Transforming Fact Landmarks to Action Landmarks for Heuristic Search

Bachelor thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr.Florian Pommerening, Esther Mugdan

Yaowen Rui
yaowen.rui@stud.unibas.ch
2021-065-586

13.11.2025

Acknowledgments

I would like to express my deepest appreciation to my supervisors Dr. Florian Pommerening and Esther Mugdan for their patience and insightful guidance. With their support and care, I could complete this thesis on time. I would like to thank Prof. Dr. Malte Helmert as well for giving me the opportunity to write my Bachelor's thesis in the Artificial Intelligence Research Group.

Abstract

Landmark heuristics are often used for guiding search in classical planning, which estimate the goal distance from given landmarks. This thesis introduces a landmark transformer that converts fact landmarks produced by common landmark generation methods into action landmarks before search begins. We then extend the landmark count heuristic to operate directly on these precomputed action landmarks, avoiding the on-the-fly derivation of action landmarks from fact landmarks in every state. We evaluate this new approach in Fast Downward against the standard fact landmark version of the landmark count heuristic across multiple landmark generation methods. Our findings show that, under the same landmark generation method, the action landmark variant typically has more state expansions, longer search time, higher memory usage, and thus lower coverage. Moreover, it often achieves a lower time per expansion, indicating cheaper per state evaluation but weaker overall search guidance. Across landmark generation methods, there are cases where the action landmark heuristic outperforms fact landmark baselines.

Table of Contents

1 Introduction

Classical planning is a fundamental concept in Artificial Intelligence where one aims to determine a sequence of actions leading from an initial state to a desired goal state. These sequences are known as *plans*. In everyday life, classic planning often operates behind the scenes. Consider this following scenario: you are on the first floor of a large shopping center and want to use one of the four elevators to reach your destination floor. Meanwhile, other customers are waiting for the elevators on various floors. How can the elevator system coordinate to dispatch the elevators efficiently? The goal is to minimize the total actions (move up, move down, board, leave) needed to deliver all passengers to their destinations, allowing elevators to operate in parallel and to perform multiple pickups and drop-offs within a single trip.

That kind of planning in general can be hard. To handle such complexity, we use search algorithms that explore possible sequences of actions to find a plan from the initial state to the goal. Additionally, an "advisor" named *heuristic* is often used which helps the search algorithm decide which states to explore first, ideally guiding it toward the goal faster. A heuristic is a function that estimates how far a given state is from the goal. However, its estimates can sometimes be less accurate. Let us consider a new elevator scenario: An elevator has a capacity of one person at a time. Two passengers and the elevator are all on the first floor, and both passengers want to go to the second floor (we ignore the cost of boarding and leaving). A naive heuristic might estimate the cost as just two (one trip for each person), but this ignores the elevator's location constraint that the elevator should be on the first floor before picking up the second passenger after the first ride. This heuristic is underestimating in this case. The perfect heuristic value should be four (two trips up and two trips down).

We can use landmarks to build heuristics. Landmarks are atoms or actions that must happen in every plan. Returning to the example: Before the elevator can transport a passenger, it must first be empty and arrive at the floor where the passenger is waiting. These necessary preconditions act as landmarks that guide the search more reliably.

In landmark-based planning from Fast Downward (Helmert (2006)), disjunctive fact landmarks are commonly used by heuristics. The objective of this thesis is to support the generation of disjunctive action landmarks and to extend an existing fact landmark based heuristic so that it correctly handles these disjunctive action landmarks. In Chapter 2, we will introduce the necessary background knowledge. In Chapter 3, we will present the generation process for both disjunctive fact and action landmarks. Then, in Chapter 4, we will describe the heuristic we extend and our modifications. In Chapter 5 we test our implementation over multiple task domains. Finally, Chapter 6 offers conclusions and describes future improvements.

2 Background

2.1 Classical Planning

We consider classical planning in the SAS^+ formalism from Bäckström and Nebel (1995).

Definition 1 (SAS^+ *planning task*)**.** A SAS^+ planning task is a 4-tuple $\Pi = \langle V, A, I, G \rangle$ where:

- V is a finite set of state variables. Each variable $v \in V$ has an associated finite domain *dom(v)*. An atom $\langle v, d \rangle$ is a tuple of a variable $v \in V$ and a value in its domain $d \in$ *dom(v)*. A partial state s is a set of atoms, containing at most one atom per variable. Let $vars(s) = \{v \mid \exists\, d, \langle v, d \rangle \in s\}$ denote the set of variables defined by s, and let $s(v)$ denote the value d with the tuple $\langle v, d \rangle \in s$. A partial state s is called a state if $vars(s) = V$.
- A is a finite set of actions. Each action $a \in A$ is a tuple $\langle pre(a), cost(a), eff(a) \rangle$ where the precondition $pre(a)$ and effect $eff(a)$ are partial states, $cost(a) \in \mathbb{R}_0^+$ is the cost of action a.
- I is a state called initial state.
- G is a partial state called the goal.

An action $a \in A$ is applicable in a state s if $pre(a) \subseteq s$. When an applicable action a is applied to s, it produces a successor state $s[a]$, defined as follows:

$$s[a] = eff(a) \cup \{\langle v, d \rangle \in s \mid v \notin vars(eff(a))\}$$

An action sequence $\pi = \langle a_1, \dots, a_n \rangle$ is applicable in s if a_1 is applicable in s, a_2 is applicable in $s[a_1]$, and so on until a_n is applicable in $s[a_{n-1}]$. For action sequence π, we write $s[\pi]$ for $s[a_1][a_2] \dots [a_n]$, which is defined if each action is applicable in the respective state. If $G \subseteq s[\pi]$, then π is a *s-plan* and an *I-plan* is called a plan.
The cost of a plan π is defined as $cost(\pi) = \sum_{i=1}^{n} cost(a_i)$. If $cost(\pi)$ is minimal among all plans, then π is an optimal plan.

2.2 Landmarks

A landmark for a state s of a planning task $\Pi = \langle V, A, I, G \rangle$ is a property that needs to be satisfied in some state along each s-plan. For example, some actions must be applied (action landmark) or some atoms must hold (fact landmark). The following definitions of two kinds of landmarks are introduced by Hoffmann et al. (2004).

Definition 2 (Disjunctive Action Landmark)**.** A disjunctive action landmark for state s is a set of actions $l^A \subseteq A$ such that $l^A \cap \{a_1, \dots, a_n\} \neq \emptyset$ for all s-plans $\pi = \langle a_1, \dots, a_n \rangle$.

Definition 3 (Disjunctive Fact Landmark)**.** A disjunctive fact landmark l^F for s is a set of atoms such that all s-plans $\pi = \langle a_1, \dots, a_n \rangle$ visit a state containing some atom in l^F.

During the generation of fact or action landmarks, the ordering between them may also be included. Ordering information can be used to infer landmarks that are required again. The following landmark ordering definition is originally from Porteous et al. (2001).

Definition 4 (Landmark Orderings)**.** Let A and B be fact or action landmarks:
There is a **natural ordering** between A and B, written $A \to B$, iff in each possible action sequence where B is true at time i, A is true at some time $j < i$.
There is a **greedy-necessary ordering** between A and B, written $A \to_{gn} B$, iff in each action sequence where B is first added at time i, A is true at time $i - 1$.

Landmarks and their associated orderings can be stored in a data structure called a landmark graph.

Definition 5 (Landmark graph)**.** A landmark graph is a labeled directed acyclic graph (V, E), where V is a set of fact or action landmarks, and E is a set of edges labeled with different kinds of landmark orderings.

2.3 Heuristic search

We can solve the planning tasks in many ways, and one of the common approaches is the heuristic search.

Definition 6 (State Space)**.** A state space is a 6-tuple $\mathcal{S} = \langle S, A, T, \mathit{cost}, s_I, S^* \rangle$ and follows directly from the planning task, where S is a finite set of states; A is a finite set of actions; $T \subseteq S \times A \times S$ is a set of transitions; $\mathit{cost} : A \to \mathbb{R}_0^+$ is a cost function; $s_I \in S$ is the initial state; $S^* \subseteq S$ is a set of goal states.

Definition 7 (Heuristic)**.** Let $\mathcal{S}$ be a state space with states S. A heuristic function or heuristic for $\mathcal{S}$ is a function

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\},$$

mapping each state to a nonnegative number or ∞.
Heuristics estimate the cost of a s-plan from s to a goal state. In this thesis, we use action landmarks to compute the heuristic values in each state. Search algorithms use heuristic values to prioritize expansion of promising states. For evaluation we use greedy best first

search (lazy), which was first introduced by Richter and Helmert (2009). Greedy best first search expands the nodes only by their heuristic:

$$f(n) = h(n),$$

where $h(n)$ is a heuristic estimate of the cost from n to the goal. Greedy best first search (lazy) is a variant of greedy best first search characterized by deferring the evaluation of a node's heuristic function until that node is popped from the front queue.

2.4 Elevator Example

We prepare a concrete example based on Figure 2.1. This example will be used in Chapter 3. There is a passenger r, waiting on the second floor of a three-story building (floors 1–3). The destination of r is the first floor. The building has two elevators: The slow one starts on the first floor and the fast one starts on the third floor. We ignore all the action costs. This elevator problem is a 4-tuple $\langle V, A, I, G\rangle$, where:

- $V = \{\text{r-at},\ \text{r-in},\ \text{slow-at},\ \text{fast-at}\}$.
  $\mathit{dom}(\text{r-at}) = \mathit{dom}(\text{fast-at}) = \mathit{dom}(\text{slow-at}) = \{1, 2, 3\}$, $\mathit{dom}(\text{r-in}) = \{\text{slow}, \text{fast}\}$
- I=$\{\langle\text{r-at}, 2\rangle,\ \langle\text{slow-at}, 1\rangle,\ \langle\text{fast-at}, 3\rangle\}$.
- G=$\{\langle\text{r-at}, 1\rangle\}$.
- $A = \{\textit{r-board-e-at-f} \mid e \in \{\text{slow}, \text{fast}\}, f \in \{1, 2, 3\}\}$
  $\cup\{\textit{r-leave-e-at-f} \mid e \in \{\text{slow}, \text{fast}\}, f \in \{1, 2, 3\}\}$
  $\cup\{\textit{e-move-}f_1\text{-}f_2 \mid e \in \{\text{slow}, \text{fast}\}, f_1, f_2 \in \{1, 2, 3\}, f_1 \neq f_2\}$
  One of the actions a is: r-board-slow-at-2
  $\mathit{pre}(a) = \{\langle\text{r-at}, 2\rangle, \langle\text{slow-at}, 2\rangle\}$, $\mathit{eff}(a) = \{\langle\text{r-in}, \text{slow}\rangle\}$, $\mathit{cost}(a) = 0$

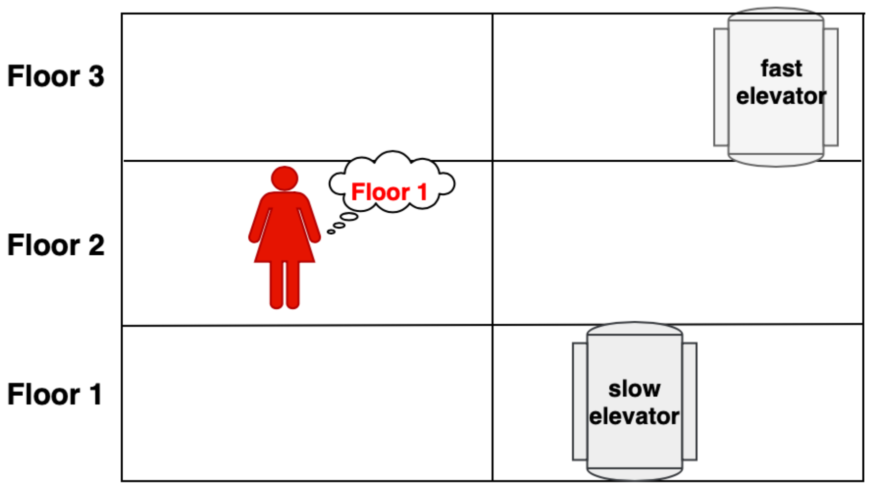


Figure 2.1: Elevator problem example instance.

3 Landmark generation

The core of this thesis is landmark generation. In Fast Downward, multiple generators produce both simple (contains just a single atom) and disjunctive fact landmarks for landmark heuristic search. In this chapter, first, we describe how fact landmarks are generated based on one of the generation methods, how they are represented in the landmark graph, and how they are evaluated during the search. Second, we show how disjunctive action landmarks can be derived from fact landmarks, how they are stored, and how they are evaluated during the search.

3.1 Fact Landmark generation

We follow the fact landmark generation method defined by Richter et al. (2008). Fact landmarks can be generated from a set of known fact landmarks (e.g., the atoms in the goal) through backchaining. Backchaining starts from known fact landmarks and works backward: For each fact landmark, we consider actions that can achieve it for the first time and add their preconditions as fact landmark candidates. Candidates are then filtered by pruning criterion. This process is iterated until no new fact landmarks are found.

The candidate generation procedure uses an approximation based on the relaxed planning graph. For a SAS^+ planning task $\Pi = \langle V, A, I, G \rangle$, we define the relaxed planning graph as follows: For $i \in \mathbb{N}_0$, let:

$$A_i = \{a \in A \mid \text{pre}(a) \subseteq S_i\} \setminus \bigcup_{j<i} A_j \tag{3.1}$$

$$S_i = \begin{cases} I, & \text{if } i = 0 \\ S_{i-1} \cup \{\langle v, d \rangle \mid a \in A_{i-1} \text{ and } \langle v, d \rangle \in \text{eff}(a)\}, & \text{if } i > 0 \end{cases} \tag{3.2}$$

The set S_i is an over-approximation of all atoms that can be reached by applying up to i actions in the delete-relaxed task (Hoffmann and Nebel, 2001) (i.e., a planning task where all negative effects of actions are ignored). The set A_i contains the actions which are applicable in the relaxation after i steps, but not previously.

We consider the actions that can achieve each atom of the fact landmark for the first time. These actions are the first achievers of this fact landmark. We define *first-time* and *first-achiever* as follows:

Let *first-time*$(\langle v, d\rangle)$ be the index i such that $\langle v, d\rangle \in S_i$ and $\langle v, d\rangle \notin S_j$ for all $j < i$. We define the first achievers of an atom:

$$\textit{first-achievers}(\langle v, d\rangle) = \{a \in A_i \mid \langle v, d\rangle \in \text{eff}(a) \text{ and } i = \textit{first-time}(\langle v, d\rangle) - 1\} \tag{3.3}$$

Fact landmark candidates are then suggested as follows: Starting from a known simple fact landmark $\langle v, d\rangle$, which first appears in the relaxed planning graph in S_i ($i > 0$, *first-time*$(\langle v, d\rangle) = i$), consider all actions in *first-achievers*$(\langle v, d\rangle)$.
For each $a \in$ *first-achievers*$(\langle v, d\rangle)$, there is a $\langle v', d'\rangle \in \text{pre}(a)$, then $\langle v', d'\rangle$ is a simple fact landmark candidate which is ordered greedy-necessarily (see in Definition 4) before $\langle v, d\rangle$.

Additionally, the disjunctive fact landmarks are found as follows: It may happen that the *first-achievers*$(\langle v, d\rangle)$ do not share a precondition, but there is an atom $\langle v', d'\rangle$, which is in turn needed for the preconditions of the first achievers. Let *first-achievers*$(\langle v, d\rangle) = \{a_1, a_2, \ldots, a_n\}$ and let $X = \{\langle v_1, d_1\rangle, \ldots, \langle v_n, d_n\rangle \mid \langle v_i, d_i\rangle \in \bigcup_{i=1}^{n} \text{pre}(a_i)\}$ be a set of atoms, each atom $\langle v_i, d_i\rangle \in X$ appears in the precondition of some first achiever of $\langle v, d\rangle$ (i.e., *first-achievers*$(\langle v, d\rangle)$ do not share a common precondition). Then X is a disjunctive fact landmark. We can then backchain from X to find more fact landmarks. In order to organize X for better future use, we partition it by variable. Firstly, we define the set of variables that actually appear in X: $K = \{v \in V \mid \exists d \text{ with } \langle v, d\rangle \in X\}$. For each $k \in K$, we define the k-partition: $X_k = \{\langle v, d\rangle \in X \mid v = k\}$. Each X_k groups those atoms in X that share the same state variable k.

Starting from the goal, we backchain to compute fact landmarks for each goal atom until no further fact landmarks are found. We then take the all discovered fact landmarks with their orderings and store them in the fact landmark graph (see Section 3.2).

Due to over-approximations in the candidate generation process, some candidate landmarks are not fact landmarks. So we set two pruning criterion to eliminate non-landmarks. The first one is: Each fact landmark candidate M is tested by removing all achievers of M from the original task, and then checking whether the resulting task still has a relaxed solution. If not, then M is indeed critical to the solution of the original task, and is thus guaranteed to be a fact landmark. Otherwise, the candidate M is rejected. We use the following example to show the elimination process in detail.
Example:
The initial state is $I = \{\langle v_1, d_1\rangle\}$, and the goal state is $\{\langle v_4, d_4\rangle\}$. A is a set of actions. For each action $a \in A$ (action costs are not considered here and are therefore omitted):

	pre	eff
a_1	$\{\langle v_1, d_1\rangle\}$	$\{\langle v_2, d_2\rangle\}$
a_2	$\{\langle v_2, d_2\rangle\}$	$\{\langle v_4, d_4\rangle\}$
a_3	$\{\langle v_1, d_1\rangle\}$	$\{\langle v_3, d_3\rangle\}$
a_4	$\{\langle v_3, d_3\rangle\}$	$\{\langle v_4, d_4\rangle\}$

Suppose $\langle v_2, d_2\rangle$ is a candidate simple fact landmark. If we remove all achievers of $\langle v_2, d_2\rangle$ (e.g., removing a_1), we can still reach the goal via a_3 then a_4. A relaxed solution still exists, so $\langle v_2, d_2\rangle$ is not necessary, we prune $\langle v_2, d_2\rangle$ as a non-landmark.

The second pruning criterion is that a simple or disjunctive fact landmark must hold at some point on every valid plan. If a fact landmark candidate M contains an atom that is already true in the initial state, M is satisfied at time 0 in every plan and imposes no achievement requirement (e.g., no action needs to be taken to achieve it). To avoid such landmarks that provide no guidance, we discard any candidate that is initially satisfied.

We use the example in Figure 2.1 to explain the whole generation process in detail:
In this example, there are three state variables: *p-at*, *p-in*, *e-in*. The variables and their values induce the following atoms:

$$\langle r\text{-}at, 1\rangle, \langle r\text{-}at, 2\rangle, \langle r\text{-}at, 3\rangle,$$

$$\langle r\text{-}in, slow\rangle, \langle r\text{-}in, fast\rangle,$$

$$\langle slow\text{-}at, 1\rangle, \langle slow\text{-}at, 2\rangle, \langle slow\text{-}at, 3\rangle,$$

$$\langle fast\text{-}at, 1\rangle, \langle fast\text{-}at, 2\rangle, \langle fast\text{-}at, 3\rangle$$

As shown in Figure 3.1. We start from the only goal, the open fact landmark is $\langle r\text{-}at, 1\rangle$. *first-achievers*$(\langle r\text{-}at, 1\rangle) = \{r\text{-}leave\text{-}slow\text{-}at\text{-}1, r\text{-}leave\text{-}fast\text{-}at\text{-}1\}$. The achievers' preconditions are $\{\langle r\text{-}in, slow\rangle, \langle r\text{-}in, fast\rangle\}$ and $\{\langle slow\text{-}at, 1\rangle, \langle fast\text{-}at, 1\rangle\}$.
These two are candidate fact landmarks. We discard any candidate that contains an atom already true initially. Here $\langle slow\text{-}at, 1\rangle$ is initially true, so $\{\langle slow\text{-}at, 1\rangle, \langle fast\text{-}at, 1\rangle\}$ is dropped. In this first backchaining round, we keep $\{\langle r\text{-}in, slow\rangle, \langle r\text{-}in, fast\rangle\}$ as a disjunctive fact landmark.

We now backchain from the new open disjunctive fact landmark $\{\langle r\text{-}in, slow\rangle, \langle r\text{-}in, fast\rangle\}$. The first achievers of each atom from that fact landmark are *r-board-slow-at*-2 and *r-board-fast-at*-2. The grouped preconditions are $\{\langle slow\text{-}at, 2\rangle, \langle fast\text{-}at, 2\rangle\}$ and $\{\langle r\text{-}at, 2\rangle\}$; the latter is shared across all achievers and is treated as a simple fact landmark:$\{\langle r\text{-}at, 2\rangle\}$. The remaining preconditions form another disjunctive fact landmark: $\{\langle slow\text{-}at, 2\rangle, \langle fast\text{-}at, 2\rangle\}$. If a simple fact landmark is already true in the initial state, it remains a fact landmark but is not expanded further (the generator backchains only from landmarks not true initially). The generation stops when no further disjunctive or simple fact landmarks are found.

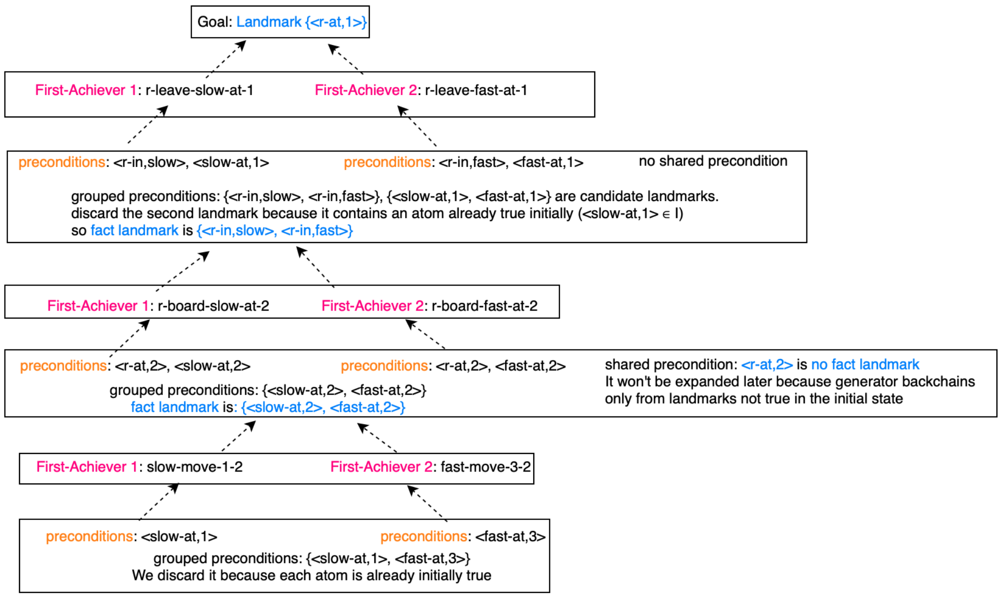

Goal: Landmark {<r-at,1>}
First-Achiever 1: r-leave-slow-at-1
First-Achiever 2: r-leave-fast-at-1
preconditions: <r-in,slow>, <slow-at,1>
preconditions: <r-in,fast>, <fast-at,1>
no shared precondition
grouped preconditions: {<r-in,slow>, <r-in,fast>}, {<slow-at,1>, <fast-at,1>} are candidate landmarks.
discard the second landmark because it contains an atom already true initially (<slow-at,1> ∈ I)
so fact landmark is {<r-in,slow>, <r-in,fast>}
First-Achiever 1: r-board-slow-at-2
First-Achiever 2: r-board-fast-at-2
preconditions: <r-at,2>, <slow-at,2>
preconditions: <r-at,2>, <fast-at,2>
shared precondition: <r-at,2> is no fact landmark
It won't be expanded later because generator backchains only from landmarks not true in the initial state
grouped preconditions: {<slow-at,2>, <fast-at,2>}
fact landmark is: {<slow-at,2>, <fast-at,2>}
First-Achiever 1: slow-move-1-2
First-Achiever 2: fast-move-3-2
preconditions: <slow-at,1>
preconditions: <fast-at,3>
grouped preconditions: {<slow-at,1>, <fast-at,3>}
We discard it because each atom is already initially true

Figure 3.1: Disjunctive fact landmark generation.

3.2 Landmark graph for disjunctive fact landmarks

The landmark graph (see in Definition 5) is a central data structure which stores the necessary information for solving a planning task. Each node represents either a simple fact landmark or a disjunctive fact landmark (a set of atoms) that must be true at some point in every valid plan. Directed edges $L_1 \rightarrow L_2$ encode orderings, indicating that in every valid plan, before L_2 is achieved for the first time, L_1 must already have been achieved at least once.

During the generation, once a fact landmark is found, it, along with any existing ordering, is added to the fact landmark graph. Let us refer back to the Figure 3.1. We establish the fact landmark graph for this example (Figure 3.2). The goal landmark $\{\langle r\text{-}at, 1\rangle\}$ becomes the first(root) node. After the first backchaining step, we obtain the disjunctive fact landmark $\{\langle r\text{-}in, slow\rangle, \langle r\text{-}in, fast\rangle\}$; we add a node for it and add an directed edge from this new node to the first node, representing their ordering. Subsequent fact landmarks discovered during backchaining are added in the same way. For simplicity, we consider only natural orderings (see Definition 4).

In Figure 3.2, the goal $\{\langle r\text{-}at, 1\rangle\}$ contains a single atom. With multiple goal atoms, the landmark graph may contain several tree-like components, one rooted at each goal atom. Finally, the fact landmark graph also supports the derivation of action landmarks. We will discuss it later in Section 3.4.

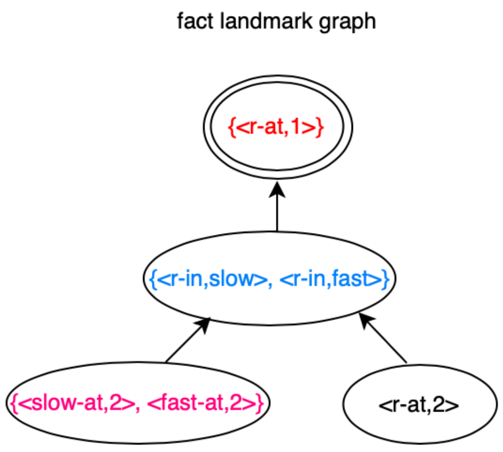


Figure 3.2: Fact landmark graph of elevator example

3.3 Status of fact landmarks

The fact landmark graph is static, meaning it does not change during search. What does change with the current state is the subset of nodes that remain to be achieved (often called the remaining or not-yet-accepted landmarks). The heuristic value for a state derives from the landmarks that still need to be achieved at that state. We now clarify how to determine, for a given state, whether a fact landmark holds and whether it needs to be (re)achieved. We say that a fact landmark F holds in a state s iff there exists $\langle v, d\rangle \in F$ with $\langle v, d\rangle \in s$ as well. This is written as $s \models F$.

Example: Given state $s = \{\langle \textit{r-at}, 2\rangle, \langle \textit{slow-at}, 1\rangle, \langle \textit{fast-at}, 3\rangle\}$, and two fact landmarks $F_1 = \{\langle \textit{r-at}, 2\rangle, \langle \textit{r-at}, 3\rangle\}$, $F_2 = \{\langle \textit{fast-at}, 2\rangle\}$, we say that F_1 holds in s because at least one of its atoms is true in s (here, $\langle \textit{r-at}, 2\rangle$). In contrast, F_2 does not hold.

Büchner et al. (2023) explain how to track each fact landmark's status during heuristic search using two sets for each state s: $\mathcal{L}_{\text{past}}$ and $\mathcal{L}_{\text{fut}}$.
Let $\mathcal{L}_I$ denote the initial set of fact landmarks $\{F_1, F_2, \ldots, F_n\}$, partitioned into past and future components such that $\mathcal{L}_{\text{past}} \cup \mathcal{L}_{\text{fut}} = \mathcal{L}_I$.

- Past fact landmarks ($\mathcal{L}_{\text{past}}(s)$) are fact landmarks that have already been achieved on every known path that leads from the initial state to current state s.
- Future fact landmarks ($\mathcal{L}_{\text{fut}}(s)$) are fact landmarks that still have to be achieved on every plan that goes from s to the goal.
- In the initial state, all fact landmarks are in $\mathcal{L}_{\text{fut}}$ and none in $\mathcal{L}_{\text{past}}$.

We summarize how to update the two sets $\mathcal{L}_{\text{past}}$ and $\mathcal{L}_{\text{fut}}$ when moving along a transition e from state s to s' produced by some action a ($e : s \xrightarrow{a} s'$).

Past update: Any fact landmark F that is true in the new state s' must be considered already achieved:

$$\mathcal{L}_{\text{past}}(s') = \mathcal{L}_{\text{past}}(s) \cup \{F \in \mathcal{L}_I \mid s' \models F\}$$

Future update: If a fact landmark F has just been achieved by this step (false in s, true in s'), then it no longer needs to be in future:

$$\mathcal{L}_{\text{fut}}(s') = \mathcal{L}_{\text{fut}}(s) \setminus \{F \in \mathcal{L}_I \mid s \not\models F \text{ and } s' \models F\}$$

If two fact landmarks F_1, F_2 from the fact landmark graph have the natural ordering $F_1 \to F_2$. This natural ordering constraint checks for out-of-order violations such that: If $s' \models F_2$ and $F_1 \notin \mathcal{L}_{\text{past}}(s)$, then e is a conflict transition because F_2 is true without having put F_1 into past. This violates the natural ordering, we discard this state transition e (i.e., fact landmarks achieved from this transition are not accepted).
The same state s' can be reached via different incoming transitions, but the ones we consider are only those observed so far in the search, not all transitions into s'. Suppose s has incoming non-conflicting edges $e_1, e_2, \ldots, e_r$. We compute per-edge updates and merge the past and future sets:

$$\mathcal{L}_{\text{past}}(s') = \bigcap_{i=1}^{r} \mathcal{L}_{\text{past}}^{(e_i)}(s') \text{ and } \mathcal{L}_{\text{fut}}(s') = \bigcup_{i=1}^{r} \mathcal{L}_{\text{fut}}^{(e_i)}(s')$$

where $\mathcal{L}_{\text{past}}^{(e_i)}(s')$ and $\mathcal{L}_{\text{fut}}^{(e_i)}(s')$ represent, respectively, the sets of past and future fact landmarks induced by the incoming edge e_i in the state s'. Edges pruned by natural ordering do not contribute to the merge. This merge rule avoids falsely declaring fact landmark past based on only one path and prevents prematurely dropping fact landmarks that are still required on some path.

3.4 Landmark transformation

Some landmark heuristics (e.g., h^{sum}) use disjunctive action landmarks at evaluation time. A common approach is to transform disjunctive fact landmarks (available as the nodes of the fact landmark graph) into disjunctive action landmarks.
For a disjunctive fact landmark $F = \{f_1, f_2, \ldots, f_k\}$, we define its associated disjunctive action landmark $L^A(F)$ as below: We collect the actions that can first achieve any atom f_i in F:

$$L^A(F) = \bigcup_{i=1}^{k} \textit{first-achievers}(f_i),$$

where *first-achievers* is defined in Section 3.1. Intuitively, $L^A(F)$ contains all actions that have any member of F as effect.

Rather than transforming landmarks in every state, we precompute and store disjunctive action landmarks before search begins. Once the landmark generation method has established the fact landmark graph for the task, we derive one disjunctive action landmark from each fact landmark node. These action landmarks form a fixed set of action landmarks:

$$\mathcal{L}^A(\mathcal{F}) = \{L^A(F) \mid F \in \mathcal{F}\},$$

where $\mathcal{F}$ is a set of fact landmarks: $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$. The set of action landmarks is stored in an action landmark graph with each action landmark as a node. Since each fact landmark $F = \{f_1, f_2, \ldots, f_k\}$ from $\mathcal{F}$ is a disjunctive fact landmark, its achiever set $L^A(F) = \bigcup_{i=1}^{k} \textit{first-achievers}(f_i)$ is also disjunctive. Moreover, natural orderings between fact landmarks are inherited by the derived action landmarks (i.e., if there is an edge between two fact landmarks F and F': $F \rightarrow F'$, then the edge between action landmarks $L^A(F)$ and $L^A(F')$ is established: $L^A(F) \rightarrow L^A(F')$).

After we know how to derive action landmarks from the fact landmarks, we can build the action landmark graph from fact landmark graph:

Definition 8 (Action landmark graph)**.** Given a fact landmark graph $G_F = (V_F, E_F)$, where nodes V_F is a set of fact landmarks $\{F_1, F_2, \ldots, F_k\}$ and E_F is a set of edges, representing the natural ordering. The action landmark graph is a directed acyclic graph $G_A = (V_A, E_A)$ constructed as follows:

- $V_A = \{L^A(F) \mid F \in V_F\}$
- $E_A = \{L^A(F) \rightarrow L^A(F') \mid F \rightarrow F' \in E_F\}$

Figure 3.3 shows the action landmark graph derived from the fact landmark graph in the elevator example.

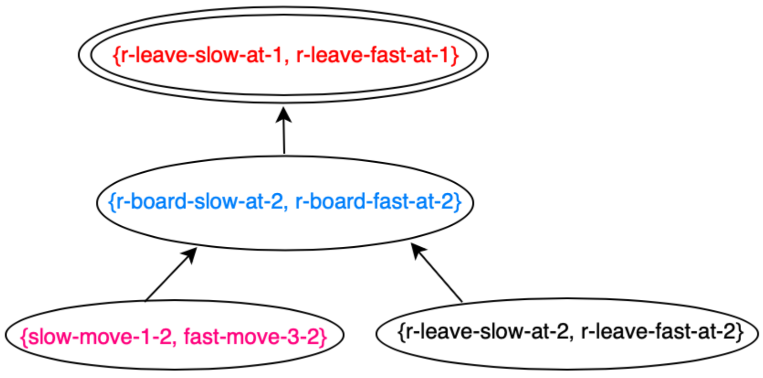

{r-leave-slow-at-1, r-leave-fast-at-1}
{r-board-slow-at-2, r-board-fast-at-2}
{slow-move-1-2, fast-move-3-2}
{r-leave-slow-at-2, r-leave-fast-at-2}

Figure 3.3: Action landmark graph of elevator example

3.5 Determination of action landmarks

Unlike fact landmarks, action landmarks are sets of actions that do not carry state values. Instead of saying *an action landmark holds in a state*, we say *an action landmark is achieved on a transition* when one of its actions is used on that transition. Concretely, consider a transition from state s to s' produced by some action a.

$$s \xrightarrow{a} s'$$

Given a set of disjunctive action landmarks $\mathcal{L}^A = \{L_1^A, L_2^A, \ldots, L_k^A\}$, we define $\textit{Hit}(a)$ as a set of action landmarks that contain a. Applying a on $s \xrightarrow{a} s'$ achieves every $L_A \in \textit{Hit}(a)$:

$$Hit(a) = \{L^A \in \mathcal{L}^A \mid a \in L^A\}$$

We first set up rules with two sets, past and future, to track action landmarks' status at each state: For each state s, $\mathcal{L}^A_{\text{past}}(s) \cup \mathcal{L}^A_{\text{fut}}(s) = \mathcal{L}^A$.

- $\mathcal{L}^A_{\text{past}}(s)$: Action landmarks that have already been achieved at least once on every path from initial state to current state s.
- $\mathcal{L}^A_{\text{fut}}(s)$: Action landmarks that must still be achieved at least once on every plan from s to a goal.

We enforce a natural ordering between two action landmarks $L^A_1 \to L^A_2$. For a transition $e:\ s \xrightarrow{a} s'$: If $a \in L^A_2$ and $L^A_1 \notin \mathcal{L}^{L^A}_{\text{past}}(s)$, we treat it as a conflict that action landmarks achieved from this transition are not accepted. We gather such action landmarks in conflict in a set *Conflict*:

$$Pruned(a) = \{a \mid a \in L^A_j \text{ and } L^A_i \notin \mathcal{L}^A_{\text{past}}(s),\ L^A_i \to L^A_j,\ i \neq j \text{ and } i, j \in [1, k]\}$$

$$Conflict(a) = \{L^A \in \mathcal{L}^A \mid a \in L^A \text{ and } a \in Pruned\}$$

Past update:

$$\mathcal{L}^A_{\text{past}}(s') = (\mathcal{L}^A_{\text{past}}(s) \cup Hit(a)) \setminus Conflict(a)$$

Future update:

$$\mathcal{L}^A_{\text{fut}}(s') = \mathcal{L}^A_{\text{fut}}(s) \setminus Hit(a) \setminus Conflict(a)$$

In a heuristic search, the same state s' can be reached via different transitions. Suppose s' has incoming edges $a_1, a_2, \ldots, a_r$, we should compute per-edge updates and merge the past and future sets:

$$\mathcal{L}^A_{\text{past}}(s') = \bigcap_{i=1}^{r} \mathcal{L}^{A(a_i)}_{\text{past}}(s') \text{ and } \mathcal{L}^A_{\text{fut}}(s') = \bigcup_{i=1}^{r} \mathcal{L}^{A(a_i)}_{\text{fut}}(s'),$$

where $\mathcal{L}^{A(a_i)}_{\text{past}}(s')$ and $\mathcal{L}^{A(a_i)}_{\text{fut}}(s')$ represent, respectively, the sets of past and future action landmarks induced by the incoming edge a_i in the state s'.
We use intersection for the past and union for the future. An action landmark is past at s' only if all incoming edges mark it past, and it remains future at s' if any incoming edge still requires it. This prevents falsely declaring an action landmark achieved. We call this merge rule conservative because it keeps $\mathcal{L}^A_{\text{past}}(s')$ safely over-approximated and $\mathcal{L}^A_{\text{fut}}(s')$ safely under-approximated.
Example:
We have a set of action landmarks: $\mathcal{L}^A = \{L^A_1, L^A_2, L^A_3\}$, where $L^A_1 = \{h\}$, $L^A_2 = \{b, c\}$, $L^A_3 = \{d, f\}$. State s has two predecessors (e.g., two edges to reach s):
$a_1:\ s_1 \xrightarrow{b} s$:
This edge hits L^A_2 (since $b \in L^A_2$). After progressing along a_1:

- $\mathcal{L}^{a_1}_{\text{past}}(s) = \{L^A_2\}$
- $\mathcal{L}^{a_1}_{\text{fut}}(s) = \{L^A_1, L^A_3\}$

$a_2: \; s_2 \xrightarrow{d} s$:
This edge hits L_3^A. After progressing along a_2:

- $\mathcal{L}_{\text{past}}^{a_2}(s) = \{L_3^A\}$
- $\mathcal{L}_{\text{fut}}^{a_2}(s) = \{L_1^A, L_2^A\}$

Merge at s:

- $\mathcal{L}_{\text{past}}(s) = \mathcal{L}_{\text{past}}^{a_1}(s) \cap \mathcal{L}_{\text{past}}^{a_2}(s) = \{L_2^A\} \cap \{L_3^A\} = \emptyset$
- $\mathcal{L}_{\text{fut}}(s) = \mathcal{L}_{\text{fut}}^{a_1}(s) \cup \mathcal{L}_{\text{fut}}^{a_2}(s) = \{L_1^A, L_3^A\} \cup \{L_1^A, L_2^A\} = \{L_1^A, L_2^A, L_3^A\}$

Intuitively, L_2^A is achieved on transition a_1 but not on a_2, and L_3^A is achieved on a_2 but not on a_1. Since at least one incoming path still needs each of them, neither can be marked past at s. Hence no action landmarks are past at s after the merge.

Alongside the conservative merge from above, we also consider an optimistic merge that treats an action landmark as past at s' if any incoming edge marks it past, and keeps it as future only if all incoming edges still mark it future:

$$\mathcal{L}_{\text{past}}^{A}(s') = \bigcup_{i=1}^{r} \mathcal{L}_{\text{past}}^{A(a_i)}(s') \text{ and } \mathcal{L}_{\text{fut}}^{A}(s') = \bigcap_{i=1}^{r} \mathcal{L}_{\text{fut}}^{A(a_i)}(s')$$

This optimistic merge rule is not reasonable for search progression that it can mark an action landmark as past in a s' even though some reaching edges have not achieved it. Everything left in the future set is required for all solutions from s', and required action landmarks may be dropped. We include it as a contrasting merge choice to examine how it affects the search behavior in the experiments.

4
Landmark count heuristic

Having generated the disjunctive action landmarks, we now introduce one of the heuristics that uses these landmarks to estimate the distance from a given state to the goal. There are several landmark-based heuristics, for instance, the cyclic landmark heuristic (h^{cycle}), the landmark count heuristic (h^{sum}). For our setting, We extend only the landmark count heuristic, which operates on fact landmarks, to correctly handle disjunctive action landmarks. In this chapter, we first introduce the original landmark count heuristic and then present our adaptation of it.

We begin with the landmark count heuristic h^{sum} which was introduced by Richter and Westphal (2010). It sums up the costs of the cheapest fact landmark achievers. Suppose $\mathcal{F} = \{F \mid F \in \mathcal{F}\}$ is a set of fact landmarks that still need to be achieved at a given state. From each F we can derive a set of first achievers $o_F = \{o \mid o \in o_F\}$, the heuristic value at a state s is:

$$h^{\text{sum}}(s) = \Sigma_{F \in \mathcal{F}} C(o_F)$$

$$C(o_F) = min_{o \in o_F} cost(o)$$

We now adapt h^{sum} to operate directly on disjunctive action landmarks. In this variant, the heuristic for a state s uses the action landmarks that still need to be achieved and accounts for action costs by assigning each action landmark the cost of its cheapest member.

Let $\mathcal{L}^A$ be a set of action landmarks that need to be achieved in state s, $L^A \in \mathcal{L}^A$ is an action landmark, we then define h^{sum} using action landmarks as

$$h^{\text{a-sum}}(s) = \Sigma_{L^A \in \mathcal{L}^A} C(L^A),$$

$$C(L^A) = min_{a \in L^A} cost(a)$$

Example:

If we have a set of disjunctive action landmarks $\mathcal{L}^A = \{L_1^A, L_2^A\}$ for state s, where $L_1^A = \{a_1, a_2\}$, $L_2^A = \{a_1, a_3\}$ with $cost(a_1) = 3$, $cost(a_2) = cost(a_3) = 2$. Then

$$C(L_1^A) = min\{3, 2\} = 2,$$

$$C(L_2^A) = min\{3, 2\} = 2,$$

so

$$h^{\text{a-sum}}(s) = C(L_1^A) + C(L_2^A) = 2 + 2 = 4.$$

5 Experimental results

In this thesis, we introduce a method that conducts the action landmark transformation (i.e., transformer) and the new heuristic, $h^{\text{a-sum}}$, which was introduced in Section 4. This transformer maps fact landmarks to action landmarks, and the new heuristic then computes values using these action landmarks. We first analyze how many landmarks are lost during the transformation and what the initial heuristic values are for both the fact and action landmark heuristics. In addition, we evaluate $h^{\text{a-sum}}$ in a series of experiments, comparing it with the baseline fact landmark heuristic h^{sum} (see Section 4) in terms of expansions, search time, memory usage and coverage. For the experiments the following setup was used:

- We use Downward Lab (Seipp et al., 2017) which is a Python package for running the experiments for the Fast Downward planning system[1].
- We run the experiments on the sciCORE compute cluster of the University of Basel. Each task was run on a single core of a Core Intel Xeon Silver 4114 CPU. The search time limit is set to 5 minutes, and the memory limit is set to 3584MB.
- The implementation is tested on a collection of IPC benchmark instances[2]. To be more specific, we use the satisficing benchmark suit, excluding those that use axioms, because some of our landmark factories do not support axioms.

Our landmark transformer is embedded in the $h^{\text{a-sum}}$ heuristic. The $h^{\text{a-sum}}$ uses one of the following methods to compute heuristic values:

- The method *rhw*, introduced by Richter et al. (2008), generates sound but incomplete fact landmarks; extracts only sound orderings (e.g., natural or greedy-necessary ordering) via first-achiever checks, the possibly-before test, and DTG path constraints.
- The method *hm*, introduced by Keyder et al. (2010), computes the complete set of causal delete-relaxation fact landmarks by label propagation on the relaxed planning graph; induces necessary/greedy-necessary orderings directly from the propagated labels.

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

- The method *zg*, introduced by Zhu and Givan (2003), provides a sound, incomplete propagation that yields a large set of fact landmarks. The orderings come from intersections/necessary-predecessor relations in the planning graph.
- The method *exhaust*, introduced by Richter et al. (2008), exhaustively checks for each fact if it is a landmark. This check is done using relaxed planning.

These methods are called landmark factories which generate fact landmarks for our transformer. We use the greedy best first search (lazy) algorithm for our experiments. Landmarks are either simple or disjunctive.

5.1 Loss of landmarks

Our landmark transformer is designed to generate exactly one action landmark for each fact landmark which is not satisfied in the initial state. Hence, for every task, the total number of action landmarks is less than or equal to the number of fact landmarks. We want to know how many action landmarks do not contribute to the heuristic value. We use the landmark loss percentage as our metric:

$$\text{landmark loss percentage} = \frac{\#\text{fact landmarks} - \#\text{action landmarks}}{\#\text{fact landmarks}}$$

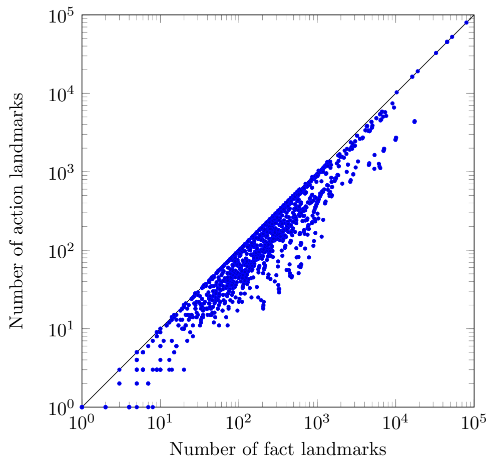
Table 5.1 reports the average per-task loss over four landmark factories. The loss is generally high. Even the best case (*hm*) still loses about one-third of landmarks on average, while *exhaust* loses nearly two-thirds. In Section 5.2, we examine whether the discarded fact landmarks are initially true.

	hm	*rhw*	*exhaust*	*zg*
Average loss per task	33.9%	46.6%	64.5%	37.9%

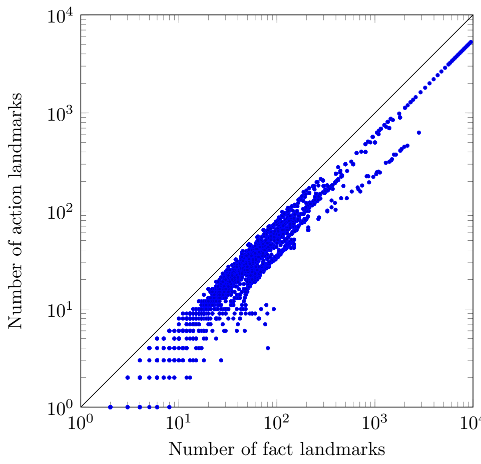
Table 5.1: Average loss percentage across landmark factories *hm*, *rhw*, *exhaust* and *zg*.

Average loss is the arithmetic mean of losses and is easily affected by extreme values, so tasks with larger loss can push the average loss percentage higher. To show the distribution behind these averages, the four subfigures of Figure 5.1 plot, on logarithmic axes, the number of action landmarks (y-axis) against the number of fact landmarks (x-axis) for each task and factory. The points on the diagonal represent no loss during the transformation. Distance below the diagonal corresponds to relative loss (the farther below, the larger the loss). Points on the x-axis indicate near-complete loss. We do not have any complete loss cases (no action landmarks are derived).
In Figure 5.1(c), we can clearly see that many points lie far below the diagonal and clusters appear close to the bottom of the plot, representing frequent near-complete losses. This aligns with the highest average loss under the factory *exhaust* (64.5%).
A high loss does not automatically indicate an error, instead, it follows from our design choices. The filtering of initially true fact landmarks contributes the high loss. The loss varies by landmark factory because each factory constructs fact landmarks differently. A higher loss means that a factory tends to produce more fact landmarks that are filtered out

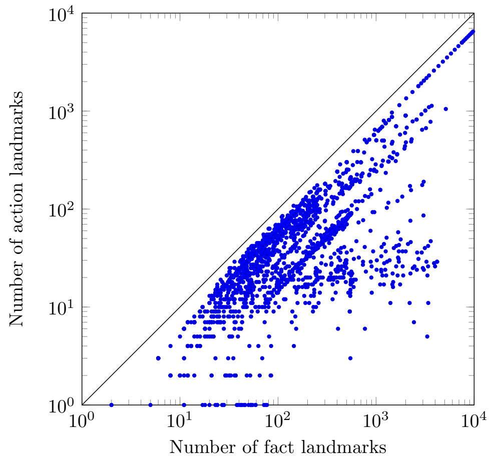
under our criteria (i.e., fact landmarks that are unnecessary for our transformer). Moreover, a high loss does not necessarily occur on complex tasks, as task complexity increases, the loss percentage does not necessarily rise.



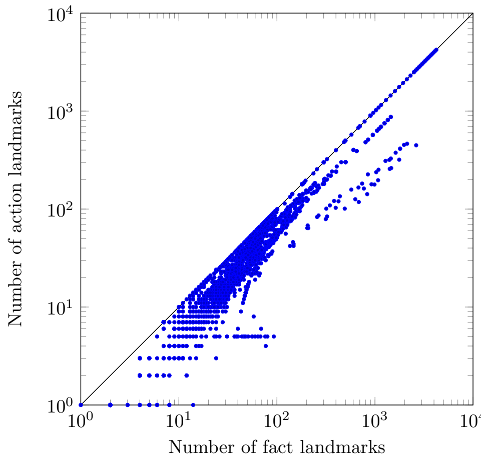
(a) Number of landmarks with factory *hm*



(b) Number of landmarks with factory *rhw*



(c) Number of landmarks with factory *exhaust*



(d) Number of landmarks with factory *zg*

Figure 5.1: Number of landmarks per factory.

5.2 Initial heuristic values

We want to check whether the loss of landmarks introduced by the transformer affects the action heuristic values in the initial state.

Our results indicate that the initial heuristic values of $h^{\text{a-sum}}$ and h^{sum} are identical across all the factories. This aligns with our design: We transform exactly those fact landmarks that are initially not satisfied into action landmarks, so both heuristics should behave the same in the initial state (i.e., the initial heuristic values should be identical). Although we lost landmarks during the transformation, the initial heuristic values are not affected. Hence, the discarded landmarks are just redundant information which we do not need for the action landmarks side.

We also see the influence of using different factories (Table 5.2). Regardless of whether we use action or fact landmarks, *hm* yields the largest initial heuristic values, while *zg* yields the smallest. Factories that produce more fact landmarks leave more unsatisfied fact landmarks in the initial state, hence a larger initial heuristic value. In particular, *rhw* is more strict than *hm* in generating fact landmarks. After the landmark transformation, this leads to fewer action landmarks and therefore smaller initial heuristic values.

	hm	*rhw*	*exhaust*	*zg*
Average initial heuristic value	1765518	59245	79547	34448

Table 5.2: Arithmetic mean of initial heuristic values with landmark factories *hm*, *rhw*, *exhaust* and *zg*.

5.3 Expansions

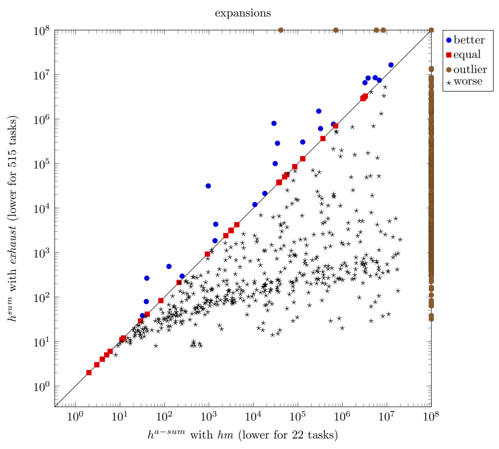
We compare search effort between the heuristics $h^{\text{a-sum}}$ and h^{sum} across four landmark factories. We use expanded states as our metric. When a heuristic can better guide the search, the search algorithm expands fewer states.

The four subfigures of Figure 5.2 plot, on logarithmic axes, the number of expansions of $h^{\text{a-sum}}$ (x-axis) against h^{sum} (y-axis) with factories *hm*, *rhw*, *zg* and *exhaust* respectively. Red points on the diagonal indicate equal performance. Blue points above the diagonal indicate that heuristic $h^{\text{a-sum}}$ expands fewer states during the search (better guidance with $h^{\text{a-sum}}$). The vertical and horizontal stacks at 10^8 correspond to runs that hit the resource limit. These points correspond to out-of-memory or out-of-time cases, we do not discuss them here. Overall, across the four factories, most points lie below the diagonal, indicating that h^{sum} typically expands fewer states. The $h^{\text{a-sum}}$ has the similar expansion performance across all factories, it tends to expand substantially more states during the search. We also observe that the heuristic values of $h^{\text{a-sum}}$ drop in a few steps and then rarely decreased further, producing long heuristic plateaus.

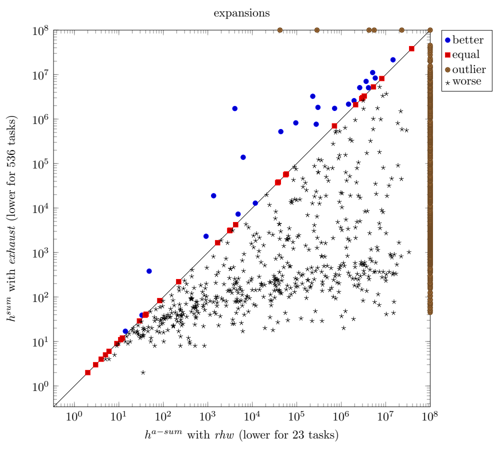
One possible reason why $h^{\text{a-sum}}$ expands many more states could be that a small number of heuristic value drops and long plateaus may mislead the search into a broad area that takes a long time to explore. Another possible reason is our conservative merge rule at join states (Section 3.5). When the same state s' is reached via different paths, we merge the per-path action landmark status (past and future sets). This keeps heuristic values large and easily builds long plateaus. To assess the impact of this choice, we also evaluate an optimistic variant that swaps the operators (union for the past set and intersection for the future set) and compare its effect on $h^{\text{a-sum}}$.

The results are interesting. In the four subfigures of Figure 5.3, although h^{sum} still expands fewer states across all factories, $h^{\text{a-sum}}$ becomes more competitive than before, especially with factory *rhw*, where it achieves its highest number of cases with fewer expansions (lower expansions for 361 tasks). We also observe many ties, with points concentrated on the diagonal, particularly for the factories *exhaust* and *zg* (Table 5.3).

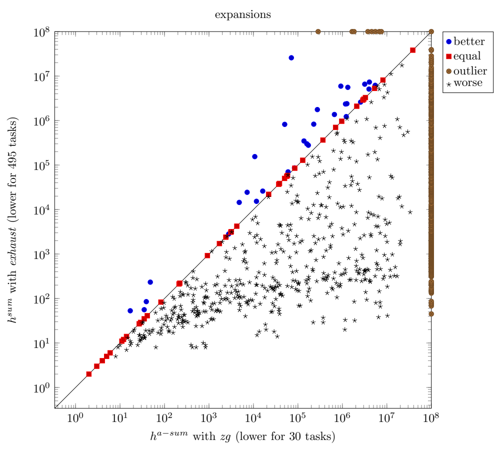
With the optimistic rule, $h^{\text{a-sum}}$ still performs more expansions overall, but creates more



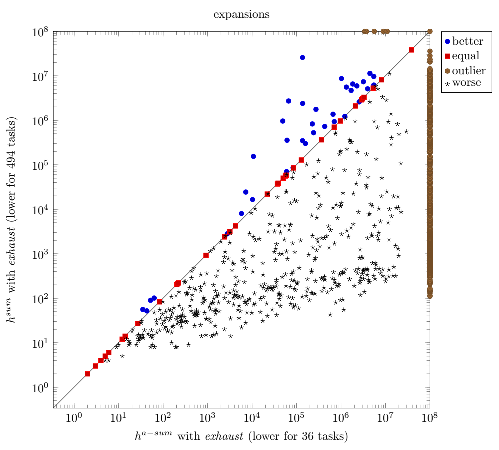
(a) Expansions with factory *hm*



(b) Expansions with factory *rhw*



(c) Expansions with factory *zg*



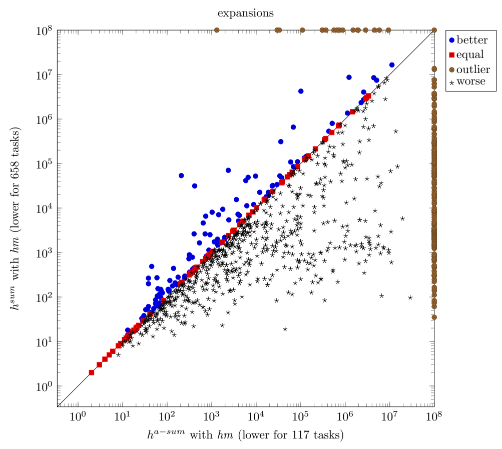
(d) Expansions with factory *exhaust*

Figure 5.2: Expansions across landmark factories.

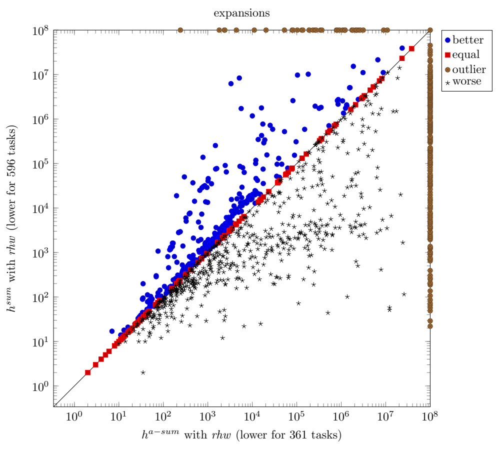
frequent small drops by underestimating each state with lower heuristic value. Those drops give the greedy best first search a bit of focus (it always expands the nodes with the smallest heuristic). The conservative rule removes that micro-guidance. With no drops, the search has to break massive ties and conducts a wider exploration. However, the optimistic merge is not proved to be theoretically valid. From this case, we can only conclude that underestimating the state (via the optimistic merge) can help the search progress.

	hm	*rhw*	*exhaust*	*zg*
Percentage of tasks	7.2%	8.0%	46.1%	45.7%

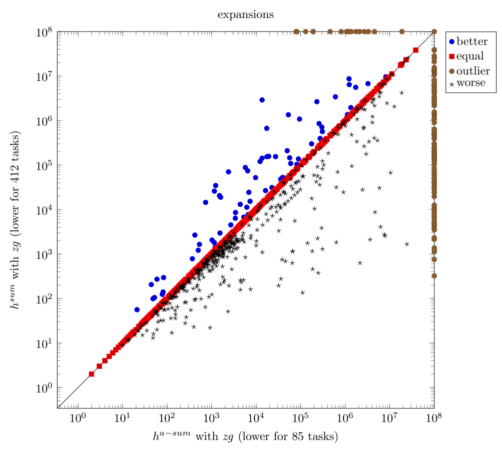
Table 5.3: Percentage of tasks on which the heuristics have identical expansions for landmark factories *hm*, *rhw*, *exhaust* and *zg*.



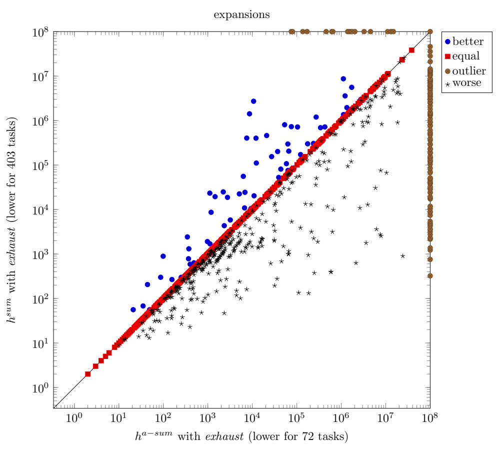
(a) Expansions with factory *hm*



(b) Expansions with factory *rhw*



(c) Expansions with factory *zg*



(d) Expansions with factory *exhaust*

Figure 5.3: Expansions across landmark factories with optimistic merge rule.

5.4 Time and Memory

In this section, we measure $h^{\text{a-sum}}$ (with conservative merge rule) and h^{sum} on the metrics time per expansion, search time score, memory, memory score and memory per expansion.

- Time per expansion (TPE): The time the search algorithm took for each state expansion. We compute TPE $= \frac{T}{E}$, where T is the geometric mean of search time over all domains, E is geometric mean of expansions over all domains. The search time is the time search algorithm took after preprocessing (task parsing, landmark generation, landmark transformation, etc.) until a solution is found or a timeout occurs .

- Search time score: We use time score for each task which was original used for measuring time in the agile tracks of the International Planning Competition[3] in 2023: The score on a solved task is 1 if it was solved within 1 second and 0 if the task was not solved within the time limit of 5 minutes (300 seconds). If the task was solved in

[3] https://ipc2023-classical.github.io

T seconds ($1 \leq T \leq 300$) then its score is $1 - \frac{log(T)}{log(300)}$. We use arithmetic mean of all domain scores as the overall time performance.

- Search memory: The peak memory used for each task in MB.
- Memory per expansion: The memory used for each state expansion in KB.
- Search memory score: The memory score on a task is 1 if the task was solved within 1 MB and 0 if it was not solved within memory limit of 3584MB (3584000KB). If the task was solved within S memory ($1 \leq S \leq 3584000$) then its score is $1 - \frac{log(S)}{log(3584000)}$.

Across all landmark factories, h^{sum} consistently achieves higher search time scores than $h^{\text{a-sum}}$ (first row of Table 5.4). The scores for $h^{\text{a-sum}}$ are tightly clustered between 0.22 and 0.24, indicating little variation across the four factories.
The second row of Table 5.4 breaks down the time per expansion. Here, we see something interesting: $h^{\text{a-sum}}$ consumes less time per expansion (e.g., *hm*: 1.43 vs 4.61, in 10^{-5} seconds). Taken together with the score differences, this indicates that h^{sum}'s better time scores come from fewer expansions, not from lower per-expansion overhead. Even when $h^{\text{a-sum}}$ is faster per expansion, it tends to expand more states overall, leading to a longer search time.
The possible reason for $h^{\text{a-sum}}$'s less time per expansion could be that $h^{\text{a-sum}}$ has fewer action landmarks after the transformation, so each state requires fewer action landmark status checks. Moreover, h^{sum} needs to derive first achievers of unsatisfied fact landmarks on-the-fly when evaluating each state, which is comparatively expensive. In contrast, $h^{\text{a-sum}}$ can use unsatisfied action landmarks directly and their costs have been prepared before the search begins.

Table 5.4: First row: Arithmetic mean of search time score using landmark factories *hm*, *rhw*, *exhaust* and *zg*. Second row: Time per expansion, numbers are multiplied by 10^{-5} and measured in second.

	$h^{\text{a-sum}}$				h^{sum}			
	hm	*rhw*	*exhaust*	*zg*	*hm*	*rhw*	*exhaust*	*zg*
Mean of search time score	0.22	0.24	0.24	0.24	0.45	0.56	0.50	0.52
Time per expansion	1.43	1.19	0.99	0.10	4.61	4.92	2.88	2.64

We have discussed the search time requirement of both heuristics, let us have a look at their memory usages. The overall memory limit is set to 3'584MB. On simple, small tasks (e.g., tasks from domain *movie*), both heuristics use comparable memory. As task complexity increases, however, $h^{\text{a-sum}}$'s memory consumption rises sharply, often reaching the memory limit and is highly sensitive even to small increases in complexity. By contrast, h^{sum} remains moderate and stable, even on large tasks.
The first row of Table 5.5 summarizes the memory score of the two heuristics (higher is better). Across the four factories, $h^{\text{a-sum}}$'s memory scores remain low, it performs best with the factories *exhaust*, *zg* (0.24) and worst with *hm* (0.15). In every factory, h^{sum} achieves a higher memory score on average. This pattern aligns with their expansions: when h^{sum} expands fewer states, it uses less memory. The second row shows the memory usage per

state expansion. Across all factories, $h^{\text{a-sum}}$ uses much less memory per expansion than h^{sum}. For factory *hm*, $h^{\text{a-sum}}$ uses 74.1 KB, compared to 681.8 KB for h^{sum}, about 9.2 times smaller. The reason is similar to the difference in time per expansion. h^{sum} performs the on-the-fly first achievers computation for unsatisfied fact landmarks and maintains more bookkeeping per-state.

Table 5.5: First row records the arithmetic mean of memory score using landmark factories *hm*, *rhw*, *exhaust* and *zg*. Second row records the memory usage per expansion which is measured in KB.

	$h^{\text{a-sum}}$				h^{sum}			
	hm	*rhw*	*exhaust*	*zg*	*hm*	*rhw*	*exhaust*	*zg*
Mean of memory score	0.15	0.17	0.24	0.24	0.27	0.40	0.50	0.52
Memory per expansion	74.1	38.5	27.4	28.4	681.8	203.7	112.0	109.9

We also test $h^{\text{a-sum}}$ with the optimistic merge rule to assess its impact on search time and memory. Table 5.6 shows higher time and memory scores in all four factories compared to the conservative variant. This matches the observation that the optimistic rule typically leads to fewer state expansions, so the search spends less time and uses less memory. However, h^{sum} remains ahead overall.

	hm	*rhw*	*exhaust*	*zg*
Mean of search time score	0.37	0.47	0.46	0.48
Mean of memory score	0.23	0.36	0.33	0.34

Table 5.6: Mean of search time score and memory score for $h^{\text{a-sum}}$ with optimistic merge rule across factories *hm*, *rhw*, *exhaust* and *zg*.

5.5 Coverage

In this section, we measure the coverage of the two heuristics $h^{\text{a-sum}}$ and h^{sum}. Coverage records the number of tasks that search algorithm can solve within the time and memory limits. Table 5.7 reports results for two settings. Coverage 1 uses our implementation with the conservative merge rule. Coverage 2 uses the optimistic merge rule mentioned in Section 3.5, we ran this variant just to study and see behavior, not because it is a valid or recommended method.

Coverage 1 shows that h^{sum} solves many more tasks than $h^{\text{a-sum}}$. The differences range from 511 (*hm*) to 741 (*rhw*), i.e., h^{sum} consistently solves about 1.9 to 2.1 times as many tasks. Here $h^{\text{a-sum}}$ solves roughly a quarter of the tasks, while h^{sum} is around the half. Factory *rhw* achieves the highest coverage for both heuristics, while *hm* yields the lowest. Earlier, we observed that with h^{sum} generally fewer expansions are needed and a better search time score is achieved. More expansions increase the chance of timeouts and memory limit hits. This effect reduces the number of solved tasks, explaining the consistent coverage advantage of h^{sum}.

Coverage 2 shows that $h^{\text{a-sum}}$ improves substantially, yet h^{sum} still leads by 114 to 225 tasks, i.e., about 1.1 to 1.2 times more solved tasks across four factories. With the optimistic rule, $h^{\text{a-sum}}$ performs fewer expansions and consumes less search time and memory compared to the conservative rule, so the number of solved tasks increases under the same resource limits.

	Coverage 1			Coverage 2		
	$h^{\text{a-sum}}$	h^{sum}	difference	$h^{\text{a-sum}}$	h^{sum}	difference
hm	577	1088	511	913	1088	175
rhw	658	1399	741	1174	1399	225
exhaust	637	1280	643	1160	1280	120
zg	641	1294	653	1180	1294	114

Table 5.7: Coverage achieved by $h^{\text{a-sum}}$, h^{sum} and their differences across four different fact landmark factories out of 2276 tasks. Coverage 1: Coverages from $h^{\text{a-sum}}$ with conservative merge rule. Coverage 2: Coverages from $h^{\text{a-sum}}$ with optimistic merge rule.

6 Conclusion

The standard landmark count heuristic sums the costs of the cheapest fact landmark achievers in each state during the search. Our objective was to evaluate whether collecting those achievers in advance as action landmarks, stored in an action landmark graph, changes search behavior comparing to the standard approach. We have implemented a landmark transformer which derives action landmarks from fact landmarks and integrated it into the landmark count heuristic. By design, the transformer converts only initially unsatisfied fact landmarks into action landmarks. Therefore, the number of action landmarks is smaller than the number of fact landmarks. Fact landmarks can be generated first by landmark factories (*hm*, *rhw*, *exhaust*, *zg*), and we compared the our action landmark heuristic variant against the standard fact landmark version in all domains in the satisficing benchmark suit. Our experiments show that, in the initial state, both heuristics have the same behavior which yield the same initial heuristic values. During the search, the action landmark heuristic requires lower time per state expansion, but performs more expansions, resulting in longer search times, higher memory consumption, and thus lower coverages. Across all evaluated metrics, the performance of the action landmark heuristic improves with optimistic merge, but it still remains more expensive than the fact landmark heuristic. Overall, using action landmarks directly within the landmark count heuristic is more expensive and provides weaker search guidance than counting fact landmarks. So pre-computing and using action landmarks directly in the landmark count heuristic does not improve search performance.

6.1 Future work

A meaningful direction for future work is to use action landmarks for other landmark heuristics, such as the LM-Cut heuristic, to assess how well they perform with action landmarks. Another direction is to refine how we evaluate the status of action landmarks during the search. The way action landmarks are updated in each state has a direct impact on the heuristic values, and more accurate update rules might improve both guidance and efficiency. Finally, in our work, action landmarks are obtained by transformation, which requires additional time and depends on the structure of fact landmarks. We could build action landmarks directly on tasks instead of transforming them from fact landmarks.

Bibliography

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

Clemens Büchner, Thomas Keller, Salomé Eriksson, and Malte Helmert. Landmark progression in heuristic search. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling*, pages 70–79, 2023.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, pages 191–246, 2006.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, pages 215–278, 2004.

Emil Keyder, Silvia Richter, and Malte Helmert. Sound and complete landmarks for and/or graphs. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence*, pages 335–340. IOS, 2010.

Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of the Sixth European Conference on Planning*, pages 174–182, 2001.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

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

Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 975–982, 2008.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation. *ICAPS Doctoral Consortium*, pages 156–160, 2003.