific actions and may be introduced at any point during the search. Our approach is inspired by the compilation of Gragera et al. (2023), who reformulate an unsolvable task so that it can self-repair during search; analogously, we reformulate the task so that variable instantiation is also resolved during search. In this sense, our compilation reifies initial state variables and their types as planning objects, allowing variable instantiation to be expressed within the planning language itself. We define the components of the compiled planning task as follows:

TYPE HIERARCHY ($\mathcal{H}'$). We extend the original type hierarchy to form a new hierarchy $\mathcal{H}'$, illustrated in Figure 2:

- The type *o_var* is used to represent the variables in the initial state.
- The type *o_type* represents the original types of the planning task and allows associating variables with them.
- The type *o_obj* encapsulates the original type hierarchy, ensuring that the extended hierarchy $\mathcal{H}'$ subsumes the original.

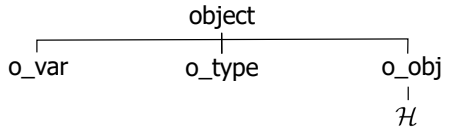

Figure 2: Type hierarchy of the reformulated planning task.

Note that the original type hierarchy can be further refined to incorporate additional knowledge about the problem that restricts variable assignments. For instance, if we want the variable *?placeA* to map exclusively to a subset of depots, we can introduce a new type *selected-depot* as a subtype of *location*, and then restrict *?placeA* to this new type. This directly injects that knowledge into the model, reducing the search space by limiting the possible instantiations of *?placeA*.

CONSTANTS ($\mathcal{C}'$). We introduce the sets of constants:

- $\mathcal{C}_{var}$ of type *o_var* to represent the variables defined in $\overline{\mathcal{I}}$, $\mathcal{C}_{var} = \{v \mid v \in vars(\overline{\mathcal{I}})\}$.
- $\mathcal{C}_{type}$ of type *o_type* to represent the types in the original hierarchy, $\mathcal{C}_{type} = \{t_{type} \mid type \in \mathcal{H}\}$.

Thus, $\mathcal{C}' = \mathcal{C} \cup \mathcal{C}_{var} \cup \mathcal{C}_{type}$, where $\mathcal{C}$ is the set of original domain constants, which is empty in many typical domains. Figure 3 shows examples of the type redefinition and the domain constants for the running example domain. Changes are highlighted in bold.

```
(:types
   o_var o_type o_obj - object
   location locatable - o_obj
   vehicle package - locatable
   truck van - vehicle)
(:constants
   t_vehicle t_location t_truck - o_type
   placeA vehicle1 placeB ... - o_var
   ...)
```

Figure 3: Reformulated types and constants for the running example.

PREDICATES ($\mathcal{P}_L$). We preserve the original predicates of the planning task and introduce an additional set of predicates ($\mathcal{P}_{lifted}$) to characterize the variables and the instantiation constraints. These predicates are as follows:

- ***obj_type***(o_obj, o_type) encodes the type of an object according to the type hierarchy and the variable types. If the lifted initial state contains a variable v and the type of an object $o \in \mathcal{O}$ belongs to $type^{\downarrow}(v, \mathcal{H})$, then we will generate the static fact $obj\_type(o, t_{type(v)})$, where $t_{type(v)}$ is the constant representing $type(v)$ in $\mathcal{C}_{type}$. For instance, if the lifted initial state contains the variable *?vehicle1* of type *vehicle*, and *truck3* is an object of type *truck*, which is a subtype of *vehicle* (see Figure 3), then the initial state includes the fact $obj\_type(truck3, t\_vehicle)$.
- ***var_type***(o_var, o_type) specifies the type of each variable in the lifted initial state. For instance, $var\_type(vehicle1, t\_vehicle)$ declares that variable *vehicle1* has type *vehicle*, and $var\_type(placeA, t\_location)$ declares that variable *placeA* has type *location*. Together with *obj_type*, this predicate ensures type consistency when assigning objects to variables during planning.

- ***free***(o_var) denotes that a variable has not been assigned a value yet. For instance, *free*(*placeB*). Once the variable is instantiated, the corresponding *free* fact is removed from the state, preventing further assignments to the same variable.
- ***assignment***(o_var, o_obj) denotes that a variable has been assigned a specific object. For example, *assignment*(*placeA*, *depot1*) indicates that σ(*placeA*) = *depot1*, i.e., the variable *placeA* is assigned the object *depot1*.
- ***applied***$\_\langle n \rangle(o\_var_1, \ldots, o\_var_n)$ indicates whether the lifted atoms of the initial state whose variables are exactly those in the set $\{o\_var_1, \ldots, o\_var_n\}$ have already been grounded using the objects assigned to those variables. For instance, the presence of *applied_2*(*vehicle1*, *placeB*) denotes that any lifted atom in the initial state whose variables are exactly *vehicle1* and *placeB*, for instance, *at(?vehicle1,?placeB)*, has already been instantiated with the objects assigned to those variables. This predicate is used to prevent re-instantiating the same lifted atom multiple times during planning.

Thus, $\mathcal{P}_L = \mathcal{P} \cup \mathcal{P}_{lifted}$, where $\mathcal{P}$ is the original set of predicates.

ACTIONS ($\mathcal{A}_L$). We keep the original actions of the planning task and introduce a new set of zero-cost actions $\mathcal{A}_{lazy}$ that contains an *assign* action scheme (Figure 4) and one or several *apply* action schemes (Figure 5). We describe them below, together with mechanisms to constrain the lifted variables and avoid generating invalid (illegal) initial states.

Assign actions. The assign action is responsible for setting the value of each free variable during planning, while the apply actions instantiate the lifted atoms in $\overline{\mathcal{I}}$ according to the computed variable assignments.

The precondition of the assign action represents that a free variable should be assigned an object of a compatible type. In the effects, the variable is associated with the corresponding object and marked as no longer free.

$$
\begin{aligned}
name =& \textit{assign} \\
par =& \{v : o\_var, t : o\_type, o : o\_obj\} \\
pre =& \{\textit{free}(v), \textit{var\_type}(v, t), \textit{obj\_type}(o, t)\} \\
add =& \{\textit{assignment}(v, o)\} \\
del =& \{\textit{free}(v)\}
\end{aligned}
$$

Figure 4: Assign action scheme.

Apply actions. To generate the apply actions, we consider the set of variables co-occurring in each atom in the lifted initial state, and denote the collection of these sets as *V*:

$$V = \{\textit{vars}(p(t)) \mid p(t) \in \overline{\mathcal{I}}\}$$

Let $V = \{V_1, \ldots, V_k\}$. Then, for each $V_i \in V$, we define a single apply action that updates the state by adding all atoms from $\overline{\mathcal{I}}$ whose variables belong to V_i, grounded according to the planner's assignments. The number of apply actions is exactly k. Figure 5 depicts the apply action schema for one of such variable sets $V_i = \{v_1, \ldots, v_m\}$. In the precondition, we check that the variables in V_i have been assigned to some objects and that the lifted atoms associated with those variables have not been instantiated yet. In the effect, we add all the grounded atoms obtained by instantiating the lifted atoms associated with V_i using the assigned objects. We also introduce the *applied*$\_\langle n \rangle$ fact to indicate that lifted atoms with those variables are already instantiated.

$$
\begin{aligned}
name =& \textit{apply\_assignment\_vars\_}V_i \\
par =& \{o_1, \ldots, o_m : o\_obj\} \\
pre =& \{\{\textit{assignment}(v_j, o_j) \mid v_j \in V_i\}, \\
& \quad \neg \textit{applied}\_\langle m \rangle(v_1, \ldots, v_m)\} \\
add =& \{\{p(\sigma(t)) \mid p(t) \in \overline{\mathcal{I}}, V_i = \textit{vars}(p(t))\}, \\
& \quad \textit{applied}\_\langle m \rangle(v_1, \ldots, v_m)\} \\
del =& \emptyset
\end{aligned}
$$

Figure 5: Apply action scheme for the set of variables V_i, with $|V_i| = m$. $\sigma(t)$ is the result of replacing in t each variable v_j by the corresponding assigned object o_j.

Then, $\mathcal{A}_L = \mathcal{A} \cup \mathcal{A}_{lazy}$, where $\mathcal{A}$ is the set of original domain-specific actions.

Following the running example in Figure 1, $V = \{\{placeA\}, \{\textit{vehicle1}, \textit{placeB}\}, \{\textit{vehicle2}, \textit{placeC}\}, \ldots\}$. Considering the second set of variables, $V_2 = \{\textit{vehicle1}, \textit{placeB}\}$, the apply action generated is shown in Figure 6. In the precondition of the action, the constants *vehicle1* and *placeB* ensure that the variables being assigned correspond to the intended ones. The effect then updates the state by grounding all lifted atoms associated with *vehicle1* and *placeB*. In the initial state of Figure 1, the only lifted atom with those variables is *(at ?vehicle1 ?placeB)*, so the effect adds *(at ?obj1 ?obj2)*. This grounding is marked as applied, preventing it from being repeated, which would otherwise be equivalent to resetting the initial state.

```
(:action apply_assign_vars_vehicle1_placeB
   :parameters (?obj1 ?obj2 - o_obj)
   :precondition (and
       (assignment vehicle1 ?obj1)
       (assignment placeB ?obj2)
       (not (applied_2 vehicle1 placeB)))
   :effect (and
       (at ?obj1 ?obj2)
       (applied_2 vehicle1 placeB)))
```

Figure 6: Example of an *apply* action scheme for the variable set $V_2 = \{\textit{vehicle1}, \textit{placeB}\}$.

Constraints on variables. Variables with different names are not required to be assigned to different objects. However, in some cases this must be enforced to avoid illegal initial states. For example, in Figure 1, the variables *?placeB* and *?placeC* may be instantiated to the same location, whereas *?vehicle1* and *?vehicle2* must denote different objects. To enforce this, we allow including the constraint $(\neq (?v_i, ?v_j))$ in the lifted initial state, which requires the variables v_i and v_j to be assigned different objects. To ensure that this inequality is not violated, the compilation generates an additional *check-inequality* action, following the schema shown in Figure 7. This action considers the set of variables $V_{\neq}$, containing the variables in the lifted initial state that must be assigned different objects $(\neq (?v_i, ?v_j))$:

$$V_{\neq} = \{v_i, v_j \in \mathit{vars}(\overline{\mathcal{I}}) \mid \neq (?v_i, ?v_j) \in \overline{\mathcal{I}}\}$$

If $|V_{\neq}| > 0$, the predicate *inequality_checked* is introduced, i.e., $\mathcal{P}_{\mathit{lifted}} \cup \{\mathit{inequality\_checked}\}$, and the action *check_inequality* is included, i.e., $\mathcal{A}_L = \mathcal{A} \cup \mathcal{A}_{\mathit{lazy}} \cup \{\mathit{check\_inequality}\}$.

$$\begin{aligned}
\mathit{name} =& \mathit{check_inequality}\\
\mathit{par} =& \{v_1, \ldots v_k : \mathit{o_var}; o_1, \ldots, o_k : \mathit{o_obj}\}\\
\mathit{pre} =& \{\{\mathit{assignment}(v_i, o_i) \mid v_i \in V_{\neq}\},\\
& \quad \{\neq (o_i, o_j) \mid v_i \neq v_j \textit{ required in } \overline{\mathcal{I}}\}\}\\
\mathit{add} =& \{(\mathit{inequality_checked})\}\\
\mathit{del} =& \emptyset
\end{aligned}$$

Figure 7: Check inequality action for $|V_{\neq}| = k$.

Also, reusing an object that already appears in the fixed part of the state may contradict the atoms in which that object is already present. For example, lifting block *B* from a single atom of the blocksworld state {*(clear B), (on B A), (ontable A)*} yields {*(clear ?block) (on B A) (ontable A)*}, where assigning *?block* = *A* produces the inconsistent state {*(clear A) (on B A)*}. Such assignments violate the domain's state invariants.

To rule out such cases, we must account for the domain's structural invariants so that variable assignments always produce legal states. This can be done in several ways: (1) by using mutex groups (Helmert 2009) to add preconditions to *apply* actions, thereby forbidding assignments that violate a mutex (e.g., when applying *(clear ?block)*, requiring that no block is on *?block*); (2) by encoding invariants as axioms that identify illegal states (Grundke, Röger, and Helmert 2024) and restricting action applicability to legal states; or (3) by deriving variable–variable and variable–object equality/inequality constraints from the invariants and enforcing them through actions such as *check_inequality* (Figure 7).

INITIAL STATE ($\mathcal{I}'$). To generate the initial state we modify $\overline{\mathcal{I}}$ by removing the atoms containing variables (denoted as $\overline{\mathcal{I}}_{\mathit{vars}}$); including all ground atoms corresponding to predicates in $\mathcal{P}_{\mathit{lifted}}$ that refer to object and variable types; and marking all variables as free (denoted as $\mathcal{I}_{\mathit{lifted}}$). Then, $\mathcal{I}' = \overline{\mathcal{I}} \setminus \overline{\mathcal{I}}_{\mathit{vars}} \cup \mathcal{I}_{\mathit{lifted}}$:

$$\begin{aligned}
\overline{\mathcal{I}}_{\mathit{vars}} =& \{\, p(t) \in \overline{\mathcal{I}} \mid \exists\, t_i \in t \text{ such that } t_i \in \mathit{vars}(\overline{\mathcal{I}}) \,\}\\
\mathcal{I}_{\mathit{lifted}} =& \{\mathit{obj_type}(o, t_{\mathit{type}(v)}) \mid o \in \mathcal{O}, v \in \mathcal{C}_{\mathit{var}},\\
& \quad \mathit{type}(o) \in \mathit{type}^{\downarrow}(v, \mathcal{H})\} \cup\\
& \{\mathit{var_type}(v, t_{\mathit{type}(v)}) \mid v \in \mathcal{C}_{\mathit{var}}\} \cup\\
& \{\mathit{free}(v) \mid v \in \mathcal{C}_{\mathit{var}}\}
\end{aligned}$$

When inequality constraints are present ($|V_{\neq}| > 0$), the *inequality_checked* predicate is also added to the problem goals to ensure that all variable assignments satisfy the specified constraints, so $\mathcal{G} \cup \{\mathit{inequality\_checked}\}$. Except in this possible case, the objects and goals from the original problem remain unchanged.

Early Variable Instantiation

The search for the early variable instantiation compilation proceeds in three sequential stages. First, the *assignment phase*, where free variables are assigned values. Then, the *application phase*, in which the initial state is instantiated using the assigned values. Finally, the *planning phase*, where the planner searches for a solution. Each branch of the search starts with a specific assignment of values to variables, which the planner then uses to try to generate a plan that satisfies the goals. The approach is intuitive: with a fully determined initial state, the complexity of solving the task depends solely on the underlying planning problem, whereas the lazy approach interleaves variable assignment and plan generation.

The early variable instantiation compilation, denoted as $\Pi_E = \langle D_E, P' \rangle$, with $D_E = \langle \mathcal{H}', \mathcal{C}', \mathcal{P}_E, \mathcal{A}_E \rangle$ and $P' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G} \rangle$, is defined as an extension of the lazy instantiation compilation. The only differences from the lazy compilation are the sets of predicates and actions, detailed below:

PREDICATES ($\mathcal{P}_E$). To effectively control the sequential progression between the three stages of the search, we introduce a set of 0-ary *control* predicates, denoted $\mathcal{P}_{\mathit{control}}$, which is formed by:

- ***application_phase***, indicating that the search is applying the previously selected variable assignments. During this phase, no new assignments can be made.
- ***planning_phase***, indicating that the search proceeds on the fully instantiated task. In this phase, neither variable assignments nor their application are allowed.

Thus, $\mathcal{P}_E = \mathcal{P}_L \cup \mathcal{P}_{\mathit{control}}$.

ACTIONS ($\mathcal{A}_E$). The set of actions is largely the same as in the lazy compilation ($\mathcal{A}_L$), but each action is extended by incorporating control atoms (using predicates from $\mathcal{P}_{\mathit{control}}$) into the preconditions and effects. We denote the resulting set of actions as $\mathcal{A}_E$.

- Each action from the original domain-specific actions $\mathcal{A}$ adds *planning_phase* to its effects, marking the transition to the planning phase once any domain action is executed.

- The apply actions require $\neg$*planning_phase* as a precondition and add *application_phase* to their effects, ensuring they can only execute during the application phase and preventing their use after domain-specific actions have begun.
- The assign action requires $\neg$*application_phase* as a precondition, ensuring that variable assignments cannot occur once the application phase has started.

This enforces a strict separation of the three stages during the search, following the order *variable assignment* $\rightarrow$ *assignment application* $\rightarrow$ *planning*. If there are constraints requiring some variables to take different values, the *check_inequality* action is introduced as a new stage, forcing the order: *variable assignment* $\rightarrow$ *inequality check* $\rightarrow$ *assignment application* $\rightarrow$ *planning*.

Since neither compilation imposes additional constraints on the variable instantiation, every valid assignment remains reachable, so both are complete: if there exists a valid assignment σ (Def. 2) for which the induced planning task $\overline{\Pi}[\sigma]$ is solvable, then Π_L or Π_E has a solution.

5 Experiments

The experimental evaluation examines three main aspects of the proposed approach: i) *coverage and overhead*, assessing the computational impact of the compilation with respect to planning task size and the number of lifted variables ii) *cost reduction*, evaluating the ability of the approach to optimize the initial state configuration, analyzing whether the resulting assignments produce lower-cost plans compared to fixed initial configurations; and iii) *practical applicability*, demonstrating that the approach is domain-independent and identifying the characteristics of domains for which it is beneficial.

Experimental Setting

Benchmarks. We selected four representative domains from the International Planning Competition (IPC) for evaluation: TRANSPORT, ROVERS, ELEVATORS and BLOCKS. This set allows us to illustrate the utility of our approach while covering a variety of planning scenarios common to many other domains: In TRANSPORT, vehicle locations are often flexible rather than fixed. Assigning trucks to cities with dense pickup demands can reduce long trips. In ROVERS, the initial distribution of rovers across waypoints and the position of the lander can reduce overall mission time. In ELEVATORS, having elevators initially waiting near passengers can reduce unnecessary movements. In BLOCKS, the initial configuration of blocks can be rearranged to better resemble the goal, minimizing the number of moves needed to reach it. For each domain, we consider the first 10 IPC problem instances that can be solved optimally, in order to compare the cost of the original planning task with the cost achieved after reconfiguring the initial state.

Table 1 identifies the parts of the initial state that we lifted, namely those that are configurable and therefore suitable for optimization. Lifting these variables yields problem instances with different numbers of free variables. Smaller instances contain fewer free variables (a problem with only two trucks has two lifted variables corresponding to their possible initial locations). We abbreviate the number of free variables in the initial state as $V = \mathit{vars}(\overline{\mathcal{I}})$, so that the table reports the minimum ($|V|_{min}$) and maximum ($|V|_{max}$) number of free variables that appear simultaneously in a problem instance within the set. This allows us to later report the maximum number of variables in the instances actually solved, indicating whether the approach can effectively handle problem instances that contain the largest number of lifted variables in the set, or only instances with fewer free variables.

Domain	Variables lifted	$\lvert V\rvert_{min}$	$\lvert V\rvert_{max}$
TRANSPORT	Position of trucks	2	3
ROVERS	Position of rovers and lander	2	5
ELEVATORS	Position of elevators	3	5
BLOCKS	Position of blocks	4	7

Table 1: Overview of lifted variables in the benchmark domains, and the minimum ($|V|_{min}$) and maximum ($|V|_{max}$) number of lifted variables present in the problem instances.

The benchmark shows that the compilations do not restrict either the number of lifted variables or the lifting of variables from multiple predicates. In ROVERS, the rover and lander positions are lifted, and in BLOCKS, the initial state is almost entirely lifted, except for *handempty*.

Approaches. We evaluate the two compilations, namely the lazy or minimum-commitment instantiation (Π_L) and the eager or early-commitment instantiation (Π_E). Both compilations achieve the same cost reduction and differ only in how they handle variable assignments.

Metrics. We evaluate coverage over the 10 original planning tasks (Cov.), and the maximum number of free variables in the initial state among the solved instances ($|V|_{max}$), compared to the values in Table 1. Cost reduction (Cost red.) is reported as the percentage decrease in the cost of the plan compared to using the original initial state, thus evaluating the ability of the approach to optimize the initial state configuration and obtain lower-cost plans. To measure the additional computational effort introduced by the compilations, we report the time overhead factor (Time overh.), defined as the ratio of the execution time for the compiled task Π_X to that of the original task Π, i.e., $\frac{T(\Pi_X)}{T(\Pi)}$.

Reproducibility. Since our aim is to find an optimal configuration for the initial state, we solve all the planning tasks using the `seq-opt-lmcut` configuration of Fast Downward, which runs A^* with the admissible `lmcut` heuristic to compute an optimal plan. Experiments were run on an Intel Core Ultra 7 255H CPU @ 5.10GHz with an 8GB memory limit and a time limit of 900s.

Results

Tables 2 and 3 report the results for the lazy and early compilations, respectively.

Coverage and overhead. As expected, solving the original planning task, which only optimizes plan cost and not

	Lazy compilation (Π_L)			
Domain	**Cov**	$\|V\|_{max}$	**Cost red.**	**Time overh.**
TRANSPORT (10)	4	2	24.1 ± 16.6	141.2 ± 48.2
ROVERS (10)	7	4	10.2 ± 12.5	37.4 ± 55.5
ELEVATORS (10)	0	0	-	-
BLOCKS (10)	8	6	76.7 ± 30.3	39292.0 ± 69642.0

Table 2: Results for Π_L. We report coverage (Cov.), maximum lifted variables solved ($|V|_{max}$), average cost reduction (Cost red., %) with std., and time overhead (Time overh.) with std.

	Early compilation (Π_E)			
Domain	**Cov**	$\|V\|_{max}$	**Cost red.**	**Time overh.**
TRANSPORT (10)	7	2	25.1 ± 14.8	22.4 ± 20.8
ROVERS (10)	7	4	10.2 ± 12.5	26.7 ± 28.4
ELEVATORS (10)	7	4	20.7 ± 9.6	27.3 ± 67.4
BLOCKS (10)	10	7	77.5 ± 26.8	2263.9 ± 4272.9

Table 3: Results for Π_E. We report coverage (Cov.), maximum lifted variables solved ($|V|_{max}$), average cost reduction (Cost red., %) with std., and time overhead (Time overh.) with std.

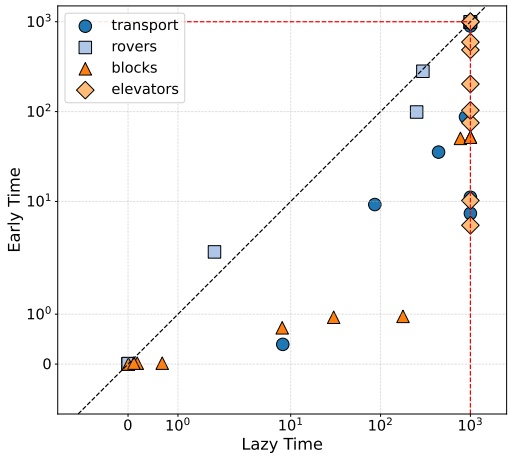

Figure 8: Runtimes for Π_L vs. Π_E. For instances below the diagonal, Π_E is faster; points on the red dashed line aat the right indicate instances solved only by Π_E.

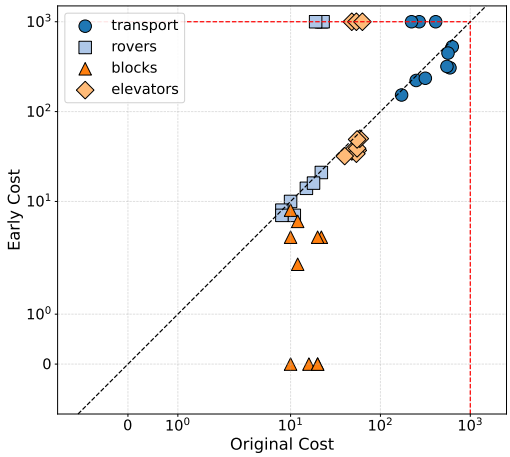

Figure 9: Plan costs for Π_E vs. Π (original task). Points below the diagonal indicate improvements by Π_E; points on the upper red dashed line are instances solved only by Π.

the configuration of a lifted initial state, is easier. For Π_L (Table 2), the time overhead is significant. In TRANSPORT, solving the compiled tasks takes on average 141.2 times longer than solving the original one. The high standard deviations are due to the increasing complexity of the problem instances: smaller instances are solved faster than larger ones. These longer times also affect coverage, with only 19 out of 40 compiled tasks being solved. Instances of ELEVATORS can be solved suboptimally or by using smaller problem instances outside the benchmark. However, due to the complexity of the problem and the number of free variables, none of the benchmark instances can be solved optimally. Regarding Π_E (Table 3), the time overhead remains considerable but is substantially lower than for the lazy compilation. This improvement is reflected in coverage, with 31 out of 40 tasks solved. These results indicate that performing variable assignment before plan search is beneficial. Figure 8 compares the solving times of the two compilations. The X-axis shows the times for Π_L (lazy), and the Y-axis shows the times for Π_E (early). Except for a single outlier in Π_L, all points lie above the diagonal, indicating lower times for Π_E. Instances solved by the early compilation but not by the lazy compilation are marked along the red dashed line on the right.

Comparing the maximum number of lifted variables in the problem instances that were successfully solved (column $|V|_{\max}$ from Tables 2 and 3) with the maximum number of lifted variables in Table 1, we observe that most domains only solve instances with fewer free variables. It is important to note that instances with more lifted variables are harder not only because they require more assignments, but also because they generally involve more objects, increasing the complexity of the underlying planning problem.

Cost reduction. Both compilations Π_L and Π_E achieve identical cost reductions on commonly solved instances, but their averages differ in Tables 2 and 3 because of coverage differences. We focus on Π_E, which surpasses the lazy compilation in coverage. In TRANSPORT and ELEVATORS, reconfiguring the initial state by lifting the variables specified in Table 1 yields cost reductions of about 20% and 25% on average, respectively. These substantial savings are partly explained by the presence of movement-dependent action costs in these domains: when the cost of moving trucks or elevators is high, eliminating even a few movements can lead to a significant reduction in total plan cost. By contrast, ROVERS relies on unitary action costs and generally produces shorter plans, which limits the potential for large savings. Still, it achieves an average cost reduction of about 10%. A special case is BLOCKS. For most IPC instances, the goal is to build a specific tower of blocks. The task is to rearrange the blocks to minimize the number of moves. The lifted initial states preserve the overall structure (e.g.,

how many blocks are on the table or in towers), but the specific block identities are left unassigned. If the initial structure matches the goal structure, assigning blocks immediately satisfies it, achieving a cost reduction of 100%.

Figure 9 compares, for each task, the cost of the plan obtained from the reconfigured initial state generated by Π_E (X-axis) with the cost of the plan obtained from the original task (Y-axis). Note that the plot uses a logarithmic scale to accommodate the wide range of plan costs in the benchmark. The compilation behaves as expected: the resulting plan cost is never higher than that of the original task, lying below the diagonal in 29 out of 40 cases. There are 9 cases in which the original task can be solved but the early compilation cannot (placed in the upper red line), and 2 cases in ROVERS where both approaches yield the same cost because the initial state is already optimally configured. Notable examples appear in TRANSPORT: one task's plan cost decreases from 594 to 307 after reconfiguration; in ELEVATORS, another decreases from 54 to 34. Triangular points at $Y = 0$ correspond to BLOCKS instances where the rearrangement already matches the goal, so the plan cost is 0.

Practical applicability. We show that, in general, the approach is useful for domains where the problem definition includes an initial state with flexibility in how resources are configured. Domains with multiple agents and spatial placements are particularly well-suited, as small differences in the initial configuration can have a significant impact on plan quality. This approach can also be useful in domains where there is a set of interchangeable objects, and it does not matter which ones are used. For example, in the BARMAN domain, a robot must prepare cocktails using clean glasses. Our approach allows flexible assignments from the set of glasses without committing to specific ones. However, the overhead of managing variable assignments might outweigh potential gains. We excluded IPC planning domains from the benchmark when the initial state lacks flexibility. For instance, in CHILDSNACK the number of sandwiches to make and serve is fixed, so there is nothing to configure. Similarly, in domains where configurable elements are directly tied to the goals, such as SOKOBAN, where the positions of the rocks cannot be altered, the approach does not bring any benefit.

We also identified domains that could, in principle, be suitable for the benchmark due to having configurable initial states. For instance, in HIKING the locations of people, tents, and camps are configurable. However, in all available problem instances, these elements are placed in the same initial locations. This lack of variability makes the domain unsuitable for our evaluation, as no improvements or meaningful comparisons across problem instances would be possible.

6 Conclusions

We introduced a novel problem that considers lifted initial states in classical planning. We addressed this problem through a compilation to classical planning that enables the planner to determine variable assignments. By integrating assignment selection with plan generation, the approach optimizes the lifted parts of the initial state and yields lower-cost plans. We propose two compilation strategies for variable assignment: a minimum-commitment approach, which can defer assignments in the search, and an early-commitment approach, which forces variables to be assigned at the start. Our evaluation shows that both strategies are effective, achieving significant plan cost reductions by exploiting reconfigured initial states. They differ in time overhead and scalability, with the early-commitment performing better overall. The approach is domain-independent and especially beneficial in domains where initial states include configurable resources.

References

Corrêa, A. B.; Giacomo, G. D.; Helmert, M.; and Rubin, S. 2024. Planning with Object Creation. In Bernardini, S.; and Muise, C., eds., *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2024, Banff, Alberta, Canada, June 1-6, 2024*, 104–113. AAAI Press.

Fikes, R.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.*, 2(3/4): 189–208.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proceedings of ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 81–88. AAAI.

Gragera, A.; Fuentetaja, R.; García-Olaya, Á.; and Fernández, F. 2023. A Planning Approach to Repair Domains with Incomplete Action Effects. In Koenig, S.; Stern, R.; and Vallati, M., eds., *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, July 8-13, 2023, Prague, Czech Republic*, 153–161. AAAI Press.

Grundke, C.; Röger, G.; and Helmert, M. 2024. Formal Representations of Classical Planning Domains. In Bernardini, S.; and Muise, C., eds., *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2024, Banff, Alberta, Canada, June 1-6, 2024*, 239–248. AAAI Press.

Haslum, P.; and Corrêa, A. B. 2024. The Universal PDDL Domain. *CoRR*, abs/2411.08040.

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

Hoffmann, J.; and Brafman, R. I. 2006. Conformant planning via heuristic forward search: A new approach. *Artif. Intell.*, 170(6-7): 507–541.

Lewis, D. 1973. Counterfactuals and comparative possibility. *J. Philos. Log.*, 2(4): 418–446.

Lindsay, A. 2023. On Using Action Inheritance and Modularity in PDDL Domain Modelling. In Koenig, S.; Stern, R.; and Vallati, M., eds., *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, Prague, Czech Republic, July 8-13, 2023*, 259–267. AAAI Press.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a

Notion of Quality. In *Proceedings of K-CAP 2017, Austin, TX, USA, December 4-6, 2017*, 14:1–14:8. ACM.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL–the planning domain definition language–version 1.2. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational ....

Palacios, H.; and Geffner, H. 2007. From Conformant into Classical Planning: Efficient Translations that May Be Complete Too. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 264–271. AAAI.

Pednault, E. P. D. 1989. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of KR 1989. Toronto, Canada, May 15-18 1989*, 324–332. Morgan Kaufmann.

Smith, D. E.; and Weld, D. S. 1998. Conformant Graphplan. In Mostow, J.; and Rich, C., eds., *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, 889–896. AAAI Press / The MIT Press.

Sreedharan, S.; Srivastava, S.; Smith, D. E.; and Kambhampati, S. 2019. Why Can't You Do That HAL? Explaining Unsolvability of Planning Tasks. In *Proceedings of IJCAI 2019, Macao, China, August 10-16, 2019*, 1422–1430. ijcai.org.