Technical Companion to

PDBs Go Numeric: Pattern-Database Heuristics for Simple Numeric Planning

A Self-Contained Guide for CS Undergraduates

Companion to the paper by D. Gnad, L. Alon, E. Weiss, and A. Shleyfman
(AAAI 2025)

Abstract

This document explains every equation, algorithm, definition, theorem, and technical concept in the paper "PDBs Go Numeric" in detail, assuming only undergraduate-level knowledge of AI planning and heuristic search. No prior knowledge of pattern databases, numeric planning, or abstraction-based heuristics is required. Each equation is accompanied by (1) a plain-English description of what it computes, (2) a symbol-by-symbol breakdown, (3) an intuitive explanation of why the computation makes sense, and (4) a concrete numerical example where helpful.

Contents

1 Overview: What This Paper Does and Why It Matters

Pattern Database (PDB) heuristics are among the most powerful admissible heuristics used in optimal classical planning. They work by projecting a planning task onto a small subset of variables (a *pattern*), completely solving that smaller problem, and storing the solution costs in a lookup table. During search, the stored costs serve as lower bounds on the true cost, guiding A* search efficiently toward optimal solutions.

The problem: PDB heuristics have only ever been applied to *classical* planning, where all state variables are finite-domain. Many real-world planning problems involve *numeric* variables—quantities that can be increased or decreased by actions (e.g., fuel level, money, distance). When a planning task includes even a single numeric variable in a pattern, the projected state space becomes potentially *infinite*, and the classical PDB construction algorithm (which enumerates all abstract states) no longer works directly.

This paper's contribution: The authors present the first adaptation of PDB heuristics to *simple numeric planning* (SNP)—planning with numeric variables that change by constant amounts. They introduce several strategies to handle the unbounded nature of numeric state spaces:

1. Restricting the numeric fluents within a projection to bound the state space.
2. Using structure-specific constraints (from domain analysis) to limit value ranges.
3. Partially constructing the abstract state space (bounded exploration).
4. Proving formal properties of LTS homomorphisms for infinite transition systems.

They also adapt pattern *generation* methods (systematic patterns and iPDB hill-climbing) and show experimentally that numeric PDBs can compete with state-of-the-art numeric planners.

Key Idea

The central challenge of this paper is bridging a fundamental gap: PDBs require enumerating all states in a projected task, but numeric planning tasks can have infinitely many states. The paper provides multiple principled approaches to make this enumeration feasible, along with theoretical guarantees that the resulting heuristics remain admissible.

2 Background: Classical Planning and the FDR Formalism

2.1 Finite-Domain Representation (FDR)

Classical AI planning is about finding a sequence of actions to get from an initial state to a goal state. The **finite-domain representation (FDR)** is a compact way to describe a planning task.

Definition 2.1 (FDR Planning Task)**.** An FDR planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$ where:

- $\mathcal{V}$ is a set of **state variables**, each $v \in \mathcal{V}$ with a finite domain $\mathcal{D}(v)$.
- $\mathcal{A}$ is a set of **actions**.
- s_0 is the **initial state** (an assignment of values to all variables).
- G is the **goal description** (a set of conditions on variable values).

Symbol breakdown:

- $\mathcal{V}$: Think of these as the "dimensions" of the world state. For example, in a logistics problem, $\mathcal{V}$ might include variables like `truck-location`, `package1-location`, `package2-location`.
- $\mathcal{D}(v)$: The set of possible values for variable v. For instance, $\mathcal{D}(\texttt{truck-location}) = \{\texttt{cityA}, \texttt{cityB}, \texttt{cityC}\}$. The key property of FDR is that every domain is *finite*.

- s_0: A complete assignment. E.g., $s_0 = \{\texttt{truck-location} = \texttt{cityA}, \texttt{package1-location} = \texttt{cityA}, \ldots\}$.
- G: A partial assignment specifying which variable values must hold in any goal state. E.g., $G = \{\texttt{package1-location} = \texttt{cityC}\}$.

A **state** $s \in \mathcal{S}$ is a complete assignment of values to all variables. The total number of states is $|\mathcal{S}| = \prod_{v \in \mathcal{V}} |\mathcal{D}(v)|$, which is finite (but can be astronomically large).

Numerical Example

Consider a tiny planning task with two variables:
- $\texttt{robot-at} \in \{\texttt{A}, \texttt{B}, \texttt{C}\}$ (3 values)
- $\texttt{door-open} \in \{\texttt{true}, \texttt{false}\}$ (2 values)

The state space has $3 \times 2 = 6$ states. The initial state might be $s_0 = \{\texttt{robot-at} = \texttt{A}, \texttt{door-open} = \texttt{false}\}$, and the goal might be $G = \{\texttt{robot-at} = \texttt{C}\}$.

2.2 Actions: Preconditions, Effects, and Costs

Each action $a \in \mathcal{A}$ is described by a tuple $\langle \text{pre}(a), \text{eff}(a), \text{cost}(a) \rangle$:

- $\text{pre}(a)$: The **precondition**—a partial state that must hold for a to be applicable.
- $\text{eff}(a)$: The **effect**—the changes to variable values after applying a.
- $\text{cost}(a) \geq 0$: The **cost** of performing a.

An action a is **applicable** in state s if $s \models \text{pre}(a)$, meaning s satisfies all preconditions. The result of applying a in s is a new state $s' = sa$ where the variables mentioned in $\text{eff}(a)$ take their new values, and all other variables keep their old values.

Numerical Example

Continuing the robot example:

Action	Precondition	Effect	Cost
`move-A-to-B`	$\texttt{robot-at} = \texttt{A}$	$\texttt{robot-at} := \texttt{B}$	1
`move-B-to-C`	$\texttt{robot-at} = \texttt{B}$, $\texttt{door-open} = \texttt{true}$	$\texttt{robot-at} := \texttt{C}$	1
`open-door`	$\texttt{robot-at} = \texttt{B}$	$\texttt{door-open} := \texttt{true}$	2

From s_0, the optimal plan is: `move-A-to-B`, `open-door`, `move-B-to-C`, with total cost $1 + 2 + 1 = 4$.

2.3 Plans and Optimal Plans

A **plan** π for Π is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ that transforms the initial state s_0 into some goal state $s_* \in S_*$, where $S_* = \{s \in \mathcal{S} \mid s \models G\}$.

The **cost of a plan** is $\text{cost}(\pi) = \sum_{i=1}^{n} \text{cost}(a_i)$.

An **optimal plan** has the minimum cost among all plans. The cost of an optimal plan from state s is denoted $h^*(s)$. If no goal state is reachable from s, we define $h^*(s) := \infty$.

The function h^* is the **perfect heuristic**: it gives the exact remaining cost from any state.

2.4 Labeled Transition Systems (LTS)

The **state space** of a planning task is captured by a **labeled transition system (LTS)**.

Definition 2.2 (Labeled Transition System)**.** An LTS is a tuple $\mathcal{T} = \langle S, L, \text{cost}, T, s_0, S_* \rangle$ where:

- S is the set of states.
- L is the set of labels (action names).
- $\text{cost} : L \to \mathbb{R}^{0+}$ is the cost function on labels.
- $T \subseteq S \times S \times L$ is the set of transitions.
- $s_0 \in S$ is the initial state.
- $S_* \subseteq S$ is the set of goal states.

What this means intuitively: An LTS is just a directed graph where:

- Nodes are states.
- Edges are transitions, each labeled with an action and carrying a cost.
- One node is marked "start" (s_0) and some nodes are marked "goal" (S_*).

A plan corresponds to a path from s_0 to some state in S_*. The optimal plan cost $h^*(s)$ is the shortest-path distance from s to the nearest goal state.

Every planning task Π defines an LTS $\mathcal{T}_\Pi$ where S is the set of all states, L is the set of action names, and $(s, s', a) \in T$ iff action a is applicable in s and $s' = sa$.

Key Idea

The crucial difference between classical and numeric planning: In classical planning, every LTS has finitely many states (since all variable domains are finite). In numeric planning, the LTS can have *infinitely many* states because numeric variables can take unboundedly many values. This infinity is the central challenge that this paper addresses.

3 Background: Pattern Database Heuristics

3.1 What Is a Heuristic?

A **heuristic** $h : S \to \mathbb{R}^{0+} \cup \{\infty\}$ assigns a non-negative estimate to each state, approximating the true optimal cost-to-goal $h^*(s)$.

A heuristic is **admissible** if it never overestimates: $h(s) \leq h^*(s)$ for all s. Admissibility is essential for A* search to guarantee finding optimal solutions.

3.2 The Idea Behind Pattern Databases

A **Pattern Database (PDB)** heuristic works by solving a simpler version of the problem:

1. **Choose a pattern:** Select a subset $P \subseteq \mathcal{V}$ of the state variables.
2. **Project:** Create a simpler planning task $\Pi|_P$ by keeping only the variables in P and ignoring all others. This is called the **projection** of Π onto P.
3. **Solve the projection:** Compute the optimal cost-to-goal $h|_P(s|_P)$ for every abstract state $s|_P$ in the projected task.
4. **Store results:** Save all these costs in a lookup table (the "database" in "pattern database").
5. **Use as heuristic:** During A* search on the original task, for any state s, project it to $s|_P$ and look up $h|_P(s|_P)$.

Why Projections Give Admissible Heuristics

When we remove variables from a planning task, we can only make the problem *easier* (or equally hard), never harder. Any plan for the original task is also a valid plan for the projected task (just ignore the missing variables' effects). Therefore, the optimal cost in the projection is $\leq$ the optimal cost in the original: $h|_P(s|_P) \leq h^*(s)$. This means PDB heuristics are always admissible.

Notation:

- $P \subseteq \mathcal{V}$: the **pattern**, a subset of variables.
- $s|_P$: the **projection** of state s onto P—the partial state that only keeps the values of variables in P.
- $\Pi|_P$: the projected planning task.
- $\mathcal{T}|_P$: the projected LTS (transition system of $\Pi|_P$).
- $h|_P(s|_P)$: the optimal cost-to-goal in the projected task, starting from abstract state $s|_P$.

A PDB for the Robot Example

Recall our robot task with variables {`robot-at`, `door-open`}.
Pattern $P = \{$`robot-at`$\}$: We project away `door-open`. In the projected task, the action `move-B-to-C` no longer requires `door-open` = `true` (that variable does not exist in the projection), and `open-door` has no effect (it only changed a variable not in P).
The projected LTS has 3 states: `A`, `B`, `C`. The optimal costs from each state to the goal (`robot-at` = `C`) are:

Abstract state $s\|_P$	$h\|_P(s\|_P)$
`A`	2 (move A→B, then B→C)
`B`	1 (move B→C)
`C`	0 (already at goal)

This PDB heuristic gives $h|_P(s_0|_P) = h|_P(\texttt{A}) = 2$. The true optimal cost is 4, so the heuristic underestimates (as expected for an admissible heuristic). The underestimate occurs because the projection ignores the cost of opening the door.
Pattern $P' = \{$`door-open`$\}$: This projection has 2 states. The goal does not mention `door-open`, so the goal is trivially satisfied in any state, giving $h|_{P'}(s|_{P'}) = 0$ for all states—a useless heuristic!
This illustrates that the choice of pattern matters enormously.

3.3 Regression Search for PDB Construction

The standard method for computing PDB heuristics uses **regression search** (backward search):

1. Start from all goal states in the projected task.
2. Work backwards: for each state, compute the cheapest path *from* that state *to* the goal.
3. Store all computed distances in a hash table.

Why regression? Regression computes distances from all states to the goal in a single exploration, starting from the goal states. Unlike forward search (which would need to be run from every possible start state), regression needs only one pass. It also automatically identifies dead ends (states from which no goal is reachable).

Regression Fails with Numeric Variables

Regression requires iterating backwards from goal states. With numeric variables, goal conditions often involve inequalities (e.g., `fuel` ≥ 10), which are satisfied by infinitely many abstract states. This means the set of goal states is infinite, and regression cannot enumerate them as starting points. This is one of the key reasons the paper uses *progression* (forward search) instead.

3.4 Combining Multiple PDBs: The Canonical Heuristic

A single PDB uses only a small subset of variables and therefore provides a weak lower bound. To get stronger heuristics, we combine multiple PDBs.

Maximum over PDBs: Given a set H of PDB heuristics, the simplest combination is:

$$h^{\max}(s) = \max_{h \in H} h(s)$$

This is admissible (the max of admissible heuristics is admissible), but it wastes information—it only uses the single best PDB for each state.

Additive PDBs: A more powerful approach is to *add* PDB values:

$$h^{\text{add}}(s) = \sum_{h \in H} h(s)$$

This is admissible only if the PDBs are **additive**—meaning no action affects variables in more than one pattern simultaneously.

Definition 3.1 (Disjoint-Additive Patterns)**.** Two patterns P_1 and P_2 are **disjoint-additive** if there is no action $a \in \mathcal{A}$ that affects at least one variable in P_1 and at least one variable in P_2.

The Canonical Heuristic h^C:

Given a collection of patterns C, the **canonical heuristic** finds the maximum over all additive combinations:

$$h^C = \max_{M \in \mathcal{M}(C)} \sum_{P \in M} h|_P \tag{1}$$

Symbol breakdown:

- C: a collection (set) of patterns.
- $\mathcal{M}(C)$: the set of all **maximal disjoint-additive subsets** of C. Each $M \in \mathcal{M}(C)$ is a subset of patterns that can be safely summed.
- $\sum_{P \in M} h|_P$: the sum of PDB values for patterns in M.
- The outer max picks the best additive combination for each state.

Numerical Example

Suppose we have four patterns $C = \{P_1, P_2, P_3, P_4\}$ and the additive subsets are: $M_1 = \{P_1, P_3\}$, $M_2 = \{P_2, P_4\}$, $M_3 = \{P_1, P_4\}$.
For a state s:

- $h|_{P_1}(s) = 3$, $h|_{P_2}(s) = 5$, $h|_{P_3}(s) = 2$, $h|_{P_4}(s) = 4$.
- M_1: $3 + 2 = 5$.
- M_2: $5 + 4 = 9$.

- M_3: $3 + 4 = 7$.
- $h^C(s) = \max(5, 9, 7) = 9$.

The canonical heuristic gives 9, much better than any single PDB's value. Since h^C is a max over admissible sums, it is itself admissible.

4 Background: Numeric Planning

4.1 From Classical to Numeric Planning

Classical planning assumes all variables have finite domains. **Numeric planning** extends this by allowing **numeric state variables**—variables whose domain is the integers $\mathbb{Z}$ (or a subset thereof).

The paper works with the **integer-restricted task (IRT)** formalism, which is a variant of the restricted task (RT) formalism. In an IRT, all numeric variable values are integers.

Definition 4.1 (Integer-Restricted Task (IRT))**.** An IRT is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$ where:

- $\mathcal{V} = \mathcal{V}_n \cup \mathcal{V}_p$ is a set of **numeric variables** $\mathcal{V}_n$ and **finite-domain (propositional) variables** $\mathcal{V}_p$.
- For $v \in \mathcal{V}_p$: the domain $\mathcal{D}(v)$ is finite (as in classical planning).
- For $v \in \mathcal{V}_n$: the domain $\mathcal{D}(v) = \mathbb{Z}$ (all integers).
- $\mathcal{A}$ is a set of actions with preconditions, effects, and costs.
- s_0 is the initial state.
- G consists of propositional conditions G_p and numeric conditions G_n.

What makes IRTs special compared to general numeric planning:

- **Numeric effects are constant additions:** An action can only change a numeric variable v by a constant amount: $v \mathrel{+}= m$ where $m \in \mathbb{Z} \setminus \{0\}$. No effects like $x := y$ or $x := x \times 2$ are allowed.
- **Conditions are linear:** Numeric preconditions and goals have the form $v \bowtie w$ where $\bowtie \in \{>, \geq, =, \leq, <\}$, $v \in \mathcal{V}_n$, and $w \in \mathbb{Z}$.

A Simple Numeric Planning Task

Consider a delivery problem where a truck must deliver packages and has limited fuel:
Variables:

- `truck-at` $\in \{$`depot`, `city`$\}$ (finite-domain)
- `package-at` $\in \{$`depot`, `city`, `truck`$\}$ (finite-domain)
- `fuel` $\in \mathbb{Z}$ (numeric, initially 10)

Actions:

Action	Precondition	Effect	Cost
`drive`	`truck-at` $=$ `depot`, `fuel` ≥ 3	`truck-at` $:=$ `city`, `fuel` $-= 3$	1
`drive-back`	`truck-at` $=$ `city`, `fuel` ≥ 3	`truck-at` $:=$ `depot`, `fuel` $-= 3$	1
`load`	`truck-at` $=$ `depot`, `package-at` $=$ `depot`	`package-at` $:=$ `truck`	1
`unload`	`truck-at` $=$ `city`, `package-at` $=$ `truck`	`package-at` $:=$ `city`	1
`refuel`	`truck-at` $=$ `depot`	`fuel` $+= 5$	2

Goal: `package-at` $=$ `city`.
Initial state: $s_0 = \{$`truck-at` $=$ `depot`, `package-at` $=$ `depot`, `fuel` $= 10\}$.
The optimal plan is: `load`, `drive`, `unload`, with cost 3. The fuel starts at 10, and we only

need 3, so refueling is unnecessary.
Note that `fuel` is a numeric variable that can in principle take any integer value. If we project onto a pattern that includes `fuel`, the abstract state space has infinitely many states (one for each possible fuel level).

4.2 States and Conditions

A **state** s in an IRT assigns a value to every variable:

$$s = \langle s_p, s_n \rangle \quad \text{where} \quad s_p \in_{v \in \mathcal{V}_p} \mathcal{D}(v), \quad s_n \in_{v \in \mathcal{V}_n} \mathbb{Z}$$

We write $s[v]$ for the value of variable v in state s. A **fact** $\langle v, d \rangle$ states that variable v has value d. We say $s \models \langle v, d \rangle$ iff $s[v] = d$.

A **partial state** s^{pt} assigns values to only a subset of variables; $s^{pt} \subseteq s$ means s agrees with s^{pt} on all variables that s^{pt} assigns.

Numeric conditions have the form $v \bowtie w$:

- `fuel` ≥ 3 means "the fuel level is at least 3."
- `distance` $= 0$ means "the distance is exactly 0."
- `money` > 100 means "we have more than 100 units of money."

State s satisfies $v \bowtie w$ (written $s \models v \bowtie w$) iff $s[v] \bowtie w$.

Infinite State Spaces

The total number of states in an IRT is:

$$|\mathcal{S}| = \left(\prod_{v \in \mathcal{V}_p} |\mathcal{D}(v)| \right) \times |\mathbb{Z}|^{|\mathcal{V}_n|}$$

If there is even a single numeric variable ($|\mathcal{V}_n| \geq 1$), this is infinite. This is the fundamental obstacle to applying PDB heuristics in numeric planning.

4.3 The Causal Graph

The **causal graph (CG)** captures dependencies between variables.

Definition 4.2 (Causal Graph)**.** The causal graph of an IRT $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$ is a directed graph $CG(\Pi) = \langle \mathcal{V}, \mathcal{E} \rangle$ where $(u, v) \in \mathcal{E}$ if $u \neq v$ and there exists an action $a \in \mathcal{A}$ such that $u \in \text{vars}(\text{pre}(a)) \cup \text{vars}(\text{eff}(a))$ and $v \in \text{vars}(\text{eff}(a))$.

In plain English: There is an edge from variable u to variable v if changing v's value (via some action) might require knowing or changing u's value.

Why the CG matters for PDBs: The CG helps determine which patterns can yield bounded projections. If a pattern P is chosen such that no variable outside P affects variables inside P (according to the CG), then the projection $\Pi|_P$ is self-contained and more likely to have a bounded state space.

Numerical Example

In the delivery example, the CG has edges:

- `truck-at` → `fuel` (driving requires and changes truck location, and changes fuel)
- `truck-at` → `package-at` (loading/unloading requires truck location)
- `fuel` → `truck-at` (driving requires sufficient fuel to change location)
- Note that there is no self-loop `package-at` → `package-at`: the CG definition requires $u \neq v$, so self-loops are excluded.

The CG reveals that `fuel` and `truck-at` are tightly coupled: each depends on the other. A pattern including one should likely include the other.

5 The Core Challenge: PDBs for Numeric Variables

5.1 Why Classical PDB Construction Fails

In classical planning, PDB construction proceeds as follows:

1. Project the task onto pattern P by removing variables $\mathcal{V} \setminus P$.
2. The projected state space $\mathcal{S}|_P$ is finite (product of finite domains).
3. Enumerate all abstract states and compute their optimal costs by regression from goal states.
4. Store in a hash table: $s|_P \mapsto h|_P(s|_P)$.

With numeric variables in P, step 2 fails: $\mathcal{S}|_P$ may be infinite. Step 3 also fails: regression from goal states is impossible when goals involve inequalities over numeric variables (infinitely many goal states to start from).

Numerical Example

Consider projecting the delivery task onto $P = \{\texttt{fuel}\}$. The projected state space has one state for every integer value of fuel: $\ldots, -2, -1, 0, 1, 2, \ldots$ This is countably infinite. The goal does not mention `fuel`, so in the projection, every state is a goal state (trivially satisfied). Hence $h|_P(s|_P) = 0$ for all states—not useful.
Now consider $P = \{\texttt{truck-at}, \texttt{fuel}\}$. The projection has states (l, f) where $l \in \{\texttt{depot}, \texttt{city}\}$ and $f \in \mathbb{Z}$. There are $2 \times |\mathbb{Z}| = \infty$ states. The `drive` action requires `fuel` ≥ 3 and decreases fuel by 3, and `refuel` increases fuel by 5. Starting from $(\texttt{depot}, 10)$, the fuel can grow without bound via repeated refueling.

5.2 The Paper's Approaches to Bounding the State Space

The paper introduces three main strategies:

1. **Restricting numeric fluents:** Remove some or all numeric variables from the pattern, reducing the infinite state space to a finite one. The extreme case (removing all numeric variables) gives a "finite-domain PDB" that ignores numeric aspects entirely.
2. **Exploiting known bounds from CG analysis:** Use structural analysis of the causal graph to determine which patterns yield bounded projections. Prior work has established conditions under which numeric variable ranges can be bounded.
3. **Partial construction (bounded exploration):** Do not try to enumerate the entire (infinite) abstract state space. Instead, explore only up to N states using progression from the initial abstract state, and use approximations for unexplored states.

6 LTS Homomorphisms and Admissibility for Infinite LTSs

Before diving into the PDB construction algorithm, the paper establishes a theoretical framework for working with infinite transition systems. This section defines *LTS homomorphisms*—structure-preserving maps between transition systems—and proves that they yield admissible heuristics even when the LTSs are infinite.

6.1 LTS Homomorphisms

Definition 6.1 (LTS Homomorphism)**.** Let $\mathcal{T} = \langle S, L, \text{cost}, T, s_0, S_* \rangle$ and $\mathcal{T}' = \langle S', L', \text{cost}', T', s'_0, S'_* \rangle$ be two LTSs. A map $\alpha : S \cup L \to S' \cup L'$ is an **LTS homomorphism** if:

1. $\alpha(S) \subseteq S'$ and $\alpha(L) \subseteq L'$. (States map to states, labels map to labels.)
2. $\alpha(T) \subseteq T'$, where $\alpha((s, s', l)) = (\alpha(s), \alpha(s'), \alpha(l))$. (Transitions are preserved.)
3. $\text{cost}(\alpha(l)) \leq \text{cost}(l)$ for each $l \in L$. (Costs do not increase.)
4. $\alpha(s_0) = s'_0$ and $\alpha(S_*) \subseteq S'_*$. (Initial and goal states are preserved.)

In plain English: A homomorphism α maps one transition system into another while:

- Preserving the structure: if there is a transition in $\mathcal{T}$, there must be a corresponding transition in $\mathcal{T}'$.
- Not increasing costs: the mapped transition is at most as expensive.
- Mapping the start to the start and goals to goals.

Why Homomorphisms Give Admissible Heuristics

If α is a homomorphism from $\mathcal{T}$ to $\mathcal{T}'$, then any plan π in $\mathcal{T}$ can be "translated" to a plan $\alpha(\pi)$ in $\mathcal{T}'$ with cost $\leq \text{cost}(\pi)$. Therefore, the optimal cost in $\mathcal{T}'$ (starting from $\alpha(s)$) is a lower bound on the optimal cost in $\mathcal{T}$ (starting from s). This is exactly what we need for an admissible heuristic.

Theorem 6.1 (Homomorphisms preserve plans)**.** *For a homomorphism α of an LTS $\mathcal{T}$ and a plan π for $\mathcal{T}$, $\alpha(\pi)$ is a plan for $\alpha(\mathcal{T})$ with* $\text{cost}(\alpha(\pi)) \leq \text{cost}(\pi)$.

Why this matters: This theorem immediately implies that $h_\alpha(s) := \text{cost}^*(\alpha(s))$—the optimal cost-to-goal in the image LTS starting from the mapped state—is an admissible heuristic for $\mathcal{T}$. The proof is straightforward: an optimal plan π^*_s from s maps to a plan $\alpha(\pi^*_s)$ from $\alpha(s)$, so $h_\alpha(s) \leq \text{cost}(\alpha(\pi^*_s)) \leq \text{cost}(\pi^*_s) = h^*(s)$.

Connection to projections: A projection $\Pi|_P$ defines a homomorphism α from $\mathcal{T}_\Pi$ to $\mathcal{T}|_P$ by $\alpha(s) = s|_P$ and $\alpha(a) = a$. Transitions are preserved because if a is applicable in s then a is applicable in $s|_P$ (the preconditions on variables in P are satisfied, and preconditions on variables outside P are simply dropped). Costs are preserved exactly ($\text{cost}(\alpha(a)) = \text{cost}(a)$).

Numerical Example

In the delivery example, the projection onto $P = \{\texttt{truck-at}\}$ defines a homomorphism:

- $\alpha((\texttt{depot}, \texttt{depot}, 10)) = \texttt{depot}$
- $\alpha((\texttt{depot}, \texttt{depot}, 7)) = \texttt{depot}$ (different fuel levels map to the same abstract state)
- $\alpha(\texttt{drive}) = \texttt{drive}$, $\alpha(\texttt{load}) = \texttt{load}$, etc.

The transition $(\texttt{depot}, 10) \xrightarrow{\texttt{drive}} (\texttt{city}, 7)$ in the original maps to $\texttt{depot} \xrightarrow{\texttt{drive}} \texttt{city}$ in the projection.

6.2 Orthogonal Projections and Additivity

To combine multiple PDB heuristics by addition, we need them to be *additive*. The paper adapts the classical notion of *orthogonal projections* to IRTs.

Definition 6.2 (Orthogonal Homomorphisms)**.** Let $\mathcal{T}$ be an LTS, and let α and β be homomorphisms for $\mathcal{T}$. We say α and β are **orthogonal** if for every transition $(s, s', l) \in T$ in every plan for $\mathcal{T}$ it holds that $\alpha(s) = \alpha(s')$ or $\beta(s) = \beta(s')$.

In plain English: Two projections are orthogonal if every transition (action application) changes at most one of the two projections. In other words, no single action can simultaneously "move" the state in both abstract spaces.

Proposition 6.1 (Orthogonal heuristics are additive)**.** *Let α and β be orthogonal LTS homomorphisms and let s be reachable from s_0. Then $h_\alpha(s) + h_\beta(s) \leq h^*(s)$.*

Proof intuition: Take an optimal plan π_s^*. By orthogonality, each transition in π_s^* either changes α's image or β's image, but not both. Split π_s^* into transitions that change α (call them π_1) and those that change β (call them π_2). Then $\alpha(\pi_1)$ is a plan for $\alpha(\mathcal{T})$ and $\beta(\pi_2)$ is a plan for $\beta(\mathcal{T})$, and their costs sum to at most $\text{cost}(\pi_s^*)$. Therefore $h_\alpha(s) + h_\beta(s) \leq \text{cost}(\pi_1) + \text{cost}(\pi_2) \leq \text{cost}(\pi_s^*) = h^*(s)$.

Numeric Variables Break Orthogonality

In classical planning, two patterns P and P' are orthogonal if no action affects variables in both patterns. In numeric planning, this condition is harder to satisfy because of the following result:

Proposition 6.2 (Numeric goal variables prevent orthogonality)**.** *Let Π be an IRT with a numeric variable $x \in \mathcal{V}_n$ such that $s_0[x]$ does not satisfy $G|_{\{x\}}$ (the goal condition on x). Let α and β be projections on patterns P and P' with $x \in P \cap P'$. Then α and β are not orthogonal.*

In plain English: If a numeric variable x appears in two patterns and the goal requires changing x from its initial value, then the two projections cannot be orthogonal. This is because any action that changes x will change the abstract state in *both* projections simultaneously (numeric variables have no self-loops—every change to x produces a genuinely new state).

Why this is unique to numeric planning: In classical planning, a finite-domain variable v can have self-loops: if action a changes v from value 1 to value 1, the transition is a self-loop. But for numeric variables, $v \mathrel{+}= m$ with $m \neq 0$ always produces a new value, so there are never self-loops on actions that affect numeric variables.

Practical consequence: This means it is impossible to have a numeric goal variable appear in two patterns that are combined additively. The canonical heuristic must ensure that numeric goal variables appear in at most one pattern per additive group.

7 Algorithm 1: Numeric PDB Construction

The paper's central algorithmic contribution is Algorithm 1, which constructs a PDB heuristic for a numeric planning task by combining *progression* (forward search from the initial state) with *regression* (backward cost computation from goal states).

7.1 The Two-Phase Approach

Progression + Regression

Classical PDB construction uses pure regression (backward search from goals). This fails for numeric tasks because there are infinitely many goal states. The paper's solution is a two-phase approach:

1. **Phase 1 (Progression):** Starting from the initial abstract state $s_0|_P$, explore the abstract state space *forward* up to N states. This discovers which abstract states are actually reachable.
2. **Phase 2 (Regression):** Among the explored states, identify goal states. Then compute shortest-path distances from all explored states *backward* to these goal states.

Why progression first? The initial state $s_0|_P$ is a single, concrete state—a well-defined starting point. By exploring forward from it, we discover only reachable abstract states. Since we limit exploration to N states, the process is guaranteed to terminate.

Why regression second? Once we have identified the finite set of reachable goal states, regression efficiently computes the optimal cost from every explored state to the nearest goal.

7.2 Detailed Walkthrough of Algorithm 1

Algorithm 1: Numeric PDB Construction

Input: IRT $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$, pattern P, max # of states N
Output: PDB heuristic $h_{\text{PDB}}(s)$

Phase 1: Progression (forward exploration)

$\texttt{open}.\mathit{insert}(s_0|_P, 0)$, $\texttt{closed} \leftarrow \emptyset$, $\texttt{goals} \leftarrow \emptyset$
while $\texttt{open} \neq \emptyset$ **and** $|\texttt{closed}| + |\texttt{open}| < N$ **do**
  $s \leftarrow \texttt{open}.\mathit{pop\_min}()$
  $\texttt{closed} \leftarrow \texttt{closed} \cup \{s\}$
  if $s \models G|_P$ **then**
    $\texttt{goals} \leftarrow \texttt{goals} \cup \{s\}$
  for each $a \in \mathcal{A} : s \models \text{pre}(a)|_P$ **do**
    $s' \leftarrow sa$
    $s'.\mathit{parents} \leftarrow s'.\mathit{parents} \cup \{(s, a)\}$
    $g(s') \leftarrow g(s) + \text{cost}(a)$
    if $s' \notin \texttt{closed}$ **then**
      $\texttt{open}.\mathit{insert}(s', g(s'))$

Phase 2: Regression (backward cost computation)

$\texttt{backward} \leftarrow \emptyset$
for each $s \in \texttt{goals}$ **do**
  $\texttt{backward}.\mathit{insert}(s, 0)$
for each $s \in \texttt{open}$ **do**
  $h_{\text{PDB}}(s) \leftarrow$ Approximate distance to goal
  $\texttt{backward}.\mathit{insert}(s, h_{\text{PDB}}(s))$
while $\texttt{backward} \neq \emptyset$ **do**
  $s' \leftarrow \texttt{backward}.\mathit{pop\_min}()$
  for each $(s, a) \in s'.\mathit{parents}$ **do**
    $h_{\text{PDB}}(s) \leftarrow \min(h_{\text{PDB}}(s),\ h_{\text{PDB}}(s') + \text{cost}(a))$
    $\texttt{backward}.\mathit{insert}(s, h_{\text{PDB}}(s))$

Line-by-line explanation:

Lines 1–12 (Progression phase):

- **Line 1:** Initialize an `open` priority queue with the initial abstract state $s_0|_P$ at cost 0. The `closed` set (fully expanded states) and `goals` set are empty.
- **Line 2:** Continue exploring as long as there are states to explore and we have not exceeded the budget N.
- **Line 3:** Pop the cheapest state from `open` (like Dijkstra's algorithm).
- **Line 4:** Mark it as fully expanded (closed).
- **Lines 5–6:** If this state satisfies the goal conditions (restricted to pattern P), record it as a goal state.
- **Lines 7–12:** For each applicable action, compute the successor state s', record the parent link (for regression later), and add s' to `open` if not already closed.

Lines 13–23 (Regression phase):

- **Lines 14–15:** Goal states get $h_{\text{PDB}} = 0$ (they are already at the goal).
- **Lines 16–18:** States still in `open` (explored but not fully expanded) are "fringe" states. Their heuristic value is approximated—the paper uses the minimum action cost as a lower bound.
- **Lines 19–23:** Process states in order of increasing h_{PDB} value. For each state s', update all its parents: the cost through s' is $h_{\text{PDB}}(s') + \text{cost}(a)$, and we keep the minimum. This is essentially Dijkstra's algorithm running backwards along the parent links.

Running Algorithm 1

Consider projecting our delivery task onto $P = \{\texttt{truck-at}, \texttt{fuel}\}$ with $N = 8$.
Phase 1 (Progression):
Initial abstract state: $(\texttt{depot}, 10)$, $g = 0$.
Expand $(\texttt{depot}, 10)$: applicable actions are `drive` ($\rightarrow (\texttt{city}, 7)$, cost 1) and `refuel` ($\rightarrow (\texttt{depot}, 15)$, cost 2) and `load` ($\rightarrow (\texttt{depot}, 10)$, cost 1, but precondition on `package-at` is projected away, so this is applicable but stays at same state).
Actually, in the projection onto $P = \{\texttt{truck-at}, \texttt{fuel}\}$, actions that only affect `package-at` (like `load`, `unload`) become no-ops (self-loops) because their effects are on variables outside P.
So the meaningful transitions from $(\texttt{depot}, 10)$ are:

- `drive` $\rightarrow (\texttt{city}, 7)$, cost 1
- `refuel` $\rightarrow (\texttt{depot}, 15)$, cost 2

Continue expanding states in cost order until we reach $N = 8$ states or run out of states to explore. The exploration builds a graph of reachable abstract states.
Phase 2 (Regression):
The goal is `package-at` = `city`. Since `package-at` $\notin P$, the goal projected onto P is empty—every abstract state is a goal state! So $h_{\text{PDB}}(s) = 0$ for all explored states.
This illustrates why the choice of pattern matters: pattern $P = \{\texttt{truck-at}, \texttt{fuel}\}$ misses the goal variable entirely, yielding a trivial heuristic. A better choice would include `package-at` in the pattern.

7.3 Bounding the Abstract State Space

The paper defines a finite sub-LTS $\mathcal{T}'$ over the explored states $S' = S_E \cup S_F$:

- S_E: the **expanded** states (in `closed`)—states that have been fully explored with all successors generated.
- S_F: the **fringe** states (in `open`)—states that have been discovered but not fully expanded.

This sub-LTS $\mathcal{T}'$ satisfies several important properties:

1. $T' \subseteq T$: all transitions in $\mathcal{T}'$ are transitions in $\mathcal{T}$.
2. $S'_* \subseteq S_*$: goal states in the sub-LTS are goal states in the original.
3. All states in $S_E \cup S_F$ are reachable from s_0.
4. All successors of states in S_E are in $\mathcal{T}'$ (either in S_E or S_F).
5. States in S_F have no outgoing edges in $\mathcal{T}'$.
6. If $s_* \in S'_*$ then $s_* \in S_E$ (goal states that are reached are always expanded, not left on the fringe).

The heuristic for states in this sub-LTS is:

$$h_{\text{PDB}}(s) = \min_{s' \in S_F} \{\text{cost}^*(s, s') + d(s')\} \tag{2}$$

Symbol breakdown:

- $\text{cost}^*(s, s')$: the optimal cost of reaching fringe state s' from state s within the sub-LTS.
- $d(s')$: an admissible estimate of the cost from fringe state s' to the goal. If s' is a goal state, $d(s') = 0$. Otherwise, $d(s')$ is the minimum action cost (a simple lower bound).
- The min is over all fringe states, finding the cheapest path from s to the boundary of the explored region, plus the estimated cost beyond.

Proposition 7.1 (h_{PDB} is admissible)**.** *h_{PDB} is admissible on the states in $S_E \cup S_F$.*

Proof sketch: For fringe states $s \in S_F$: no state in $S_E \cup S_F$ can reach s from within (fringe states have no outgoing edges in $\mathcal{T}'$), so $h_{\text{PDB}}(s) = d(s)$ which is admissible by definition.

For expanded states $s \in S_E$: the optimal plan π^*_s from s to a goal must eventually leave the explored region through some fringe state s^*_F. The cost of π^*_s is at least $\text{cost}^*(s, s^*_F) + h^*(s^*_F)$. Since $d(s^*_F) \leq h^*(s^*_F)$ (admissible estimate), we get:

$$h^*(s) = \text{cost}^*(s, s^*_F) + h^*(s^*_F) \geq \min_{s' \in S_F} \{\text{cost}^*(s, s') + d(s')\} = h_{\text{PDB}}(s).$$

7.4 Handling Missing Abstract States (Lookup Misses)

A consequence of partial construction is that during the main A* search, we may encounter a concrete state s whose abstract counterpart $s|_P$ was *not* generated during PDB construction. This happens when $s|_P$ lies outside the explored region $S_E \cup S_F$.

The paper's solution: Return $h_{\text{PDB}}(s) = 0$ for such states. This is trivially admissible (0 never overestimates), but uninformative. The rationale is that large PDB collections are typically used, so if one PDB returns 0 for a state, another PDB is likely to provide useful guidance.

Lookup Misses Can Degrade Performance

If many states during the main search map to unexplored regions of the PDB, the heuristic becomes effectively useless for those states. This is why the state limit N is an important

parameter: too small and many states will be missed; too large and PDB construction becomes expensive. The paper reports that iPDB (which uses many small patterns) has significantly fewer misses than single-pattern greedy PDB.

8 Exploiting Tractable IRT Fragments

8.1 Bounded Domains from CG Structure

Rather than exploring up to an arbitrary limit N, we can sometimes *prove* that a pattern's projected state space is finite.

The idea: Prior work has established conditions on the causal graph that guarantee bounded numeric variable domains in the projection. If we select a pattern P such that the projection $\Pi|_P$ satisfies these conditions, the abstract LTS $\mathcal{T}|_P$ is guaranteed to be finite, and we can compute the PDB exactly (no approximation needed).

Specifically, the paper uses results from Shleyfman, Gnad, and Jonsson (2023) that provide "recipes" for computing bounded domain intervals for numeric variables based on CG structure.

Selecting Patterns That Guarantee Finiteness

The key insight is: not all patterns that include numeric variables lead to infinite projections. If the CG structure of the pattern is "well-behaved" (e.g., acyclic dependencies among numeric variables, or variables with bounded effects), the projection may have a finite state space. The paper uses CG analysis to identify such patterns and prioritize them during pattern generation.

8.2 Handling Out-of-Bound States

Even when bounds on numeric variable ranges are computed, they are *approximations*. The bounds consider only the variables inside the pattern, ignoring dependencies on variables outside it.

A numeric variable $v \in P$ might depend cyclically on a variable $u \notin P$. In the original task, u might constrain v's range, but in the projection (where u is ignored), v might take values outside the predicted bounds.

The paper addresses this by **discarding PDBs for out-of-bound states**: if a state s maps to an abstract state $s|_P$ where some numeric variable exceeds the predicted bounds, the PDB returns 0.

9 Pattern Generation for Numeric Variables

Having a single PDB is usually not enough for strong heuristic guidance. The paper adapts two established pattern generation methods from classical planning:

9.1 Systematic Pattern Generation (sysC)

Systematic pattern generation enumerates all "interesting" patterns with k variables, focusing on patterns that include goal variables and causally related variables.

In the classical setting, variables are considered "interesting" if they appear in goals or are connected to goal variables through the causal graph.

Adaptation for numeric planning: The paper constructs systematic patterns using the numeric CG, starting with goal variables and adding causally related variables. For the numeric adaptation:

- The paper uses patterns with two variables (which performed best empirically).
- The number of abstract states is limited to 50,000 per pattern.
- Patterns are combined with the canonical heuristic h^C.

9.2 Incremental PDB Hill-Climbing (iPDB)

iPDB is a more sophisticated method that uses hill-climbing to find good pattern collections:

1. Start with a collection of small patterns (e.g., one variable each for goal variables).
2. Evaluate the current heuristic quality by running a sample of searches.
3. Try extending each pattern by adding one variable (chosen from the CG neighborhood).
4. If the extended pattern improves heuristic quality, keep the extension.
5. Repeat until no improvement is found or time/size limits are reached.

Adaptations for numeric planning:

1. **Pattern size bound:** In classical planning, iPDB bounds pattern size by the domain-size product: $\prod_{v \in P} |\mathcal{D}(v)| \leq B$ for some budget B. For numeric variables, $|\mathcal{D}(v)| = |\mathbb{Z}| = \infty$, so this bound is meaningless.

   The paper's solution: Instead of bounding the domain-size product, bound the *relevant range of values* for each numeric variable. Using the estimated numeric intervals (ENI) $[M_x^-, M_x^+]$ and maximum additive constants C_x^+ and C_x^-, estimate the number of reachable values as:

   $$\frac{M_x^+ + C_x^+ - (M_x^- - C_x^-)}{\text{gcd(effects on } x)} + 1 \tag{3}$$

   Symbol breakdown:

   - $[M_x^-, M_x^+]$: the estimated range of values for numeric variable x based on domain analysis.
   - C_x^+: the maximum amount by which any single action can increase x.
   - C_x^-: the maximum amount by which any single action can decrease x (as a positive number).
   - $M_x^- - C_x^-$: the lowest value x might reach (lower bound minus maximum decrease).
   - $M_x^+ + C_x^+$: the highest value x might reach (upper bound plus maximum increase).
   - gcd(effects on x): the greatest common divisor of all constant effects on x. Since effects are integer additions, the reachable values form an arithmetic progression with this step size.
   - Dividing the range by the GCD gives the number of distinct reachable values (approximately).

2. **State limit during construction:** Each PDB is limited to at most N abstract states (default: 100,000 for greedy, 50,000 for systematic patterns, 10,000 for iPDB). The hill-climbing process itself is limited to 15 minutes.

3. **Sample states:** iPDB evaluates heuristic quality on sampled states. The paper initially samples 1,000 states to estimate improvement, similar to the finite-domain variant.

Numeric iPDB Pattern Extension

Suppose we have a pattern $P = \{\texttt{truck-at}\}$ with 2 abstract states. The CG suggests adding `fuel` (a numeric variable).
Before adding: domain size = 2.
After adding: we estimate `fuel`'s range. Suppose analysis gives $M^-_{\texttt{fuel}} = 0$, $M^+_{\texttt{fuel}} = 20$, $C^+_{\texttt{fuel}} = 5$, $C^-_{\texttt{fuel}} = 3$, and $\gcd(3, 5) = 1$. Estimated number of fuel values:

$$\frac{(20 + 5) - (0 - 3)}{1} + 1 = \frac{28}{1} + 1 = 29$$

New pattern size estimate: $2 \times 29 = 58$ abstract states. If the budget B allows this, the extension is accepted, and the pattern is evaluated for heuristic improvement.
Note that this is an approximation—the actual number of reachable abstract states during PDB construction (with the progression-based algorithm) might be different.

10 Theoretical Results: Formal Properties

10.1 Theorem 1: Homomorphisms Preserve Plans (Cost Non-Increasing)

Theorem 10.1. *For a homomorphism α of an LTS $\mathcal{T}$ and plan π for $\mathcal{T}$, $\alpha(\pi)$ is a plan for $\alpha(\mathcal{T})$ with* $\text{cost}(\alpha(\pi)) \leq \text{cost}(\pi)$.

What this says: Any plan in the original system can be translated (via α) into a plan in the image system, and the translated plan costs no more than the original.

Why it matters: This immediately gives admissibility. If π^* is an optimal plan from s in $\mathcal{T}$, then $\alpha(\pi^*)$ is a plan from $\alpha(s)$ in $\alpha(\mathcal{T})$, so:

$$h_\alpha(s) \leq \text{cost}(\alpha(\pi^*)) \leq \text{cost}(\pi^*) = h^*(s)$$

Proof sketch: Let $\pi = \langle t_1, \ldots, t_n \rangle$ be a sequence of transitions from s_0 to $s_* \in S_*$. By Property 2 of Definition 6.1, each $\alpha(t_i)$ is a transition in $\mathcal{T}'$. By Property 4, $\alpha(s_0) = s'_0$ and $\alpha(s_*) \in S'_*$. By Property 3, $\text{cost}(\alpha(l)) \leq \text{cost}(l)$ for each label. So $\alpha(\pi)$ is a valid path from s'_0 to S'_* with cost $\leq \text{cost}(\pi)$.

Key Idea

This theorem is the foundation for *all* PDB heuristics: projections are homomorphisms, and homomorphisms yield admissible heuristics. The paper's contribution is extending this to *infinite* LTSs, where the proof requires care because plans are finite but the LTS may be infinite.

10.2 Proposition 1: Orthogonal Projections Are Additive

Proposition 10.1. *Let α and β be orthogonal LTS homomorphisms and let s be reachable from s_0. Then $h_\alpha(s) + h_\beta(s) \leq h^*(s)$.*

What this says: If two homomorphisms (projections) are orthogonal—meaning no single transition changes both abstract states simultaneously—then their heuristic values can be safely added while maintaining admissibility.

Proof walkthrough: Since s is reachable from s_0, each s-plan can be extended into an s_0-plan. Let $\pi_s = \langle t_1, \ldots, t_n \rangle$ be an optimal s-plan. By orthogonality, each transition t_i satisfies either $\alpha(s_i) = \alpha(s_{i+1})$ (a self-loop in α's image) or $\beta(s_i) = \beta(s_{i+1})$ (a self-loop in β's image).

Partition the transitions: let π_1 be those that are non-trivial in α's image, and π_2 be those non-trivial in β's image. These sets are disjoint (by orthogonality). Then:

- $\alpha(\pi_1)$ is an $\alpha(s)$-plan for $\alpha(\mathcal{T})$, so $h_\alpha(s) \leq \text{cost}(\alpha(\pi_1)) \leq \text{cost}(\pi_1)$.
- $\beta(\pi_2)$ is a $\beta(s)$-plan for $\beta(\mathcal{T})$, so $h_\beta(s) \leq \text{cost}(\beta(\pi_2)) \leq \text{cost}(\pi_2)$.

Adding: $h_\alpha(s) + h_\beta(s) \leq \text{cost}(\pi_1) + \text{cost}(\pi_2) \leq \text{cost}(\pi_s) = h^*(s)$.

Numerical Example

Consider a task with three variables: `A`, `B`, `C` and actions:

- a_1: affects only `A`, cost 3
- a_2: affects only `B` and `C`, cost 5
- a_3: affects only `A`, cost 2

Patterns $P_1 = \{\texttt{A}\}$ and $P_2 = \{\texttt{B}, \texttt{C}\}$ are orthogonal: actions a_1, a_3 affect only P_1 (self-loop in P_2), and a_2 affects only P_2 (self-loop in P_1).
If $h|_{P_1}(s) = 5$ and $h|_{P_2}(s) = 10$, then $h|_{P_1}(s) + h|_{P_2}(s) = 15 \leq h^*(s)$.

10.3 Proposition 2: Numeric Variables Prevent Orthogonality

Proposition 10.2. *Let Π be an IRT with a numeric variable $x \in \mathcal{V}_n$ such that $s_0[x] \not\models G|_{\{x\}}$, and let α and β be projections on patterns P and P' with $x \in P \cap P'$. Then α and β are not orthogonal.*

What this says: If a numeric variable x must change its value to reach the goal (i.e., the initial value does not satisfy the goal condition on x), and x appears in two different patterns, those patterns cannot be orthogonal.

Why: Since $s_0[x]$ does not satisfy the goal, there must be some action a along any plan that changes x. The corresponding transition (s, s', a) has $s[x] \neq s'[x]$. Since $x \in P$, the projected states satisfy $s|_P \neq s'|_P$ (the transition is not a self-loop in α's image). Since $x \in P'$, similarly $s|_{P'} \neq s'|_{P'}$ (not a self-loop in β's image either). Both projections change simultaneously, violating orthogonality.

Practical implication: When constructing the canonical heuristic, a numeric goal variable should appear in at most one pattern per additive group. This limits how aggressively patterns can overlap on numeric dimensions.

10.4 Proposition 3: h_{PDB} Is Admissible on the Sub-LTS

Proposition 10.3. *h_{PDB} is admissible on the states in $S_E \cup S_F$.*

What this says: The heuristic computed by Algorithm 1 never overestimates the true optimal cost for any state that was explored during PDB construction.

Proof: For fringe states $s \in S_F$: by definition, $d(s)$ is admissible (the minimum action cost lower bound), so $h_{\text{PDB}}(s) = d(s) \leq h^*(s)$.

For expanded states $s \in S_E$: consider the optimal plan π_s^*. This plan must eventually reach a state outside S_E (unless the goal is within S_E, in which case the argument is simpler). Let s_F^* be the first fringe state encountered along π_s^*. Then:

$$h^*(s) = \text{cost}^*(s, s_F^*) + h^*(s_F^*) \geq \text{cost}^*(s, s_F^*) + d(s_F^*) \geq \min_{s' \in S_F} \{\text{cost}^*(s, s') + d(s')\} = h_{\text{PDB}}(s)$$

The first inequality uses the admissibility of d. The second uses the fact that the minimum over all fringe states is at most the value for any specific fringe state.

11 Experimental Evaluation: What the Results Show

The paper evaluates numeric PDB heuristics within the Numeric Fast Downward (NFD) framework using A* search.

11.1 Experimental Setup

- **Benchmarks:** IPC 2023 simple numeric domains, domains from the literature (counters, delivery, farmland, plant-watering, etc.), and two novel Minecraft-inspired domains (forestfire, minecraft-pogo, minecraft-sword).
- **Baselines:** $h^{\max}$ (imax), repetition $h^{\max}$ (rmax), numeric landmark heuristic (LM), LM-cut (LMc), and operator counting (OPC).
- **PDB variants tested:**
  - **FDR iPDB:** Classical iPDB ignoring all numeric variables (finite-domain only).
  - **Greedy PDB:** Single-pattern numeric PDB using Fast Downward's greedy pattern selection.
  - **sysC:** Systematic patterns (size 2) with canonical heuristic.
  - **iPDB:** Hill-climbing iPDB with numeric variable support and canonical heuristic.
- **Resources:** 30-minute time limit, 8 GiB memory, Intel Xeon Gold 6130 CPUs.

11.2 Key Findings

1. Numeric PDBs dramatically reduce search effort.

Figure 1 in the paper compares the number of state expansions between finite-domain iPDB and numeric iPDB on a per-instance basis. Taking numeric variables into account reduces expansions by up to six orders of magnitude in some instances. This demonstrates that numeric information provides crucial heuristic guidance that finite-domain PDBs miss entirely.

2. Numeric PDBs compete with strong baselines.

Table 1 shows total coverage (number of solved instances):

Planner	Solved instances
blind search	203
$h^{\max}$ (imax)	136
$h^{\max}$ repetition (rmax)	237
LM	216
OPC	303
LMc	311
FDR iPDB	222
greedy PDB (numeric)	224
sysC (numeric)	241
iPDB (numeric)	250

Numeric iPDB solves 250 instances (vs. 222 for FDR iPDB—28 more instances from incorporating numeric variables). While iPDB lags behind OPC (303) and LMc (311) overall, it excels in certain domains, particularly the Minecraft variants where it solves 24 additional instances compared to OPC and LMc (which solve very few in those domains).

3. iPDB and LMc are complementary.

The per-instance scatter plots (Figure 2) show that iPDB and LMc have complementary strengths: each solves instances that the other cannot. This suggests that combining them (e.g., via a portfolio approach) could yield a stronger planner.

4. Lookup misses are manageable with iPDB.

The percentage of states with no PDB lookup (lookup misses) is significantly lower for iPDB than for single-pattern greedy PDB. This confirms the benefit of using multiple smaller patterns that collectively cover more of the state space.

12 Connections to Prior Work

12.1 Classical PDB Heuristics

PDBs were introduced by Culberson and Schaeffer (1996, 1998) for solving combinatorial puzzles. They were brought to AI planning by Edelkamp (2002) and extended with multiple patterns by Holte et al. (2004, 2006), Felner et al. (2004), Haslum et al. (2005, 2007), and Katz and Domshlak (2009). The canonical heuristic was introduced by Haslum et al. (2007).

12.2 Numeric Planning Heuristics

Prior admissible heuristics for numeric planning include:

- $h^{\max}$: Interval-based relaxation (Aldinger and Nebel 2017).
- LM-cut: Adapted for optimal numeric planning (Kuroiwa et al. 2021, 2022).
- Operator counting: Constraint-based heuristics (Kuroiwa et al. 2021).
- Numeric landmarks (Scala et al. 2017).

This paper is the first to bring PDB heuristics into the numeric planning setting.

12.3 Abstraction and Homomorphisms

LTS homomorphisms generalize the classical notion of abstraction. The paper builds on the model-theoretic view of LTSs as relational structures (Libkin 2004) and on prior work on symmetries and endomorphisms in planning (Shleyfman et al. 2015; Horčík and Fišer 2021; Shleyfman et al. 2023).

13 Summary of Notation

For quick reference, here is a table of all major symbols used in the paper:

Symbol	Type / Dimensions	Meaning
$\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, G \rangle$	Planning task	Integer-restricted task (IRT)
$\mathcal{V} = \mathcal{V}_n \cup \mathcal{V}_p$	Set of variables	Numeric ($\mathcal{V}_n$) and finite-domain ($\mathcal{V}_p$) variables
$v \in \mathcal{V}$	Variable	A state variable
$\mathcal{D}(v)$	Set	Domain of variable v ($\mathbb{Z}$ for numeric, finite otherwise)
$\mathcal{A}$	Set of actions	Available actions
s_0	State	Initial state
$G = G_p \cup G_n$	Goal description	Propositional (G_p) and numeric (G_n) conditions
$\mathcal{S}$	Set of states	All possible states
S_*	Set of states	Goal states: $\{s \mid s \models G\}$
$s[v]$	Value	Value of variable v in state s
$\langle v, d \rangle$	Fact	Variable v has value d
s^{pt}	Partial state	Assignment to a subset of variables
$\mathrm{pre}(a)$	Conditions	Precondition of action a
$\mathrm{eff}(a)$	Effects	Effect of action a
$\mathrm{cost}(a)$	Scalar ≥ 0	Cost of action a
sa	State	Result of applying a in s
$v \mathrel{+}= m$	Numeric effect	Increase v by integer m
$v \bowtie w$	Numeric condition	v compared to integer w
$\mathcal{T} = \langle S, L, \mathrm{cost}, T, s_0, S_* \rangle$	LTS	Labeled transition system
$T \subseteq S \times S \times L$	Transitions	Set of labeled transitions
$\mathrm{cost}^*(s, s')$	Scalar	Optimal cost from s to s'
$h^*(s)$	Scalar	Perfect heuristic (optimal cost to goal from s)
$\mathcal{T}_\Pi$	LTS	State space of planning task Π
$P \subseteq \mathcal{V}$	Pattern	Subset of variables for PDB
$s\|_P$	Partial state	Projection of state s onto variables in P
$\Pi\|_P$	Projected task	Task restricted to variables in P
$\mathcal{T}\|_P$	Projected LTS	LTS of the projected task
$h\|_P$	Heuristic	PDB heuristic for pattern P
$CG(\Pi)$	Directed graph	Causal graph of Π
$\mathrm{vars}(\Psi)$	Set of variables	Variables mentioned in formula set Ψ
α	Map	LTS homomorphism
$\alpha(\mathcal{T})$	LTS	Image of $\mathcal{T}$ under α
h^C	Heuristic	Canonical heuristic over pattern collection C
$\mathcal{M}(C)$	Set of sets	Maximal disjoint-additive subsets of C
S_E	Set of states	Expanded (closed) states in Algorithm 1
S_F	Set of states	Fringe (open) states in Algorithm 1
N	Integer	Maximum number of states to explore
$h_{\mathrm{PDB}}(s)$	Scalar	PDB heuristic value for state s
$d(s')$	Scalar	Admissible estimate for fringe state s'
$[M_x^-, M_x^+]$	Interval	Estimated numeric interval for variable x
C_x^+, C_x^-	Scalars	Maximum additive constants for x

14 Putting It All Together: The Complete Pipeline

Here is how the entire system works from start to finish:

1. **Input:** A simple numeric planning (SNP) task in PDDL.
2. **Translation:** The PDDL task is translated into an IRT using NFD's translator component. This ensures all numeric effects are constant additions and the task is in the required form.
3. **CG Construction:** Build the causal graph $CG(\Pi)$ to understand variable dependencies.
4. **Pattern Generation:** Select patterns using one of:
   - **Greedy:** Fast Downward's default single-pattern selection.
   - **sysC:** Enumerate all interesting 2-variable patterns.
   - **iPDB:** Hill-climbing over pattern collections with numeric-aware size estimation.
5. **PDB Construction (for each pattern):**
   (a) Project the IRT onto the pattern P.
   (b) Run Algorithm 1: progression from $s_0|_P$ up to N states, then regression to compute h_{PDB}.
   (c) Store the PDB as a hash table: $s|_P \mapsto h_{\text{PDB}}(s|_P)$.
6. **Combine PDBs:** If multiple patterns are generated, combine them using the canonical heuristic h^C, identifying maximal disjoint-additive subsets.
7. **A* Search:** Run A* with h^C as the heuristic. For each state s encountered during search:
   (a) Project s onto each pattern P: compute $s|_P$.
   (b) Look up $h_{\text{PDB}}(s|_P)$ in each PDB. If $s|_P$ is not in the PDB, return 0 (lookup miss).
   (c) Compute $h^C(s)$ as the max over additive sums of PDB values.
   (d) Use $f(s) = g(s) + h^C(s)$ as the priority in A*.
8. **Output:** The optimal plan (if found within time/memory limits).

End-to-End Example: Counters Domain

The **counters** domain is a common benchmark for numeric planning. It has n counter variables, each starting at 0, and the goal requires specific counter values. Actions increment or decrement counters at a cost.
For a task with counters c_1, c_2, c_3:
Step 1: Pattern generation (iPDB) might produce:
- $P_1 = \{c_1, c_2\}$ (captures interactions between counters 1 and 2)
- $P_2 = \{c_3\}$ (captures counter 3 independently)

Step 2: For $P_1 = \{c_1, c_2\}$, Algorithm 1 explores abstract states (v_1, v_2) starting from $(0, 0)$. With $N = 10000$, it might explore states up to $v_1, v_2 \in [-10, 50]$ (depending on goal values and actions).
Step 3: If P_1 and P_2 are disjoint-additive (no action affects both $\{c_1, c_2\}$ and $\{c_3\}$ simultaneously), then:

$$h^C(s) = h|_{P_1}(s|_{P_1}) + h|_{P_2}(s|_{P_2})$$

If some action affects c_1 and c_3 simultaneously, they are not additive, and:

$$h^C(s) = \max(h|_{P_1}(s|_{P_1}),\ h|_{P_2}(s|_{P_2}))$$

Step 4: A* uses h^C to guide search. The numeric PDB captures the actual costs of changing counter values, providing much tighter bounds than a PDB that ignores numeric variables.

15 Frequently Asked Questions

Q: Why not just ignore numeric variables and use classical PDBs?

A: You can—this is the "FDR iPDB" baseline in the paper. But the experiments show it performs significantly worse than numeric PDBs. Figure 1 demonstrates up to six orders of magnitude more state expansions when numeric information is ignored. Numeric variables carry crucial cost information (e.g., fuel consumption, resource usage) that finite-domain PDBs completely miss.

Q: What happens if the abstract state space is truly infinite even with bounds?

A: Algorithm 1 handles this by limiting exploration to N states. The resulting PDB is a *partial* PDB: it provides admissible estimates for explored states and returns 0 (trivially admissible) for unexplored states. The hope is that the explored region covers the states most relevant to the search.

Q: Can numeric PDBs be combined with other heuristics like LM-cut?

A: Yes, you can take the maximum of a numeric PDB heuristic and LM-cut: $h(s) = \max(h^C(s), h_{\text{LMc}}(s))$. The max of admissible heuristics is admissible. The paper's experiments show that iPDB and LMc are complementary (each solves instances the other cannot), suggesting this combination would be beneficial. However, the paper does not evaluate this combination directly; it is left as future work.

Q: Why are regression-based PDB methods not used?

A: Regression starts from goal states and works backward. In numeric planning, goal conditions often involve inequalities (e.g., `fuel` ≥ 10), which are satisfied by infinitely many states. You cannot enumerate infinitely many starting points for regression. The paper's progression-based approach starts from the single initial abstract state, which is always well-defined.

Q: How does the state limit N affect heuristic quality?

A: Larger N means more abstract states are explored, giving a more complete (and typically more informative) PDB. However, construction time and memory increase with N. The paper uses $N = 100{,}000$ for greedy PDBs, $N = 50{,}000$ for systematic pattern PDBs, and $N = 10{,}000$ for iPDB. The empirical evaluation shows these limits provide a good trade-off between heuristic quality and construction cost.