Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.
Diese Arbeit wurde vorgelegt am Lehrstuhl für Maschinelles Lernen und Inferenz.

Learning Action Preconditions from State-Action Traces using Inductive Logic Programming

Lernen von Aktionsvoraussetzungen aus Zustands-Aktions-Spuren mit Induktiver Logikprogrammierung

Master Thesis
Masterarbeit

Presented by / Vorgelegt von

Xiaolu Zhang
430095

Supervised by / Betreut von Jonas Gösgens, M.Sc.

1st Examiner / 1. Prüfer Prof. Hector Geffner, Ph.D.

2nd Examiner / 2. Prüfer Prof. Dr. rer. nat. Christopher Morris

Aachen, February 18, 2026

Abstract

This thesis investigates whether symbolic predicates in planning domains can be reconstructed automatically from grounded state–action traces. A domain-independent pipeline is proposed that combines Answer Set Programming[1] with Inductive Logic Programming(ILP) [2]. Execution traces are generated using Clingo [3], and Popper [2] induces logical rules under a unified Boolean-valued representation with explicit negation.

Two symbolic representations are derived from the same traces: a minimal domain [4] containing primitive fluents and a maximal domain [4] extended with SIFT-extracted predicates [4]. Learning targets follow the max–minus–min strategy, where each predicate appearing only in the maximal domain is treated as an independent ILP task.

Experiments across six classical benchmarks show three outcome types: predicates that are fully reconstructed with perfect precision and recall; predicates partially recovered due to missing relational structure; and predicates for which no consistent hypothesis is found. These results demonstrate how ILP can serve as an analytical tool to evaluate the expressive adequacy of symbolic representations and guide predicate design in planning domains.

Contents

1 Introduction

A key challenge in symbolic learning is that the predicate vocabulary itself is often unknown. Most existing approaches to learning action models assume a fixed set of predicates and focus solely on reconstructing preconditions and effects. However, the expressiveness of the learned rules depends critically on the quality of this vocabulary: if essential relational concepts are missing, then no learning algorithm—symbolic or neural—can recover the correct rules. This creates a circular dependency: predicates are needed to learn rules, yet rules are needed to justify which predicates should exist. The purpose of this thesis is to break this dependency by providing a principled pipeline that constructs, evaluates, and defines symbolic predicates directly from grounded observations.

Learning symbolic rules from observed behavior is a central problem in automated planning and reasoning. In many planning domains, symbolic predicates describe when a property holds in a state or when an action is applicable. These predicates often encode spatial relations, object configurations, or consistency constraints. In classical settings, such symbolic knowledge is specified manually. This thesis studies how such rules can instead be recovered automatically from grounded state–action traces.

The input to our approach consists of logical facts extracted from demonstrations or simulated executions. Each state is represented as a finite set of ground atoms. For example, in a grid-based setting, a fact such as `at(p1,c1,s1)` states that person `p1` is located at cell `c1` in state `s1`. Based on such observations, target predicates can be defined through logical conditions. A simple example is the predicate `can_pick_up(p1,k1,s1)`, which holds when a person and a key are co-located:

```
can_pick_up(P, K, S) :- at(P, C, S), at(K, C, S).
```

Positive and negative examples constrain the hypothesis space and prevent overly general rules. Clingo is used not merely as a simulator, but as a way to obtain fully grounded, noise-free, and type-consistent state–action traces. This guarantees that the learning process evaluates representational choices rather than artifacts of imperfect demonstrations. By controlling the traces at the logical level, we can study the limits of symbolic reconstruction under idealised but informative conditions.

This thesis uses a pipeline composed of two logic-based components. Clingo [3]is used to represent symbolic environments and to generate grounded state–action traces through Answer Set Programming. Popper [2] is used to learn logic programs from labeled examples using

Inductive Logic Programming (ILP) [2] under an explicit search bias. Together, these tools support an end-to-end workflow that moves from low-level grounded facts to human-readable symbolic rules.

A central idea of this work is the max–minus–min strategy. For each domain, two symbolic representations are derived from the same execution traces. The minimal domain [4] contains only primitive predicates that directly describe the environment. The maximal domain [4] extends this representation with higher-level predicates extracted by SIFT [4]. Learning targets are defined as the set difference

$$\Delta = \text{Predicates(Maximal Domain)} \setminus \text{Predicates(Minimal Domain)},$$

and each predicate in Δ is treated as an independent learning task. Popper then attempts to define each maximal-only predicate using only minimal-domain predicates and Boolean-valued fluents. This setup provides a systematic way to test which symbolic concepts can be reconstructed from basic observations and which require richer background knowledge. The max–minus–min strategy provides a controlled way to evaluate representational sufficiency. If a maximal-domain predicate can be defined purely from minimal fluents, then the minimal representation is expressive enough; if not, the missing predicate captures structure that the minimal vocabulary fails to encode. This turns predicate learning into a systematic test of representational power rather than an open-ended search problem. A worked example illustrating this transformation is given in Appendix A, Section A.1.

A direct application of ILP to raw traces is infeasible, because the hypothesis space over all possible predicate symbols is combinatorially large. SIFT addresses this problem by analysing how hypothetical predicates would have to change across actions and retaining only those whose behaviour is consistent with the observed transitions. The resulting maximal predicate set acts as a data-driven vocabulary from which ILP can selectively reconstruct meaningful definitions.

The same pipeline is applied across several planning benchmarks. The core domains are Block World, Sokoban, N-Puzzle, and Miconic. In addition, the approach is evaluated on two logistics-style domains, Logistics and Driverlog, using the same max–minus–min task construction. Across all domains, the learning mechanism, background encoding, and evaluation protocol remain unchanged. Popper is chosen because it learns compact, human-readable logical rules under an explicit, finite bias. This makes it possible to determine with certainty whether a predicate can be defined from the available background knowledge. Unlike neural or probabilistic approaches, Popper provides deterministic guarantees: if no definition is found, either none exists within the hypothesis space or essential information is missing from the minimal domain. Appendix A, Section A.1 shows how the files `exs.pl`, `bk.pl`, and `bias.pl` are instantiated for a single target predicate in N-Puzzle.

The experiments reveal several recurring patterns. In Block World, Popper reconstructs three maximal-only predicates with perfect precision and recall. The learned rules correspond to interpretable abstractions over primitive stacking relations, including existential patterns and Boolean consistency constraints. For readers interested in the exact inputs that lead to such hypotheses, Appendix A, Section A.1 provides a complete worked example for `learned_pred_0` in N-Puzzle.

In Sokoban, Popper learns an abstract occupancy predicate by combining the minimal predicates `has_box` and `has_player`, using explicit Boolean negation to encode a domain invariant. In the N-Puzzle domain, several predicates are learned as local projections of minimal spatial fluents such as `at_val` and `blank_val`, while other targets yield `NO SOLUTION` due to missing relational structure in the background. In Miconic, Popper learns predicates that depend on lift position with perfect precision but incomplete recall, while predicates involving passenger goals remain only partially recoverable. In the Logistics and Driverlog domains, some maximal predicates reduce to simple projections of minimal relations, while others cannot be learned due to either missing expressiveness or lack of positive examples.

Across all domains, two representation choices prove critical for stable learning. First, encoding fluents in a Boolean-valued form provides a uniform interface across heterogeneous domains. Second, explicit negation through `neg/2` avoids unsafe classical negation in learned hypotheses while still allowing Popper to express exclusion and consistency constraints. Overall, the results show that many maximal-domain predicates can be reconstructed from minimal representations when they correspond to local or projection-style conditions. Conversely, `NO SOLUTION` outcomes serve as diagnostic signals of insufficient background expressiveness or limited inductive supervision.

2 Background

This chapter provides the conceptual foundations on which the remainder of the thesis is built. It introduces the learning and reasoning formalisms that underpin our approach, and explains how they interact in the construction of symbolic representations from state–action data. We begin with Inductive Logic Programming (ILP), focusing in particular on Popper and its role in synthesizing interpretable logical rules under explicit representational constraints. We then turn to classical planning and describe how action preconditions can be inferred from observed traces when domain models are not available. To support the generation of such traces, the chapter also outlines the basics of Answer Set Programming (ASP) [1] and the use of Clingo as a grounding and solving engine. Finally, we discuss the structure and function of predicates in symbolic learning, and introduce SIFT as a mechanism for extracting candidate relational features from trace data. Together, these components establish a unified background for the learning pipeline developed in later chapters.

2.1 Inductive Logic Programming and Popper

Inductive Logic Programming (ILP) is a learning paradigm that induces symbolic rules from labeled examples and background knowledge. Given positive and negative examples, an ILP system searches for logical hypotheses that explain why some instances hold while others do not. The resulting hypotheses are usually expressed as logic programs, which makes them interpretable and suitable for symbolic reasoning.

Popper is a recent ILP system that combines program synthesis with logical testing. It explores a space of candidate rules defined by bias constraints, evaluates them against examples, and retains only those that are consistent. The output of Popper is a set of definite clauses written in a Prolog-style syntax. Because the learned rules are explicit and human-readable, Popper is well suited for tasks where understanding the structure of symbolic domains is important.

2.1.1 Learning Setting, Bias, and Practical Constraints

Most ILP systems, including Popper, operate in a setting where a hypothesis H is learned from background knowledge B and labeled examples E. Conceptually, H should explain the positive examples while rejecting the negative ones. In Popper, this is implemented as a constrained search over definite clauses, guided by a *bias* that limits, for instance, the number of literals, the number of variables, and the allowed predicate symbols.

These constraints are not merely computational; they shape the form of the resulting rules. Smaller clause bounds favour compact hypotheses such as existential projections or simple conjunctions, whereas more expressive relational rules may require larger search spaces. Moreover, Popper typically assumes a definite-clause hypothesis language and therefore does not freely support classical negation inside learned clauses. In practice, negation must often be encoded explicitly in the background knowledge (e.g., via Boolean truth values or dedicated helper predicates), so that the learned program remains within the safe and supported fragment.

2.2 Classical Planning and Learning from Traces

Classical planning [5] typically relies on symbolic domain models defined in formalisms such as STRIPS [6] or PDDL [7]. In these models, actions are specified by preconditions and effects written as logical predicates. Such domain descriptions allow planners to reason about action applicability and goal reachability.

In many settings, however, explicit domain models are not available. Instead, only observed sequences of actions and resulting states are given. Learning symbolic representations directly from these traces offers an alternative to manual modeling. It allows a system to recover structured symbolic knowledge from observations and to approximate the information encoded in hand-crafted planning domains.

2.2.1 Learning Action Preconditions from Traces

From the perspective of classical planning, learning from traces can be framed as recovering a logical description of action applicability. For an action schema $a(\bar{x})$, the goal is to learn a predicate that characterises when a is applicable in a state s. In trace data, each transition (s, a, s') provides evidence that a was applicable in s, while states in which a was not selected (or is known to be impossible) can be used to construct negative evidence.

This thesis focuses on learning such precondition-like predicates in a controlled setting where traces are generated from a known dynamics model. The objective is not to replace planning, but to analyse to what extent higher-level symbolic predicates can be reconstructed from a restricted minimal vocabulary.

2.3 Answer Set Programming and Clingo

Answer Set Programming (ASP) is a declarative logic programming paradigm used for solving search and reasoning problems. Unlike ILP, ASP is not designed for learning rules but for computing models that satisfy a given set of logical constraints. A logic program in ASP describes facts, rules, and constraints, and an ASP solver computes its stable models.

Clingo is a widely used as ASP solver that integrates grounding and solving. It takes a logic program as input and produces answer sets that represent valid configurations of the modeled problem. In planning-related settings, ASP is often used to encode world states, actions, and transition constraints. In this work, Clingo is used to generate grounded state–action traces [8] that later serve as structured input for learning.

2.3.1 State–Action Traces as Structured Data

A key advantage of ASP-based planning encodings is that they produce explicit and fully grounded state descriptions. A typical output consists of a sequence of states together with the action applied at each step, often referred to as a *state–action trace*. Such traces provide a machine-readable account of how the world evolves under the modeled dynamics.

For learning, traces play a dual role. First, they provide supervised signals about action applicability, since an applied action must have been applicable in the originating state. Second, they provide rich relational snapshots of the state, which can be transformed into example sets for ILP. This makes Clingo-generated traces a suitable intermediate representation between simulators and symbolic rule induction.

2.4 Predicates and Their Role in Symbolic Learning

A predicate is a symbolic relation that describes a property of the world or a relation between entities. Formally, a predicate is written as $p(t_1, t_2, \dots, t_n)$, where p is the predicate name and $t_1, \dots, t_n$ are its arguments. Each argument can be a constant representing an object or a variable ranging over such objects. A predicate together with concrete arguments is called a *ground atom* [9].

In this thesis, symbolic states are represented as finite sets of ground atoms. Each atom expresses a fact that holds in a given state, such as the position of an object or a relation between entities. Actions transform one state into another by changing which ground atoms are true.

Two types of predicates are distinguished. *Primitive predicates* [10] are directly extracted from the environment or simulator, such as object positions or occupancy facts. *Derived predicates* [11] are symbolic abstractions defined in terms of primitive ones. For example, a predicate like `co_located/3` expresses that two entities occupy the same position, even though this relation is not directly observed.

Derived predicates play an important role in symbolic learning. They allow compact representations of recurring patterns and reduce the complexity of learned rules. Learning such predicates makes it possible to move from low-level observations toward higher-level symbolic descriptions that are suitable for planning and reasoning.

2.4.1 Fluents, Types, and Boolean-Valued Encodings

In planning domains, predicates that can change across states are often called fluents [5]. A state can then be seen as a valuation over fluents, i.e., a specification of which ground atoms hold. In learning settings based on traces, it is useful to make this valuation explicit. One common strategy is to represent fluents with an additional truth-value argument, yielding a Boolean-valued encoding such as $\texttt{p\_val}(\bar{t}, v)$ where v denotes whether the fluent holds.

This encoding has two practical benefits. First, it allows a learner to reason about absence information without using classical negation in learned clauses. Second, it makes it possible to express consistency constraints between complementary truth values via explicit helper relations (e.g., a predicate that maps `true` to `false` and vice versa). In addition, most domains require type information (e.g., object categories such as *block*, *tile*, or *location*) to keep hypotheses well-typed and to avoid accidental variable bindings during learning.

2.4.2 From Traces to Candidate Predicates: SIFT

While raw state–action traces provide rich information about how actions change the world, they are typically too low-level to serve directly as a hypothesis space for learning. In particular, a learning system requires a structured set of state predicates whose truth values vary across the trace, yet such predicates are not available when the domain vocabulary is hidden. SIFT addresses this gap by inferring a set of hypothetical domain predicates directly from traces.

SIFT operates by constructing features, each of which represents a candidate predicate together with a hypothesis about which action argument positions may affect its truth. Formally, a feature $f = \langle k, B \rangle$ specifies a k-ary predicate whose value may change only through the *action patterns* in B, where each pattern identifies a particular binding between action parameters and predicate arguments. Given a typed vocabulary extracted from the traces, SIFT enumerates all possible features compatible with these types.

For each feature, SIFT tests whether the assumption that all and only the patterns in B affect the corresponding hypothetical atom is *consistent* with the traces. Consistency is determined by a set of tractable constraints derived from the traces—most importantly, "consecutive" and "fork" pattern constraints—which capture how action instances must alternate or agree in sign if they are to affect the same atom. A feature is retained exactly when these constraints admit a valid assignment of signs, yielding a Boolean-valued predicate whose truth can be propagated along the traces.

The resulting set of consistent features constitutes a library of candidate domain predicates. These predicates can then be used to reconstruct action preconditions and effects, and they serve as inputs to subsequent learning stages such as ILP. Primitive predicates from the hidden domain appear as admissible features, while the method may additionally produce redundant but valid predicates corresponding to the maximal domain representation.

3 Related Work

This chapter builds on several concepts from symbolic planning and logic-based learning. The following detailed definitions and examples are provided to support readers who may not be familiar with the underlying terminology.

3.1 Background Concepts and Terminology

STRIPS is a classical formalism for representing planning problems through actions with explicitly defined preconditions and effects. A STRIPS domain describes how symbolic world states change as actions are applied.

PDDL, the Planning Domain Definition Language, extends the ideas of STRIPS and has become a standard language for specifying planning domains and problems. It is widely used as an interface between domain models and automated planners.

Clingo is a system based on Answer Set Programming (ASP), a declarative paradigm for modeling combinatorial search and reasoning problems. In planning-related work, Clingo is often used to encode domain constraints, state transitions, and execution traces in a symbolic and executable form.

SIFT is a method for inducing lifted symbolic models from state–action traces. It analyses how predicates behave across actions and identifies those that are stable, informative, or action-relevant. As a result, SIFT can extract both primitive and abstract predicates from observed executions.

Popper is an Inductive Logic Programming (ILP) system that learns logic rules from background knowledge and labeled examples under an explicit bias. It focuses on inducing small, interpretable clauses that explain observed regularities in relational data.

3.2 Learning Action Models from Traces

A large body of research addresses the problem of learning action models from observed executions. The main motivation is to reduce the manual effort required to specify STRIPS or PDDL domains, where actions, preconditions, and effects are traditionally defined by experts.

Many approaches in this line of work assume that the predicate vocabulary is given and fixed. Learning then focuses on reconstructing full action schemas, often using statistical analysis,

constraint satisfaction, or optimisation over plan traces. The output of these systems is typically a complete STRIPS-style action model that can be used directly by a planner.

While effective for recovering action definitions, these approaches usually do not question the predicate vocabulary itself. Abstract or derived predicates are assumed to be part of the domain description rather than objects of analysis. As a result, the learning problem is framed at the level of actions, not at the level of symbolic representation.

3.3 Inductive Logic Programming for Symbolic Learning

Inductive Logic Programming provides an alternative framework for learning symbolic knowledge from examples. ILP systems aim to induce logical rules that explain positive observations while excluding negative ones, given background knowledge and bias constraints.

Inductive Learning of Answer Set Programs (ILASP) [12] is a prominent ILP system that learns Answer Set Programs. It supports non-monotonic reasoning, preferences, and constraints, and has been applied to learning planning-related knowledge. Its expressive power allows it to capture rich theories about an agent's environment.

However, ILASP typically targets complete logical theories rather than compact definitions of individual predicates. Its focus lies on learning full models rather than analysing how specific predicates can be grounded in simpler ones. This makes it less suitable for studying representational questions in highly structured but simple domains.

In this project, we aim to automate the process of learning symbolic rules from structured observations in grid-based environments. For instance, we may want to discover under what conditions an agent can pick up a key, or what spatial relationships support certain actions. These rules are not always explicitly encoded, and must be inferred from data.

Sometimes, useful intermediate relations like `co_located/3`—which indicates that two objects occupy the same position—are not directly observed but are essential for learning compact and generalizable rules. Our goal is to identify such concepts automatically when possible, using a pipeline composed of two logic-based tools: Clingo and Popper.

Together, these tools enable an end-to-end workflow: Clingo is used to simulate structured environments and generate logical traces, while Popper performs rule induction by learning interpretable logic programs from examples. This supports not only automated rule discovery but also abstraction through predicate invention.

Clingo: Declarative Modeling with Answer Set Programming

Clingo is a system for Answer Set Programming (ASP), a form of declarative logic programming. ASP is well-suited for modeling problems involving constraints, search, and combinatorics.

Instead of specifying procedures, the user defines logical rules, and Clingo computes consistent models that satisfy all the constraints.

For example, in the classic graph coloring problem, we can write:

```
node(a). node(b). node(c).
edge(a,b). edge(b,c). edge(c,a).
color(red;green;blue).

1 { assign(N,C) : color(C) } 1 :- node(N).
:- assign(X,C), assign(Y,C), edge(X,Y).
```

Clingo will find valid colorings where no two connected nodes share the same color. In our project, Clingo serves to encode grid-based worlds and generate symbolic traces of agent actions and environment states.

Popper: Inductive Logic Programming for Rule Learning

Popper is a system for Inductive Logic Programming (ILP), used to learn logical rules from data. Given background knowledge, labeled examples, and a bias specification (mode declarations), Popper generates human-readable Prolog-style rules that are consistent with the observations.

Suppose we observe that a person becomes happy when they like something. We can represent this with:

```
% General facts of example
person(alice). person(bob).
likes(alice, music).
likes(bob, sports).

% Facts of Round 1
playing(music).

pos(happy(alice)).
neg(happy(bob)).
```

Popper can learn the rule:

```
happy(X) :- likes(X, music).
```

This rule captures the idea that a person is happy if they like music which is currently playing.

Now suppose we introduce new examples with updated world information:

```
% New facts (Round 2)
playing(sports).
```

```
% Updated examples
neg(happy(alice)).
pos(happy(bob)).
```

With these new examples, Popper will revise its hypothesis and learn a more general rule:

```
happy(X) :- likes(X, Y), playing(Y).
```

This updated rule generalizes the concept of happiness as being tied to whether a person likes something that is currently happening. It demonstrates Popper's ability to adapt its learned logic in response to new evidence, supporting both refinement and generalization of symbolic knowledge.

Popper uses a generate-and-test method to search the space of logic programs:

- It first generates hypotheses based on mode declarations and background knowledge.
- It then tests each candidate to see if it covers all positive and none of the negative examples.
- Internally, Popper uses Clingo as a reasoning backend to perform this search efficiently.

Because Popper outputs interpretable rules, it is especially suitable for applications where explainability and formal guarantees are required.

The two tools are used in sequence: Clingo generates logical traces, and Popper induces symbolic rules from those traces.

Together, these tools enable a workflow that supports both declarative modeling and explainable machine learning.

This combination of symbolic modeling and inductive logic learning allows for an automated and interpretable rule discovery process. It supports the goals outlined in our introduction: to abstract, generalize, and explain action applicability in structured planning domains.

It enables the learning of generated intermediate predicates like `co_located/3`, which capture reusable spatial relationships between entities and help compress symbolic rules.

3.4 Predicate Induction from Traces with SIFT

SIFT introduces a complementary perspective on learning from traces. Instead of learning explicit rules, it evaluates candidate predicates by analysing how they change across actions. Predicates that behave consistently are retained as useful symbolic features.

The output of SIFT is a structured symbolic description of the domain that includes both primitive and abstract predicates. In practice, this information can be exported as JSON files that list all states, predicates, and their truth values. These files are commonly produced in two

variants: a maximal version, which contains all extracted predicates, and a minimal version, which keeps only primitive ones.

SIFT therefore addresses the problem of *predicate selection* rather than predicate definition. It identifies which predicates are informative, but it does not explain how abstract predicates relate logically to primitive ones.

3.5 Learning Predicate Definitions with Popper

Popper has been used in several works to induce symbolic rules from relational data. Its strength lies in learning compact and interpretable definitions under tight bias constraints.

In this thesis, Popper is not applied to learn complete action schemas. Instead, it is used to define individual predicates that appear in the maximal domain but are absent from the minimal one. The learning task is therefore shifted from actions to predicates.

The background knowledge consists only of minimal-domain predicates, encoded in a Boolean-valued form. Given positive and negative examples of a target predicate, Popper attempts to induce a definition using only this minimal information.

Compared to ILASP, Popper operates in a more restricted hypothesis space. It learns definite clauses and avoids classical negation by relying on explicit Boolean encodings. This restriction makes it well suited for analysing small, interpretable rules in grid-based and relational domains.

3.6 Logic-Based Tools for Symbolic Rule Learning

Declarative modeling and ILP are often combined in symbolic learning pipelines. Declarative systems generate precise logical descriptions of states and transitions, while ILP systems abstract over these descriptions by inducing rules or higher-level predicates.

Clingo is frequently used to model environments and generate grounded execution traces. These traces provide structured data that can be consumed by ILP learners. Popper, in turn, relies on Clingo internally to evaluate candidate hypotheses during its generate-and-test search.

This combination supports a workflow in which symbolic knowledge is both generated and analysed within a unified logical framework. Intermediate predicates induced or defined in this process can capture reusable relations, such as spatial configurations or consistency constraints.

3.7 Minimal and Maximal Domain Representations

Most work on learning from traces implicitly assumes a single symbolic domain representation. In contrast, this thesis explicitly distinguishes between minimal and maximal domains derived from the same executions.

The minimal domain contains only primitive predicates that directly encode local properties of states. The maximal domain extends this representation with abstract predicates that capture higher-level regularities.

In practice, the choice of a minimal domain is not unique. Different minimal representations may be semantically equivalent while using different predicate vocabularies. When predicates are compared across domains, such representational choices can affect the observed differences.

This thesis addresses this issue by fixing one minimal representation as a canonical reference. Differences between minimal variants are treated as representation choices rather than learning errors. This makes it possible to interpret the set difference between maximal and minimal domains as a meaningful learning target.

Overall, this perspective shifts the focus of learning from reconstructing actions to analysing the expressive power of symbolic representations. The goal is not to recover full planning models, but to understand which abstract predicates can be defined in terms of simpler ones, and where the limits of such definitions lie.

4 Methods

Learning symbolic rules from observed traces has been explored in many forms of artificial intelligence research. This chapter describes the experimental methodology used in this thesis and explains how the proposed *max–minus–min* strategy is instantiated across six planning domains using the Popper Inductive Logic Programming system.

The chapter is structured as follows. Section 4.1 introduces the max–minus–min setting and defines the learning targets. Section 4.2 explains how state–action traces are converted from SIFT JSON outputs into Popper projects. Section 4.3 describes the common Boolean-valued representation and explicit negation scheme used in all domains. Section 4.4 details the Popper configuration and evaluation protocol, including how "solved" and "unsolved" targets are defined. Section 4.5 presents domain-specific instantiations for Block World, Sokoban, N-Puzzle, Miconic, Logistics, and Driverlog. Section 4.6 summarises the quantitative outcomes, and Section 4.7 discusses cross-domain observations.

4.1 Problem Setting and Max–Minus–Min Targets

Across all domains, the overall goal is to learn interpretable symbolic rules for predicates that appear only in a richer, *maximal* representation of the domain but are absent from a restricted, *minimal* representation.

Let $\mathcal{P}_{\max}$ denote the set of predicate symbols in the maximal domain, and $\mathcal{P}_{\min}$ the set of predicate symbols in the minimal domain. The *max–minus–min* strategy defines the set of learning targets as

$$\Delta \;=\; \mathcal{P}_{\max} \setminus \mathcal{P}_{\min}.$$

Each predicate $p \in \Delta$ is treated as an independent ILP task. For every such target, the learning problem is:

> *Given labeled examples of p obtained from the maximal domain, and background knowledge restricted to minimal-domain predicates, induce a logical definition of p that is consistent with the examples and uses only minimal-domain fluents.*

Concretely, the experimental report explores how predicates in Δ can be induced by Popper in six planning domains:

- a classical **Block World** domain with objects, stacking relations, and table positions,
- a grid-based **Sokoban** domain with a player and movable boxes,

- a puzzle-based **N-Puzzle** domain with tile locations and blank cell movement,
- a lift-based **Miconic** domain with floors, elevators, and service requests,
- a transportation-based **Logistics** domain with locations and containment relations,
- a transportation-based **Driverlog** domain with drivers, trucks, packages, and locations.

In every domain, the learning objective is to induce symbolic rules that define the maximal-only predicates using only minimal-domain features. The experiments focus on the following questions:

- Which maximal predicates can be reconstructed from minimal-domain fluents alone?
- What kinds of relational structure (local vs. non-local) can be expressed using the minimal representation?
- How do precision, recall, and rule size vary across domains and targets?

4.1.1 State–Action Traces and Ground Atoms

Throughout this thesis, learning is performed on state–action traces that are generated from classical planning instances and processed by SIFT. A trace has the form

$$\tau = (s_0, a_0, s_1, a_1, \dots, a_{T-1}, s_T),$$

where each s_t is a fully specified symbolic state and each a_t is a grounded action schema applicable in s_t. SIFT processes such traces and, for every time step t, produces a set of ground atoms

$$\mathcal{A}_{\max}(s_t) \subseteq \mathsf{Ground}(\mathcal{P}_{\max}), \qquad \mathcal{A}_{\min}(s_t) \subseteq \mathsf{Ground}(\mathcal{P}_{\min}),$$

where $\mathsf{Ground}(\mathcal{P})$ denotes all ground atoms that can be formed from the predicate symbols in $\mathcal{P}$ and the domain's object constants.

The ILP tasks considered here are *state-based*: for each target predicate $p \in \Delta$ and each time step t, the learner observes whether a particular grounding of p is true or false in state s_t. Temporal information (such as the index t or the surrounding actions a_t) is not used directly in the background knowledge. Instead, the temporal traces serve only as a source of diverse states and ground atoms.

Formally, for a target predicate p of arity k, each example corresponds to a ground atom $p(\bar{o})$ instantiated with a k-tuple of objects $\bar{o} = (o_1, \dots, o_k)$ and a state identifier S. The example is labelled positive if

$$p(\bar{o}) \in \mathcal{A}_{\max}(s_t),$$

and negative otherwise (under a closed-world assumption, see Section 4.2.2). The state identifier S is a symbolic index that uniquely refers to s_t within the dataset; all domain fluents used as background knowledge are keyed by the same identifier S.

4.2 Data and Conversion Pipeline

The goal of the pipeline is to transform the symbolic information extracted by SIFT into a collection of Popper learning tasks, each corresponding to a single predicate whose definition is to be learned. SIFT provides a structured, Boolean-valued representation of all predicates that hold in each state–action trace, while Popper requires background knowledge, labelled examples, and bias declarations in a Prolog-style format. Bridging these two representations requires a series of domain-specific conversion tools that interpret the JSON traces, identify learning targets, and generate consistent Popper task directories. The following subsections describe this conversion process in detail, including how examples are constructed, how tasks are organised, and how the overall workflow is executed for each domain.

4.2.1 From SIFT JSON to Popper Tasks

For each domain, SIFT (Structured Induction of Features and Traces) provides two JSON files:

- a *maximal* JSON file, encoding state-based predicates including SIFT-extracted features;
- a *minimal* JSON file, encoding only primitive domain fluents.

Both JSON files describe the same underlying set of state–action traces, but with different predicate vocabularies. A Python-based toolchain compares the two JSON files, extracts the set Δ of maximal-only predicates, and constructs one Popper learning task per predicate.

Each Popper task directory contains:

- **bk.pl** — background knowledge, derived from minimal-domain facts;
- **exs.pl** — positive and negative examples of the target predicate, derived from the maximal JSON;
- **bias.pl** — mode declarations and search constraints;
- **config.pl** — optional configuration parameters.

The conversion scripts are domain-specific in how they map JSON fields to logical predicates, but identical in their overall structure and in how they implement the max–minus–min filter.

4.2.2 Example Generation and Closed-World Assumption

The SIFT JSON files contain, for every state s_t and for every candidate grounding of each predicate symbol, an explicit Boolean truth value. The conversion scripts exploit this representation to construct positive and negative examples for each target predicate $p \in \Delta$.

For a fixed target predicate p of arity k, the converter iterates over all states s_t and all groundings $p(\bar{o})$ that appear in the maximal JSON. For each pair $(s_t, \bar{o})$, it inspects whether $p(\bar{o})$ is present in the maximal set of atoms $\mathcal{A}_{\max}(s_t)$:

- if $p(\bar{o}) \in \mathcal{A}_{\max}(s_t)$, the converter emits a `pos/1` fact for the corresponding Popper example;
- if $p(\bar{o}) \notin \mathcal{A}_{\max}(s_t)$, the converter emits a `neg/1` fact.

This corresponds to a standard closed-world assumption on the maximal representation: every ground atom that is not explicitly marked as true in the maximal JSON is treated as false. The minimal JSON is never used to label examples; it only contributes background predicates.

In practice, this procedure yields highly imbalanced datasets, with many more negative examples than positive ones. The conversion scripts therefore include simple filtering options (e.g. limiting the number of states per trace or subsampling negatives) that can be used to keep individual Popper tasks tractable. When such filtering is enabled, it always operates symmetrically on positives and negatives per state, so that the empirical distributions of states and groundings remain comparable across targets and domains.

4.2.3 Project Layout and Tools

Inside the experimental container, the Popper engine and conversion tools follow a common directory layout:

```
~/popper/
 popper.py                          # Main Popper engine
tools/
     gen_from_sift.py               # Convert SIFT JSONs -> Popper tasks (
         Block World)
     gen_max_minus_min_projects.py  # Convert SIFT JSONs -> Popper tasks (
         Sokoban)
     rewrite_lean_bias_from_exs.py  # Auto-generate correct bias heads
     filter_exs_by_state.py         # Keep only first N states (reduce
         complexity)
projects/
     diff_domain_max_minus_min/        # Different 'domains tasks from
         SIFT data
```

Several scripts in the `tools/` directory share a common purpose: they convert SIFT-generated JSON traces into Popper learning tasks. Although each script targets a different planning domain—such as Block World, Sokoban, N-Puzzle, Miconic, Logistics, or Driverlog—they all implement the same conversion pipeline. Given a directory of SIFT traces, each script produces a Popper project consisting of background knowledge, positive and negative examples, and a domain-specific bias file. The differences between these scripts lie only in the domain schemas and type declarations; the overal functionality is identical. Each task directory (e.g. `learned_pred_1`, `learned_pred_4`) represents an independent Popper learning problem.

The same learning mechanism is used in all domains; only the JSON-to-logic converters and the choice of targets differ.

4.2.4 Workflow

The high-level workflow for a single domain is:

1. **Generate Popper tasks from SIFT JSON outputs:**
```
python3 tools/gen_<domain>_preconds.py \
    data/<domain>_sift_maximal.json \
    data/<domain>_sift_minimal.json \
    projects/<domain>_preconds
```

2. **Rewrite bias files automatically (optional):**
```
python3 tools/rewrite_lean_bias_from_exs.py
```

3. **Filter examples by state (optional):**
```
python3 tools/filter_exs_by_state.py   # keep only first N states
```

4. **Run Popper on a specific target:**
```
python3 popper.py projects/<domain>_preconds/learned_pred_k
```

Popper incrementally generates and evaluates candidate hypotheses of increasing size, computing precision, recall, and a score that balances correctness and compactness.

4.3 Representation and Explicit Boolean Negation

To keep the representation uniform across domains and to avoid issues with classical negation, all domain fluents are encoded as predicates with an explicit Boolean argument. For example:

- Block World: `on_val(State, BlockAbove, BlockBelow, Val)`
- Sokoban: `has_box_val(State, Cell, Val)`,`has_player_val(State, Cell, Val)`
- N-Puzzle: `at_val(State, Tile, Row, Col, Val)`,`blank_val(State, Cell, Val)`
- Miconic: `lift_pos(State, Floor)`,`in_floor(State, Passenger, Floor)`,`in_lift(State, Passenger)`
- Logistics: `at(State, Object, Location, Val)`, `in(State, Object, Container, Val)`
- Driverlog: Boolean-valued variants `*_val` for location and auxiliary features.

In addition, two Boolean constants and an explicit negation predicate are provided:

```
1 true(1).
```

```
false(0).
neg(0,1).
neg(1,0).
```

Every minimal-domain fluent is thus represented as a valued predicate, and the absence of a fluent is encoded via the Boolean value and the `neg/2` relation rather than via classical negation in the clause body.

4.3.1 Examples and Targets

Examples in `exs.pl` follow Popper's standard format:

```
:- discontiguous pos/1.
:- discontiguous neg/1.

pos(learned_pred_1(S, C)).
pos(learned_pred_1(S', C')).
...

neg(learned_pred_1(S, C'')).
neg(learned_pred_1(S'', C)).
...
```

Each `pos/1` and `neg/1` fact is derived from SIFT's maximal JSON by checking whether the target predicate holds in a given state. The background knowledge for that task only contains minimal-domain predicates and Boolean helpers.

Explicit Boolean encoding has two advantages:

- it avoids unsafeness and grounding issues associated with classical negation;
- it allows Popper to learn constraints involving mutual exclusion and Boolean consistency, such as XOR-style relations.

4.4 ILP Learning Setup with Popper

Popper uses a bias file `bias.pl` to constrain the hypothesis space. Across domains, the bias follows the same general pattern:

- limit the maximum number of clauses per hypothesis (typically two);
- limit the maximum number of body literals per clause (typically three or four);
- limit the maximum number of distinct variables (typically four to six);
- restrict the set of predicates allowed in the body to minimal-domain fluents and Boolean helpers.

These constraints encourage short, interpretable rules and keep the search space finite. Where necessary (e.g. in N-Puzzle and Driverlog), bias files are slightly customised to reflect the arity of domain predicates.

4.4.1 Evaluation Metrics [13]

For each hypothesis, Popper computes the standard classification metrics:

$$\text{Precision} = \frac{TP}{TP + FP},$$
$$\text{Recall} = \frac{TP}{TP + FN},$$

where TP, FP, FN, and TN denote true positives, false positives, false negatives, and true negatives, respectively. In addition, Popper reports the hypothesis size (measured in literals) and the number of clauses.

These metrics are used to classify the outcome of each target as one of the following:

- **Solved:** Popper returns a hypothesis together with a complete evaluation summary (precision, recall, TP, FP, FN, TN). The hypothesis is consistent with all examples under the given bias.
- **NO SOLUTION:** Popper exhausts the search space defined by the bias and reports that no consistent hypothesis exists.
- **No verified solution:** Popper does not produce a parseable evaluation summary (e.g. due to time or memory limits) and no hypothesis is reported, even though consistent hypotheses may exist in principle.

In the quantitative summary (Section 4.6), the "Unsolved" column aggregates both `NO SOLUTION` and "no verified solution" cases, but they are explicitly distinguished in the text for each domain.

4.4.2 Search Configuration and Stopping Criteria

Popper explores the hypothesis space in a bottom-up fashion, ordered primarily by hypothesis size (number of literals and number of clauses). For each target predicate, the search is constrained by the bias file (`bias.pl`) and by a small number of global configuration parameters.

First, hypotheses are enumerated in increasing order of size. Within a fixed size, Popper generates all clause structures that are compatible with the mode declarations and then instantiates them with predicate symbols that are allowed in the body. This guarantees that, if a consistent hypothesis exists within the bias bounds, it will eventually be considered during search.

Second, each Popper run is given a fixed resource budget. If the search completes within this budget, Popper returns one of the best hypotheses found (according to its internal scoring function) together with a complete evaluation summary over all examples. If the search space is exhausted and no hypothesis consistent with the examples is found, Popper reports `NO SOLUTION`. If the resource budget is exceeded before a hypothesis can be fully evaluated and logged, no parseable summary is produced; such cases are reported in this thesis as "no verified solution".

The distinction between `NO SOLUTION` and "no verified solution" is methodologically important. `NO SOLUTION` indicates that, under the given bias, the minimal background representation is not expressive enough to capture the target predicate. "No verified solution" instead points to computational limits of the current configuration (for example, a very large example set or a particularly rich bias), and does not rule out the existence of suitable hypotheses outside the explored search frontier.

4.4.3 Execution Environment

All experiments are executed inside an Apptainer container that bundles Python, Popper, and Clingo. Each domain has its own directory under `projects/` with the three core input files and a script to launch the learner. This setup ensures that software versions and dependencies remain stable and that the experiments can be replayed on other machines with compatible container support.

4.4.4 Result Collection and Post-processing

For each Popper task, the solver outputs a textual summary that includes the learned hypothesis (if any) and the corresponding evaluation statistics (`TP`, `FP`, `FN`, `TN`, precision, recall, and hypothesis size). A small Python script parses these summaries and aggregates them into CSV files, one per domain.

The tables reported later in this chapter (e.g. Tables 4.1–**??**) are generated directly from these CSV files. This ensures that all quantitative results in the thesis are consistent with the raw Popper outputs and can be reproduced by rerunning the conversion and learning pipeline on the same SIFT JSON inputs.

4.5 Domain Instantiations

This section instantiates the general max–minus–min pipeline in each of the six domains. For every domain, it describes the minimal and maximal predicate sets, the resulting targets Δ, the construction of Popper tasks, and the main learning results.

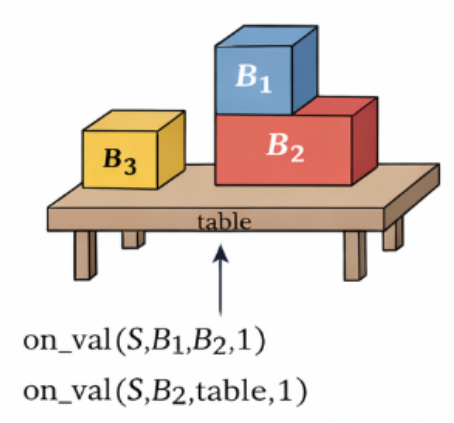

Figure 4.1: Block World example.

4.5.1 Block World Domain

Minimal and maximal domains. The Block World domain contains primitive fluents that describe stacking relations between blocks. In the setting used for this experiment, the minimal domain includes Boolean-valued variants of the core spatial predicate on, encoded as `on_val/4`, together with Boolean helper predicates. The maximal domain extends the minimal one with additional predicates extracted by SIFT from execution traces.

In this dataset, three predicates appear only in the maximal domain:

$$\Delta_{\text{BlockWorld}} = \{\texttt{learned\_pred\_3}, \texttt{learned\_pred\_4}, \texttt{learned\_pred\_5}\}.$$

Each of these is converted into a Popper task.

Task construction. For each `learned_pred_k`, the converter generates a Popper project directory containing `bk.pl`, `exs.pl`, `bias.pl`, and `config.pl`. The background knowledge encodes Boolean-valued stacking facts via `on_val/4`, along with `true/1`, `false/1`, and `neg/2`. The examples file `exs.pl` contains positive and negative instances of `learned_pred_k/2` or `/3` extracted from the maximal JSON.

Learning results. Popper successfully learns all three target predicates with perfect precision and recall. The learned hypotheses and their quantitative results are summarised in Table 4.1.

Interpretation. The learned predicates reveal two abstraction patterns over `on_val/4`. `learned_pred_3` and `learned_pred_4` are existential abstractions: they hold when a block has some block below it, or when some block is stacked above it, respectively. `learned_pred_5` captures a Boolean consistency constraint: the truth values of `on(V0,V1)` and `on(V1,V0)` must be opposite. All three predicates are reconstructed solely from the primitive stacking relation and Boolean inversion.

Table 4.1: Summary of learned predicates in the Block World domain (4ops).

Target Predicate	Precision	Recall	Size	Learned Rule
`learned_pred_3/2`	1.00	1.00	3	`learned_pred_3(V0,V1) :- true(V3), on_val(V0,V2,V1,V3).`
`learned_pred_4/2`	1.00	1.00	3	`learned_pred_4(V0,V1) :- true(V2), on_val(V3,V0,V1,V2).`
`learned_pred_5/3`	1.00	1.00	4	`learned_pred_5(V0,V1,V2) :- on_val(V0,V1,V2,V3), neg(V3,V4), on_val(V1,V0,V2,V4).`

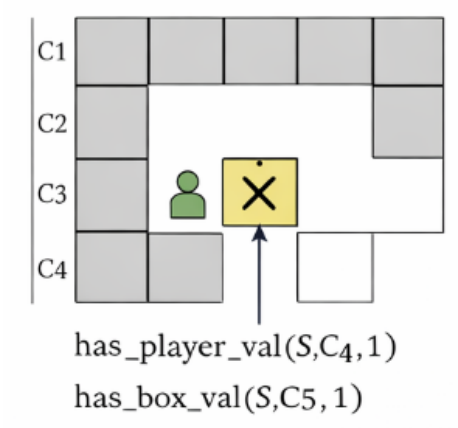

Figure 4.2: Sokoban example.

4.5.2 Sokoban Domain

Minimal and maximal domains. The Sokoban environment is a grid world in which each state is defined over a fixed set of cells. Exactly one cell contains the player, and zero or more cells may contain boxes. Walls and goal locations are part of the level description but are not explicitly represented in the background knowledge.

The minimal SIFT domain contains two primitive fluents:

$$\mathcal{P}_{\text{min}} = \{\texttt{has\_box}, \texttt{has\_player}\},$$

both defined over pairs of states and grid cells. The maximal domain introduces one additional abstract predicate:

$$\mathcal{P}_{\text{max}} = \{\texttt{has\_box}, \texttt{has\_player}, \texttt{learned\_pred\_1}\}.$$

Thus

$$\Delta_{\text{Sokoban}} = \mathcal{P}_{\text{max}} \setminus \mathcal{P}_{\text{min}} = \{\texttt{learned\_pred\_1}\}.$$

Both JSON files contain 1676 states. For each state, SIFT enumerates all grid cells and assigns Boolean truth values to the three predicates.

Task construction. A Python script (`gen_max_minus_min_projects.py`) automatically constructs a Popper task for `learned_pred_1`, resulting in:

```
/mnt/popper_projects/sokoban_p6_max_minus_min/
    learned_pred_1/
        bk.pl
        exs.pl
        bias.pl
        config.pl
```

`bk.pl` encodes Boolean-valued versions of `has_box` and `has_player` as `*_val` predicates, together with `true/1`, `false/1`, and `neg/2`. `exs.pl` contains positive and negative examples of `learned_pred_1(State, Cell)`, and `bias.pl` constrains the hypothesis space to short rules with at most two clauses, three body literals, and four variables.

Learning results. During the search, Popper first identifies an intuitive two-clause hypothesis (logical disjunction) and then selects a shorter, single-clause hypothesis that exploits explicit negation:

```
1 learned_pred_1(S,C) :-
2     has_box_val(S,C,Vb),
3     has_player_val(S,C,Vp),
4     neg(Vp, Vb).
```

The final rule achieves:

- Precision: 1.00
- Recall: 1.00
- True Positives (TP): 5028
- False Negatives (FN): 0
- False Positives (FP): 0

Interpretation. The learned predicate denotes that cell `C` is occupied in state `S`. In Sokoban, each cell can be occupied by either the player or a box, but never by both. Although the final hypothesis is expressed in an XOR-style form using `neg/2`, it is equivalent on this domain to the disjunction

$$\texttt{learned\_pred\_1}(S, C) \leftrightarrow \texttt{has\_box}(S, C) \vee \texttt{has\_player}(S, C).$$

This is a direct realisation of the max–minus–min strategy: a maximal-only occupancy predicate is fully reconstructed from minimal grid-level fluents.

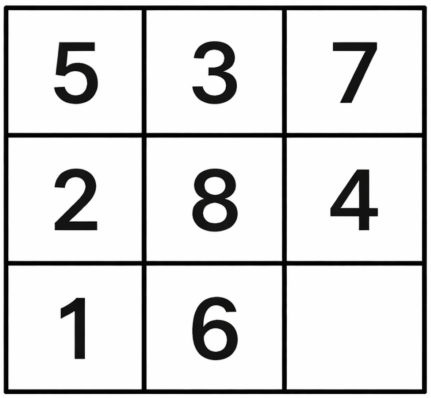

Figure 4.3: N-Puzzle example.

4.5.3 N-Puzzle Domain

Minimal and maximal domains. The N-Puzzle domain models a sliding-tile puzzle on a fixed-size grid. Tiles occupy grid cells, and one cell is blank. The minimal representation uses Boolean-valued predicates for tile locations and blank cells, such as `at_val/5` and `blank_val/4`, plus a type relation `obj_type/2`. The maximal representation adds SIFT-extracted precondition-like predicates.

The max–minus–min filter identifies eight target predicates in the updated dataset:

$$\Delta_{\text{NPuzzle}} = \{\texttt{learned\_pred\_0}, \texttt{learned\_pred\_1}, \texttt{learned\_pred\_10}, \texttt{learned\_pred\_11}, \texttt{learned\_pred\_12}, \texttt{learned\_pred\_13}, \texttt{learned\_pred\_14}, \texttt{learned\_pred\_16}\}. \quad (4.1)$$

Task construction. The converter places the original JSON files under `puzzleSIFT/sift_json` and generates one Popper project directory per target. For each target, `bk.pl` contains grid-level facts via `at_val/5`, `blank_val/4`, `obj_type/2`, and Boolean helpers. `exs.pl` lists positive and negative examples of the target predicate, and `bias.pl` restricts clause size and variable usage.

Learning results. Popper produces solutions for six targets: `learned_pred_0`, `learned_pred_1`, `learned_pred_10`, `learned_pred_12`, `learned_pred_14`, and `learned_pred_16`. Two targets, `learned_pred_11` and `learned_pred_13`, result in `NO SOLUTION`. The successful rules are strongly local: they rely only on direct state facts such as whether a location is blank or occupied by a tile, sometimes combined with `obj_type/2`.

Interpretation. Two patterns emerge. First, some rules (e.g. `learned_pred_0`, `learned_pred_1`, `learned_pred_14`) are almost direct projections of a background predicate and achieve perfect precision and recall. Second, other rules (`learned_pred_10`, `learned_pred_12`, `learned_pred_16`)

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule
learned_pred_0	1.00	1.00	3	360/0/360/0	learned_pred_0(S,C) :- obj_type(C,T), blank_val(S,C,_,T).
learned_pred_1	1.00	1.00	3	180/0/180/0	learned_pred_1(S,C) :- obj_type(C,T), blank_val(S,C,_,T).
learned_pred_10	1.00	0.67	3	1200/600/1800/0	learned_pred_10(S,A,B) :- true(T), at_val(S,A,_,B,T).
learned_pred_12	1.00	0.20	4	180/720/900/0	learned_pred_12(S,A,B) :- true(T), blank_val(S,B,_,T), at_val(S,A,_,_,T).
learned_pred_14	1.00	1.00	4	180/0/180/0	learned_pred_14(S,C) :- true(T), blank_val(S,_,_,T), at_val(S,_,C,_,T).
learned_pred_16	1.00	0.67	4	240/120/360/0	learned_pred_16(S,A,B) :- true(T), blank_val(S,_,A,T), at_val(S,_,_,B,T).

Table 4.2: Learned preconditions induced by Popper in the updated N-Puzzle experiment.

keep precision at 1.00 but lose recall: they capture only partial conditions because the background lacks relational predicates such as adjacency or ordering. The `NO SOLUTION` outcomes for `learned_pred_11` and `learned_pred_13` are consistent with this limitation.

4.5.4 Miconic Domain

Minimal and maximal domains. The Miconic domain models an elevator system with a lift and multiple floors. Passengers wait on specific floors and need to be transported to their target floors. Actions involve moving the lift, boarding passengers, and unloading them.

The minimal-domain background includes primitive predicates such as:

- `lift_pos(State, Floor)` — position of the lift;
- `in_floor(State, Passenger, Floor)` — passenger currently on a floor;
- `in_lift(State, Passenger)` — passenger currently inside the lift.

The maximal domain adds SIFT-extracted predicates that encode preconditions or higher-level state patterns. The max–minus–min filter yields three targets:

$$\Delta_{\text{Miconic}} = \{\texttt{learned\_pred\_1}, \texttt{learned\_pred\_5}, \texttt{learned\_pred\_6}\}.$$

Task construction. For each target, a Popper project directory is created under `miconic_preconds/`. The background knowledge is restricted to the minimal predicates above and Boolean helpers;

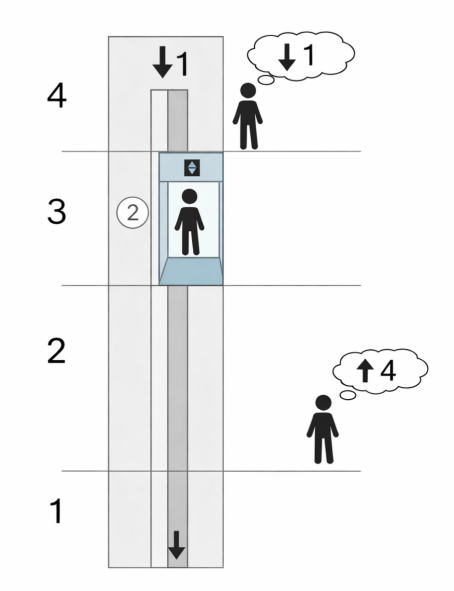

Figure 4.4: miconic example.

no action symbols, goal predicates, or target-floor information are provided. The bias limits the hypothesis space to rules with at most four body literals and six variables.

Learning results. All three target predicates are learned with perfect precision but varying recall. Table 4.3 summarises the quantitative results and schematic rule structures.

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule (schematic)
`learned_pred_1`	1.00	0.72	9	276/108/384/0	combination of `lift_pos(S,F)` with `in_floor(S,P,F')`
`learned_pred_5`	1.00	0.72	6	138/54/192/0	combination of `lift_pos(S,F)` with passenger on some floor
`learned_pred_6`	1.00	0.53	6	101/91/192/0	mainly defined in terms of `lift_pos(S,F)`

Table 4.3: Learned predicates induced by Popper in the Miconic domain using only minimal-domain background knowledge.

Interpretation. For `learned_pred_6`, Popper induces a simple hypothesis essentially equivalent to `lift_pos`, indicating that this maximal predicate adds no information beyond the minimal representation. In contrast, `learned_pred_1` and `learned_pred_5` combine lift position with existential conditions over passengers on floors, capturing higher-level relational abstractions. All rules are conservative (precision 1.00) but incomplete (recall < 1), suggesting that additional relational information—such as explicit goal conditions or service requests—would be needed for full coverage.

4.5.5 Logistics Domain

Minimal and maximal domains. The Logistics domain models packages, vehicles, and locations. The minimal representation uses Boolean-valued predicates for object locations and containment relations, such as `at/4` (object at location) and `in/4` (object in container), plus Boolean helpers. The maximal representation adds SIFT-extracted precondition-like predicates over the same traces.

The max–minus–min comparison yields three targets:

$$\Delta_{\text{Logistics}} = \{\texttt{learned\_pred\_0}, \texttt{learned\_pred\_3}, \texttt{learned\_pred\_4}\}.$$

Task construction. Each target predicate is converted into a Popper project under `logistics_preconds/`. `bk.pl` only contains minimal-domain predicates (`at/4`, `in/4`, Boolean helpers). `exs.pl` lists the positive and negative examples obtained from the maximal JSON. The bias again favours simple relational rules.

Learning results. Popper finds a complete solution for `learned_pred_4` but no verified solutions for `learned_pred_0` and `learned_pred_3` under the current configuration. For `learned_pred_4`, the induced rule is:

```
learned_pred_4(V0,V1,V2) :- in(V0,V1,V3,V2).
```

The evaluation reports perfect performance (Precision 1.00, Recall 1.00) with TP = 16704, TN = 18912, FP = 0, FN = 0 at size 2.

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule
`learned_pred_4`	1.00	1.00	2	16704/0/18912/0	`learned_pred_4(S,X,V) :- in(S,X,O,V).`
`learned_pred_0`	–	–	–	–	no verified solution under current configuration
`learned_pred_3`	–	–	–	–	no verified solution under current configuration

Table 4.4: Popper results in the Logistics domain under the max–minus–min setting.

Interpretation. `learned_pred_4` acts as an existential projection of the minimal-domain predicate `in/4`: one argument is abstracted away, while the remaining arguments are preserved. The learned predicate therefore behaves as a wrapper around an existing containment relation, rather than introducing new relational structure. For `learned_pred_0` and `learned_pred_3`, the lack of a verified solution is consistent with the combination of a restricted background, large example sets, and the cost of search; distinguishing representational limits from computational limits is left to future work.

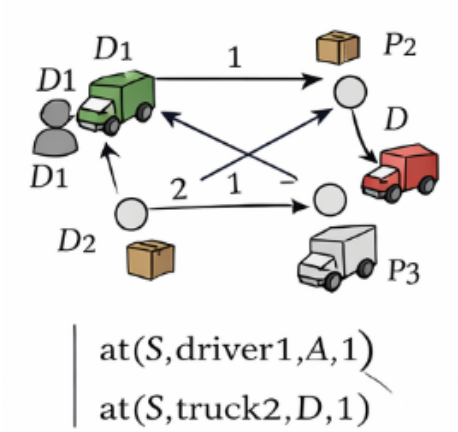

Figure 4.5: Driver's Log example.

4.5.6 Driverlog Domain

Minimal and maximal domains. The Driverlog domain models vehicles, drivers, packages, and locations. The minimal-domain background consists of Boolean-valued variants of location and auxiliary features (`*_val` predicates), together with `true/1`, `false/1`, and `neg/2`. The maximal domain adds precondition-like predicates extracted by SIFT.

The max–minus–min filter produces three target predicates that are treated as learning tasks:

$$\Delta_{\text{Driverlog}} = \{\texttt{learned\_pred\_0}, \texttt{learned\_pred\_4}, \texttt{learned\_pred\_12}\}.$$

Task construction. Each target has its own Popper project under `driverlog_preconds/`. `bk.pl` contains Boolean-valued minimal-domain predicates such as `at_val/4` and auxiliary features like `learned_pred_1_val/5` and `learned_pred_2_val/4`. `exs.pl` lists positive and negative instances of each target, and the bias limits body size and variable usage.

Learning results. Popper produces successful hypotheses for all three targets. The learned rules and their scores are summarised in Table 4.5.

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule
`learned_pred_0`	1.00	1.00	3	240/0/624/0	`learned_pred_0(S,X,Y) :- learned_pred_2_val(S,X,Y,V), true(V).`
`learned_pred_4`	1.00	0.82	3	236/52/1152/0	`learned_pred_4(S,X,Y) :- at_val(S,X,Y,V), true(V).`
`learned_pred_12`	1.00	1.00	3	48/0/816/0	`learned_pred_12(S,X,Y) :- learned_pred_1_val(S,X,Y,V1,V2), true(V2).`

Table 4.5: Learned preconditions successfully induced by Popper in the Driverlog domain.

Interpretation. `learned_pred_0` and `learned_pred_12` are defined solely in terms of auxiliary Boolean-valued predicates `learned_pred_2_val/4` and `learned_pred_1_val/5`, and require the respective value to be `true`. They therefore behave as thresholded versions of existing features. `learned_pred_4` depends directly on `at_val/4` and states that the precondition holds whenever the corresponding location fluent is true. Recall ranges between 0.82 and 1.00, indicating that the induced rules are conservative and do not overgeneralise; they reconstruct preconditions from a small set of Boolean-valued minimal predicates.

4.6 Quantitative Summary Across Domains

The max–minus–min filter produces a set of target predicates in each domain, and each target is learned as an independent Popper task. Table **??** summarises the number of targets, the number of solved and unsolved tasks, and the solved predicates under the current setting. In all domains, solved tasks report precision equal to 1.00, while recall depends on the expressiveness of the minimal background and the example structure.

4.7 Cross-Domain Discussion and Observations

Several common patterns emerge from the six domains:

- **Boolean-valued fluents and explicit negation.** Encoding all fluents as valued predicates with explicit Boolean arguments, together with `true/1`, `false/1`, and `neg/2`, provides a uniform representation across domains and avoids grounding problems associated with classical negation. The same mechanism supports mutual-exclusion and consistency constraints in Block World, Sokoban, and Driverlog.
- **Max–minus–min as a target selection method.** Restricting learning targets to predicates that appear in the maximal domain but not in the minimal domain yields a focused set of meaningful predicates. In Block World and Sokoban, these targets correspond to stacking abstractions and occupancy predicates. In N-Puzzle, Miconic, Logistics, and Driverlog, they often behave as wrappers or projections of minimal-domain fluents.
- **Local vs. relational expressiveness.** Many successful rules are based on local spatial properties: `on_val/4` in Block World, `has_box_val/3` and `has_player_val/3` in Sokoban, `at_val/5` and `blank_val/4` in N-Puzzle, and `lift_pos/2` in Miconic. Whenever the target predicate can be expressed using such local observations, Popper tends to find short hypotheses with high recall. When preconditions depend on relational structure not present in the minimal background (e.g. adjacency, distance, or goal relations), recall decreases or `NO SOLUTION` is reported.
- **Interpretable rule formats.** All learned rules across domains have simple and human-readable forms, typically using at most two or three body atoms. They can be described as

occupancy, stacking, containment, or position statements, often enriched with Boolean consistency.

- **`NO SOLUTION` and "no verified solution" as diagnostics.** `NO SOLUTION` in N-Puzzle indicates that the minimal background cannot express the target under the given bias, even though labeled examples exist. "No verified solution" in Logistics highlights computational limits (e.g. search cost on large example sets) rather than representational impossibility. Together, these outcomes provide diagnostic signals about which maximal predicates are expressible from the minimal representation and which require additional relational structure or different search settings.

Overall, the experiments demonstrate that the combination of a max–minus–min target selection scheme, a Boolean-valued representation, and Popper's structured hypothesis search can recover many maximal-domain predicates from minimal-domain fluents alone, and that the resulting rules are both quantitatively accurate (precision 1.00) and qualitatively interpretable.

4.8 Recall Fixing and Relational Extensions

To investigate potential strategies for improving recall, we explored two extensions in N-Puzzle domain: (i) reducing positive examples via sequential covering, and (ii) enriching the background knowledge with relational predicates.

4.8.1 Two Strategies Applied

1. Positive reduction (sequential covering). The first strategy aimed to improve recall by isolating the structure not captured by the initial learned clause. After learning the first hypothesis H_1 for `learned_pred_10/3`, we evaluated H_1 on all training examples and removed every positive example already entailed by H_1. The remaining training set therefore consisted only of uncovered positive examples together with all original negative examples. Popper was then executed again to attempt to induce a second clause H_2, with the intended final hypothesis being $H = H_1 \vee H_2$.

In practice, Popper returned *no solution* in this second round. The remaining uncovered positives did not share a coherent structural pattern that could be generalised into a consistent clause. This indicates that the primary learnable regularity of the predicate was already captured by H_1, and that the residual examples do not form a stable secondary rule.

2. Background enrichment (relational predicates). The second strategy focused on increasing the expressive power of the background knowledge. We introduced relational predicates derived from the grid structure, such as `left_of/2`, `neighbor/2`, and `same_row/2`, and allowed them as admissible body literals in the bias specification. These predicates provide

relational information that is absent in the minimal-domain encoding, enabling Popper to reason about relative positions rather than absolute coordinates alone.

With this enriched background knowledge, Popper was rerun on the original training set. In this setting, recall exhibited a modest improvement from 0.67 to 0.78 while precision remained high. The learned clause incorporated relational structure, suggesting that the additional predicates provided useful signals for generalisation. However, the improvement was limited in magnitude.

4.8.2 Analysis

The contrasting outcomes of these two approaches highlight an important distinction. Sequential covering attempts to extract additional structure from the remaining examples, but its success depends on the existence of a consistent secondary pattern, which was not present in this case. By contrast, enriching the background knowledge expands the hypothesis space and may allow Popper to exploit relational regularities, though the effect remains constrained by the diversity and structure of the available traces.

Overall, these experiments suggest that recall improvements depend more on the structural richness of the traces than on purely procedural adjustments. While relational extensions show potential, their empirical impact in the current dataset is limited.

5 Results

This chapter presents the experimental results obtained with the Popper ILP system. All rules were learned from state–action traces generated by Clingo and exported through SIFT JSON files, as described in Chapter 4. The chapter is organised in two parts:

- Section 5.1 reports preliminary experiments on small, hand-crafted domains (grid world, classical Block World, and Sokoban), which were used to probe Popper's behaviour before introducing the max–minus–min pipeline.
- Section 5.2 then reports the main results for the six planning domains that use the max–minus–min target selection strategy: Block World, Sokoban, N-Puzzle, Miconic, Logistics, and Driverlog.

The final sections summarise quantitative outcomes and analyse how representation choices influence the observed learning behaviour.

5.1 Preliminary Experiments on Toy Domains

Before applying the max–minus–min strategy to SIFT-based domains, a set of smaller experiments was conducted to validate the use of Popper in spatial and temporal settings. These experiments operate on hand-crafted domains with manually specified background knowledge and examples.

5.1.1 Grid World: Learning Co-location Rules

The grid domain contains a person and a key placed on a 3×3 grid. The target predicate is `can_pick_up/3`, which represents the condition under which the person can pick up the key. Popper receives examples in which the person and key are located either in the same cell or in different cells.

The learned rule is:

```
1 can_pick_up(P, K, S) :-
2     at(P, C, S),
3     at(K, C, S).
```

This rule expresses that a person can pick up the key exactly when both occupy the same cell. Popper reaches precision $= 1.00$ and recall $= 1.00$, meaning that every positive example is

captured and no false positives occur. The rule is minimal (one head literal and two body literals) and matches the intuitive human definition of co-location.

5.1.2 Classical Block World: Learning Pick-Up Preconditions

The classical Block World experiment aims to learn the predicate `pickable/2`, which states that a block can be picked up in a given state. The environment includes facts such as `on/3`, `clear/2`, and `on_table/2`.

In an early version of the experiment, the learner failed because Popper prunes hypotheses containing classical negation. This issue was resolved by defining `clear/2` and `on_table/2` as positive background knowledge instead of using negative literals in hypotheses. After this modification, Popper produces the following hypothesis:

```
1 pickable(B, S) :-
2     clear(B, S),
3     on_table(B, S).
```

The rule captures that a block is pickable when it is clear and lies on the table. The learned program matches all positive and negative examples (precision = 1.00, recall = 1.00). This experiment shows that Popper can recover the logical structure of classical planning preconditions once negation is encoded at the knowledge level and clauses remain safe.

5.1.3 Sokoban: Spatial Preconditions for Push

In the Sokoban domain, an agent pushes boxes through a maze of walls and empty cells. The learning goal is to decide when a push action is possible. The target predicate is `can_push/3`, defined over the person, the box, and the current state. The background knowledge includes predicates such as `box/3`, `free/2`, and `adj/3`. To simplify learning, abstract predicates such as `co_located/3` and `box_has_space_right/2` are introduced to encode mid-level spatial patterns.

The final rule learned by Popper is:

```
1 can_push(S, P, B) :-
2     box(B, S, Cb),
3     free(S, Cd),
4     adj(Cb, Cd, Dir),
5     adj(Cp, Cb, Dir),
6     person(P, S, Cp).
```

The rule states that a person can push a box when the box has a free cell in the pushing direction and the person stands adjacent to the box from the same direction. Popper reaches precision =

1.00 and recall = 0.67: all predicted pushes are correct, but some valid cases are missed. The missing cases occur primarily in directions that are underrepresented in the examples. The experiment illustrates how directional symmetry and data distribution influence generalisation, and how invented predicates can make complex spatial dependencies learnable.

5.1.4 Temporal Preconditions in Sokoban

A temporal variant of the Sokoban experiment extends the background knowledge to two consecutive states. The goal is to learn when a push action succeeds, considering both spatial configuration and recent movement. The background includes `box/3`, `free/2`, `adj/3`, `apl_move/4`, and `prev_state/2`.

The rule learned by Popper is:

```
1 can_push(S, P, B) :-
2     box(B, S, Cb),
3     free(S, Cd),
4     adj(Cb, Cd, Dir),
5     adj(Cp, Cb, Dir),
6     apl_move(Sp, S, P, Cp).
```

This rule indicates that a push action occurs if the box has a free cell in the target direction, the person is located at the adjacent cell on the opposite side, and the previous state shows that the person moved into that cell. The hypothesis reaches precision = 1.00 and recall = 1.00, showing that Popper can learn temporal preconditions when state transitions are explicitly represented in the background knowledge.

These preliminary experiments motivated the design choices used in the max–minus–min pipeline, in particular the avoidance of classical negation and the use of mid-level abstractions in the background knowledge.

5.2 Max–Minus–Min Results Across Six Domains

The main experiments apply the max–minus–min strategy to six planning domains using SIFT-generated traces: Block World, Sokoban, N-Puzzle, Miconic, Logistics, and Driverlog. For each domain, the learning targets are the predicates that occur only in the maximal domain representation and not in the minimal one. The background knowledge is restricted to minimal-domain predicates encoded in Boolean-valued form, together with Boolean helpers (`true/1`, `false/1`, `neg/2`). Popper is run once per target predicate.

5.2.1 Block World

In the SIFT-based Block World experiments, the minimal domain contains Boolean-valued variants of the stacking predicate `on/3` encoded as `on_val/4`. The maximal domain adds three SIFT-extracted predicates that appear only in the maximal representation:

$$\Delta_{\text{BlockWorld}} = \{\texttt{learned\_pred\_3}, \texttt{learned\_pred\_4}, \texttt{learned\_pred\_5}\}.$$

For each target, Popper receives background knowledge `bk.pl` with `on_val/4` and Boolean helpers, examples `exs.pl` derived from the maximal JSON, and a bias file restricting hypothesis size.

Popper successfully induces all three targets with perfect precision and recall. The learned hypotheses are:

```
learned_pred_3(V0,V1) :-
    true(V3),
    on_val(V0,V2,V1,V3).

learned_pred_4(V0,V1) :-
    true(V2),
    on_val(V3,V0,V1,V2).

learned_pred_5(V0,V1,V2) :-
    on_val(V0,V1,V2,V3),
    neg(V3,V4),
    on_val(V1,V0,V2,V4).
```

All three rules obtain precision = 1.00 and recall = 1.00. The first two predicates form an existential pair: they hold when a block has some block below it, or when some block is stacked above it, respectively. The third predicate captures a Boolean consistency constraint between the truth values of `on(V0,V1)` and `on(V1,V0)`. These results show that the maximal-only predicates in this Block World dataset can be fully reconstructed from minimal-domain stacking facts and Boolean inversion.

5.2.2 Sokoban

In the SIFT-based Sokoban experiments, the minimal domain contains two primitive fluents, `has_box` and `has_player`, each encoded in Boolean-valued form as `has_box_val/3` and `has_player_val/3`. The maximal domain adds one additional predicate:

$$\Delta_{\text{Sokoban}} = \{\texttt{learned\_pred\_1}\}.$$

The learned rule is:

```
learned_pred_1(S,C) :-
    has_box_val(S,C,Vb),
    has_player_val(S,C,Vp),
    neg(Vp, Vb).
```

This rule says that `learned_pred_1(S,C)` holds exactly when the cell `C` is occupied in state `S`, i.e. when either the player or a box is present but not both. On this dataset, the rule achieves precision $= 1.00$ and recall $= 1.00$ (TP $= 5028$, FP $= 0$, FN $= 0$). The XOR-style form using `neg/2` is equivalent to the disjunction

$$\texttt{has\_box}(S, C) \vee \texttt{has\_player}(S, C),$$

illustrating how a maximal-only occupancy predicate can be reconstructed from minimal fluents.

5.2.3 N-Puzzle

In the N-Puzzle domain, the minimal representation provides Boolean-valued predicates for tile and blank locations, such as `at_val/5` and `blank_val/4`, as well as a type relation `obj_type/2`. The maximal domain adds SIFT-extracted preconditions that do not appear in the minimal vocabulary. The max–minus–min filter identifies eight targets in the updated dataset:

$$\Delta_{\text{NPuzzle}} = \left\{ \begin{matrix} \texttt{learned\_pred\_0}, \texttt{learned\_pred\_1}, \texttt{learned\_pred\_10}, \texttt{learned\_pred\_11}, \\ \texttt{learned\_pred\_12}, \texttt{learned\_pred\_13}, \texttt{learned\_pred\_14}, \texttt{learned\_pred\_16} \end{matrix} \right\}.$$

Popper finds solutions for six of these. The successful rules are local: they use only direct state facts about tile or blank positions and, in some cases, type information. Table 5.1 summarises the learned rules and their metrics.

All solved predicates achieve precision $= 1.00$, but several have recall below 1.00. Rules for `learned_pred_0`, `learned_pred_1`, and `learned_pred_14` can be seen as direct projections of `blank_val/4` and `at_val/5`, and reach perfect recall. Rules for `learned_pred_10`, `learned_pred_12`, and `learned_pred_16` combine tile and blank information but only express partial conditions; they cover a subset of the true cases while rejecting all negatives.

For `learned_pred_11` and `learned_pred_13`, Popper reports `NO SOLUTION` under the current bias. In one case, the maximal JSON contains no positive instances, so the task degenerates. In the other, inspection of argument patterns suggests that the predicate depends on implicit ordering relations between grid positions (e.g. vertical order between coordinates), which are

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule
learned_pred_0	1.00	1.00	3	360/0/360/0	learned_pred_0(S,C) :- obj_type(C,T), blank_val(S,C,_,T).
learned_pred_1	1.00	1.00	3	180/0/180/0	learned_pred_1(S,C) :- obj_type(C,T), blank_val(S,C,_,T).
learned_pred_10	1.00	0.67	3	1200/600/1800/0	learned_pred_10(S,A,B) :- true(T), at_val(S,A,_,B,T).
learned_pred_12	1.00	0.20	4	180/720/900/0	learned_pred_12(S,A,B) :- true(T), blank_val(S,B,_,T), at_val(S,A,_,_,T).
learned_pred_14	1.00	1.00	4	180/0/180/0	learned_pred_14(S,C) :- true(T), blank_val(S,_,_,T), at_val(S,_,C,_,T).
learned_pred_16	1.00	0.67	4	240/120/360/0	learned_pred_16(S,A,B) :- true(T), blank_val(S,_,A,T), at_val(S,_,_,B,T).

Table 5.1: Learned preconditions induced by Popper in the updated N-Puzzle experiment.

not present in the minimal background. Because Popper does not invent numerical comparison predicates, such ordering constraints are not expressible in the hypothesis language, leading to the observed `NO SOLUTION` outcome.

5.2.4 Miconic

The Miconic domain models an elevator system with a lift and multiple floors. The minimal-domain background provides primitive predicates `lift_pos/2`, `in_floor/3`, and `in_lift/2`, together with Boolean helpers. The maximal domain adds several SIFT-extracted preconditions. The max–minus–min filter yields three targets:

$$\Delta_{\text{Miconic}} = \{\texttt{learned\_pred\_1}, \texttt{learned\_pred\_5}, \texttt{learned\_pred\_6}\}.$$

Popper learns hypotheses for all three predicates with precision = 1.00 but recall below 1.00. The induced rules primarily rely on `lift_pos/2` and, in some cases, existential conditions over passengers on floors. A schematic description is given in Table 5.2.

For `learned_pred_6`, the dominant rule is essentially equivalent to `lift_pos/2`, indicating that this maximal predicate does not encode additional information beyond the minimal representation. The other two predicates link lift position with passenger distribution on floors, capturing higher-level relational abstractions but only partially covering the examples. The reduced recall reflects missing directional and ordering information in the background: SIFT

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule (schematic)
`learned_pred_1`	1.00	0.72	9	276/108/384/0	combination of `lift_pos(S,F)` with `in_floor(S,P,F')`
`learned_pred_5`	1.00	0.72	6	138/54/192/0	combination of `lift_pos(S,F)` with passenger on some floor
`learned_pred_6`	1.00	0.53	6	101/91/192/0	mainly defined in terms of `lift_pos(S,F)`

Table 5.2: Learned predicates induced by Popper in the Miconic domain using minimal-domain knowledge.

features suggest that some maximal predicates depend on whether the lift lies above or below another floor, but no explicit ordering relations between floors are available to Popper.

5.2.5 Logistics

The Logistics domain models packages, vehicles, and locations, with actions such as driving, loading, and unloading. The minimal representation uses Boolean-valued predicates for object locations and containment relations, including `at/4` and `in/4`, plus Boolean helpers. The maximal representation adds precondition-like predicates derived by SIFT. The max–minus–min filter identifies three target predicates:

$$\Delta_{\text{Logistics}} = \{\texttt{learned\_pred\_0}, \texttt{learned\_pred\_3}, \texttt{learned\_pred\_4}\}.$$

Under the current configuration, Popper finds a stable hypothesis only for `learned_pred_4`. The induced rule is:

```
1 learned_pred_4(S,X,V) :-
2     in(S,X,O,V).
```

This rule achieves precision = 1.00 and recall = 1.00 (TP = 16704, FN = 0, FP = 0, TN = 18912) at size 2. It defines `learned_pred_4` as an existential projection of `in/4`, acting as a wrapper around an existing containment relation.

For `learned_pred_0` and `learned_pred_3`, the runs do not yield a verified hypothesis. Although these tasks contain many labeled examples, the search does not converge to a hypothesis with a complete and parsable evaluation summary within the current bias and runtime limits. These cases are therefore reported as *no verified solution* rather than `NO SOLUTION`. A plausible explanation is that these targets encode transport-mode and connectivity-related structure (e.g. road vs. air links) that is not present in the minimal background; under such restrictions, Popper may require richer background relations or larger hypotheses than the current bias permits.

5.2.6 Driverlog

The Driverlog domain models drivers, trucks, packages, and locations. The minimal-domain background contains Boolean-valued variants of location predicates and auxiliary features, such as `at_val/4`, `learned_pred_1_val/5`, and `learned_pred_2_val/4`, plus Boolean helpers. The maximal domain adds precondition-like predicates identified by SIFT. The max–minus–min filter produces three target predicates:

$$\Delta_{\text{Driverlog}} = \{\texttt{learned\_pred\_0}, \texttt{learned\_pred\_4}, \texttt{learned\_pred\_12}\}.$$

Popper induces successful hypotheses for all three targets. The learned rules and their metrics are shown in Table 5.3.

Predicate	Precision	Recall	Size	TP/FN/TN/FP	Learned Rule
`learned_pred_0`	1.00	1.00	3	240/0/624/0	`learned_pred_0(S,X,Y) :- learned_pred_2_val(S,X,Y,V), true(V).`
`learned_pred_4`	1.00	0.82	3	236/52/1152/0	`learned_pred_4(S,X,Y) :- at_val(S,X,Y,V), true(V).`
`learned_pred_12`	1.00	1.00	3	48/0/816/0	`learned_pred_12(S,X,Y) :- learned_pred_1_val(S,X,Y,V1,V2), true(V2).`

Table 5.3: Learned preconditions successfully induced by Popper in the Driverlog domain.

All three predicates achieve precision = 1.00, while recall ranges from 0.82 to 1.00. `learned_pred_0` and `learned_pred_12` are defined solely in terms of auxiliary Boolean-valued predicates and require the corresponding value to be `true`, effectively acting as thresholded versions of existing features. `learned_pred_4` depends directly on `at_val/4` and states that the precondition holds whenever the corresponding `at` fluent is true in the current state. These patterns show that several Driverlog preconditions can be reconstructed from simple state occupancy relations and auxiliary features in the minimal representation.

5.3 Summary of Quantitative Outcomes

Across the six max–minus–min domains, the filter produces a total of 19 target predicates. Table 5.4 summarises the number of targets per domain, the number of solved and unsolved tasks, and the typical ranges of recall for solved predicates. Precision is 1.00 in all solved cases.

A recurring pattern is that all solved targets reach precision 1.00, but some have recall strictly below 1.00, and some targets are reported as `NO SOLUTION` or have no verified solution. In practice, these three outcome types delimit the expressive frontier of the minimal representation: high-recall rules correspond to maximal predicates that are fully reconstructable from

Domain	Targets	Solved	Unsolved	Precision	Recall	Solved targets
Block World	3	3	0	1.00	1.00	learned_pred_3, learned_pred_4, learned_pred_5
Sokoban	1	1	0	1.00	1.00	learned_pred_1
N-Puzzle	8	6	2 (NO SOLUTION)	1.00	0.20–1.00	learned_pred_0, learned_pred_1, learned_pred_10, learned_pred_12, learned_pred_14, learned_pred_16
Miconic	3	3	0	1.00	0.53–0.72	learned_pred_1, learned_pred_5, learned_pred_6
Logistics	3	1	2 (no verified solution)	1.00	1.00	learned_pred_4
Driverlog	3	3	0	1.00	0.82–1.00	learned_pred_0, learned_pred_4, learned_pred_12

Table 5.4: Quantitative summary of max–minus–min learning outcomes across domains. "Unsolved" includes both NO SOLUTION and cases without a verified hypothesis.

local, Boolean-valued fluents; low-recall rules correspond to predicates that are only partially captured by the available background; and NO SOLUTION or missing solutions indicate predicates whose behaviour depends on relational structure that is absent from the minimal domain or too expensive to search under the current bias.

5.4 Cross-Domain Learning Patterns

Beyond domain-specific observations, the results exhibit several systematic patterns that recur across all six max–minus–min domains. These patterns characterise the types of maximal-only predicates that can be reconstructed from minimal-domain fluents, the kinds of predicates for which recall remains partial, and the structural reasons underlying cases where Popper reports NO SOLUTION or fails to verify a hypothesis.

5.4.1 Projection-Like Predicates Are Consistently Learnable

Across all domains, predicates that correspond to existential projections of minimal-domain fluents are learned with precision 1.00 and, in most cases, recall 1.00. These predicates typically depend on the presence of a single fluent with no additional relational constraints. Examples include:

- Block World: learned_pred_3 and learned_pred_4, both defined in terms of on_val/4;

- Sokoban: `learned_pred_1`, reconstructed as an XOR-style combination of `has_box_val/3` and `has_player_val/3`;
- Logistics: `learned_pred_4`, an existential projection of `in/4`;
- Driverlog: `learned_pred_0` and `learned_pred_12`, both reductions of Boolean-valued auxiliary predicates.

These results show that maximal predicates which collapse to local occupancy conditions or Boolean projections are fully expressible under the constraints of the minimal representation and Popper's bias.

5.4.2 Recall Degradation Correlates with Missing Relational Structure

Predicates that require relational information absent from the minimal-domain background regularly exhibit reduced recall. The most common missing structure includes:

- ordering relations between grid coordinates (N-Puzzle);
- vertical relations between floors (Miconic);
- directionally constrained neighbourhoods (Sokoban).

In these cases, Popper consistently induces hypotheses with precision 1.00 but recall strictly below 1.00. These partial rules capture a subset of the true positive cases but cannot express the full relational dependency required by the maximal predicate. The behaviour is therefore attributable not to the data but to the expressiveness of the minimal background.

5.4.3 Two Distinct Failure Modes

The results also exhibit two qualitatively different forms of failure:

(1) `NO SOLUTION`. This outcome occurs when the maximal predicate depends on relational structure that is not expressible in the minimal-domain hypothesis language. In the N-Puzzle domain, for instance, one target predicate contains no positive instances, while another requires implicit coordinate ordering.

(2) No verified solution. This case arises when Popper explores a hypothesis space that is too large to fully evaluate under the current bias or time limits. The Logistics domain exhibits this behaviour: large numbers of examples combined with weakly constrained argument structure lead to search divergence, even though the predicate may in principle be expressible.

Together, these patterns clarify the empirical frontier of what can and cannot be reconstructed from minimal-domain fluents using ILP.

5.5 Failure Case Typology

To better understand the conditions under which learning fails, the observed failure outcomes can be grouped into three distinct categories. These categories recur across domains and correspond to different structural limitations of the minimal representation or the hypothesis search.

5.5.1 Type I: Degenerate Targets (No Positive Instances)

Some maximal-only predicates contain no positive instances in the SIFT-derived JSON files. In these cases, Popper correctly reports `NO SOLUTION`. The failure is not indicative of a limitation of the minimal background or the hypothesis language; the task degenerates because no rule can satisfy the example constraints. One of the N-Puzzle targets exhibits this behaviour.

5.5.2 Type II: Relationally Under-Specified Targets

A second class of failures arises when the maximal predicate depends on relational or ordering information not represented in the minimal domain. For such targets, Popper either reports `NO SOLUTION` or induces a rule with precision 1.00 but low recall. Examples include:

- predicates requiring vertical ordering between floors in Miconic;
- predicates requiring coordinate ordering or distance in N-Puzzle;
- predicates requiring explicit direction relations in Sokoban.

These targets are semantically meaningful but not expressible under the Boolean-valued minimal representation, which lacks relations such as numerical comparison, adjacency, or reachability.

5.5.3 Type III: Search-Limited Targets

A third class of failures occurs when Popper does not verify a hypothesis within the given bias limits, despite the predicate being in principle expressible. This behaviour appears in the Logistics experiments, where the large number of examples and weak argument constraints yield a hypothesis space too large for the configured search bounds. These outcomes are reported as “no verified solution” rather than `NO SOLUTION` because the lack of a hypothesis arises from search complexity, not representational insufficiency.

This typology helps distinguish between representational limitations and search-related failures, providing a more fine-grained interpretation of learning outcomes across domains.

5.6 Impact of Representation Choices on Learning Outcomes

The experimental results are shaped not only by the learning algorithm itself, but also by a set of explicit representation and bias choices that are kept fixed across domains. This section summarises how these choices influence the observed outcomes.

5.6.1 Boolean-Valued Fluents and Explicit Negation

All background predicates are encoded in Boolean-valued form, where each fluent carries an explicit truth value and absence information is represented numerically. This avoids the use of classical negation inside learned hypotheses and allows negation to be handled via the `neg/2` predicate.

The results show that this encoding is sufficient to express a wide range of constraints observed in the maximal domains. In Block World and Sokoban, Popper learns mutual exclusion and consistency relations using `neg/2` over Boolean values. Without Boolean-valued fluents, several learned rules would not be expressible within Popper's hypothesis space, because clauses containing classical negation are pruned during search.

5.6.2 Local Background Knowledge and Relational Limits

Another consistent pattern concerns the expressiveness of the minimal background knowledge. In N-Puzzle and Miconic, predicates that depend only on local state properties, such as tile presence or lift position, are often learnable with high precision and, sometimes, high recall. In contrast, predicates that require relational structure beyond local observations—such as adjacency, ordering, or connectivity—frequently result in reduced recall or `NO SOLUTION`.

These behaviours are not caused by insufficient data, but by the absence of relational predicates in the minimal domain. Since Popper does not invent new comparison or connectivity predicates, constraints that rely on relations such as $y_1 > y_2$ or reachability between locations are simply not expressible in the hypothesis language.

5.6.3 Effect of Bias Constraints

The hypothesis space explored by Popper is constrained through bias settings that limit clause length, number of variables, and number of literals. These limits are necessary to keep search tractable across domains with thousands of examples. The learned rules show that these constraints favour compact abstractions, such as existential projections or simple Boolean combinations. While this leads to high precision and interpretable hypotheses, it also restricts the learner's ability to express more complex relational conditions.

5.6.4 Single-Clause Hypotheses and Disjunctive Concepts

A further limitation of the current configuration is that, under the imposed bias, Popper is effectively restricted to producing a single clause per target predicate. This is sufficient when the underlying concept is structurally uniform, but it becomes problematic when positive instances arise from multiple distinct patterns. In such genuinely disjunctive cases, any single clause can only capture one of the patterns, inevitably leaving some positive examples uncovered and leading to reduced recall.

A standard remedy in ILP is *sequential covering* (also known as separate-and-conquer). Instead of learning a single clause, the learner iteratively constructs a set of rules:

1. learn an initial rule R_1 that covers a subset of the positive examples while avoiding negatives;
2. remove all positive examples correctly entailed by R_1 while keeping all negative examples;
3. learn a new rule R_2 on the remaining dataset and repeat.

The resulting rule set $\{R_1, R_2, \dots, R_k\}$ can express disjunctive concepts as a union of simpler patterns, potentially raising recall towards 1.00 even when each individual clause is only partially correct. The prevalence of high-precision but low-recall hypotheses in domains such as N-Puzzle and Miconic suggests that some maximal predicates are indeed multi-modal in this sense. While implementing a full sequential-covering regime lies outside the scope of the present experiments, the observed outcome patterns indicate that multi-clause learning would be a natural next step for improving coverage without abandoning interpretability.

5.6.5 Role of the Max–Minus–Min Strategy

Finally, the max–minus–min strategy helps stabilise learning across domains by focusing on predicates that are empirically meaningful: they appear in the maximal domain but not in the minimal one. Many of these predicates collapse to simple projections of minimal fluents, leading to rules with precision and recall equal to 1.00. When learning fails, the failure is itself informative: either the predicate has no positive instances, or it depends on relations that are absent from the minimal domain or too costly to search with the current bias.

Overall, the combination of Boolean-valued fluents, explicit negation, controlled bias, and the max–minus–min target selection creates a learning setting in which both success and failure are interpretable. The rules learned in this chapter thus serve not only as models of domain preconditions, but also as diagnostics of the expressive sufficiency of the underlying representations.

Relation to the Research Question. Taken together, the empirical results provide a direct answer to the central question of whether maximal-domain predicates can be reconstructed from minimal representations using ILP. Three outcome categories emerge. First, predicates

that correspond to local Boolean projections are fully recoverable, achieving precision and recall equal to 1.00 across domains. Second, predicates that depend on relational structure missing from the minimal domain are only partially recoverable: Popper induces precise but incomplete abstractions with reduced recall. Third, some predicates cannot be recovered at all, either because they contain no positive instances or because their defining relations—such as ordering or connectivity—lie outside the expressive limits of the minimal background. Thus, the experiments reveal a clear expressive frontier: minimal-domain fluents support reconstruction of local conditions but not relationally structured preconditions.

6 Conclusion

This thesis investigated a specific predicate-learning problem at the intersection of symbolic planning and inductive logic programming. Given two symbolic representations of the same state–action traces, a *minimal* and a *maximal* domain, the central question was:

> When can predicates that appear only in the maximal representation be reconstructed from the minimal one using ILP over grounded execution traces?

From a set of traces generated by a planning domain, two vocabularies are derived. The minimal domain contains primitive predicates that encode low-level properties of the environment. The maximal domain extends this vocabulary with additional predicates extracted by SIFT, intended to capture higher-level regularities observed in the traces. Let $\mathcal{P}_{\text{min}}$ and $\mathcal{P}_{\text{max}}$ denote the predicate sets of the minimal and maximal domains. The learning targets are defined by their set difference

$$\Delta = \mathcal{P}_{\text{max}} \setminus \mathcal{P}_{\text{min}}.$$

Each predicate $p \in \Delta$ is treated as an independent Popper task: given labelled examples of p extracted from the maximal-domain traces, the goal is to induce a compact, interpretable logical definition of p using only predicates from $\mathcal{P}_{\text{min}}$ plus Boolean helper relations.

The aim of this setting is not to reconstruct full action models, but to assess the expressive power of minimal-domain representations. Successful learning shows that a maximal predicate does not introduce fundamentally new information beyond what is already present in the minimal vocabulary, whereas systematic failure indicates that additional relational structure or richer supervision is required.

6.1 Summary of Contributions and Findings

The thesis makes three main contributions.

6.1.1 A Cross-Domain Max–Minus–Min Pipeline

First, it introduces a unified learning pipeline that applies the max–minus–min strategy across multiple planning benchmarks. The pipeline combines:

- grounded state–action traces generated by Clingo,
- SIFT-derived minimal and maximal domain vocabularies,
- Boolean-valued encodings of minimal-domain fluents, and

- Popper as the ILP engine under a fixed, interpretable bias.

This setup is instantiated in six domains: Block World, Sokoban, N-Puzzle, Miconic, Logistics, and Driverlog. In all cases, the learning mechanism, background encoding, and evaluation protocol remain unchanged; only the domain and the SIFT-derived targets differ. Preliminary experiments on small, hand-crafted domains motivate design choices such as Boolean-valued fluents and explicit negation via `neg/2`.

6.1.2 Three Outcome Types Across Domains

Second, the thesis empirically characterises the outcome space induced by max–minus–min learning. Across all domains, the targets fall into three qualitative categories:

- **Full reconstruction**: Popper finds definitions with precision and recall equal to 1.00. These predicates typically correspond to existential projections, simple Boolean combinations, or consistency constraints over minimal fluents.
- **Partial reconstruction**: Popper finds high-precision but low-recall rules. These hypotheses capture necessary but not sufficient conditions for the targets, reflecting either missing relational structure in the minimal domain or the preference for compact programs under the current bias.
- **Unsolved targets**: Popper reports `NO SOLUTION` or fails to produce a verified hypothesis. This occurs when traces provide too few positive instances, when relevant relations are not expressible in the minimal vocabulary, or when valid definitions exceed the imposed bias limits.

Taken together, these outcome types delimit the expressive frontier of the minimal representation: fully reconstructed predicates are redundant with respect to the minimal vocabulary, partially reconstructed predicates expose representational gaps, and unsolved predicates flag concepts that require richer background knowledge or looser bias constraints.

6.1.3 ILP as a Diagnostic for Symbolic Representations

Third, the thesis argues that ILP, combined with the max–minus–min strategy, can be used as a diagnostic tool for symbolic representations rather than only as a predictive mechanism. By systematically testing whether maximal-domain predicates can be defined from minimal fluents, the pipeline identifies:

- predicates that can safely be removed or replaced by learned definitions, simplifying domain descriptions without loss of expressiveness; and
- predicates whose behaviour cannot be reconstructed, indicating where additional relational predicates, comparison operators, or structural background knowledge are genuinely needed.

In this way, ILP becomes a method for probing the adequacy of symbolic interfaces between simulators, feature extractors such as SIFT, and planners.

6.2 Limitations

The empirical analysis also highlights several limitations of the current setup.

6.2.1 Data coverage

Trace-based learning is inherently dependent on behavioural coverage. When the maximal JSON contains few or no positive instances of a target predicate, the corresponding Popper task becomes ill-defined and typically results in `NO SOLUTION`. Even when positives exist, skewed distributions can bias the learned rules toward particular spatial or relational patterns.

6.2.2 Representational gaps

Some maximal-only predicates implicitly depend on relations that are not present in the minimal vocabulary, such as adjacency, ordering, or connectivity. Since Popper does not invent such relations, they must be provided as background predicates; otherwise, correct definitions may simply lie outside the hypothesis space.

6.2.3 Bias and search constraints

Finally, the hypothesis space is restricted by bias settings that limit clause length, number of literals, and variables. These constraints are necessary for tractability, but they favour compact, high-precision rules over more complex definitions with higher recall. In this sense, the observed failures reflect a trade-off between interpretability and completeness.

6.3 Outlook and Future Work

The findings of this thesis suggest several directions for further research.

6.3.1 Richer background relations

One natural extension is to enrich the minimal background with domain-specific relational predicates, such as adjacency, order, or connectivity. This would allow a more direct test of whether unresolved targets become learnable once the missing structure is made explicit.

6.3.2 Temporal and action-level targets

The present work focuses on state-level predicates derived by the max–minus–min filter. Extending the approach to temporal and action-specific targets, using background knowledge that links successive states, would bring the setting closer to full action-model learning.

6.3.3 Search strategies and multi-clause learning

For domains with large example sets, more sophisticated search strategies or sampling schemes may be needed to keep Popper runs tractable. In addition, multi-clause learning regimes such as sequential covering could help address targets whose concepts are genuinely disjunctive and cannot be captured by a single clause under the current bias.

6.3.4 Integration with domain and feature design.

Finally, the learned rules can feed back into the design of planning domains. Reconstructable predicates can be replaced by their definitions, reducing redundancy, while irreducible predicates mark where additional symbolic features or numerical comparisons are necessary. Integrating this feedback loop into automated feature engineering and domain design is a promising direction for future work.

In summary, this thesis shows that inductive logic programming, combined with the max–minus–min strategy, can be used not only to learn symbolic rules but also to analyse the expressive power of planning representations. The results demonstrate that many abstract predicates are reducible to simpler fluents, while others mark the boundary at which richer relational structure becomes necessary. Understanding and exploiting this boundary remains a central challenge for interpretable learning in planning domains.

A Appendix

This appendix presents a complete Popper task example for the `learned_pred_0` target predicate in the N-Puzzle domain. The purpose is to illustrate how examples, background knowledge, and bias constraints interact to define a single ILP learning task. Only shortened excerpts of `exs.pl` and `bk.pl` are shown for readability; the full files are available in the accompanying repository. An end-to-end worked example is provided in Section A.1.

A.1 Worked Example: From Minimal/Maximal JSON to a Popper Task

This section provides an end-to-end worked example that links the SIFT representations to the Popper learning task for `learned_pred_0`. Specifically, we show (i) a minimal-state JSON excerpt, (ii) the corresponding maximal-state JSON excerpt containing the target predicate, (iii) the induced difference Δ used to label examples, (iv) the resulting Popper `pos/neg` instances, and (v) the hypothesis induced by Popper.

A.1.1 Step 1: Minimal JSON excerpt (state `st0`)

Listing A.1 shows a shortened excerpt of the *minimal-domain* JSON for state `st0`. Only the Boolean-valued fluents required for this example are shown.

Listing A.1: Minimal JSON excerpt for state `st0` (shortened).

```
{
  "state": "st0",
  "fluents": [
    {"name": "blank_val", "args": ["p_x_0", 1]},
    {"name": "blank_val", "args": ["p_x_1", 0]},
    {"name": "at_val",    "args": ["tile_1", "p_x_0", 1]},
    {"name": "at_val",    "args": ["tile_1", "p_x_1", 0]}
  ]
}
```

A.1.2 Step 2: Maximal JSON excerpt (same state `st0`)

Listing A.2 shows the corresponding *maximal-domain* excerpt for the same state `st0`, where the target predicate `learned_pred_0` appears.

Listing A.2: Maximal JSON excerpt for state `st0` (shortened).

```
{
  "state": "st0",
  "predicates": [
    {"name": "learned_pred_0", "args": ["p_x_0"]}
  ]
}
```

A.1.3 Step 3: Difference Δ and labeling rule

For each state S, the difference set $\Delta(S)$ contains instances of predicates that occur in the maximal representation but not in the minimal one. In this worked example, the maximal-only fact is:

$$\Delta(\texttt{st0}) = \{\texttt{learned\_pred\_0(p\_x\_0)}\}.$$

Therefore, `p_x_0` is labeled as a positive instance and `p_x_1` as a negative instance for `st0`.

A.1.4 Step 4: Popper examples generated from Δ

This yields the following Popper instances (cf. Listing A.5):

Listing A.3: Examples for state `st0` generated from Δ(`st0`).

```
pos(learned_pred_0(st0,p_x_0)).
neg(learned_pred_0(st0,p_x_1)).
```

A.1.5 Step 5: Popper output hypothesis

Finally, Popper induces a hypothesis defining `learned_pred_0/2` from the minimal-domain fluents in `bk.pl`. One representative induced rule for this task is shown in Listing A.4.

Listing A.4: Representative Popper hypothesis for `learned_pred_0/2` (shortened).

```
learned_pred_0(S,C) :-
    blank_val(S,C,1).
```

This rule illustrates the intended bridge: although `learned_pred_0/1` is maximal-only in the SIFT output, Popper can define it using minimal-domain Boolean-valued fluents.

A.2 Example File: exs.pl

Listing A.5 shows a shortened excerpt of the examples file. Positive and negative examples are given in the standard Popper format `pos/1` and `neg/1`. The `discontiguous` directives ensure that the clauses are accepted by the Prolog parser.

Listing A.5: Excerpt of `exs.pl` for `learned_pred_0` in the N-Puzzle domain.

```
:- discontiguous pos/1.
:- discontiguous neg/1.

pos(learned_pred_0(st0,p_x_0)).
neg(learned_pred_0(st0,p_x_1)).

pos(learned_pred_0(st1,p_x_0)).
neg(learned_pred_0(st1,p_x_1)).

pos(learned_pred_0(st2,p_x_1)).
neg(learned_pred_0(st2,p_x_0)).

pos(learned_pred_0(st3,p_x_1)).
neg(learned_pred_0(st3,p_x_0)).

pos(learned_pred_0(st4,p_x_1)).
neg(learned_pred_0(st4,p_x_0)).

pos(learned_pred_0(st5,p_x_0)).
```

These examples encode, for multiple states `st0–st5`, which positions satisfy the maximal-only predicate `learned_pred_0`. The target is unary in the maximal domain but becomes binary `learned_pred_0(S,C)` in the Popper version due to explicit state arguments.

A.3 Example File: bk.pl

Listing A.6 shows selected background knowledge. Boolean-valued fluents are encoded using explicit truth values, together with a `neg/2` predicate that links `true` and `false`. Only a subset of states is shown; the real dataset contains all 20 states.

Listing A.6: Excerpt of `bk.pl` for the N-Puzzle domain.

```
% Boolean constants and explicit negation
true(1).
false(0).
neg(0,1).
neg(1,0).

```

```
% States
state(st0).
state(st1).
state(st2).
state(st3).
state(st4).
state(st5).
state(st6).
state(st7).
state(st8).
state(st9).
state(st10).
state(st11).
state(st12).
state(st13).
state(st14).
state(st15).
state(st16).
state(st17).
state(st18).
state(st19).
```

The full background file additionally defines type relations (`obj/1`, `obj_type/2`) and Boolean-valued fluents such as `at_val/5` and `blank_val/4`, which originate from the minimal representation extracted from SIFT traces.

A.4 Bias File: bias.pl

The bias determines the hypothesis space explored by Popper. It specifies the allowed head predicate, admissible body predicates, and upper bounds on clause size, number of variables, and number of body literals.

Listing A.7: Bias specification for `learned_pred_0` in the N-Puzzle domain.

```
head_pred(learned_pred_0,2).

body_pred(state,1).
body_pred(obj,1).
body_pred(obj_type,2).
body_pred(val,1).

body_pred(at_val,5).
body_pred(blank_val,4).

body_pred(true,1).
body_pred(false,1).
```

```
body_pred(neg,2).

max_clauses(2).
max_body(4).
max_vars(6).
```

This bias limits hypotheses to at most two clauses with up to four body literals each, using no more than six variables. This encourages compact, projection-like rules and prevents excessively large or relationally complex hypotheses that would increase search cost.

A.5 Origin of the Target Predicate in the Maximal JSON

For the state-level maximal occurrence used for labeling examples, see Section A.1. Listing A.8 presents a shortened view of the maximal-domain JSON entry corresponding to `learned_pred_0`. The predicate appears only in the maximal representation and is annotated with SIFT pattern information. Popper receives only the labeled instances of this predicate, not the feature patterns themselves.

Listing A.8: Excerpt of the maximal SIFT JSON for `learned_pred_0`.

```
{
  "name": "learned_pred_0",
  "arity": 1,
  "knowledge": "concluded",
  "behavior": "fluent",
  "arg_types": [[1]],
  "feature": [
    {
      "selected_patterns": [
        {"name": "move-left",  "args": [1]},
        {"name": "move-left",  "args": [3]},
        {"name": "move-right", "args": [1]},
        {"name": "move-right", "args": [3]}
      ],
      "all_patterns": [
        {"name": "move-down",  "args": [1]},
        {"name": "move-left",  "args": [1]},
        {"name": "move-left",  "args": [3]},
        {"name": "move-right", "args": [1]},
        {"name": "move-right", "args": [3]},
        {"name": "move-up",    "args": [1]}
      ]
    }
  ]
}
```

```
26 ]
```

This metadata shows that the predicate is related to spatial patterns in tile movement, although such patterns do not appear in the minimal-domain background knowledge. Popper therefore learns a rule exclusively from Boolean-valued occupancy fluents.

A.6 Summary

This example demonstrates the structure of a full Popper learning task: `exs.pl` defines the labeled instances of the target predicate, `bk.pl` provides minimal-domain fluents and type information, and `bias.pl` restricts the hypothesis space. Section A.1 closes the loop by showing how a single maximal-only predicate instance in $\Delta(S)$ is converted into `pos/neg` examples and the resulting Popper hypothesis. The maximal-domain JSON defines where the target predicate originates in the SIFT representation. Together, these components form the input to Popper for inducing symbolic rules from N-Puzzle traces.

List of Symbols

B	Background knowledge used by Popper.
E	Set of labelled examples (positive and negative).
H	Learned hypothesis / induced logic program.
$p(t_1, \dots, t_n)$	Predicate with arguments $t_1, \dots, t_n$.
$p(\bar{o})$	Ground atom instantiated with an object tuple $\bar{o} = (o_1, \dots, o_k)$.
$\mathcal{P}_{\text{min}}$	Predicate set of the minimal domain representation.
$\mathcal{P}_{\text{max}}$	Predicate set of the maximal domain representation.
$\Delta = \mathcal{P}_{\text{max}} \setminus \mathcal{P}_{\text{min}}$	Set of max–minus–min learning targets (maximal-only predicates).
Ground($\mathcal{P}$)	Set of all ground atoms formable from predicate symbols in $\mathcal{P}$ and domain constants.
τ	A state–action trace.
s_t	State at time step t.
a_t	Action executed at time step t.
T	Trace horizon (number of transitions).
$\mathcal{A}_{\text{max}}(s_t)$	Set of ground atoms true in state s_t under the maximal representation.
$v \in \{0, 1\}$	Boolean truth value (false/true).
$p_{\text{val}}(\cdot, v)$	Boolean-valued fluent encoding, storing truth value v explicitly.
`true(1)`	Boolean constant representing true.
`false(0)`	Boolean constant representing false.
`pos(p(…))`	Positive example of target predicate p.
`neg(p(…))`	Negative example of target predicate p.
CWA	Closed-world assumption: atoms not listed as true are treated as false.
TP	True positives.
FP	False positives.
FN	False negatives.
TN	True negatives.
Precision	Precision score, defined as $TP/(TP + FP)$.
Recall	Recall score, defined as $TP/(TP + FN)$.
Size	Hypothesis size (number of literals/clauses, as reported by Popper).
`max_clauses`	Maximum number of clauses allowed in a hypothesis.
`max_body`	Maximum number of body literals per clause.
`max_vars`	Maximum number of variables allowed in a clause.

List of Figures

List of Tables

List of Listings

List of References

[1] V. Lifschitz, "Answer set programming," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, AAAI Press, 2008, pp. 1594–1597.

[2] A. Cropper and R. Morel, "Learning programs by learning from failures," *Machine Learning*, vol. 110, no. 4, pp. 801–856, 2021. DOI: 10.1007/s10994-020-05934-z [Online]. Available: https://doi.org/10.1007/s10994-020-05934-z

[3] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo: Asp with control," in *Technical Communications of the 35th International Conference on Logic Programming (ICLP 2019)*, 2019. [Online]. Available: https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf_iclp_GebserKKS19.pdf

[4] J. Gösens, N. Jansen, and H. Geffner, "Learning lifted strips models from action traces alone: A simple, general, and scalable solution," *arXiv preprint arXiv:2411.14995*, 2025. [Online]. Available: https://arxiv.org/abs/2411.14995

[5] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[6] R. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3–4, pp. 189–208, 1971.

[7] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—the planning domain definition language," AIPS-98 Planning Competition Committee, Tech. Rep., 1998.

[8] N. Jansen, J. Gösgens, and H. Geffner, *Learning lifted action models from traces of incomplete actions and states*, 2025. arXiv: 2508.21449 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2508.21449

[9] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Springer, 1987.

[10] D. W. Zimmerman, Ed., *Oxford Studies in Metaphysics, Volume 1*. Oxford: Oxford University Press, 2004.

[11] J. Hoffmann and S. Edelkamp, "The deterministic part of ipc-4: An overview," *Journal of Artificial Intelligence Research*, vol. 24, pp. 519–579, 2005.

[12] M. Law, A. Russo, and K. Broda, "Inductive learning of answer set programs," in *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA)*, Springer, 2014, pp. 311–325.

[13] G. Naidu, T. Zuva, and E. M. Sibanda, “A review of evaluation metrics in machine learning algorithms,” in *Artificial Intelligence Application in Networks and Systems*, R. Silhavy and P. Silhavy, Eds., Cham: Springer International Publishing, 2023, pp. 15–25, ISBN: 978-3-031-35314-7.