Revisiting Landmarks: How To Learn from Previous Plans to Generalize over Problem Instances

Issa Hanou, Sebastijan Dumančić, Mathijs de Weerdt

Delft University of Technology, The Netherlands
i.k.hanou@tudelft.nl

Abstract

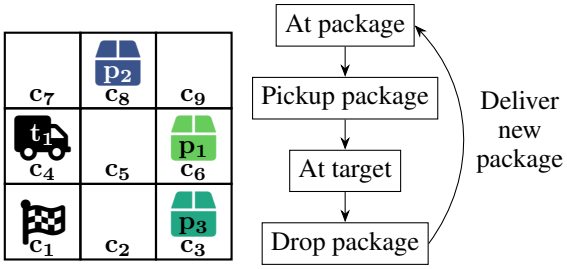
Landmarks have greatly improved AI planning by identifying the must-reach states for any plan that solves an instance. However, as landmarks are ground atoms, they are instance- and object-specific, and thus do not capture any general knowledge about the problem domain. Moreover, landmark graphs often contain repetitive chains of landmarks, thereby increasing graph size. We propose generalized landmarks that describe general intermediate goals for an entire problem domain. Generalized landmarks extend beyond the predicates of a domain by using *state functions*, which are independent of a specific object and capture repetition. A directed generalized landmark graph is constructed that includes loops to identify repetitive subplans. We give examples of hand-constructed graphs and discuss our approach to learning them from a small set of solved instances. To use generalized landmarks for planning, we present a simple counting heuristic and show how it exploits loop behavior and local landmark guides to find plans more efficiently.

Code & Data — https://doi.org/10.5281/zenodo.20591819
Extended version — https://arxiv.org/abs/2508.21564

Introduction

Real-world planning problems often have complicated structures and dependencies on subgoals to reach the final goal, making plans difficult to interpret and symmetries hard to recognize. While many problems have inherent symmetries on partial plans that are repeated with different objects, this knowledge is currently unused in planners. Prior work introduced subgoal hierarchies to draft policy for generalized planning (Drexler, Seipp, and Geffner 2024) and landmarks to identify subgoals in an instance (Porteous, Sebastia, and Hoffmann 2001). However, we propose a heuristic for classical planning that searches over states rather than state transitions and generalizes from the original landmarks.

Landmarks were introduced by Porteous, Sebastia, and Hoffmann (2001) as facts that must hold at some point in every plan that solves that planning problem. These landmarks have been used in heuristics, such as counting the ones achieved in previous states, which have greatly improved the planning approaches for many problems (Richter, Helmert, and Westphal 2008; Karpas and Domshlak 2009). However, landmarks have always been computed as ground atoms of an instance, resulting in two main limitations: i) they only apply to one problem instance and thus have to be computed individually, and ii) they are dependent on the specific objects of that instance, and thus do not capture repetitive subplans across objects.



(a) `Delivery` instance with three packages. (b) Generalized landmarks for `Delivery` domain.

Figure 1: Example of generalized landmarks. (a) shows a `Delivery` problem instance where the truck has to deliver three packages to the target cell and (b) gives a manually constructed landmark graph for the `Delivery` domain.

We address this lack of generalization by introducing *generalized landmarks*, and refer to the original ground-atom landmarks as *traditional landmarks* for clarity. Instead of ground atoms, generalized landmarks are composed of first-order functions that express a Boolean relation over the objects in a domain, which are evaluated for a given state. These state functions allow us to represent the notion of any object rather than a specific instantiation. This allows us to generalize over all objects at once, enabling generalized landmarks to hold for any number of objects within an instance, as well as across objects between instances.

Figure 1 shows an example of generalized landmarks, for the `Delivery` domain, restricted to one truck with capacity one.[1] For each package, the same four steps must be repeated, and when the truck is restricted to only carry one package at a time, these steps must be done sequentially for

[1] https://github.com/rleap-project/dlplan/blob/main/benchmarks/delivery/domain.pddl

each package. The first-order functions that constitute the generalized landmarks are functions that take a state s and return a truth value. For example, d_1 returns `true` if there is an empty truck, d_2 returns `true` if there is a cell with only trucks at that cell and no packages, d_3 returns `true` if a truck is at the goal cell, and d_4 returns `true` if there is an empty truck at a goal cell. In the `Delivery` domain, a goal cell is a cell where a package should be delivered.

The generalized landmark graph shows a loop with a condition: Deliver new package. This condition also takes the form of a first-order function, and when the condition does not hold, the next landmark must be achieved, or the goal is reached. In the `Delivery` example, the loop condition is a function that compares each package's goal location with its current location and counts the differences. If this function returns 0, the loop is complete, and the goal is reached; if it is bigger than 0, the loop can be traversed.

We propose a generalized landmark counting heuristic that can be used in any planner setup. The heuristic employs information from the landmarks themselves and from the loop condition. For each loop, we use this condition to compute the number of loops for an instance in the initial state. This allows us to quickly determine the total number of landmarks to accept, and we use it to efficiently prune the search space. Generalized landmarks form an abstract plan that serves as a long-horizon guide to the goal. Additionally, we can provide a state function for each landmark that serves as an inherent local heuristic to guide the short-horizon search to the next landmark. We construct generalized landmark graphs by hand for three domains of varying complexity and demonstrate that we can solve difficult instances of these domains very efficiently.

This paper presents four contributions. First, we define generalized landmarks that generalize over problem instances in a domain. Second, we present a heuristic for using generalized landmarks in planning applications to identify the problem's subgoals. Third, we show how exploiting loop conditions and local guide heuristics enables more efficient plan finding. Finally, while we discuss several manually constructed generalized landmark graphs, we also outline the basic idea for a discovery algorithm that learns the graph from just a few small solved instances. Generalized landmarks provide four advantages over traditional ones: i) they generalize over all instances in a domain, ii) they only have to be computed once, iii) they are learned from small instances and scale to larger instances, and iv) they capture more general relations and repetition in a domain.

The Graph of Generalized Landmarks

This section formally introduces the concept of *generalized landmarks* and the associated concepts to create a graph of generalized landmarks. First, we introduce the scope and then define state functions and the components of the graph.

Generalized landmarks apply to an entire domain, given that the instances share a domain goal. We build on the work on domain goals and take the domain as annotated with a domain goal, and the instances as generated according to this domain goal (Grundke, Helmert, and Röger 2025a). The goal predicates $\mathcal{P}^{\mathcal{G}}$ introduced in the domain goal (Grundke, Röger, and Helmert 2024) are also included in the domain formulation. For example, in `Delivery`, we have predicates $\mathcal{P} = \{at, carrying, empty, adjacent\}$. The problem instances considered in this domain require packages to be at at a certain location, and define the goal as a conjunction of literals in the form $at(pk, loc)$. This is translated using the goal predicates $\mathcal{P}^{\mathcal{G}} = \{at^g\}$ such that the domain goal is $\forall(pk, loc) : at(pk, loc) \rightarrow at^g(pk, loc)$.

The generalized landmark graph in Figure 1b uses different kinds of state functions to define the conditions. Each of the generalized landmarks (Definition 2) is defined by a conjunction of *state descriptors* (Definition 1).

Definition 1 (State Descriptor)**.** *A **state descriptor** d is a function that takes a state as input and returns a truth value. For a state s and a state descriptor d, we use $s \vDash d$ to indicate that the descriptor returns `true` in this state.*

Definition 2 (Generalized Landmark)**.** *For a domain $\mathcal{D}$, a **generalized landmark** L is a set of state descriptors such that in the state trajectory $\mathbf{t}$ of a plan that solves an instance of $\mathcal{D}$, there is at least one state $s \in \mathbf{t}$ such that L holds. A generalized landmark holds if every descriptor $d \in L$ returns `true` in that state, which we indicate with $s \vDash L$.*

Example 1. *For the `Delivery` example in Figure 1b, we can formally define the state descriptors described in the introduction (see Appendix A). However, they can also be formulated more precisely to capture the full extent of the landmarks. For example, we could define the* At Package *generalized landmark with d_1 returning `true` when an empty truck is at a cell where there is a package, and it is not its goal predicate.* Pickup package *is achieved with d_2 returning `true` if a truck is carrying a package which is not at the goal. Descriptor d_3 achieves* At target *when returning `true` if the truck carries a package and is at the goal location of that package. Finally, the* Drop package *landmark is defined by d_4 that returns `true` when there is an empty truck at a cell where there is also a package for which it is the goal cell. For the exact formulation, see Appendix A.*

Loops of Generalized Landmarks

To capture repetitive subplans, we explicitly define loops of generalized landmarks. While a graph of generalized landmarks can be constructed without loops, loops are more efficient in scaling to instances with more objects. However, we must ensure proper landmark progression when traversing a loop. We thus need to know exactly when the loop can be traversed, and when to continue achieving the next landmarks. Moreover, we need to prevent 'duplication' of landmarks. For example, in `Delivery`, the truck that just delivered a package can pick it up again, move away, return to the goal cell, and 'redeliver' the same package. These challenges are addressed by introducing *loop conditions*, which are enforced on the last landmark of the loop, which we refer to as the *loop landmark*.

Consider the example graph in Figure 1b, we have a loop between L_4 (Drop package) and L_1 (At package), and the node with the outgoing edge to a previously-ordered landmark is thus the loop landmark (Drop package). The condition on the edge is Deliver new package, to prevent dupli-

cated behavior, and when no *new* package can be delivered, the loop traversal is complete. Then, the next landmark can be traversed, or in this case, the goal is achieved.

We define a *state progression condition* on the edge that connects to an earlier landmark, which guarantees that each loop completion achieves a different subgoal. On each loop traversal, this condition ensures that some numerical value changes (e.g., the number of delivered packages). Moreover, we use an *exit condition* on the edge connecting to the next ordered landmark to determine when the loop can no longer be traversed (e.g., all packages are delivered). If this condition holds, then the next landmark should be achieved, or the goal is reached.

More formally, for a loop $\ell = (L_i, L_j)$ from generalized landmark node v_i to v_j, the *loop condition* $\mathcal{L}_\ell$ (Definition 4) is a pair that gives the exit condition of the loop (when to continue in the landmark chain) and state progression condition (comparing the loop landmark's state assigned value after the first traversal). To define the loop condition, we use state descriptors and state progressors (introduced in Definition 3). An example can later be found in Example 2.

Definition 3 (State progressor)**.** *A **state progressor** ρ is a function that takes two states as input and returns a truth value. For states s and s' and a state progressor ρ, we use $s, s' \vDash \rho$ to indicate that the progressor returns* `true`*, meaning that state s' is a valid progression after state s, according to the state progressor ρ.*

Definition 4 (Loop condition)**.** *For a loop $\ell = (L_i, L_j)$, the **loop condition** $\mathcal{L}_\ell = (\mathcal{L}^{\text{exit}}, \mathcal{L}^{\text{progress}})$ specifies when the loop must be exited ($\mathcal{L}^{\text{exit}}$) and when a loop traversal is valid ($\mathcal{L}^{\text{progress}}$), where*

i) $\mathcal{L}^{\text{exit}}$ is a set of state descriptors such that if state s accepts loop landmark L_i ($s \vDash L_i$) and each $d \in \mathcal{L}^{\text{exit}}$ holds ($s \vDash \mathcal{L}^{\text{exit}}$), the loop from must be exited; and

ii) $\mathcal{L}^{\text{progress}}$ is a set of state progressors such that when visiting a state s that accepts the loop landmark L_i and a later state s' that accepts the loop landmark L_i again, each $\rho \in \mathcal{L}^{\text{progress}}$ must return true ($s, s' \vDash \mathcal{L}^{\text{progress}}$).

Finally, to efficiently use generalized landmarks in a counting heuristic, the number of times a loop must be traversed has to be known for each instance. We introduce a *state value* that is evaluated in the initial state and computes exactly the number of loop traversals by indicating how often a loop landmark must be reached. This also allows some loops to be skipped altogether for a specific instance. For each loop ℓ, we thus have a loop landmark counter (Definition 6) that defines how many times a loop landmark can be achieved, which is a set of state values (Definition 5).

Definition 5 (State value)**.** *A **state value** γ is a function that takes a state and returns a nonnegative integer value.*

Definition 6 (Loop landmark counter)**.** *For a loop $\ell = (L_i, L_j)$, the **loop landmark counter** $\mathcal{C}_\ell$ is a set of state values that identify in the initial state the number of times the loop landmark L_i can be achieved, such that each $\gamma \in \mathcal{C}_\ell$ returns this nonnegative number.*

Now, we can formally define the loop condition in our running example (Example 2). All the introduced functions are formalized in Appendix A.

Example 2. *In the landmark graph in Figure 1b, we have the loop $\ell = (\text{Drop package}, \text{At package})$. We define the loop condition $\mathcal{L}_\ell$ using a state descriptor d_5 which holds if all packages are at their goal location; a state progressor ρ_1 which holds if more packages are at their goal location in the latter state than in the former state; and a state value γ_1 which counts the number of packages that are not at their goal location. When no packages are left to deliver, the exit condition is* `true`*, so we have $\mathcal{L}^{\text{exit}} = \{d_5\}$. A loop can be traversed whenever a package is delivered, so we have the state progression condition $\mathcal{L}^{\text{progress}} = \{\rho_1\}$. Finally, the state value γ_1 returns in the initial state the total number of packages to be delivered, which is the number of times that the loop landmark 'Drop package' can be achieved, so $\mathcal{C}_\ell = \{\gamma_1\}$. So, we know right away that the loop can only be traversed three times in the instance in Figure 1a.*

Next, we define the graph of generalized landmarks (Definition 7). This compact and interpretable representation illustrates the repetition and generalization of plans for problems in this domain.

Definition 7 (Graph of Generalized Landmarks)**.** *For a domain $\mathcal{D}$, a **graph of generalized landmarks** $G_\mathcal{D} = (V, E^{\text{O}}, E^{\text{L}}, \mathcal{L}, \mathcal{C})$ has a set of generalized landmark nodes V, a set of ordered edges E^{O}, and a set of loop edges E^{L}. Each loop edge $e = (L_i, L_j) \in E^{\text{L}}$ is associated with a loop $\ell = (L_i, L_j)$ with loop condition $\mathcal{L}_\ell = (\mathcal{L}^{\text{exit}}, \mathcal{L}^{\text{progress}}) \in \mathcal{L}$ and a loop landmark counter $\mathcal{C}_\ell \in \mathcal{C}$.*

This allows us to also formalize the loop landmarks.

Definition 8 (Loop landmark)**.** *Given a graph of generalized landmarks $G_\mathcal{D} = (V, E^{\text{O}}, E^{\text{L}}, \mathcal{L}, \mathcal{C})$, a loop landmark is a landmark $L_i \in V$ which has at least one outgoing edge to some landmark $L_j \in V$ such that $j < i$ and $e = (L_i, L_j) \in E^{\text{L}}$, and either has another edge the next landmark L_{i+1} or it is the last landmark in the chain (i.e., $i = |V|$).*

Our framework introduces a set of state descriptors $\mathcal{S}^{\text{d}}$, a set of state progressors $\mathcal{S}^\rho$, and a set of state values $\mathcal{S}^\gamma$, which together we refer to as the set of *state functions* $\mathcal{S} = \{\mathcal{S}^{\text{d}}, \mathcal{S}^\rho, \mathcal{S}^\gamma\}$. We provide the formal representation of the state functions used as examples throughout this section in Appendix A.

Local Heuristic: Guide the Landmarks

Generalized landmarks and the use of state functions provide one final advantage. While generalized landmarks provide a long-horizon heuristic, keeping track of all subgoals in the problem instance, we also allow the use of short-horizon heuristics, in the form of *landmark guides*. For example, in `Delivery`, we know from Figure 1 that the first subgoal is for the truck to reach a cell with a package (that is not delivered). However, if our grid is not just 3x3 but 10x10, this information provides little guidance. So, we provide the option of a *guided graph of generalized landmarks* (Definition 9) that uses a set of state values to provide a local

heuristic for each generalized landmark. In a state where the landmark is accepted, the value should be 0; in one state before this landmark state, the value should be 1, and so on, such that the value decreases (or at least never increases) as the state is closer to accepting the landmark.

Definition 9 (Guided Graph of Generalized Landmarks)**.** *For a domain $\mathcal{D}$, a **guided graph of generalized landmarks** $G_{\mathcal{D}} = (V, E^{\mathrm{O}}, E^{\mathrm{L}}, \mathcal{L}, \mathcal{C}, \Lambda)$ has a set of guiding state values Λ, such that for each generalized landmark $L_i \in V$ we have exactly one guide $\lambda_i \in \Lambda$ such that in each state s that accepts landmark L_i, we have $\lambda_i(s) = 0$ and for each state s' that is reachable in δ actions to achieving L, we have $\lambda_i(s') \leq \delta$.*

Example 3. *In our example in Figure 1b, the guide λ_1 to the first landmark* $\mathrm{At}\,\mathtt{package}$ *is the distance between the truck and the nearest undelivered package, using the* `adjacent` *predicate in the domain, and comparing package locations with their goal locations to determine whether it is delivered. Similarly, the guide λ_3 to the third landmark* $\mathrm{At}\,\mathtt{target}$ *uses the* `adjacent` *predicate and the goal location of the package being carried. For the second landmark, the guide λ_2 simply counts the* `empty` *atoms in the current state, which is 0 if a package was picked up. Contrary, the guide λ_4 of the last landmark counts the* `carrying` *atoms.*

Searching with Generalized Landmarks

To use generalized landmarks in planning, we design a heuristic that counts the accepted landmarks, which we name the *generalized landmark counting heuristic* LM^G. Similarly to Richter, Helmert, and Westphal (2008), we estimate the distance from state s to the goal by computing the number of landmarks that still need to be achieved from state s onward. Algorithm 1 gives the pseudocode for the LM^G heuristic. Before the search, we calculate the number of times each landmark must be achieved using the loop counter $\mathcal{C}_\ell$. Given a graph of generalized landmarks $G_{\mathcal{D}} = (V, E^{\mathrm{O}}, E^{\mathrm{L}}, \mathcal{L}, \mathcal{C})$ and a specific instance, we evaluate the state values γ of the landmark counter $\mathcal{C}_\ell$ on the initial state, which returns a nonnegative number (Definition 5-6). Using this number, we compute the maximum value of LM^G, which we denote by m, by rolling out the loops and counting the total number of landmarks to be achieved in this instance (Equation 1). If a loop is skipped, fewer landmarks have to be achieved in total.

$$
\begin{aligned}
m \leftarrow |V| + & \sum_{\substack{\forall \ell = (L_i, L_j) \in E^{\mathrm{L}} \\ \forall \gamma \in \mathcal{C}_\ell \\ \gamma(initState) > 0}} \gamma(initState) \cdot (i - j + 1) \\
& - \sum_{\substack{\forall \ell = (L_i, L_j) \in E^{\mathrm{L}} \\ \forall \gamma \in \mathcal{C}_\ell \\ \gamma(initState) = 0}} (i - j + 1)
\end{aligned}
\tag{1}
$$

During the search, we keep track of the next generalized landmark to achieve, as well as the total number of accepted landmarks. This reference is updated as landmarks are accepted, and loops are traversed (line 5). When evaluating a potential next state, we check if the next generalized landmark to be achieved is accepted in this state (line 6). We immediately check if this landmark is a loop landmark, and if so, we check the loop exit condition. If this does not hold yet, we update the reference to the next landmark to be achieved (first in the loop), while storing the value of the state progression condition in the current loop landmark (line 10-11). This way, we guide the planner to re-achieve the ordered landmarks in the loop while keeping track of the total number of loop traversals. If this is not the first loop landmark encountered and the exit condition does not hold, the state progression condition must hold for this loop landmark to be accepted; otherwise, the landmark is not (yet) accepted, and the loop is not traversed. Finally, we also check how many of the state descriptors of the next landmark are already accepted in the current state, which acts as a short guide (value c) to the complete landmarks (line 19). Our total heuristic value is the precomputed maximum value minus the currently accepted landmarks, plus the value c.

In general, we restrict landmark progression to accept only one landmark per state. However, when a state s accepts the loop landmark, satisfies the state progression condition, and the guide to the next landmark to achieve is already 0 in this state, we also immediately accept that first landmark (line 12-13), but no further landmarks are achieved. For example, in `Delivery` if a package is delivered at a cell with another undelivered package, we immediately progress in the landmark chain. By allowing this exception in the heuristic, we can handle fully defined landmark graphs and show all intermediate steps of the abstract plan, where sometimes two landmarks are accepted in a single state.

After checking whether the next landmark is accepted (and a possible loop is taken), the guide value h_λ (if used) is computed for the next landmark to be accepted (line 20). This value is used in a lexicographical evaluation combined with the value h_{LM} computed for LM^G. The priority queue in A^*'s `expand` function is based on the default priority defined by $(f,\ h,\ |searchTree|)$, where f is the weighted path cost plus weighted heuristic value. Instead, we prioritize on the heuristic value h_{LM}, then on the path cost (f without the h value), and add an extra parameter h_λ: $(h_{LM},\ f,\ h_\lambda,\ |searchTree|)$. Finally, the priority queue is emptied when a loop is taken or exited.

As we use information from previously accepted landmarks, our heuristic depends on the previous state and is thus path-dependent. However, we do not actually require information about the previous state, only access to the landmarks accepted in that state. In the initial state, only the first landmark is checked, though no loops can be taken here yet.

Evaluation

To demonstrate the use of generalized landmarks in a planner, we implemented the LM^G heuristic in SymbolicPlanners.jl (Zhi-Xuan 2022), which enabled easy heuristic implementation, although we recognize that this is not a state-of-the-art planner. The state functions used to construct the generalized landmarks, their guides, and the loop conditions are all functions over a state that return either a truth or

Algorithm 1: Pseudocode for the *generalized landmark counting heuristic* LM^G.

Initialize the `landmarkCounter`, loop counters, the pointer `nextLM` and the empty `visited` array;
Calculate the maximum value m of the heuristic based on the landmark graph $G_{\mathcal{D}}$ and the current task T;
Function `ComputeHeuristic`($state$, $prevState$, $G_{\mathcal{D}} = (V, E^{\mathrm{O}}, E^{\mathrm{L}}, \mathcal{L}, \mathcal{C})$, $T \in \mathcal{D}$)**:**
  if $state \in visited$ **then return** $m -$ `landmarkCounter`$[state]$;
  Copy counters, the pointer `nextLM` and `visited` list from $prevState$ and mark $state$ as `visited`;
  if $state \vDash L_{nextLM}$ **then**
    if *If a loop* $\ell = (L_{nextLM}, L_{loopStart})$ *exists from* L_{nextLM} *to a previously-ordered landmark* $L_{loopStart}$ **then**
      if *The exit condition* $\mathcal{L}^{\text{exit}}$ *of the loop does* ***not*** *hold in* $state$ **then**
        if *The state progression condition* $\mathcal{L}^{\text{progress}}$ *holds (if applicable)* **then**
          Accept landmark $L_{\texttt{nextLM}}$ in $state$ and update loop traversal counter;
          `nextLM` $\leftarrow$ `loopStart`;
          if $\lambda_{nextLM}(state) = 0$ **then**
            `nextLM` $\leftarrow +1$;
        else
          `nextLM` $\leftarrow +1$;
      else
        Accept landmark $L_{\texttt{nextLM}}$ in $state$;
        `nextLM` $\leftarrow +1$;
      $c \leftarrow |L_{\texttt{nextLM}}| - \big|\{d \in L_{\texttt{nextLM}} \mid state \vDash d\}\big|$;
      Compute guide value $\lambda_{\texttt{nextLM}}$ on $state$;
  return $\Big(m + c -$ `landmarkCounter`$[state],\ \ \lambda_{\texttt{nextLM}}(state)\Big)$;

a numeric value. While the implementation of these functions is not restricted to any form, we use the features computed by DLplan (Drexler, Francès, and Seipp 2022), though they can be substituted with a different form, as long as an evaluator is supplied to evaluate the functions on a specific state. We manually constructed the (guided) graph of generalized landmarks for two different domains from the IPC benchmark with domain goals (Grundke, Helmert, and Röger 2025b), as well as the `Delivery` domain used as an example throughout this paper, which was taken from the DLplan library set of benchmarks. We use these manual graphs to show the benefits of generalized landmarks and the efficiency of our heuristic. Future work is learning these graphs automatically, more on this in the next section. First, we describe the domains and landmark graphs; then, we present our setup and results. Our goal is to show that generalized landmarks outperform a well-known and generally good domain-independent heuristic h_{add}.

Delivery (D) The `Delivery` domain has three actions: $move$, $pickPackage$, and $dropPackage$, the object types $cell$ and $locatable$, with subtypes $truck, package$ of $locatable$, four predicates: $empty$, $carrying$, at, and $adjacent$, and goal predicate at^g. The domain goal is formulated as $\forall(pk, loc) : at(pk, loc) \rightarrow at^g(pk, loc)$. The domain provides difficulty in terms of the number of packages, their distribution over the grid, and the size of the grid. The generalized landmark graph was already described in this paper and is given in full in Appendix A.

Ferry (F) The `Ferry` domain is similar to `Delivery` as it has three actions $sail, board$, and $debark$, and should pick up cars and sail them to their destination: $\forall(car, loc) : at(car, loc) \rightarrow at^g(car, loc)$. However, the ferry object is not explicitly included like the truck, and the locations are not connected via an $adjacent$ predicate. While the packages are distributed over a large grid, many cars are often waiting at the same location. We construct a graph with four landmarks: go to a car that is not at its goal, pick up the car, sail the car to its goal location, and debark. We use a guide in this landmark graph to allow direct loop continuation: after dropping off a car at some location, thereby achieving the fourth landmark, the guide to the first landmark is used, and when evaluated to 0, it allows immediate acceptance of the first landmark again, to pick up a car waiting at this location.

Childsnack (C) The `ChildSnack` domain has six actions: $moveTray$, $serveSandwich$, $serveSandwichNoGluten$, $makeSandwich$, $makeSandwichNoGluten$, and $putOnTray$, and the goal is to serve all children (distributed over several tables) a sandwich, taking into account whether they are allergic to gluten or not. The domain goal is $\forall(child) : serve^g(child)$. We define eight landmarks with two sequential loops. First, we create a gluten-free sandwich, put it on a tray, move to a table with a child who is allergic to gluten and has not received a sandwich yet, and serve the child the sandwich. This is the first loop, which is repeated until all children allergic to gluten have been served. The number of loop landmarks depends on the number of allergic children, and if this is 0, the loop (and all contained landmarks) is skipped altogether. Then we create a sandwich for all children who are not allergic and serve them, which is a loop that depends on the number of non-allergic children. If an instance does not have any (non) allergic children, then that loop and associated landmarks are skipped. While this

Table 1: Number of solved instances for the different heuristic configurations and average runtime over instances that were solved by all heuristic settings.

	Solved			Avg Runtime (s)		
	h_{add}	LM^G	LM^G_λ	h_{add}	LM^G	LM^G_λ
C (30)	4	**30**	-	453.73	**11.69**	-
D (30)	29	**30**	**30**	104.58	**11.32**	12.88
F (57)	49	39	**57**	41.98	22.38	**15**

may result in suboptimal plans, as some sandwiches can be served together on the limited number of trays, the domain is difficult due to the number of consecutive, interdependent steps, and these generalized landmarks provide an efficient abstract plan. We do not use guides in this landmark graph.

Experimental Setup

We used the A^*-planner from SymbolicPlanners.jl and used the h_{add} heuristic as a baseline. For the `Delivery` domain, we used the thirty instances provided by the DLplan benchmark. For the other domains, we used the provided test set for evaluating our heuristic; the number of available instances is reported in Table 1. The experiments were run on the DelftBlue supercomputer (Delft High Performance Computing Centre DHPC), where each heuristic tested each instance of the different domains with 8GB of memory and a time limit of 30 minutes per instance per heuristic.

Results

Table 1 shows the number of solved instances per domain for both h_{add} and our heuristic, where we also compare performance with (LM^G_λ) and without (LM^G) the landmark guides (where applicable). The table also reports the average runtime to solve the instances, averaged over the number of instances solved by that heuristic. We see that generalized landmarks clearly outperform the baseline heuristic of h_{add} both in terms of instances solved as well as runtime. For `Delivery`, the landmark graph without guides already solves all instances, and the guides increase the runtime (as more computations must be made).

For `Ferry`, we do see a difference: without the guide, the condition from Algorithm 1 line 12 does not hold, so after dropping a car, the next car cannot be picked up immediately, which is where the guide is very helpful. However, if we add a redundant action to the `Ferry` domain, which takes as precondition: $emptyFerry$, $at(car1, loc)$, $at(car2, loc)$, and $atFerry(loc)$, with the single effect to set a new predicate $delivered(car1)$ to `true` (which is never used in a precondition), this action can be used to find plans more efficiently without the guide. In this case, only one instance is solved by LM^G_λ and not by LM^G (Appendix B).

Figure 2 shows the expanded states and plan length for the successful instances for the different domains. Again, we clearly see that our heuristic LM^G outperforms the h_{add} baseline. For the `Delivery` domain, the guiding of the heuristic toward each landmark reduces the number of expanded states, which in some instances has a big influence. This is because the packages are distributed across a large grid, so a local heuristic to guide the planner toward the next landmark can greatly improve performance. We note that the generalized landmark graphs are not necessarily optimal, so while no major differences in the plan length are visible, it might happen that the plan found by h_{add} is slightly shorter (generally not more than roughly 5% of the plan length).

Learning Generalized Landmarks

Graphs of generalized landmarks can be constructed manually by reasoning about a domain, but we also want to generate these graphs automatically. We propose a method to learn them from a small set of solved instances. The training instances are treated as state trajectories based on the provided plan, allowing us to infer the goal and how to reach it. The benefit of state trajectories is that we can use the states as training data to generate state functions that serve as generalized landmarks, their guides, or the loop conditions. Moreover, landmarks are must-reach states, so we want to define them over states, not actions. This also provides a smaller search space than when using state transitions, such as in policy sketches (Drexler, Seipp, and Geffner 2024).

The training trajectories form the state space, and we generate a set of state functions for this state space using the DLplan library (Drexler, Francès, and Seipp 2022). Then, each state in the training trajectories is evaluated using each state function in the available set. While each state function type is evaluated differently, they can share the same underlying function. For example, say we have a function returning all the packages in `Delivery` that have some at^g goal location defined but are currently not at that location. We create a state descriptor indicating whether the set is empty, a state progressor comparing the set sizes across two states, and a state value returning the exact number of packages.

The main idea for our learning algorithm is that we identify the same states across trajectories using a subset of state functions. We use an iterative process to find the earliest next state in each trajectory for which we have a set of state descriptors that identify it. This is constrained such that i) the state directly previous to this identified state must not satisfy this state descriptor, ii) it must be strictly later than the previously found state, and iii) each trajectory must have at least one such state. For each landmark, we also identify a guide feature, which must be 0 in the state accepting this landmark, 1 in the state directly previous, and increase in the state further away from this landmark state.

In each iteration, we also check whether the landmark L_i we are currently discovering, which is first accepted in some state s_i, is achieved again in a later state s' for some of the trajectories. If this is true, we find the earliest previously identified landmark L_j for which any landmark L_k found between it (such that $\forall k : j < k < i$) are also satisfied in some state between s and s'. Then, we move on to find a function that gives the number of loop landmarks for each loop in the initial state, and we find the exit and state progression conditions on each loop.

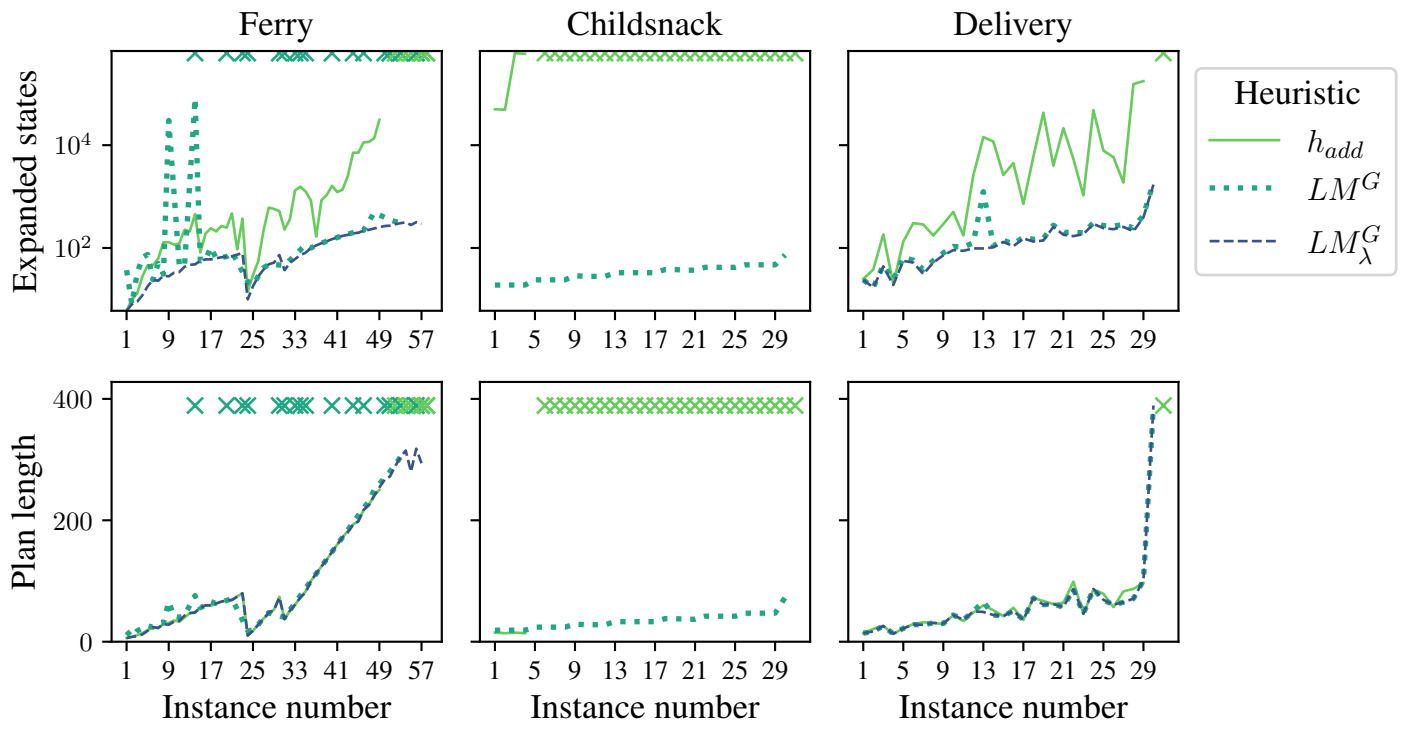


Figure 2: Number of expanded states (top) and plan length (bottom) for instances solved by the different heuristics.

We implemented this method in an Answer Set Program (ASP) (Gebser et al. 2012), which can be quickly evaluated, and the full model is given in Appendix C. The iterative process calls the ASP model for each landmark, given an updated set of previously found landmarks and their associated landmark states. Each iteration, we minimize the next state that identifies a landmark. The optimization is not very efficient yet, as we struggle with the trade-off between more expressive functions and generality to find more landmarks.

Initial Results

We use the method described above to learn a landmark graph for each domain. We generate a feature pool using the DLplan library, with a feature limit of 1000 times the number of non-static (plus the goal) predicates in the domain, a feature complexity limit of 9, and a time limit of 1800 seconds. We created five training instances for each domain by hand and used Fast Downward (FD) to generate the plans. Additionally, we manually altered the plans to contain expected patterns (based on the manual landmark graphs) to see if this increases performance. The resulting plans were often slightly longer. The reported learning runtime also includes pre- and postprocessing.

For `Delivery`, the five instances are all on a 3x3 grid, with either 2 or 3 packages to be delivered. The custom plans changed the delivery order for some packages. The graph learned from the FD plans has four landmarks, each with a guide and a loop from the fourth to the first landmark, and the graph was learned in 26.4 seconds. The graph learned from the manual plans is very similar, but in the third and fourth landmarks, an extra state descriptor is included, and the graph was produced in 28.4 seconds. For `ChildSnack`, the five instances each had 4 children distributed across the tables, with either a 2/2 or 1/3 allergic/non-allergic distribution. The custom plans ensured a sandwich was made, brought to a child, and then the next sandwich was made. The graph learned from the FD plans has three landmarks and no loops, and was produced in 73.7 seconds. The graph learned from the custom plans has eight landmarks and three loops (2,1), (3,1), and (4,1), which was learned in 547.4 seconds. For `Ferry`, the custom instances ensured that the goal location of one car was not the starting location of another. Each instance has three or four cars. For this domain, no further custom plans were needed. The graph learned from the FD plans has four landmarks and three loops (2,1), (3,1), and (4,1), and was produced in 22.9 seconds.

We reran the experiment described in the previous section for the LM^G with these landmark graphs. Table 2 shows the results from this experiment. Failures happen when the state progression takes a loop, thereby emptying its priority queue, which was a bad decision. For `Delivery`, we see that the manual plans provided more information, which improved the heuristic performance, but for `ChildSnack`, this did not have an impact. Two graphs (C-manual and F-FD) contain several nested loops, which our heuristic is currently not optimized for, leading to improper landmark progression. We want to tackle this in future work.

Table 2: Solved instances by $LMHeur$ heuristic with learned landmark graphs, trained on Fast-Downward plans (FD) or on custom-made plans (manual). For the three domains, we show the number of instances that were successfully solved (S), failed (F), timed out (T/O), or ran out of memory (O/M). For each variant of the landmark graph, we show the compute the average planning time in seconds for instances solved by both this LM^G variant and h_{add}, reporting the average time for both LM^G (Time) and h_{add} (HAdd).

		S	F	T/O	O/M	Time	HAdd
C (30)	FD	2	0	8	20	95.60	60.96
	manual	2	0	0	28	135.67	60.96
D (30)	FD	6	8	16	0	35.14	13.53
	manual	10	0	20	0	26.99	16.18
F (57)	FD	11	0	5	41	26.12	14.74

Related Work

While we propose generalized landmarks for classical planning, this work is very related to generalized planning. For example, Chen et al. (2026) proposed a generalized planner MOOSE that constructs lifted first-order rules over the state and goal conditions, and decides the action(s) to take. They computed these rules using goal regression to find the relevant facts with respect to the goal. Different rules for generalized planning were proposed in policy sketches (Drexler, Seipp, and Geffner 2024), using the same state functions we use, to define a sketch rule that specifies how function evaluations should change across states, thereby effectively identifying subgoals. While they learned their rules from a small set of instances, as we aim to do for generalized landmarks as well, the main difference is that they take a state-transition perspective, whereas we focus on state evaluation. Additionally, recent work proposed a state-centric formulation of generalized planning that learns to predict the successor state rather than the action sequence. While their method solved out-of-distribution instances, it relies on a state embedding to learn the transition model, providing a neural prediction from which the eventual plan is extracted. However, generalized landmarks yield more interpretable abstractions, and our learning idea remains symbolic in nature.

In terms of traditional landmarks, several works have considered generalizing them. Traditional disjunctive landmarks were extracted by symmetrically reducing the problem (Porteous and Cresswell 2002; Gregory et al. 2004), overcoming several symmetries within a problem instance. Moreover, landmarks were introduced for generalized planning by using pointers to the objects in a planning instance (Segovia-Aguas et al. 2022). Landmarks were also proposed in lifted planning, providing an abstraction by parameterizing predicates and actions over a finite universe of objects (Wichlacz, Höller, and Hoffmann 2022), though they remain restricted to domain predicates. While each of these methods generalizes over some aspect of a problem instance, they must still be computed for each instance individually. Finally, domain landmarks were proposed in hierarchical planning (Fine-Morris et al. 2022), which must hold in at least one plan for every instance. These were learned from plan traces (or trajectories) using natural language processing on the action definitions, which were then converted to the methods used in hierarchical task networks to identify the (hierarchical) subgoal structure. However, the repetitive nature captured by generalized landmarks is not included.

Other works that have sought to combat symmetries include a framework that solved families of problems by generalizing over objects, yielding a policy that can be applied to individual instances (Illanes and McIlraith 2019). Additionally, equivalence classes of objects in the lifted planning space reduced symmetries and limited the number of applicable actions (Ridder and Fox 2014). Both methods generalized explicitly over objects, while our method finds general relations in the problem (domain) structure. In terms of abstraction generation, Liu et al. (2025) used an LLM to decompose a complex task into subtasks, and then constructed a behavior rule library that expresses the subroutines and subgoal hierarchy of previously solved tasks.

Conclusion

Generalized landmarks are the must-reach high-level subgoals that any plan solving an instance in a domain must satisfy. Using domain knowledge, we construct graphs of generalized landmarks out of first-order functions over a state. By including loops in these graphs, we generalize over objects and capture repetitive subplans, which can be efficiently used in a landmark-counting heuristic. We show that generalized landmarks give a compact representation of the subgoals in a domain, providing an interpretable plan.

In future work, we want to improve the learning method to extract a generalized landmark graph from a few small solved instances. Currently, we generate a set of state functions and identify the same abstract states by finding a subset of features that describe that state across instances and objects. While many different landmark graphs can be described for one domain, one is more useful in heuristic search than another, based on the descriptiveness of the state functions and the presence of one or more loops. The remaining challenge is how to optimize the learning process to automatically find a representative and efficient landmark graph while interleaving loop discovery.

Acknowledgements

This work is part of the NWO LTP-ROBUST RAIL Lab, a collaboration between the Delft University of Technology, Utrecht University, NS, and ProRail. More information at https://icai.ai/icai-labs/rail/.

References

Chen, D. Z.; Hofmann, T.; Klassen, T. Q.; and McIlraith, S. A. 2026. Satisficing and Optimal Generalised Planning via Goal Regression. In *The Fortieth AAAI Conference on Artificial Intelligence (AAAI-26)*.

Delft High Performance Computing Centre (DHPC). 2024. DelftBlue Supercomputer (Phase 2).

https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2. Ark:/44463/DelftBluePhase2.

Drexler, D.; Francès, G.; and Seipp, J. 2022. DLPlan. https://doi.org/10.5281/zenodo.5826139. Zenodo.

Drexler, D.; Seipp, J.; and Geffner, H. 2024. Expressing and Exploiting the Common Subgoal Structure of Classical Planning Domains Using Sketches. *Journal of Artificial Intelligence Research*, 80: 171–208.

Fine-Morris, M.; Floyd, M. W.; Auslander, B.; Pennisi, G.; Gupta, K.; Roberts, M.; Heflin, J.; and noz Avila, H. M. 2022. Learning Decomposition Methods with Numeric Landmarks and Numeric Preconditions. In *Proceedings of the 5th Workshop on Hierarchical Planning*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gregory, P.; Cresswell, S.; Long, D.; and Porteous, J. 2004. On the extraction of disjunctive landmarks from planning problems via symmetry reduction. In *The 4th International Workshop on Symmetry and Constraint Satisfaction Problems*, 34–41. Springer.

Grundke, C.; Helmert, M.; and Röger, G. 2025b. Code, benchmarks and experiment data for the KR 2025 paper: "Domain-Independent Instance Generation for Classical Planning". Zenodo: https://doi.org/10.5281/zenodo.16875683.

Grundke, C.; Helmert, M.; and Röger, G. 2025a. Domain-Independent Instance Generation for Classical Planning. In *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning*, 805–809.

Grundke, C.; Röger, G.; and Helmert, M. 2024. Formal Representations of Classical Planning Domains. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*, 239–248.

Illanes, L.; and McIlraith, S. 2019. Generalized Planning via Abstraction: Arbitrary Numbers of Objects. In *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*.

Karpas, E.; and Domshlak, C. 2009. Cost-Optimal Planning with Landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)s*, 1728–1733.

Liu, P.; Tenenbaum, J.; Pack Kaelbling, L.; and Mao, J. 2025. Lifelong Experience Abstraction and Planning. In *ICML 2025 Workshop on Programmatic Representations for Agent Learning.*

Porteous, J.; and Cresswell, S. 2002. Extending landmarks analysis to reason about resources and repetition. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG '02) - Delft, The Netherlands*.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *6th European Conference on Planning*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks Revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, AAAI-08.

Ridder, B.; and Fox, M. 2014. Heuristic Evaluation Based on Lifted Relaxed Planning Graphs. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, ICAPS 2014.

Segovia-Aguas, J.; Jiménez, S.; Jonsson, A.; and Sebastiá, L. 2022. Scaling-up Generalized Planning as Heuristic Search with Landmarks (extended version). In *Symposium on Combinatorial Search*.

Wichlacz, J.; Höller, D.; and Hoffmann, J. 2022. Landmark Heuristics for Lifted Classical Planning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, 4665–4671.

Zhi-Xuan, T. 2022. *PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning*. MSc thesis, Massachussets Institute of Technology.

A State Functions and Generalized Landmark Graph for `Delivery`

Generalized Landmarks for `Delivery` with Simple State Descriptors

These are the simple state descriptors described in the introduction for us in a generalized landmark graph for `Delivery`.

d_1 $|\{\, \texttt{Item} \mid \mathit{empty}(\texttt{Item}) \,\}|$
Count the empty objects (trucks)

d_2 $|\{\, \texttt{Item} \mid \mathit{at}(\texttt{Item}, \texttt{Cell}) \wedge \forall \texttt{Item}_2 : \mathit{at}(\texttt{Item}_2, \texttt{Cell}) \rightarrow \mathit{truck}(\texttt{Item}_2) \,\}|$
Count the objects at the cell where there are only trucks

d_3 $|\{\, \texttt{Item} \mid \mathit{at}^g(\texttt{Item}) \wedge \exists \texttt{Item}_2 : \mathit{at}(\texttt{Item}_2, \texttt{Cell}) \wedge \mathit{empty}(\texttt{Item}_2) \,\}|$
Count the objects that have a goal location defined, and there is a $truck$ object at that goal location

d_4 $|\{\, \texttt{Item} \mid \mathit{at}^g(\texttt{Item}) \wedge \exists \texttt{Item}_2 : \mathit{at}(\texttt{Item}_2, \texttt{Cell}) \wedge \mathit{empty}(\texttt{Item}_2) \,\}|$
Count the objects that have a goal location defined, and there is an $empty$ item at that goal location

These constitute the landmarks for `Delivery` as follows: $L_1 = \{\neg d_2\}$, $L_2 = \{\neg d_1, d_2\}$, $L_3 = \{d_3\}$, and $L_4 = \{d_1, d_4\}$

Generalized Landmarks for `Delivery` with Complex State Descriptors

The more complicated state descriptions to define the landmarks exactly per function are formulated as follows.

d'_1 $|\{\, \texttt{Item} \mid \mathit{empty}(\texttt{Item}) \wedge \exists \texttt{Item}_2 : \mathit{at}(\texttt{Item}_2, \texttt{Cell}) \rightarrow \big(\mathit{at}^g(\texttt{Item}_2, \texttt{Cell}_2) \setminus \mathit{at}(\texttt{Item}_2, \texttt{Cell}_2)\big) \,\}|$
Count the empty objects (trucks) that are at a cell at which there are also packages that are not at their goal cell

d'_2 $|\{\, \texttt{Item} \mid \mathit{at}^g(\texttt{Item}, \texttt{Cell}) \wedge \neg \mathit{at}(\texttt{Item}, \texttt{Cell}) \wedge \mathit{carrying}(\texttt{Item}_2, \texttt{Item}) \wedge \mathit{truck}(\texttt{Item}_2) \,\}|$
Count the objects that have a goal (at^g) but are not there and are being carried by a truck

d'_3 $|\{\, \texttt{Item} \mid \mathit{carrying}(\texttt{Item}, \texttt{Item}_2) \wedge \exists \texttt{Item}_2 : \big(\mathit{at}^g(\texttt{Item}_2, \texttt{Cell}) \setminus \mathit{at}(\texttt{Item}_2, \texttt{Cell})\big)$
$\wedge \exists \texttt{Item}_3 : \mathit{at}(\texttt{Item}_3, \texttt{Cell}) \rightarrow \mathit{truck}(\texttt{Item}_3) \,\}|$
Count the objects that are carrying something which is not at its goal location, but at that goal location, there is a truck

d'_4 $|\{\, \texttt{Item} \mid \mathit{at}^g(\texttt{Item}) \wedge \mathit{at}(\texttt{Item}, \texttt{Cell}) \wedge \exists \texttt{Item}_2 : \mathit{at}(\texttt{Item}_2, \texttt{Cell}) \wedge \mathit{empty}(\texttt{Item}_2) \,\}|$
Count the objects that are at their goal location, and there is an empty object (truck) at that goal location

Loops of Generalized Landmarks for `Delivery`

To formalize the loop in the running example for `Delivery`, we have one state function, which counts the objects that have a goal location defined but are not at that goal location. This function is used as the exit condition when the set is empty. The state progressor ensures that between each loop landmark, this value decreases. And the state value of the loop landmark counter gives the size of this set.

$$|\{\, \texttt{Item} \mid \mathit{at}^g(\texttt{Item}, \texttt{Cell}) \wedge \neg \mathit{at}(\texttt{Item}, \texttt{Cell}) \,\}|$$

Local Heuristic Landmark Guides for `Delivery`

Each landmark is associated with a guide state value. Some of these state values compute a distance to a state that satisfies the latter part.

λ_1 Distance between a state where an $empty$ truck is at a cell to a state such that a truck is at a cell, such that a package is at this cell, and it is not the goal cell for the package.

λ_2 Distance between a state with where something is $empty$ and a state with a non-empty truck.

λ_3 Distance between a state where there is a non-empty truck at a cell to a state where the truck is carrying a package and at the goal cell of that package.

λ_4 Distance between a state with where something is $empty$ and a state with a non-empty truck.

Resulting Graph

Figure 3 shows the resulting landmark graph that we manually constructed for the `Delivery` domain.

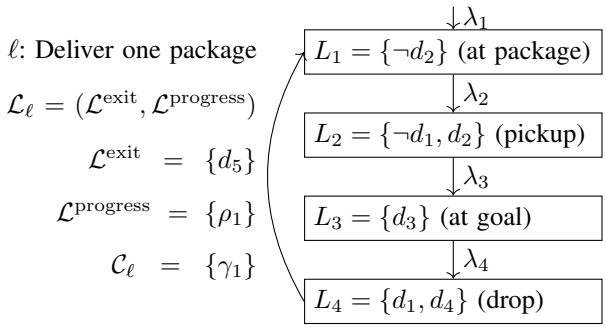


Figure 3: Example graph of generalized landmarks for `Delivery` domain, showing the state descriptors on the right.

B Extra Results for `Ferry`

Extra Results for Domain with Redundant Action

We adapted the `Ferry` domain slightly to see the benefit the guide originally provided and to evaluate the un-guided variant on this adapted domain. We introduce the predicate $delivered(car)$, which is never used in a precondition, but does alter the state, so the state is seen as new and not as visited. Then, we introduce the redundant action $justDelivered$, which takes as precondition: $emptyFerry$, $at(car1, loc)$, $at(car2, loc)$, and $atFerry(loc)$, with the single effect to set a predicate $delivered(car1)$ to `true`. This action can then be used after delivering a car to a location where the ferry should also pick up its next passenger.

Figure 4 shows the expanded states and plan length when the heuristic as described in our evaluation section is used to plan the `Ferry` instances. We see that the difference between the guided and not guided versions decreases, and both outperform the h_{add} baseline. $\text{LM}^G - \lambda$ solves one instance more than LM^G.

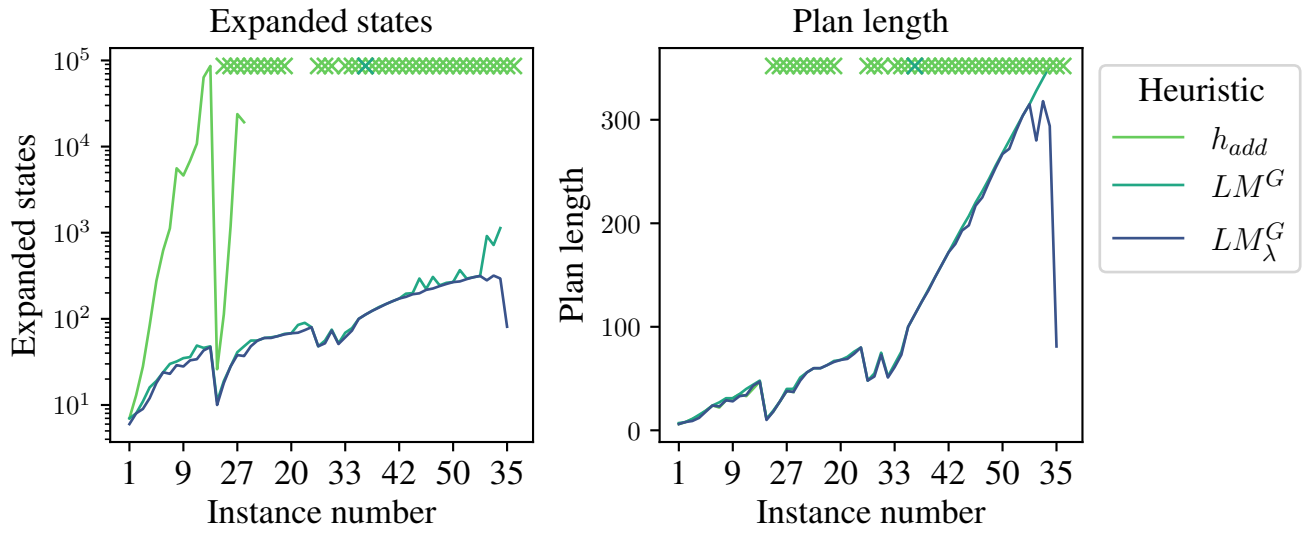


Figure 4: Expanded states and plan length for the adapted `Ferry` domain.

C Learning Model as an Answer Set Program

```
%%% Constants %%%
num_value(0;1).
opposite_value(0, 1).
opposite_value(1, 0).
infinite_value(2147483647).
opposite_condition(Up, Down) :- feature_condition_increase(Up), feature_condition_decrease(Down).
opposite_condition(Down, Up) :- feature_condition_increase(Up), feature_condition_decrease(Down).

%%% Select compound feature %%%
{ select(Feature, Value) } :- feature(Feature), num_value(Value), landmark_feature_type(Type), feature_type(Feature, Type), b_value(Trajectory, _, Feature, Value) : state(Trajectory, _).
% Identify all states for each trajectory for which the selection of features holds.
landmark_state(Trajectory, State) :- state(Trajectory, State), b_value(Trajectory, State, Feature, Value) : select(Feature, Value).
% Get the first state in every trajectory in which the landmark holds.
first_landmark_state(Trajectory, First) :- state(Trajectory, _), First = #min {State : landmark_state(Trajectory, State) }, landmark_state(Trajectory, First).

%%% Constraints %%%%
% Cannot have a Trajectory without a landmark state.
no_landmark :- state(Trajectory, _), not landmark_state(Trajectory, _).
:- no_landmark.
% Cannot have a new landmark that occurs in the same state as the previous landmark
:- first_landmark_state(Trajectory, State), previous_first_landmark_state(Trajectory, PrevState, _), State <= PrevState.

% Select features that changed their value with the directly previous state, must be at least one feature that has a value change directly before the landmark
direct_feature_change(Feature, Value) :- feature(Feature), select(Feature, Value), opposite_value(Value, Opposite), b_value(Trajectory, State - 1, Feature, Opposite) : first_landmark_state(Trajectory, State).
:- not direct_feature_change(_, _).

% Select features that have changed their once and kept it consistent until this landmark - should change between loops.
feature_change(Feature, Value, Trajectory, State, PrevState) :- feature(Feature), select(Feature, Value), opposite_value(Value, Opposite), first_landmark_state(Trajectory, State), b_value(Trajectory, PrevState, Feature, Opposite), PrevState < State, b_value(Trajectory, IntermediateState, Feature, Value) : state(Trajectory, IntermediateState), IntermediateState > PrevState, IntermediateState <= State.
:- feature(Feature), select(Feature, Value), state(Trajectory, _), not feature_change(Feature, Value, Trajectory, _, _).
% Make sure that the selected features changes its value between loop iterations
change_between_loops(Feature, Value, Trajectory, LmState1, LmState2):- feature(Feature), select(Feature, Value), landmark_state(Trajectory, LmState1), landmark_state(Trajectory, LmState2), LmState1 < LmState2, state(Trajectory, IntermediateState), IntermediateState > LmState1, IntermediateState < LmState2, opposite_value(Value, Opposite), b_value(Trajectory, IntermediateState, Feature, Opposite).
:- feature(Feature), select(Feature, Value), not change_between_loops(Feature, Value, _, _, _).

%%% Construct landmark graph %%%
% Create new landmark node for the currently selected compound feature.
most_recent_landmark_node(N) :- N = #max { Index : previous_landmark_node(Index, _, _) }, previous_landmark_node(N, _, _).
current_landmark_node(NodeIndex, Feature, Value) :- select(Feature, Value), most_recent_landmark_node(N), NodeIndex = N + 1.
```

```
%%% Loop detection %%%
% We can loop between the CurrentLandmark, achieved in the CurrentState and again in the NextState, and a PreviousLandmark, which is achieved again in between the Current and NextState.
landmark_achieved_again(Trajectory, CurrentState, NextState, PreviousLandmark) :- landmark_state(Trajectory, CurrentState), landmark_state(Trajectory, NextState), NextState > CurrentState, previous_landmark_state(Trajectory, IntermediateState, PreviousLandmark), IntermediateState < NextState, IntermediateState > CurrentState.

% Helper function to check which IntermediateLandmarks are accepted again before NextState after achieving PreviousLandmark in CurrentState.
intermediate_landmark_achieved(Trajectory, CurrentState, IntermediateState, NextState, PreviousLandmark, IntermediateLandmark) :- landmark_achieved_again(Trajectory, CurrentState, NextState, PreviousLandmark), current_landmark_node(CurrentLandmark, _, _), previous_landmark_node(IntermediateLandmark, _, _), previous_landmark_state(Trajectory, IntermediateState, IntermediateLandmark), state(Trajectory, IntermediateState), IntermediateLandmark < CurrentLandmark, IntermediateLandmark >= PreviousLandmark, IntermediateState > CurrentState, IntermediateState < NextState.

% A landmark loop is only valid if all IntermediateLandmarks (between PreviousLandmark and CurrentLandmark) are achieved again between achieving the CurrentLandmark in the CurrentState and the NextState
check_landmarks_loop(Trajectory, CurrentState, NextState, CurrentLandmark, PreviousLandmark) :- landmark_achieved_again(Trajectory, CurrentState, NextState, PreviousLandmark), current_landmark_node(CurrentLandmark, _, _), previous_landmark_node(PreviousLandmark, _, _), intermediate_landmark_achieved(Trajectory, CurrentState, _, NextState, PreviousLandmark, IntermediateLandmark) : previous_landmark_node(IntermediateLandmark, _, _), IntermediateLandmark < CurrentLandmark, IntermediateLandmark > PreviousLandmark.

% Get the biggest loop: get the earliest PreviousLandmark such that all landmarks between CurrentLandmark and PreviousLandmark are achieved again before achieving CurrentLandmark again in NextState.
biggest_loop(Trajectory, CurrentState, NextState, CurrentLandmark, PreviousLandmark) :- first_landmark_state(Trajectory, CurrentState), landmark_state(Trajectory, NextState), current_landmark_node(CurrentLandmark, _, _), PreviousLandmark = #min { IntermediateLandmark : check_landmarks_loop(Trajectory, CurrentState, NextState, CurrentLandmark, IntermediateLandmark) }, previous_landmark_node(PreviousLandmark, _, _).

% If there is a loop from CurrentLandmark to PreviousLandmark, which shifts the landmark state from CurrentState to NextState, then PreviousLandmarkState is the state where PreviousLandmark holds and is later than CurrentState. We want the features that change their numeric value between CurrentState and PreviousLandmarkState, which forms the conditions
loop_condition_change(Feature, Trajectory, CurrentState, PreviousLandmarkState, CurrentLandmark, PreviousLandmark, CurrentValue, NewValue) :- numerical(Feature), first_landmark_state(Trajectory, CurrentState), biggest_loop(Trajectory, CurrentState, NextState, CurrentLandmark, PreviousLandmark), previous_landmark_state(Trajectory, PreviousLandmarkState, PreviousLandmarkNode), PreviousLandmarkState > CurrentState, PreviousLandmarkState < NextState, value(Trajectory, CurrentState, Feature, CurrentValue), value(Trajectory, PreviousLandmarkState, Feature, NewValue).

% Find the first loop: earliest next state where full loop is achieved
complete_loop(Trajectory, CurrentState, EarliestLoopState, CurrentLandmark, PreviousLandmark) :- first_landmark_state(Trajectory, CurrentState), biggest_loop(Trajectory, CurrentState, _, CurrentLandmark, PreviousLandmark), EarliestLoopState = #min { NextState : biggest_loop(Trajectory, CurrentState, NextState, CurrentLandmark, PreviousLandmark) }.
% Find more loops recursively: after the previous loop ended (PrevLoopEndState) and the minimal NextLoopEndState such that all landmarks are achieved again in between
complete_loop(Trajectory, PrevLoopEndState, NextLoopEndState, CurrentLandmark, PreviousLandmark) :- complete_loop(Trajectory, PrevLoopStartState, PrevLoopEndState, CurrentLandmark, PreviousLandmark), NextLoopEndState = #min { NextState : check_landmarks_loop(Trajectory, PrevLoopEndState, NextState, CurrentLandmark, PreviousLandmark) }, biggest_loop(Trajectory, _, NextLoopEndState, CurrentLandmark, PreviousLandmark).

```

```
%%% Constraints %%%
% Make sure that a loop occurs in more than one trajectory to rule out incidentals
non_single_trajectory_loop(CurrentLandmark, PreviousLandmark) :- complete_loop(Trajectory, _, _, CurrentLandmark, PreviousLandmark), complete_loop(OtherTrajectory, _, _, CurrentLandmark, PreviousLandmark), Trajectory != OtherTrajectory.
single_trajectory_loop(CurrentLandmark, PreviousLandmark) :- complete_loop(_, _, _, CurrentLandmark, PreviousLandmark), not non_single_trajectory_loop(CurrentLandmark, PreviousLandmark).
:- complete_loop(_, _, _, CurrentLandmark, PreviousLandmark), single_trajectory_loop(CurrentLandmark, PreviousLandmark).

%%% State index helpers for loops %%%
% If no loop is found the CurrentState is the first_landmark_state.
last_end_of_loop_state(Trajectory, CurrentState) :- first_landmark_state(Trajectory, CurrentState), current_landmark_node(CurrentLandmark, _, _), not complete_loop(Trajectory, _, _, CurrentLandmark, _).
% If a loop is found, reset the CurrentState index to the last state in the last loop.
last_end_of_loop_state(Trajectory, LastState) :- first_landmark_state(Trajectory, CurrentState), current_landmark_node(CurrentLandmark, _, _), complete_loop(Trajectory, _, _, CurrentLandmark, PreviousLandmark), LastState = #max { NextState : complete_loop(Trajectory, _, NextState, CurrentLandmark, PreviousLandmark) }, landmark_state(Trajectory, LastState).

% The end of the first loop is the first landmark state, also if there are no loops in this trajectory.
end_of_loop_state(Trajectory, CurrentState) :- first_landmark_state(Trajectory, CurrentState), current_landmark_node(CurrentLandmark, _, _).
% For each loop, get the end state.
end_of_loop_state(Trajectory, NextState) :- first_landmark_state(Trajectory, CurrentState), current_landmark_node(CurrentLandmark, _, _), complete_loop(Trajectory, _, NextState, CurrentLandmark, _).


%%% Loop conditions %%%
% Get the numeric features that identify the last state of a loop for the last loop iteration
last_end_of_loop_value(Feature, Value) :- numerical(Feature), value(_, _, Feature, Value), value(Trajectory, State, Feature, Value) : last_end_of_loop_state(Trajectory, State).
% The features that identifies the last_end_of_loop_state should also change in each loop iteration
end_of_loop_iteration_value(Feature, Trajectory, State, NextState, Value, NextValue) :- last_end_of_loop_value(Feature, _), end_of_loop_state(Trajectory, State), end_of_loop_state(Trajectory, NextState), State < NextState, value(Trajectory, State, Feature, Value), value(Trajectory, NextState, Feature, NextValue), Value != NextValue.

% Identify whether the features that change in a loop iteration increase
end_of_loop_iteration_value_change(Trajectory, State, NextState, Feature, Condition) :- numerical(Feature), feature_condition_increase(Condition), end_of_loop_iteration_value(Feature, Trajectory, State, NextState, Value, NextValue), Value < NextValue.
% Identify whether the features that change in a loop iteration decrease
end_of_loop_iteration_value_change(Trajectory, State, NextState, Feature, Condition) :- numerical(Feature), feature_condition_decrease(Condition), end_of_loop_iteration_value(Feature, Trajectory, State, NextState, Value, NextValue), Value > NextValue.
% Check that the increase or decrease in the loop identifying feature is consistent across all loops
end_of_loop_iteration_consistent_value_change(Feature, Condition) :- end_of_loop_iteration_value_change(_, _, _, Feature, Condition), end_of_loop_iteration_value_change(Trajectory, State, NextState, Feature, Condition) : complete_loop(Trajectory, State, NextState, _, _).

% Identify the feature which signals that the current landmark state is the last loop iteration
last_end_of_loop_condition(Feature, Value, CurrentLandmark, PreviousLandmark) :- last_end_of_loop_value(Feature, Value), complete_loop(_, _, _, CurrentLandmark, PreviousLandmark).
```

```
% Find the feature that identifies the exact number of loops in the initial state, by finding features that give the number of end of loop states
number_of_loop_end_states(Trajectory, Value, Feature, CurrentLandmark, PreviousLandmark) :- initial(Trajectory, InitialState), complete_loop(_, _, _, CurrentLandmark, PreviousLandmark), NumStates = #count { State : end_of_loop_state(Trajectory, State) }, numerical(Feature), value(Trajectory, InitialState, Feature, Value), Value == NumStates.
% Make sure that the number-of-loops identifying feature is consistent for all trajectories
compute_loop_feature(CurrentLandmark, PreviousLandmark, Feature) :- numerical(Feature), complete_loop(_, _, _, CurrentLandmark, PreviousLandmark), number_of_loop_end_states(Trajectory, _, Feature, CurrentLandmark, PreviousLandmark) : state(Trajectory, _).
% Make sure that all features that indicate the number of loops actually give the same value.
:- compute_loop_feature(CurrentLandmark, PreviousLandmark, Feature1), compute_loop_feature(CurrentLandmark, PreviousLandmark, Feature2), Feature1 != Feature2, number_of_loop_end_states(Trajectory, Value1, Feature1, CurrentLandmark, PreviousLandmark), number_of_loop_end_states(Trajectory, Value2, Feature2, CurrentLandmark, PreviousLandmark), Value1 != Value2.

%%% Wrap up loops %%%
% We find a loop if there is at least one complete loop for some trajectory, there is a condition that changes throughout the loop and we can identify the end of the loop with that same feature
found_loop(CurrentLandmark, PreviousLandmark, ConditionalFeature, Condition, FinalValue) :- complete_loop(_, _, _, CurrentLandmark, PreviousLandmark), end_of_loop_iteration_consistent_value_change(ConditionalFeature, Condition), last_end_of_loop_condition(ConditionalFeature, FinalValue, CurrentLandmark, PreviousLandmark).

%%% Guiding %%%
% Helper for all states in between the two landmark states
intermediate_state(Trajectory, State) :- most_recent_landmark_node(PrevNode), first_landmark_state(Trajectory, Latest), previous_first_landmark_state(Trajectory, Earliest, PrevNode), state(Trajectory, State), State >= Earliest, State < Latest.

% Select guide features that count down to 0 in the first landmark state for every trajectory, which are 1 in the directly previous state
guide_to_zero(Feature) :- numerical(Feature), value(Trajectory, LandmarkState, Feature, 0) : first_landmark_state(Trajectory, LandmarkState).
guide_from_one(Feature) :- numerical(Feature), value(Trajectory, LandmarkState-1, Feature, 1) : first_landmark_state(Trajectory, LandmarkState).
guide_feature(Feature, PrevNode, NodeIndex) :- numerical(Feature), current_landmark_node(NodeIndex, _, _), most_recent_landmark_node(PrevNode), guide_to_zero(Feature), guide_from_one(Feature).

% For the selected features, it is better if they must be 0 in the landmark.
:- select(Feature, 0).
% Conditional loop feature values cannot be infinite
:- infinite_value(Inf), last_end_of_loop_condition(_, _, _, Inf).


%%% Optimization %%%
% Selection criteria: first minimize the first state that (priority=3 -> higher) is a landmark (across trajectories)
#minimize { State@3,Trajectory : state(Trajectory, _), first_landmark_state(Trajectory, State) }.


%%% Postprocessing optimization %%%
% Give guides a value based on how well they count down
guide_value(Feature, Trajectory, Intermediate, Value, Diff) :- guide_feature(Feature, PrevNode, _), most_recent_landmark_node(PrevNode), previous_first_landmark_state(Trajectory, PrevLandmarkState, PrevNode), intermediate_state(Trajectory, Intermediate), value(Trajectory, Intermediate, Feature, Value), Diff = (Intermediate - PrevLandmarkState + 1) - Value.
```

```
% Give the feature identifying the number of loops in the initial state a value
loop_identification_score(CurrentLandmark, PreviousLandmark, Feature, Score) :-
    current_landmark_node(CurrentLandmark, _, _), found_loop(CurrentLandmark,
    PreviousLandmark, _, _, _), compute_loop_feature(CurrentLandmark, PreviousLandmark,
    Feature), complexity(Feature, Score).
loop_identification_score(CurrentLandmark, PreviousLandmark, none, -1) :-
    current_landmark_node(CurrentLandmark, _, _), found_loop(CurrentLandmark,
    PreviousLandmark, _, _, _), not compute_loop_feature(CurrentLandmark, PreviousLandmark, _
    ).

condition_score(Decrease, 0) :- feature_condition_decrease(Decrease).
condition_score(Increase, 1) :- feature_condition_increase(Increase).
% If EndOfLoopValue = 0 this is better, and decrease is preferred
loop_condition_score(CurrentLandmark, PreviousLandmark, Feature, Score) :- found_loop(
    CurrentLandmark, PreviousLandmark, Feature, Condition, EndOfLoopValue), condition_score(
    Condition, Lower), complexity(Feature, Complexity), Score = Complexity + EndOfLoopValue +
     Lower.


% Landmark selection
#show select/2.
#show first_landmark_state/2.
#show landmark_state/2.
#show current_landmark_node/3.

% Guide selection
#show guide_feature/3.

% Loop selection
#show found_loop/5.

% Optimization helpers
#show guide_value/5.
#show loop_condition_score/4.
#show loop_identification_score/4.
```