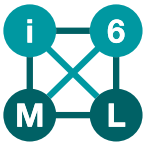

Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.

Learning General Policies for Partially Observable Deterministic (POD) Planning

Master Thesis

Presented by

Samridhi Kalra
441937

Supervised by Dr. Till Hofmann

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

2nd Examiner Prof. Gerhard Lakemeyer, Ph.D.

Aachen, 27th February, 2026

Abstract

In AI planning, the aim is to compute a plan or a policy that leads from the initial state to a goal state of a problem, given a model of the available actions. Generalized planning takes it one step further by attempting to compute a policy that can solve a whole family of problems, rather than addressing each problem instance separately. While classical planning assumes deterministic actions and complete information about the initial state, in Partially Observable Deterministic (POD) planning, the agent does not have full information about the state. Instead, it can perform sensing actions to obtain (partial) information about the current state. One approach to POD planning is to translate the problem into a Fully Observable Non-Deterministic (FOND) planning problem by introducing the literals KL and $\neg KL$, which express that L is known or not known, respectively, making the uncertainty about knowledge explicit in the problem representation. Sensing actions are compiled into non-deterministic actions whose outcomes correspond to the possible observations. The translated problem can then be solved using a classical online planner or a FOND planner that computes an offline contingent plan. Recently, generalized planning was extended to FOND based on a combinatorial approach, in which a general policy is described using rules derived from a collection of features. Such a policy can be learned without supervision by classifying transitions into 'good' and 'non-good' and identifying distinguishing features for these classes. The goal of this thesis is to learn general policies for POD problems by translating the POD problem into a FOND problem and then adapting the FOND learning approach to generate strong policies.

Contents

1 Introduction

In Artificial Intelligence (AI), an indicator of intelligence is 'acting rationally' [35], and AI planning attempts to solve this by deriving a plan or a strategy to achieve a defined goal from an initial state, given a model of the world. In the simplest setting, a classical planning instance can be thought of as a state transition system having a finite set of (fully observable) states, each representing a possible configuration of the world, a finite set of actions, and a deterministic transition function [19]. Thus, planning is essentially a search problem for a goal state in this state space. A toy version of a planning problem involving a robot vacuum, first introduced in [32] as Vacuum World, can be seen in Figure 1.1, represented as a state transition system. It is a world with two rooms that are initially both dirty, and a robot is assigned the task of cleaning them both using only three actions: 'Move Right', 'Move Left', and 'Clean'. In the figure, we can see eight different configurations (or possible states) of the objects in the two rooms, and the possible transitions between them. Today, state-of-the-art classical planners can derive plans or sequence of actions to reach the goal state, given a description of the planning problem as input.

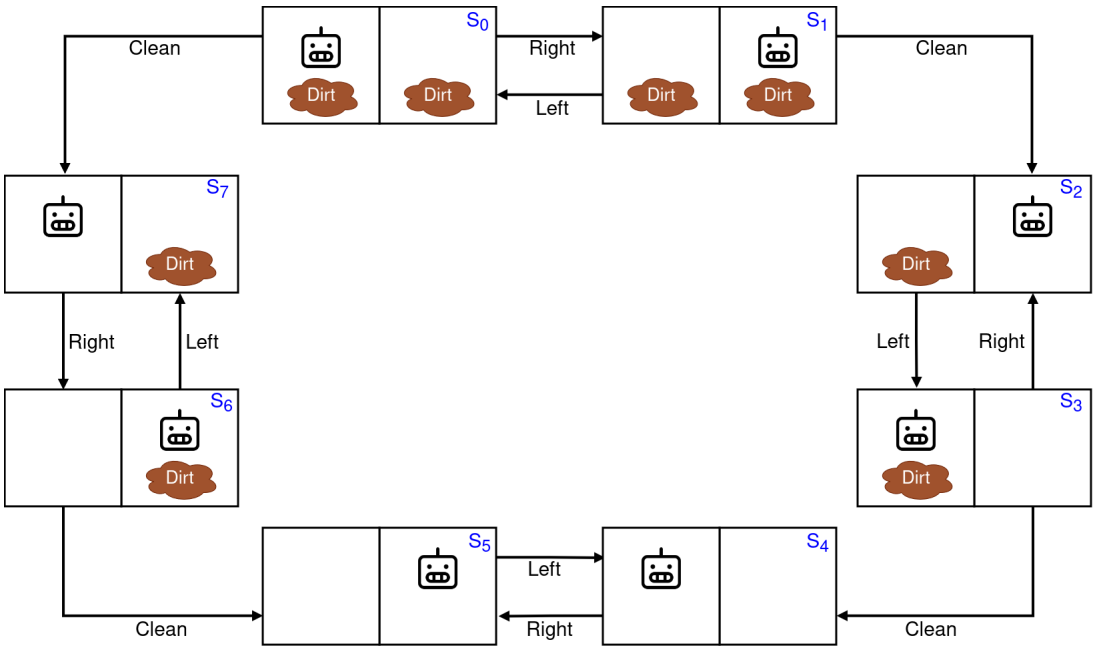

Figure 1.1: Vacuum World represented as a state transition system, with all possible states and valid actions.

When we relax or adjust certain assumptions about the world to represent it more realistically, the planning problem becomes more complex. For example, in the Vacuum World, the robot may not have prior knowledge of whether a room is dirty or clean; it can only gather this information through sensing. In scenarios where complete information about the states is unavailable, but partial information can be obtained through sensing, it is a *contingent planning* problem in a closed-loop setting. Sensing allows the agent to obtain (partial) information about the world and adjust its actions accordingly. This is also referred to as *partially observable* planning, as the agent does not have full visibility of the world states. A restricted subset of contingent planning with *deterministic* actions, called Partially Observable Deterministic (POD) planning, will be the focus of this research.

In domains with partial observability, uncertainty about the current state is tracked using sets of concrete world states believed to be true based on the current information from the environment. These sets of states are called beliefs, and planning with incomplete information can be formulated as a search problem in *belief space* [5]. The objective is to find a plan (often a tree-shaped policy branching on observations) that guarantees reaching a goal state from every possible state within the initial belief, typically by using sensing actions to reduce uncertainty along the way by eliminating states from the belief that are inconsistent with the observations made by the agent while acting in the world [2]. For example, in the Vacuum World with partial observability, the robot may not know whether a room is dirty or clean at the start, and hence the initial belief would consist of four states, one for each possibility of dirt in the two rooms, given that the robot starts in one of them.

Another key area of focus in AI planning is *generalized planning*, which leverages the structure of the world model to find solutions for a family of planning instances, rather than solving each one individually. For example, each instance in the Vacuum World could have a different number of rooms or initial positions of the robot, while adhering to common underlying 'rules'. An implication of generalized planning is that the solution form is no longer a distinct sequence of actions; rather, it is a strategy that can be followed to reach the goal without the need to plan for each problem instance separately. Most recently [9], a *general policy* that solves a generalized planning problem is represented using rules that resemble conditions and effects, defined on a pool of features representing world states. These rules describe the transitions that lead to the goal states in this feature space. The reason for using features is that they provide an abstraction over the state space of a problem, which can be too specific to describe a general policy expected to be reusable across different problem instances.

For classical domains, there now exist reliable methods to learn general policies, both by using combinatorial and deep learning approaches [3, 10, 25, 26, 36]. Here, we are interested in the former approach as it offers a couple of advantages. Firstly, the solutions can be theoretically validated for correctness if the problem was modeled correctly, and secondly, the rationale behind the generated solution is clearly visible in its representation. A combinatorial approach

that learns rule-based general policies in a self-supervised manner using a small set of training examples was proposed by Frances *et al.* [14]. Here, the problem is treated as a Weighted Max-SAT optimization that, given an input of the state space of training instances and a feature pool, learns a logical distinction between 'good' and 'bad' transitions along with a set of distinguishing features, which is then used to write the rules for the general policy. The features are selected from a large feature pool, generated from domain predicates via a context-free description logic grammar. The resulting policy rules map feature conditions to effect sets, yielding compact, expressive policies.

Recently, Hofmann and Geffner [24] extended generalized planning to Fully Observable Non-Deterministic (FOND) domains, expanding on the idea of using combinatorial optimization to learn general policies with self-supervision. By nature, FOND problems are more complex than classical ones, as they involve non-deterministic actions and require policies that can handle multiple possible outcomes. The proposed approach is to learn general policies for FOND problems by treating the problem as a Weighted Max-SAT optimization, similar to the method used for classical domains, but with adjustments to account for the non-deterministic nature that could lead us to dead-ends. This is done by *determinizing* each FOND problem by replacing every non-deterministic action by the set of its outcomes as deterministic actions, and learning a safe policy that avoids dead-ends.

Although uncertainty in a partially observable setting arises from uncertainty in the initial state, it can be compiled into a fully known world by adding a dummy non-deterministic action that randomly picks the true initial state [5, 7]. This inspires a translation-based approach that POD planners take to solve the problem by searching in a non-deterministic search space, as opposed to a belief space. The resulting FOND problem can then be solved with FOND planners while still correctly handling all the uncertainty that originally came from the unknown initial situation [30]. A recent idea is to translate the problem into a FOND planning problem by introducing the KL and $\neg KL$ literals, which express what is *known* or *not known*, respectively [2, 6, 8]. The K-literals give the planner a *knowledge-level* language that turns an uncertain belief state into a regular propositional state using $\neg KL$. This allows the partially observable deterministic problem to be compiled into a fully observable non-deterministic one, where the only non-determinism is the outcome of sensing actions as their outcomes correspond to the possible observations.

In this work, we turn to look at generalized planning for POD models. As there exist sound translations of POD problems to FOND problems [2, 8, 30], the proposed approach is to learn general policies by translating the POD problem into a FOND one and then leveraging the self-supervised combinatorial method to learn general policies for the FOND model [24]. Specifically, we will use a linear translation proposed by Bonet and Geffner [8] to translate the POD problem.

As in [24], the learning process will require us to generate a state space for a set of training instances, that will be used by a SAT solver to find satisfying assignments for good transitions in the state space and the features that specifically capture them, which will then be used to write the rules for the general policy. For our approach, we will generate the state space on the translated POD problem, by applying domain actions on states modeled as non-deterministic ones that can lead to multiple outcomes based on the possible observations. We will show through the proposed approach that it is possible to learn general policies for POD problems by leveraging the translation to FOND problems and using a combinatorial optimization method, and that the learned policies are compact, interpretable, and generalize up to a certain extent.

Overall, the following research questions are addressed in this work:

1. How can we represent the POD domain such that it can be used to learn generalized policies in a self-supervised manner?
2. What is the nature of the solution and what assumptions are necessary for its validity?
3. How well does the proposed approach perform on benchmark POD domains?

First we will review the necessary background and related work in Chapter 2 and Chapter 3, respectively, before describing our approach in Chapter 4. We will then present the results of our experiments in Chapter 5 and conclude with a discussion of the implications of our findings in Chapter 6.

2 Background

In this section, we review classical planning, generalized classical planning, generalized FOND planning, POD planning, and a linear translation of POD domains relevant for the approach.

2.1 Classical Planning

Classical planning involves planning for a world with a finite and fully observable state space, and a finite set of possible actions that are deterministic actions. Hence, such a problem can be easily described using a state transition system. A compact representation of this state space can be in the form of a collection of (boolean or multivalued) variables that correspond to various properties of the world, and possible states are derived implicitly by assigning different values to each variable in the domain. Actions modify these assignments in pre-defined ways to move from one state to another. Specialized languages exist for describing planning domains and instances using the compact representation, e.g., the Planning Domain Definition Language (PDDL) [18], which extends STRIPS [13] operators. In this form of representation, a planning *domain* encodes a set of instances by defining the predicates, object types and action schemas that describe the world; whereas a planning *instance* is a specific configuration of the domain in terms of the initial state, goal state and a defined set of objects. PDDL has virtually become the standard and is even officially used for representing benchmark planning domains for International Planning Competitions [29] organized by International Conference on Planning and Scheduling (ICAPS). State-of-art domain-independent AI planners read problems encoded in PDDL and solve them to output valid plans.

More formally, a planning problem in the classical setting is defined as a pair $P = \langle D, I \rangle$, where D is the domain and I is the planning instance [17, 23, 32]. The domain D consists of first-order predicate symbols and a set of action schemas with preconditions and effects. Preconditions are sets of atoms $p(x_1, x_2, \dots, x_k)$ in conjunction, and the effects are a set of positive $p(x_1, x_2, \dots, x_k)$ or negative $\neg p(x_1, x_2, \dots, x_k)$ atoms for add or delete effects, respectively, where p is a predicate symbol of arity k and x_n is a schema argument. The second part of the problem is the instance I, which is a tuple $I = \langle O, Init, Goal \rangle$, where O is a finite set of objects c_i in the world; $Init$ is the set of ground predicates $p(c_1, c_2, \dots, c_k)$ that hold true in the initial state; and $Goal$ represents the ground predicates that must hold true to solve the problem.

As hinted before, this compact representation of planning instance $P = \langle D, I \rangle$ implicitly defines a state model $M = \langle S, s_0, S_G, Act, A, f \rangle$ where the states are possible permutations of grounded atoms; s_0 is the initial configuration defined by $Init$, and S_G contains the states $s \in S$ that satisfy

the goal, i.e. $Goal \subseteq s$. Act is the set of ground actions based on the actions-schemas in domain D. $A(s)$ is the set of actions applicable in a state s, which is exactly the actions that s satisfies the preconditions for. Lastly, f is a deterministic state transition function $s' = f(a, s)$ where $f(a, s) = (s \setminus del(a)) \cup add(a)$. The solution for a classical domain is a sequence of actions (i.e., a $plan$) $a_0, a_1, ..., a_n$ from the initial state such that $pre(a_i) \subseteq s_i$, where $s_i = f(a_{i-1}, s_{i-1})$, and $s_n \in S_G$.

Sometimes world states are represented using multivalued variables [17] instead of the aforementioned boolean predicates for compact representations that are more natural to the problem at hand; for example, $at(robot) = 1$ instead of $at(robot, 1) = true$. Even though this way of state representation seems more expressive, it does not mean that it is more complex for planning, as this representation can be easily compiled into the previous form by introducing negative effects for other unused values in the domain of the variable that is affected by the action. This representation will be made more explicit later when we introduce a formalism for POD planning.

2.2 Generalized Classical Planning

In generalized planning, the goal is to solve a class Q of planning instances together. Since each instance could have different groundings or initial and goal states, the solution form is not a definite sequence of actions anymore. This representation is more nuanced than simply mapping states to actions as they can have different groundings over different problem instances. Here, we will represent a general policy π as a set of feature-based conditional rules $C \mapsto E$ as proposed in [9]. The features describe the world states more quantitatively, and can be boolean p or numerical (non-negative) n. In one of the approaches where classical general policies are learned through self-supervision, these features are generated from problem predicates and objects using Description Logic in a context-free manner [14].

A rule condition C is a set of feature literals of the form $p, \neg p, n = 0, n > 0$ that must hold true in a state for the rule to be applicable. The effects E are the changes a feature should see when the domain action that is enabled by the rule is applied. They are sets of the form $p, \neg p, p?, n \uparrow, n \downarrow, n?$. A transition (s, s') *satisfies* a rule $C \mapsto E$ when C holds true and the features mentioned in E change accordingly. If a boolean feature p is in E then its value in s', $p(s') = true$; if $\neg p$ appears in E, then $p(s') = false$; if a numerical feature effect $n \uparrow$ is in E, then $n(s') > n(s)$; if $n \downarrow$ is in E, then $n(s') < n(s)$; If $n?$ in E, then its value can change in any manner. If a feature is not mentioned in E, then its value should remain the same, i.e., $n(s') = n(s)$ or $p(s') = p(s)$. As the features on which the general policy π is based are generated from the underlying domain of the family of instances Q, they are shared by all instances of the class. Hence, π determines the concrete policy π_P for each instance P. Transitions satisfying some rule in a policy are said to satisfy the policy π_P, using which a π-trajectory can be generated.

If all maximal π-trajectories reach a goal state, then the policy solves the given problem. The general policy π is said to solve a class of problems Q if each π_P solves each instance P in Q.

The self-supervised approach to learning the rules for a general policy employs combinatorial optimization, where a propositional theory $T(\mathcal{S}, \mathcal{F})$ is formulated to be solved as a min-cost SAT problem [14]. One input to the theory is the state space $\mathcal{S}$ which is expanded for a small set of training instances $P_i \in Q$. The other input is the feature pool $\mathcal{F}$ generated using Description Logic. The aim is to find satisfying assignments of particular transitions (marked as 'good') in the expanded state space that would contribute to a solution policy, while also selecting most relevant and simple features that help capture how the states evolve in those transitions and distinguish them from the 'bad' transitions that do not contribute to a solution. Consequently, policy rules are determined on the basis of how the selected features change in the 'good' transitions. The constraint of finding the simplest features is defined as the optimization objective of the SAT problem; specifically, it is the cumulative syntactic complexity of the selected features in terms of their syntax trees. This constraint results in the selection of the simplest features which is effectively good for generalization.

2.3 Generalized FOND Planning

Extending classical domains with non-deterministic actions gives us FOND domains. Like a classical planning model, a FOND model P can also be defined by a state model $M = \langle S, s_0, S_G, Act, A, f \rangle$, with the only difference that the state transition function F is now a non-deterministic function that maps a state-action pair to a *set* of successors $s' \in F(a, s)$ where $s \in S$ and $a \in A(s)$. For simplicity, a non-deterministic action a can be considered as a set of deterministic actions $a = \{b_1, b_2..., b_n\}$, each one associated with a deterministic transition function $F(a, s) = \{f(b_1, s), f(b_2, s)..., f(b_n, s)\}$ [17]. While planning on a FOND problem instance, the solution cannot take the form of a fixed sequence of actions, as each action sequence could lead to different states each time. Hence, solving a FOND problem P would require a partial mapping from states to actions in the form of a policy π. In FOND domains, the model is realistic, or *fair*, only if the state trajectories that are infinite visit all possible $s' \in F(a, s)$ an infinite number of times, if a occurs infinitely in the trajectory. A policy π *solves* P if all fair, maximal $\pi-$trajectories reach a goal state.

For generalized planning in FOND model, Hofmann and Geffner [24] propose to use the same combinatorial optimization approach as mentioned above for learning general policies over a determinized relaxation of the FOND model obtained by considering all underlying deterministic actions $b_i \in a$ separately. This is also known as the all-outcome relaxation. More concretely, the idea is to determinize the collection Q of solvable FOND problems to class Q_D using the all-outcome relaxation, and learn the general policy π_D for Q_D using a small set of training examples. For policy π_D, all the maximal π-trajectories would reach the goal states

of relaxed problems P_D in Q_D, however, if the learned policy is directly interpreted to use the original non-deterministic actions a instead of deterministic $b_i \in a$, then this policy might lead to dead-end states during inference due to non-determinism. Hence, the notion of a *safe* policy is established to constrain the learning process. A general policy π_D for a determinized Q_D of Q is regarded as *safe* in P_D if, for every reachable state s in P_D and for each deterministic action b_i that belongs to $\pi_D(s)$, there exists a non-deterministic action a such that b_i is included in a, and no state s' produced by $F(a, s)$ is a dead-end. Furthermore, π_D is considered safe in Q_D if it is safe across all the instances within the class.

To obtain general policies that are safe in the original FOND class of problems Q, two steps are taken. First, the propositional theory $T(\mathcal{S}, \mathcal{F})$ is adapted to classify only safe transitions as 'good'. This is achieved by identifying dead-end states in the sample problems beforehand, so that only *alive* states are considered, that are reachable and have a path leading to a goal state. Secondly, the solution incorporates additional descriptions of the dead-end states as constraints to prevent entering these states. This description, represented by B, consists of a conjunction of boolean conditions based on features that differentiate dead-end states from alive states. B is also learned in combinatorial optimization process and is included in the policy as a constraint. Specifically, it must not hold true for any of the next states $s' \in F(a, s)$ for an applicable policy rule to be applied in s. Hence, the general policy for FOND is a set of feature-based rules R of the form $C \mapsto E$, and a set of constraints B.

Specifically, the SAT theory $T(\mathcal{S}, \mathcal{F})$ for learning the general policy for FOND in [24] consists of the following sentences:

1. For every alive state s where $Safe(s) = \{a \in A(s) : \forall s' \in F(a, s), \neg DeadEnd(s')\}$,

$$\bigvee_{s' \in f(a,s), a \in Safe(s)} Good(s, s')$$

2. For all s that are alive states,

$$Exactly\text{-}1_{d \in \mathbb{N}} : V(s, d)$$

   where $V(s, d)$ is true if length of the shortest path from s to a goal state is d, and $V(s, 0)$ is true if s is a goal state.
3. For all transitions (s, s'),

$$Good(s, s') \wedge V(s, d) \rightarrow \bigwedge_{a \in A(s), s' \in F(a,s)} \bigvee_{s'' \in F(a,s)} V(s'', d'') \rightarrow d'' < d$$

4. For all alive states s leading to a dead state s',

$$\neg Good(s, s')$$

5. For all transitions (s_1, s_1') and (s_2, s_2'),

$$Good(s_1, s_1') \wedge \neg Good(s_2, s_2') \rightarrow D(s_1, s_2) \vee D_2(s_1, s_1', s_2, s_2')$$

where,

$$D(s_1, s_2) = \bigvee_{f : [[f(s_1)]] \neq [[f(s_2)]]} Select(f)$$

and

$$D_2(s_1, s_1', s_2, s_2') = \bigvee_{f : \Delta f(s_1, s_1') \neq \Delta f(s_2, s_2')} Select(f)$$

where $Select(f)$ is true if the feature f is selected, and $[[f(s)]]$ is the value of feature f in state s.

These sentences ensure that:

- A good transition never directly leads to a dead-end state in the non-deterministic problem.
- An alive state has a path to a goal state.
- The non-deterministic action associated with a good transition may lead us closer to the goal.
- A transition that leads to a dead-end state cannot be good.
- At least one selected feature can distinguish between alive states and dead-ends.
- At least one selected feature can distinguish between goal and non-goal states.
- At least one selected feature can distinguish between good transitions and bad transitions.

The solver finds satisfying assignments for the above sentences while also optimizing for the simplicity of the selected features, which results in a general policy that is safe in the original FOND class of problems Q.

2.4 POD Planning

The model for a POD domain $M = \langle S, s_0, S_G, Act, A, f, O \rangle$ is like the classical planning model extended with a sensing model O [17]. Additionally, because even the initial state is not fully known, s_0 is a set of possible initial states. Under partial observability, there is uncertainty in the current world state due to lack of information. This means that actions are not taken on single states but sets of states, one of which could be true. Here on, we will denote sets of states as beliefs b. The sensing model is a function $O(s, a, b)$ that maps a state, action and belief to a set of observation tokens, i.e., if an action a is taken in the belief b and the resulting (true) state is s, then a token $o \in O(s, a, b)$ is observed. Thus, each observation gives us a bit more

information about the true state of the world by eliminating beliefs where those observations are not possible. Here, the solution form is a policy π which is a function that assigns actions to belief states. The sequences of actions and observations $a_0, o_0, a_1, o_1, ...$, generated by the policy π represent the executions where $a_i = \pi(b_i)$. The policy solves the model if all these π-trajectories lead to a goal state with certainty, meaning a belief state $b \subseteq S_G$.

A more compact representation of the POD domain can be made using multivalued state variables [8], where it is represented as a tuple $P = \langle V, I, A, G, W \rangle$, where:

- V denotes a set of state variables X, each having a finite domain D_X.
- I consists of X-literals that describe the initial situation.
- G is a conjunction of X-literals that defines the goal.
- A includes actions characterized by preconditions $\text{Pre}(a)$ and conditional effects (rules) $C \Rightarrow E$, where $\text{Pre}(a)$ and C are sets of X-literals, and E is a set of positive X-literals.
- Positive literals take the form $X = x$ for $X \in V$ and $x \in D_X$.
- Negative literals $\neg(X = x)$ are expressed as $X \neq x$.
- The states associated with a problem P are the different valuations of the state variables X.

The sensing component W in P consists of a set of observable multivalued variables Y, each with a domain D_Y:

- Each variable Y has a state formula $\text{Pre}(Y)$, known as the Y-sensor precondition, which specifies the conditions under which the variable Y can be observed.
- There are state formulas $W(Y = y)$ for each value y in D_Y, which describe the conditions necessary for Y to take the value y.
- Assuming that sensing is deterministic, the formulas $W(Y = y)$ must be mutually exclusive and collectively exhaustive in the states where $\text{Pre}(Y)$ holds true, meaning that in such states, only one value of Y can be observed.
- An observable variable Y can also serve as a state variable, in which case $W(Y = y)$ simplifies to $Y = y$.
- Both $\text{Pre}(Y)$ and $W(Y = y)$ are in Disjunctive Normal Form (DNF), and the terms C in $W(Y = y)$ contain only positive X-literals.
- If negative literals such as $X \neq x$ are necessary, they must be substituted with the disjunction $\bigvee_{x'}(X = x')$, where x' varies over the possible values of X in D_X that are different from x, and the resulting formula must also be converted to DNF.

The planning problem $P = \langle V, I, G, A, W \rangle$ defines the model $S(P) = \langle S, S_0, S_G, A, f, O \rangle$ introduced earlier, where S represents the possible set of valuations for the variables in V, and S_0 and S_G are the sets of states (or beliefs) that satisfy the initial conditions I and the goal conditions G, respectively. The function $A(s)$ denotes the set of actions in P for which the preconditions are satisfied in state s, and $f(a, s)$ is the state transition function defined by the conditional

effects associated with action a. Similarly, the sensor model O is such that $o(s, a, b)$ represents a valuation over the variables Y in W for which $\text{Pre}(Y)$ holds true in state b.

Example: Consider a simple vacuum cleaner robot operating in a two-room environment. The robot's task is to clean both rooms, but it does not know which rooms are dirty. In the compact multivalued representation, this problem can be encoded as $P = \langle V, I, A, G, W \rangle$ where:

- $V = \{\text{loc}, \text{dirt}_L, \text{dirt}_R\}$ with domains $D_{\text{loc}} = \{L, R\}$, where, $D_{\text{dirt}_L} = \{clean, dirty\}$, and $D_{\text{dirt}_R} = \{clean, dirty\}$.
- $I = \{\text{loc} = L\}$ (initial location is known to be Left, dirt status is unknown).
- $G = \{\text{dirt}_L = clean, \text{dirt}_R = clean\}$ (both rooms must be clean).
- $A = \{\text{move}_L, \text{move}_R, \text{suck}\}$ with:
  - move_L: $\text{Pre}(\text{move}_L) = \{\text{loc} = R\}$, effects: $\text{loc} = L$.
  - move_R: $\text{Pre}(\text{move}_R) = \{\text{loc} = L\}$, effects: $\text{loc} = R$.
  - suck: $\text{Pre}(\text{suck}) = \emptyset$, with conditional effects: $\text{loc} = L \Rightarrow \text{dirt}_L = clean$ and $\text{loc} = R \Rightarrow \text{dirt}_R = clean$.
- W includes an observable variable $\text{dirt}_{\text{here}}$ with $D_{\text{dirt}_{\text{here}}} = \{clean, dirty\}$ and:
  - $\text{Pre}(\text{dirt}_{\text{here}}) = \top$ (dirt can always be sensed at the current location).
  - $W(\text{dirt}_{\text{here}} = clean) = (\text{loc} = L \wedge \text{dirt}_L = clean) \vee (\text{loc} = R \wedge \text{dirt}_R = clean)$.
  - $W(\text{dirt}_{\text{here}} = dirty) = (\text{loc} = L \wedge \text{dirt}_L = dirty) \vee (\text{loc} = R \wedge \text{dirt}_R = dirty)$.

2.5 Linear Translation of the POD model

The above formulation of the POD model defines a non-deterministic search problem for the goal in *belief space*. A few translations of contingent and conformant models have been introduced [2, 6, 8], that project them onto an epistemic space using KL and $\neg KL$ literals. These literals explicitly express what is *known* or *unknown*, thereby compiling away uncertainty. Specifically, there are four K-literals for each $X = x$: $K(X = x)$, $K(X \neq x)$, $\neg K(X = x)$, and $\neg K(X \neq x)$. Consequently, the planning problem transforms into a search problem in *state space*, which relieves us of the non-trivial task of belief tracking. For contingent planning (planning with sensing), the only actions that remain non-deterministic are *sensing actions*, as their effects depend on a hidden true world state that is not initially known. Hence, such a translation can be interpreted as a FOND model.

We will mainly refer to the translation $X(P)$ proposed by Bonet and Geffner [8] for belief tracking in the LW1 Planner. This translation is linear in the number of problem fluents in P, and is width-1 complete. Roughly, the width of a problem is the maximum number of undetermined state variables at a time, that appear in the goal, observations, or the conditional effects of actions. This property of the translation is useful as the width of most POD benchmarks is 1.

For a problem $P = \langle V, I, G, A, W \rangle$, $X_0(P)$ produces a classical problem $P' = \langle F', I', G', A' \rangle$ along with a set of deductive axioms D', where:

- $F' = \{KL : L \in \{X = x, X \neq x\}, X \in V, x \in D_X\}$,
- $I' = \{KL : L \in I\}$,
- $G' = \{KL : L \in G\}$,
- $A' = A$ but with each action precondition L replaced by KL, and each conditional effect $C \Rightarrow X = x$ replaced by effects $KC \Rightarrow Kx$ and $\neg K\neg C \Rightarrow \neg K\bar{x}$,
- $D' = \{Kx \Rightarrow \bigwedge_{x' : x' \neq x} K\bar{x}', \bigwedge_{x' : x' \neq x} K\bar{x}' \Rightarrow Kx\}$, for all $x \in D_X$ and $X \in V$.

This translation is based on one used in the K-replanner [6] which is linear but incomplete. To address this, Bonet and Geffner [8] introduce an additional step of *action compilation* to make it width-1 complete. Action compilation introduces some explicit action effects that use the knowledge of a finite domain and help eliminate some states from the belief. Formally, additional condition effects are added to each action a in A' as follows: for each effect of the form $C, X = x \Rightarrow X = x'$ such that $x' \in D_X$ and $x' \neq x$, the effect $KC, K\neg L1, \ldots, K\neg L_m$ is added to a, where $L_1, \ldots, L_m$ come from any action effect of the form $(L_1, \ldots, L_m) \Rightarrow X = x$, that results in $X = x$. $X_0(P)$ along with compiled actions gives us $X(P)$.

The last missing piece is the translation of the sensing model. Updating states upon sensing is done through unit resolution. More precisely, if s_a is the state following the execution of an action a in state s in $X(P)$, then the state s_a^o that results from obtaining the observation o is: $s_a^o = \text{UNIT}(s_a \cup D' \cup K_o)$. Here, $\text{UNIT}(C')$ stands for the set of unit literals in the unit resolution closure of C', D' is the set of deductive rules, and K_o stands for the codification of the observation o given the sensing model W. For each term $C_i \cup \{L_i\}$ in $W(Y = y)$ such that o makes $Y = y$ false, K_o contains the formula $KC_i \Rightarrow K\neg L_i$. The intention is that, if the observation does *not* turn out to be y, we would know that the states that make $Y = y$ should not be a part of the belief anymore. So, we add appropriate $K\neg L$ literals to ensure that. A step-by-step example of the LW1 translation can be found in Section 4.1.

As this translation is sound and complete for width-1 problems, an execution of a trajectory in the translated problem reaching the goal state corresponds to the same trajectory in the original problem that also reaches the goal state. We will be using this translation to learn general policies for POD problems.

3 Related Work

The foundations of AI planning rest on computing a sequence of actions that reaches a goal from an initial state under a deterministic, fully observable model [20], i.e., classical planning. Classical planners exploit state-space search and heuristics such as the FF relaxation [17], but they assume complete knowledge of the world.

3.1 Generalized Planning

Generalized planning extends the classical setting by seeking a single policy that solves an entire family of instances. Early work formalized generalized plans as algorithms or finite-state controllers that operate over a shared set of fluents and actions [9]. Different representations such as macro-actions, procedural programs, and feature-based rules have been explored, with top-down search and bottom-up reuse offering complementary scalability [26].

More recent work shows that general policies can be obtained by projecting domains onto a common feature set and employing combinatorial optimization on the expanded state space of small training instances [14]. This is cast as a SAT problem that distinguishes 'good' from 'bad' transitions. Compared with deep-learning or deep-reinforcement-learning approaches [11, 21, 34, 37, 38], which scale well but yield opaque neural controllers that cannot be formally verified, this combinatorial method produces transparent, interpretable rule sets whose correctness can be proved mathematically and formally verified on the benchmark domains, offering provable guarantees of safety and completeness while remaining compact and expressive.

A similar approach has also been adopted for learning general policies for FOND domains, where the non-deterministic nature of the environment leads to a more complex state space and solution structure [24]. By treating the problem as a Weighted Max-SAT optimization, similar to the method used for classical domains, but with adjustments to account for the non-deterministic nature that could lead us to dead-ends, the approach learns a safe policy that avoids dead-ends.

3.2 Planning Under Partial Observability

Planning under partial observability (POD) replaces the deterministic assumption with sensing actions and incomplete initial information. The prevailing technique translates a POD problem into a FOND problem by introducing knowledge literals KL and $\neg KL$ that make epistemic

uncertainty explicit, and by compiling each sensing action into a non-deterministic action whose outcomes correspond to possible observations [16].

This translation underlies several contingent planners. The CLG planner uses a quadratic translation and the FF-additive heuristic, but it is complete only for width-1 problems [2]. CLG+ augments CLG with assumption actions that are penalized, enabling robust action selection even when the goal cannot be guaranteed because dead-end beliefs are present [1]. The K-replanner introduced linear-size translations for belief tracking and action selection, achieving scalability comparable to CLG, but the translation is not complete [6]. LW1 utilizes a linear translation while retaining completeness for width-1 domains [8].

Belief-tracking complexity has been analyzed via the width parameter, which bounds the exponential cost of exact tracking [7]. Factored belief tracking runs in time exponential in the problem width, while causal belief tracking exploits causal width to obtain practical approximations such as beam tracking. These results provide the theoretical foundation for tractable POD planning and motivate translation-based approaches that compile uncertainty away [31].

Online POD planners formalize soundness and completeness in terms of belief-tracking and action-selection components. Monotonic inference, combined with randomized action selection, guarantees completeness for solvable deterministic tasks [4]. The LW1 family of planners exemplifies this line of work, achieving high coverage on Minesweeper benchmarks while remaining simple to implement.

3.3 Positioning of This Work

This thesis builds on the aforementioned work by: (1) using the K-literal translation to convert POD problems into FOND instances, and (2) applying combinatorial learning to derive general feature-based policy rules. Together, these contributions integrate the scalability of linear translations, the theoretical guarantees of width-bounded belief tracking, and the explainability of symbolic policy learning to advance generalized planning for partially observable domains.

4 Approach

Finding a correct policy for partially observable deterministic (POD) tasks is computationally demanding: the need to reason over all possible belief states makes policy synthesis PSPACE-hard in the deterministic case and EXPSPACE-hard when dead-ends can appear, so that exhaustive offline partially-observable planning quickly becomes prohibitive for realistic instances [22, 33]. An approach to address this complexity is to introduce the epistemic literals KL, $\neg KL$, $K\neg L$, $\neg K\neg L$ which encode 'L is known to be true', 'L is not known to be true', 'L is known to be false', and 'L is not known to be false', respectively [2, 6, 8]. The large majority of POD benchmarks the resulting problem have a contingent width 1, and under this condition the K-literal translation is guaranteed to be complete while remaining polynomial in the number of fluents, and planners such as CLG (or its linear-translation variants such as LW1) can solve the translated instances efficiently [2, 8]. Thus, the K-literal encoding both removes uncertainty from the search space and isolates an interesting, tractable subset of POD problems that includes most existing benchmark domains. Moreover, by compiling each sensing action into a non-deterministic action whose outcomes correspond to the possible observations, the original POD problem is transformed into a FOND model that no longer requires explicit belief-tracking.

Recently, Hofmann and Geffner [24] broadened the scope of generalized planning to FOND domains, building on an earlier work that uses combinatorial optimization to learn general policies for classical problems [14] in an self-supervised manner using a small set of example problems P_i, belonging to a family Q. The learning problem is formulated as a weighted Max-SAT optimization, similar to the technique applied to classical domains but incorporating additional constraints that prevent the policy from entering states that become dead-ends in the non-deterministic setting. The key step is to *determinize* each FOND instance by replacing every non-deterministic action with a set of deterministic actions representing all its possible outcomes; a policy is then learned over this deterministic relaxation, with the safety requirement that it never selects an action whose deterministic successors correspond to dead-end states of the original FOND problem. The SAT theory $T(\mathcal{S}, \mathcal{F})$ expects as input a state space $\mathcal{S}$ generated from the training instances, along with a feature pool $\mathcal{F}$, and outputs a set of 'good' transitions that are used in the policy, and the most appropriate features capturing these transitions and help distinguish them from bad transitions. The policy rules of the form $C \rightarrow E$ are then derived from the set of these 'good' transitions by capturing how the values of the selected features change within them.

Utilizing this approach of learning general policies for FOND, our aim is to learn general policies for POD problems by translating them into FOND problems using a linear K-literal translation by Bonet and Geffner [8], expanding the state space of the translated problem to be used as input to the SAT theory $T(\mathcal{S}, \mathcal{F})$, and then applying the combinatorial optimization method to learn general policies for the resulting FOND model. The feature pool $\mathcal{F}$ will be generated from the domain predicates using a context-free description logic grammar, as in [14], and the state space $\mathcal{S}$ will be generated by applying the domain actions on states modeled as non-deterministic ones that can lead to multiple outcomes based on the possible observations.

Notably, while typical FOND problems admit strong cyclic solutions under fairness assumptions, the monotonic nature of sensing in POD problems (where uncertainty only decreases) enables us to learn strong policies. This is discussed in detail in Section 4.3.3.

In this chapter, we will first describe the translation of POD problems to FOND problems in more detail, and then discuss the state expansion of the translated problem, followed by the learning process for the general policy. POD problems can encode unsolvable instances, and the state space of the translated problem can grow exponentially with the number of fluents, so we will also discuss how these challenges are addressed. Finally, we will describe some important implementation details.

4.1 Translating POD Problems to FOND Problems

Following the linear translation $X(P)$ [8] of POD problems P as described in Section 2.5, we can obtain a FOND model from a given POD problem. Let us look at a running example of the Wumpus World (also depicted in Figure 4.1) to understand the translation better.

In this domain, an agent acts in a grid world, looking for hidden gold that glitters, while avoiding pits that she could fall into, or Wumpuses that she could be eaten by. Pits and Wumpuses are also hidden, but they emit a breeze, and a stench, respectively, to adjacent cells, which the agent can sense. The agent can also sense the presence of glitter in the current cell. For simplicity, we will only consider a domain with a 4x4 grid, one Wumpus, and no pits.

Original POD Model. The original POD model $P = \langle V, I, G, A, W \rangle$ can be defined as follows:

- **State variables:** $V = \{Agent\_pos, Wumpus\_pos, Gold\_pos, Alive\}$ with:
  - $D_{Agent\_pos} = D_{Wumpus\_pos} = D_{Gold\_pos} = \{p1, p2, p3, p4\}$.
  - Variable $Adj\_i\text{-}j$ for each pair of adjacent cells i and j, with domain $\{True, False\}$, which is true if the cells i and j are adjacent.
  - The $Alive$ variable is $True$ if the agent is alive, and $False$ otherwise.
- **Initial state:** I includes:
  - $Agent\_pos = p1$.
  - $Adj\_i\text{-}j = True$ for all adjacent cells i and j.

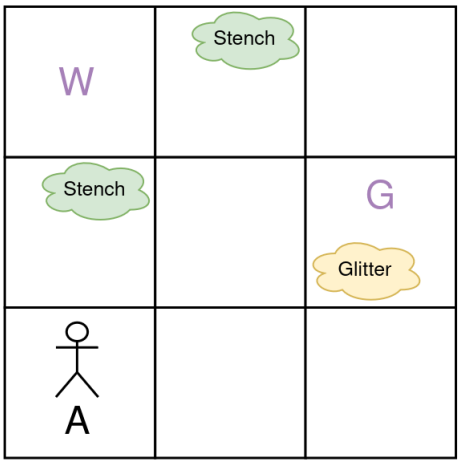

Figure 4.1: An instance of the Wumpus World. In this domain, an agent in a grid looks for treasure while being wary of Wumpuses that emit a 'stench' to adjacent cells. The agent is represented by A, the Wumpus by W, and the gold by G. Gold and Wumpus positions are hidden. The agent can sense no stench and no glitter in the current cell.

- **Goal state:** $G = \{Gold\_pos = Agent\_pos\}$.
- **Actions:** $A = \{Move\_i\text{-}j\}$ for all $i, j \in D_{Agent\_pos}$ with:
  - Precondition: $Pre(Move\_i\text{-}j) = \{Agent\_pos = i, Adj\_i\text{-}j = True, Alive = True\}$.
  - Conditional effects:
    - $True \Rightarrow Agent\_pos = j$.
    - $Wumpus\_pos = j \Rightarrow Alive = False$.
- **Sensor model:** $W = \{Stench\_i, Glitter\_i\}$ for all $i \in D_{Agent\_pos}$, where $D_{Stench_i} = D_{Glitter_i} = \{True, False\}$:
  - **Stench sensing:**
    - For $Stench\_i = True$:
      Precondition: $Pre(Stench\_i = True) = \{Agent\_pos = i\}$.
      State formula: $\bigvee_{j|Adj\_i\text{-}j=True}(Wumpus\_pos = j)$.
    - For $Stench_i = False$:
      Precondition: $Pre(Stench\_i = False) = \{Agent\_pos = i\}$.
      State formula: $\bigvee_{j|Adj\_i\text{-}j=False}(Wumpus\_pos = j)$, ensuring that there is a possibility of a Wumpus being in other cells that are not adjacent to cell i.
  - **Glitter sensing:**
    - For $Glitter\_i = True$:
      Precondition: $Pre(Glitter\_i = True) = \{Agent\_pos = i\}$.
      State formula: $Gold\_pos = i$.

    - For $Glitter\_i = False$:
      Precondition: $Pre(Glitter\_i = False) = \{Agent\_pos = i\}$.
      State formula: $\bigvee_{j \in D_{Gold\_pos}, j \neq i}(Gold\_pos = j)$.

Translated Model. The translated model $X(P) = \langle F', I', G', A' \rangle$ (explained in Section 2.4) using the above encoding is defined below. It also includes the set of deductive closure rules D' that are generated as part of the translation, which are used to infer new knowledge from the known literals.

- **State variables:** F' includes K-literals for all state variables:
  - $\{KAgent\_pos = i, KAgent\_pos \neq i\}$ for all i in D_{Agent_pos}.
  - $\{KWumpus\_pos = i, KWumpus\_pos \neq i\}$ for all i in D_{Agent_pos}.
  - $\{KGold\_pos = i, KGold\_pos \neq i\}$ for all i in D_{Agent_pos}.
  - $\{KAdj\_i\text{-}j = True, KAdj\_i\text{-}j = False\}$ for all $Adj\_i\text{-}j$ in V.
- **Initial state:** I' includes:
  - Agent position: $KAgent\_pos = p1$.
  - Adjacency information: $KAdj\_p1\_p2 = True$, $KAdj\_p1\_p4 = True$, $KAdj\_p2\_p3 = True$, $KAdj\_p3\_p4 = True$.
  - To make this state consistent with the deductive closure rules D', we also need to add:
    - Negative agent positions:
      $KAgent\_pos \neq p2, KAgent\_pos \neq p3, KAgent\_pos \neq p4$.
    - Non-adjacency: $KAdj\_i\text{-}j = False$ for all non-adjacent pairs of cells i and j.
- **Goal state:** $G' = \{KGold\_pos = KAgent\_pos\}$.
- **Actions:** $A' = \{Move\_i\text{-}j\}$ for all $i, j \in D_{Agent\_pos}$ with:
  - Precondition: $Pre(Move\_i\text{-}j) = \{KAgent\_pos = i, KAdj\_i\text{-}j = True\}$.
  - Conditional effects:
    - $True \Rightarrow KAgent\_pos = j$.
    - $True \Rightarrow KAgent\_pos \neq i$ (from action compilation).
    - $KWumpus\_pos = j \Rightarrow KAlive = False$.
    - $\neg KWumpus\_pos \neq j \Rightarrow \neg KAlive$.
  - The last effect is interesting because it reasons about incomplete knowledge about the Wumpus position and alive-ness of the agent. If the agent does not know that the Wumpus is not in cell j, then it cannot know that it is alive after moving to cell j. This is an example of how the translation captures the uncertainty explicitly.
- **Deductive closure rules:** D' for each variable X and value x in its domain, the following rules to ensure mutual exclusivity and exhaustivity:

$$D' = \{Kx \Rightarrow \bigwedge_{x' : x' \neq x} K\bar{x}', \bigwedge_{x' : x' \neq x} K\bar{x}' \Rightarrow Kx\}.$$

Translated Sensor Model (Observations). Observations are codified, as mentioned in Section 2.5, by looking at all the sensor models for $Y \neq y'$ when $Y = y$ is observed. The sensor models are translated as follows:

- **Glitter sensing:**
  - For $Glitter\_i = True$:
    Precondition: $\{KAgent\_pos = i\}$ for all i in D_{Agent_pos}.
    State formula: $\bigwedge_{j \in D_{Gold\_pos}, j \neq i} KGold\_pos \neq j$.
    This means that if the agent senses glitter in cell i, then it knows that the gold is not in any other cell. The deductive closure rules D' will then infer that $KGold\_pos = i$.
  - For $Glitter\_i = False$:
    Precondition: $\{KAgent\_pos = i\}$ for all i in D_{Agent_pos}.
    State formula: $True \Rightarrow KGold\_pos \neq i$, or simply $KGold\_pos \neq i$.
    This means that if the agent does not sense glitter in cell i, then it knows that the gold is not in cell i.
- **Stench sensing:**
  - For $Stench\_i = True$:
    Precondition: $\{KAgent\_pos = i\}$ for all i in D_{Agent_pos}.
    State formula:

$$\bigwedge_{j \in D_{Wumpus\_pos}} (KAdj\_i\text{-}j = False \Rightarrow KWumpus\_pos \neq j) \wedge (KWumpus\_pos = j \Rightarrow KAdj\_i\text{-}j = True)$$

    As we don't have prior knowledge of wumpus position, observing stench in cell i tells the agent that the wumpus must not be in any cell not adjacent to cell i.
  - For $Stench\_i = False$:
    Precondition: $\{KAgent\_pos = i\}$ for all i in D_{Agent_pos}.
    State formula:

$$\bigwedge_{j \in D_{Wumpus\_pos}} (KAdj\_i\text{-}j = True \Rightarrow KWumpus\_pos \neq j) \wedge (KWumpus\_pos = j \Rightarrow KAdj\_i\text{-}j = False)$$

It is evident from the translated sensor model that incomplete information about the world state, expressed as disjunctions in the original sensor model, is not considered, and new observations help in eliminating underlying belief states that are inconsistent with the observation, which is a common approach in contingent planning.

The recipe for adding observations according to Bonet and Geffner [8] involves performing unit resolution on the combined set of clauses in s_a, D', and K_o, where s_a is the state resulting

from applying an action a in state s, D' is the set of deductive closure rules, and K_o is the state formulas of the translated sensor model for the observation o, expressed as conditional rules. This process generates the state s_a^o for the observation o in s_a. In the following section, we will discuss how this process is captured in the state expansion of the translated POD problem as non-deterministic state transitions.

4.2 State Expansion of the Translated POD Problem

Now that the translation of the POD model allows us to work with a states instead of beliefs, the next step in learning a general policy is to expand the state space of the translated POD problem as a non-deterministic state transition system. This is a crucial step for learning the general policy, as it provides the necessary transitions and states for the SAT theory to learn from. This section aims to describe the state expansion of a small training example $X(P_i)$, used for learning the general policy.

In the translated POD model $X(P)$, there are two points of interest, as they are not explicitly encoded as actions. First, we address the application of the deductive axioms D' during the expansion, and secondly, the progression to next state through observations.

4.2.1 Applying Deductive Axioms During Expansion

As formalized in Section 2.5, the deductive axioms D' consist of two rules that ensure the mutual exclusivity and exhaustivity of the multivalued state variables in F'. The conditions of both the rules are non-overlapping, hence they can be applied to a state s_a (the state resulting from taking an action a in s) in any order. Moreover, they only need to be applied once to achieve closure, and the operation is fast and polynomial [8]. For the expansion of the state space, we would not perform the closure under these rules as a separate operation; instead, simply add the resulting literals to the existing state s_a.

4.2.2 Modeling Sensing as Non-Deterministic Effects

An observation is a single valuation of all the observable variables Y in a POD problem P for which the sensor preconditions $Pre(Y)$ are satisfied. In Wumpus World, when the agent is in cell i, one out of four possible observations can be $Stench\_i = True, Glitter\_i = False$. The translation provides a mechanism to update the states through observation by performing unit resolution on the combined set of clauses in s_a, D', and K_o, as described in the previous section. This inference step generates the state s_a^o from s_a. Since sensing outcomes cannot be pre-determined, we do not want this operation in the state space as an explicit sensing action, rather, modelled as the non-deterministic effect of the action $a \in A'$. This distinction is important so that only the actions a are exposed for action selection in learning the general policy from the state space, and sensing is done automatically for all observable variables whose

sensor preconditions hold. Hence, the progression from state s to s_a^o will be captured as a single state transition with these intermediate steps:

1. Updating to state s_a after applying action a,
2. Attaining closure of s_a under the deductive axioms, and finally,
3. Inferring s_a^o for the observation o in s_a through unit resolution.

For each action a applicable in in state s, the non-deterministic effects come from adding the possible observations o in s_a. The intermediate state s_a is hidden and not part of the state space exposed to the SAT theory to solve for a general policy. Figure 4.2 illustrates this non-deterministic transition.

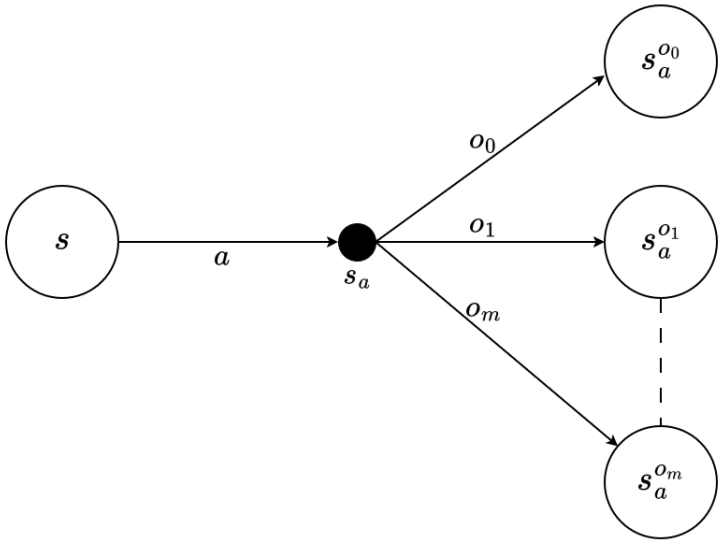

Figure 4.2: Representing the non-deterministic transition of applying action a in state s due to multiple possible sensing outcomes $\{o_0, o_1, \dots, o_m\}$. The intermediate state s_a is hidden and not part of the state space.

4.2.3 Ensuring Consistency with Hidden Problem Instances

As we are dealing with partial observability, the encoding does not explicitly capture the underlying true state of the world, but rather, is a framework for how the agent gathers knowledge about it, collected through acting, sensing, and reasoning. Hence, one POD problem implicitly encodes all the possible instances with different configurations of the hidden predicates, e.g., all the possible configurations of the Wumpus and gold positions in the Wumpus World domain, as depicted in Figure 4.3. This is not a limitation of the encoding, it is rather a consequence of the nature of the problem itself, and solving a partially observable problem is essentially finding a strategy to reach the goal state for all these possible configurations.

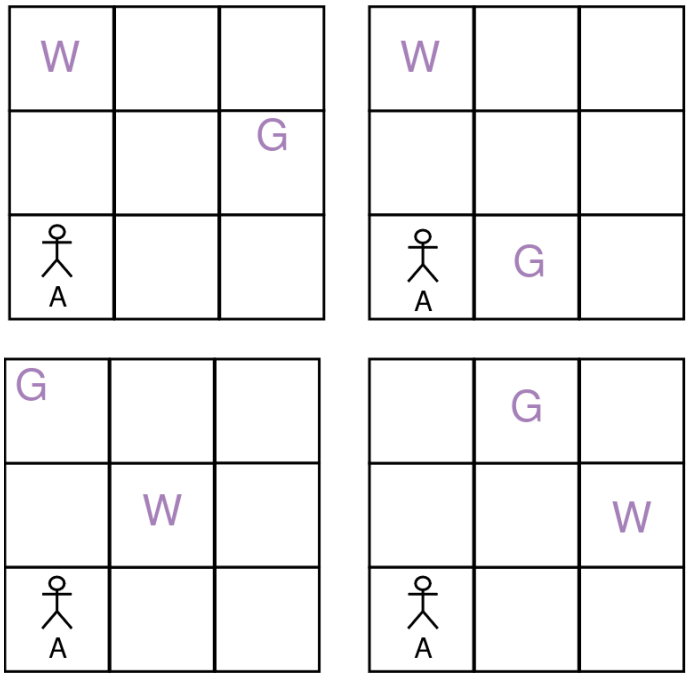

Figure 4.3: A subset of all possible instances of a Wumpus World problem where the agent is initially at the cell p_1. Observation tokens are not shown for simplicity.

Consequently, each trajectory in the state space corresponds to a particular sequence of actions and observations, which should be consistent with a particular configuration of the hidden predicates. During sensing, it is important to consider that while the sensor preconditions indicate whether a sensor reading is possible in a state, they do not specify which actual sensor readings are feasible with respect to the underlying hidden problem instance. For example, in the Wumpus World domain, the sensor preconditions for both $Stench\_i = True$ and $Stench\_i = False$ are the same, namely $\{KAgent\_pos = i\}$. However, if the agent senses stench in cell i, and later returns to it after a few moves, the agent should only sense stench again and not the absence of stench, as it cannot forget prior observations. Or if the agent senses glitter in cell i, then it should not be able to sense it in another cell j, even though the sensor preconditions allow it, as the gold can only be in one cell. Allowing the agent to sense conflicting readings would result in inconsistencies with the underlying hidden problem instance corresponding to the state trajectory, leading to unsound expansions. To address this, we perform two checks during state expansion, while expanding from a state s_a:

1. First, we check if an observation o is applicable in s_a by checking if the sensor preconditions for o are satisfied in s_a.
2. If any applicable observation results in a state s_a^o that is the same as s_a, we can conclude that this observation is the only one possible according to current knowledge about the underlying hidden problem instance, and other observations should not be considered for expansion. Alternatively, if all applicable observations lead to unseen states, then all

of them are considered for expansion. This check works because repeated sensing does not add any new information to the state.

The following pseudocode illustrates this process:

Algorithm 1 Observation Consistency Check

Input: State s_a after applying action a and deductive closure

Input: Set of observable variables $W = \{Y_1, Y_2, \dots, Y_n\}$ with domains D_{Y_i}
Output: Set of valid observations $ValidObs$ for expansion

```
ApplicableObs ← ∅
ValidObs ← ∅
for each observation o = {Y_1 = y_1, Y_2 = y_2, ..., Y_n = y_n} where y_i ∈ D_{Y_i} do
    Pre_o ← ⋃_{i=1}^{n} Pre(Y_i = y_i)          Union of all sensor preconditions
    if Pre_o ⊆ s_a then                          Preconditions satisfied if they are subset of state
        ApplicableObs ← ApplicableObs ∪ {o}
for each observation o in ApplicableObs do
    s_a^o ← UNIT(s_a ∪ D' ∪ K_o)                 Apply observation via unit resolution
    if s_a^o = s_a then                          Observation adds no new information
        return {o}                               This is the only valid observation
return ApplicableObs                             All applicable observations lead to new states
```

Following this blueprint, each problem P_i in the training set is expanded from the initial state in a breadth-first manner, until no new states are generated, and the resulting state space $\mathcal{S}$ is used for learning the general policy, as described in the next section.

4.3 Learning the Generalized Policy

Now that we have a non-deterministic state space representing our original set of POD problems, we can apply the learning approach for FOND models to learn a general policy. This section describes how the SAT-based learning framework is applied to our translated POD problems, how features are generated to describe good transitions, and what characteristics the resulting policies have.

4.3.1 The SAT-Based Learning Framework

As described in Section 2.3, the learning approach for FOND models involves finding a satisfiable assignment to a propositional theory $T(\mathcal{S}, \mathcal{F})$ that captures the transitions which are productive towards achieving the goal state, and the features that best describe them [24]. Here, $\mathcal{S}$ is the set of transitions (s, s') in the expanded state space of the training problems, and $\mathcal{F}$ is the pool of features generated for the problem domain in a context-free description

logic grammar. To deal with non-determinism, the theory is designed such that the satisfying good transitions are also *safe* in the original FOND problem, meaning that they do not lead to dead-end states.

Additionally, the input states are pre-partitioned into alive, goal and dead-end states to inform the solver for learning the good transitions. This is done by checking the guaranteed reachability of the goal state from each state in the expanded state space, and marking the states accordingly, while being aware of non-determinism in the transitions [24]. We use this learning approach out-of-the-box for our translated POD problems by providing the expanded non-deterministic state space $\mathcal{S}$ and the feature pool $\mathcal{F}$ as inputs to the SAT theory. To promote generalization, the optimization target for the weighted Max-SAT solving is the combined complexity of the selected features, which favors simpler features, and has empirically shown to lead to better generalization [14, 24].

The SAT theory, formally described in Section 2.3, encodes these sentences in propositional theory to ensure the learned policy is sound and effective:

- Good transitions are guaranteed to be safe, never leading directly to dead-end states in the non-deterministic problem.
- Alive states are defined as those from which at least one safe path to a goal state exists, i.e., every outcome of all actions in the path leads to states that are also alive.
- Non-deterministic actions associated with good transitions must have the potential to bring states closer to the goal.
- Any transition that leads to a dead-end state is explicitly excluded from being classified as good.
- The selected features must be capable of distinguishing between alive states and dead-end states.
- The selected features must be capable of distinguishing between goal states and non-goal states.
- The selected features must be capable of distinguishing between good transitions and bad transitions.

4.3.2 Feature Pool Generation

To generate the feature pool $\mathcal{F}$, we use the same context-free description logic grammar for the problem domain as used in the learning of general policies for classical and FOND models [14, 24], which generates features that are not directly tied to specific objects and predicates in the training problems. The only difference is that the domain predicates are now the K-literals in the translated POD problem. In practice, this is done using the DLPlan library [12].

DLPlan takes as input the set of states, problem predicates and objects, and incrementally generates a set of features using a description logic grammar:

- First, concept and role compositions on primitive ones are derived from the problem predicates.
- The grammar rules are applied to generate compositions until a specified upper limit $c_{\max}$ on the depth of the syntax tree is reached.
- Boolean features are generated using conditions like whether a concept is satisfied by any object, or whether a role is satisfied by any pair of objects, or whether a nullary predicate is true in a state.
- Numerical features are generated using counting and distance measures on the concepts and roles.

The pool of generated features is then used in the learning process to find satisfying assignments that help describe good transitions in the state space. These assignments are used to derive policy rules.

4.3.3 Policy Characteristics and Solution Properties

An important question is what kind of solutions we get from the learning process. In typical FOND problems, due to the behaviour of non-deterministic actions, a strong cyclic policy is acceptable as a solution under the assumption of *fairness*, as a strong policy is not feasible due to the dynamic nature of the environment and the possibility of cycles in the state space. The fairness assumption dictates that infinitely applying a non-deterministic action a in a state s would also result in all possible successor states s' of the action to occur infinitely. In simple terms, we assume that the environment is not adversarial and does not intentionally avoid certain outcomes, and that if a safe action can lead to a state that is closer to the goal, it will eventually do so if applied enough times. In contrast, in a typical POD problem, the only non-determinism in actions arises from sensing, and once a value becomes known, it remains known; i.e., uncertainty decreases monotonically [30]. The sensing outcomes are not random, but rather, determined by the underlying hidden problem instance. Hence, sensing does not lead to cycles. It only adds more information about the underlying, unchanging world, which helps to eliminate states from the current belief. Hence, we get *strong policies* as solutions.

Claim: *Although the SAT theory is designed to learn strong cyclic policies for FOND problems, it learns strong policies for our translated POD problems, as the non-determinism in the state space is only due to sensing and does not lead to cycles.*

During the state expansion process, described in Section 4.2, we took care to ensure that sensing does not lead to states that are inconsistent with the underlying hidden problem instance corresponding to that state trajectory, as each problem implicitly encodes all possible instances with different configurations of the hidden states. This is a crucial point to consider when learning general policies for POD problems, as the policy must be able to react to all possible sensing outcomes and configurations of the hidden states. In fact, this is an inherent

aspect of the problem, and a solution policy should inherently be designed to respond to all possible sensing outcomes. Moreover, like a generalized policy for FOND models, a policy solving the problem P in Q is also expected to solve $P[s]$, where s is a state that is reachable and *alive*. In this sense, the class of problems Q is closed.

4.4 Unsolvable Instances and State Space Explosion

Firstly, as discussed in Section 4.2, a POD problem implicitly encodes all possible configurations of the hidden predicates. Hence, there could be certain configurations of the hidden states that do not have guaranteed solutions, even though the encoding allows for them. During expansion, the state expansion will arrive at states that are dead-ends, not by virtue of the domain-specific obstacles, like encountering pits or Wumpuses, but by the characteristic that a guaranteed path to the goal does not exist. An example of such a world state can be seen in Figure 4.4. This is a known and a common problem for partially observable environments and affects action selection in contingent planners [1]. These states corresponding to unsolvable configurations render the whole FOND problem unsolvable, as there now exists no safe path to the goal state from the initial state. An obvious solution to this problem is to exclude the unsolvable configurations from the state space, as they mislead the learning process if they are included, and the resulting policy would not be able to solve the training problems. This could be done by identifying 'bad observations' that correspond to unsolvable configurations, and pruning the state space by not allowing transitions corresponding to these observations. For example, in the Wumpus World, if the agent senses a stench in the initial cell, then there is no safe path to the goal. Hence, this observation should be omitted from the state space. In brief, the idea is to go over the expanded state space from bottom-up and identify dead-end states with alive siblings from the same parent (corresponding to 'good' observations), and prune the state space by not allowing transitions to these dead-end states, while retaining the transitions to the alive siblings. This only works if the domains do not have surprising dead-ends, which is the case for the domains we are working with and how the sensing models are defined, but it may not be the case for all domains.

However, we observed a second challenge during experimentation: the state space of the translated POD problem can be quite large, even for small problems, due to all the possible combinations of sensing outcomes corresponding to different configurations of the hidden states. This is a problem for learning the general policy, as the SAT theory becomes too large to be solved. To address this problem, it becomes pertinent to limit the number of underlying hidden problem instances that are captured in the state expansion for the training set of POD problems. We do this by fixing a set of (hidden) ground truth instances and limiting sensing outcomes that are consistent with those ground instances. For example, in the Wumpus World domain, we can pre-define a subset of possible hidden Wumpus and Gold positions for each of the training problems.

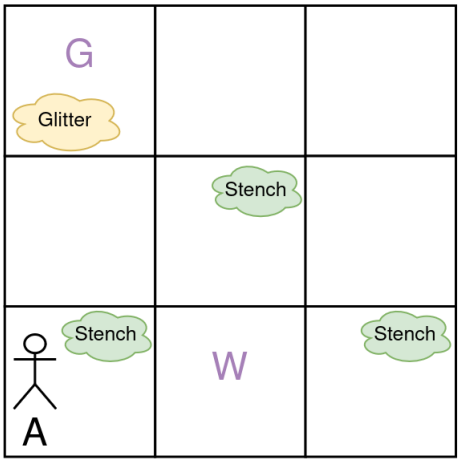

Figure 4.4: An unsolvable instance of the Wumpus World. In this domain, an agent in a grid looks for treasure while being wary of Wumpuses that emit a 'stench' to adjacent cells. At the current position, the agent senses a stench and therefore knows that a Wumpus is nearby. Consequently, it cannot move either up or left.

This allows us to generate a more manageable state space for learning and also to automatically exclude unsolvable configurations, addressing the first problem. For instance, we can simply exclude Wumpus positions that are adjacent to the initial cell, as they lead to an unsolvable problem. This relieves us of the effort of algorithmically identifying unsolvable configurations and pruning the state space, as the pre-defined hidden predicate values already take care of that. We go forward with this approach for our experiments.

4.5 Implementation

In this section, we will discuss some details for the implementation of the approach described in the previous sections. We will talk about the representation of the state variables, the learning process through iterative SAT solving.

4.5.1 Multivalued State Variables

The POD model is defined on multivalued state variables, whereas the generalized planning approach that we will utilise expects the domain to be defined using boolean state variables. For practical purposes, we adapt to the expected representation (without increasing complexity) by expressing the variable assertions $KX = x$ and $KX \neq x$ as boolean predicates, $KX(x)$ and $K\neg X(x)$. This is similar to how PDDL expresses propositions, and hence, allows us to leverage existing PDDL parsers for domain and problem definition. As long as the action compilation

is correct, and deductive axioms are correctly applied for the specified domains of the state variables, the boolean representation is sufficient for the state expansion and learning process.

4.5.2 Learning through Iterative SAT Solving

The learning algorithm we use for generalized policies in POD domains is the same as the framework proposed by [24] for FOND domains. The propositional theory $T(\mathcal{S}, \mathcal{F})$ is modelled with Answer Set Programming (ASP) [27] using the solver clingo [15]. The features are generated using the DLPlan library [12] with the same description logic grammar as used in [24] for FOND problems, but with the K-literals as the primitive predicates. For domain and problem parsing, we use the `pddl` library [28], with modifications to the parser to handle sensing models and domain definitions for multivalued state variables for the POD problems.

Once we have the state expansion of the translated POD problems, we can apply the same learning algorithm to find generalized policies. The complete learning algorithm by [24] is shown in Algorithms 2 and 3. The algorithm iteratively processes problems in the class $\mathcal{Q}$, starting with the smallest instance. For each problem P, the algorithm first checks whether the current policy π solves it; if so, P is added to the solved set $\mathcal{S}$ and the algorithm continues to the next problem. Otherwise, P is added to the training set $\mathcal{T}$, and a new policy must be learned by invoking the incremental complexity solver.

Algorithm 2 Control Loop for Learning Generalized Policies

Input: Problem class $\mathcal{Q} = \{P_1, \ldots, P_k\}$, maximum complexity $c_{\max}$
Output: Generalized policy π for $\mathcal{Q}$

```
c_min ← 1                                              Minimum complexity for solver
cost_max ← ∞                                           Cost bound for solver
T ← ∅                                                  Training set
S ← ∅                                                  Solved set
π ← ∅
for each problem P ∈ Q do
    if CHECKPOLICY(π, P) then
        Add P to S
        continue
    Add P to T
    (π, c_min, cost_max) ← INCREMENTALSOLVER(T, c_min, c_max, cost_max)
    Add P to S
return π
```

The incremental complexity solver (Algorithm 3) employs a strategy to balance feature expressiveness with policy cost. The solver increments the maximal complexity of all features in $\mathcal{F}$ from the minimum complexity $c_{\min}$ needed for the last policy up to the maximal complexity $c_{\max}$. For each complexity level c, features up to that complexity are generated using GENERATEFEATURES, and the SAT theory $T(\mathcal{T}, \mathcal{F})$ is solved with an upper bound $\text{cost}_{\max}$

on the total policy cost. When a new policy π_{new} is found, it is returned immediately, the minimum complexity $c_{\min}$ is updated to c for future invocations, and the cost bound is tightened to $\text{cost}(\pi_{\text{new}}) - 1$ to force the solver to find cheaper solutions at higher complexity levels in subsequent calls. This way, the solver often finds an expensive first policy with low-complexity features and then iteratively improves this policy with features of higher complexity while exploiting the upper bound on the total cost. The process continues until all problems in $\mathcal{Q}$ are solved.

Algorithm 3 Incremental Complexity Solver

Input: Training set $\mathcal{T}$, minimum complexity $c_{\min}$, maximum complexity $c_{\max}$, cost bound $\text{cost}_{\max}$
Output: Policy π, updated $c_{\min}$, updated $\text{cost}_{\max}$

```
for c ← c_min to c_max do
    F ← GENERATEFEATURES(c)
    π_new ← SOLVE(T, F, c, cost_max)
    if π_new ≠ ∅ then
        c_min ← c
        cost_max ← cost(π_new) − 1
        return (π_new, c_min, cost_max)
return (∅, c_min, cost_max)                    No solution found
```

The key steps in the overall implementation include:

- Translating the POD problems into a format suitable for state expansion.
- Performing consistent state expansion to generate the state space of the problems.
- Applying the learning algorithm (as shown in Algorithms 2 and 3) iteratively starting with the smallest problem in $\mathcal{Q}$, incrementally building the training set $\mathcal{T}$ and updating the policy π with features of increasing complexity whenever a problem fails.
- Testing the learned policy on each problem in $\mathcal{Q}$ and adding successfully solved problems to $\mathcal{S}$ until all problems are solved, at which point the final policy π is returned.

5 Results

In this chapter, we will present the implementation details and the results of our experiments on learning generalized policies for POD domains. We will see how the learning process performs on various benchmark domains and compare the results with those obtained for FOND domains. We will also discuss the solution policies learned and their generalization capabilities across different problem instances.

5.1 Evaluation

To evaluate our approach, we present experimental results on several benchmark domains with varying complexity. While these domains have been previously used in the literature for partially observable deterministic planning, learning generalized policies that can scale across varying problem sizes is a novel task for POD domains. We evaluate our methods on interesting benchmark domains from previous work on planning under partial observability [2, 6, 8], including Wumpus, Kill-Wumpus, Coloured-Balls, Localize, Binary Search, Bomb in the Toilet, and Doors. These domains provide diverse challenges involving grid-based navigation, sensing, and manipulation under partial observability. To assess both the correctness and generalization capabilities of the learned policies, we start with small training instances (up to about 5 objects each) and evaluate whether the policies can generalize to larger, unseen problem instances through iterative solving and verifying (as discussed in Section 4.5.2), while managing the inherent state-space explosion that characterizes POD domains. These domains include:

- **Doors**: An agent moves in a grid with the objective of reaching a known goal position. On the way, it might come across walls as obstacles or doors that can be sensed and passed through. Each even column is a wall with a door. Depending on the problem instance, the agent might have to pass through multiple doors to reach the goal in the rightmost column.
- **Localize**: An agent starting at an unknown position in a grid moves in a grid with walls blocking its way to the goal position at the top-right corner. It knows the layout of the grid and the position of the walls, but not its own position. The agent can sense whether walls exist in an adjacent cell in all four directions, and move to free cells to localize itself and reach the goal. We start with a grid of size 3×3 for training and evaluate on grids of size up to 7×7. The layout of walls is similar across all problem instances, with the number of walls increasing as the grid size increases.

- **Wumpus**: An agent in a grid world looks for treasure. It has to be wary of Wumpuses that emit a 'stench' to adjacent cells. The domain consists of one Wumpus and one Gold. We start with a grid of size 2×2 for training and evaluate on grids of size up to 7×7.
- **Kill-wumpus**: Similar to the wumpus world but the objective is only to locate and kill one Wumpus in the grid that emits a cell to the adjacent cells
- **Coloured-balls** In a grid world, there are cells with coloured baskets that expect the same coloured balls which are scattered all over the grid. The objective is to collect all the balls in the appropriate baskets. The agent can sense whether a ball is present in the current cell, and its colour when picked up. We start with a grid of size 2×2 for training and evaluate on grids of size up to 5×5. There are two colours of baskets and one ball that may be of any of the two colours.
- **BTCS** In this domain, there are packages that may or may not contain a bomb which can be defused by dunking it in an unclogged toilet and then flushing it. The objective is to defuse the bomb. The agent can sense whether a package contains a bomb. We start with a problem instance with 3 package for training and evaluate on instances with up to 15 packages.
- **Binary Search** In this domain, there are n nodes in a list, one of which is the secret node. The agent can sense whether the secret node is to the left or right of a given node, and the objective is to find the target node. We start with a problem instance with 5 nodes for training and evaluate on instances with up to 50 nodes.

The domain definitions these domains can be found in the Appendix A.

Results: The following table summarizes the results for the domains defined above.

Domain	$\|P\|$	$\|T\|$	$\|S\|$	$\|O\|_T$	$\|O\|_P$	t_{solve}	t_{wall}	*mem*	$\|F\|$	$\|\Phi\|$	k^*	c_Φ
doors	5	3	4	20	32	73.95	4453.47	42935.76	1025	3	7	15
localize	3	3	3	49	49	977.68	8441.99	8560.91	1710	3	6	16
wumpus-1	5	2	5	6	49	1400.73	1781.94	23647.91	1138	3	7	9
k-wumpus	6	3	6	12	49	388.76	3177.80	33950.15	234	4	7	16
cballs	5	3	2	14	30	8867.29	2667.33	48090.89	2241	4	9	28
btcs	5	1	5	5	17	0.097	1850.67	40.0	14	3	2	4
search	3	1	3	5	50	102.79	2363.14	6234.83	288	2	6	7

Table 5.1: Experimental results for learning generalized policies across benchmark POD domains. $|P|$: Total number of problems, $|T|$: Number of problems used in training, $|S|$: Number of solved problems (including training and testing), $|O|_T$: Maximum number of objects in all training instances, $|O|_P$: Maximum number of objects in all instances, t_{solve}: CPU time needed for finding the best policy by the SAT solver, t_{wall}: Total wall time, *mem*: Maximum memory consumption (in MB), $|F|$: Size of the feature pool, $|\Phi|$: Number of selected features (bounded at 15 for all the experiments), k^*: Maximum cost of the selected features, c_Φ: Total cost of all selected features.

The results demonstrate successful learning and generalization of policies across most evaluated domains using the proposed approach. Most problems were successfully solved, including test instances larger than those used for training, though some domains (Doors and Coloured-Balls) achieved partial coverage with 4/5 and 2/5 problems solved respectively. Notably, the learned policies generalize effectively: for instance, in the Wumpus domain, a policy trained on instances with up to 6 objects successfully solved problems with up to 49 objects, while in Binary Search, training on instances with 5 objects enabled solving problems with 50 objects. The learned policies are compact, using only 2-4 selected features from feature pools ranging from 14 to 2241 candidates, with feature complexities bounded at 2-9. Computational requirements vary substantially across domains: BTCS exhibits the most efficient learning (0.097s solve time, 40 MB memory), while the Coloured-Balls domain represents the most challenging case (148 minutes solve time, 47 GB memory), followed by Wumpus (23 minutes solve time, 23.6 GB memory). Total wall times, which include state expansion and the iterative SAT solving process, range from approximately 30 minutes (Wumpus and BTCS) to 2.4 hours (Localize), with Doors also requiring substantial time (1.2 hours), reflecting the computational demands of handling belief-state spaces in POD domains.

A common pattern for learning generalized policies for benchmark POD domains is that the learning process is more computationally intensive than for typical FOND domains, due to much larger state spaces resulting from the epistemic literals introduced in the translation, which basically encodes all possible belief states. While benchmark FOND domains can solve problems with up to even around 100 objects for simpler domains [24], the POD domains we

evaluated on could only solve problems with up to around 50 objects within feasible time and memory limits. Moreover, during experimentation, we observed that the size of individual states in larger problem instances causes a bottleneck in feature evaluation of states during policy execution for verification. For example, in the Localize domain, execution of the learned policy on a test instance beyond the training size (with 49 objects) was not feasible due to the large state size and the resulting large feature evaluation time (2-3 hours per ground instance).

5.2 Example Learned Policies

The learned policies for the evaluated domains are compact and interpretable, consisting of a small number of selected features that capture the essential conditions for action selection. In this section, we present examples of the learned policies for the Localize, Wumpus, and Binary Search domains, highlighting their structure and generalization capabilities.

5.2.1 Wumpus World

The general policy for the Wumpus World domain with 1 Wumpus and no pits selects three features, represented both symbolically and as description logic features:

- $A \equiv |K(alive)|$: boolean feature which is true exactly when the agent *knows* it is alive.

```
b_nullary(K_pos_alive)
```

- $G \equiv |K(got\text{-}treasure)|$: boolean feature which is true when the treasure is *known* to have been collected.

```
b_nullary(K_pos_got_treasure)
```

- $d \equiv dist(pos, safe\text{-}adj, position\_KG)$: the distance from the agent's current cell to the goal state via adjacent cells that are *known* to not be occupied by a Wumpus and not contain gold.

```
n_concept_distance(c_primitive(K_pos_at,0),
r_restrict(r_primitive(K_pos_adj,0,1),c_primitive(K_neg_wumpus-at,0)),
c_not(c_primitive(K_neg_gold-at,0)))
```

The set of learned rules are:

$$\{A, d = 0, \neg G\} \mapsto \{\}$$
$$\{A, d = 0, \neg G\} \mapsto \{\uparrow d, G\}$$
$$\{A, d > 0, \neg G\} \mapsto \{\downarrow d\}$$

The three rule clauses can be read as follows:

- Start : The first rule corresponds to applying the `start` action when the agent is in the initial state and senses glitter, which means it now knows that the treasure is in the current cell, and hence the distance to the goal is 0. Applying the start action does not change the features, hence the empty effect set.
- Pick up the treasure : If the agent is alive, on a safe cell, and current cell is known to contain the treasure, then the policy selects an action that leads to picking up the treasure, leading to the goal state.
- Explore a safe cell : If the agent is alive, the distance to a known safe cell where the treasure is not known to be *not* present (yet) is greater than 0, then the policy selects an action that leads to a decrease in this distance, which means moving towards the nearest unexplored safe cell.

As we can see, the policy captures rule conditions that are built over $\neg KL$ or $\neg K\neg L$ literals, which enable reasoning over uncertainty and thus enabling exploration, which is a key aspect of planning under partial observability.

5.2.2 Binary Search

This domain is interesting as it is a non-spatial domain, and the learned policy is not based on distance features but rather on features that capture the relative position of the current node to the target one. The learned policy selects two features:

- $T \equiv |Ksecret\text{-}not\text{-}testing|$: boolean feature which is true when there is no known secret node that is being tested.

  ```
  b_empty(c_and(c_primitive(K_pos_secret,0),c_primitive(K_pos_testing,0)))
  ```

- $D \equiv |K\_not\_discover|$: boolean feature which is true when there has been no node 'discovered' yet. In the domain, once a node is discovered, it means that either we reach the goal state after discovering the secret correctly, or a trapping state.

  ```
  b_nullary(K_pos_discover-not-yet-attempted)
  ```

- $F \equiv |K\_finish|$: boolean feature which is true when the secret node is known to have been found, i.e., the goal has been reached.

  ```
  b_nullary(K_pos_finish)
  ```

The rules for the learned policy are:

$$\{T, D, \neg F\} \mapsto \{\neg T\}$$
$$\{D, \neg T, \neg F\} \mapsto \{F, \neg D\}$$

These rules can be read as follows:

- Test until the secret is known: The first rule corresponds to testing a node when there is no known secret node being tested and no node has been discovered yet. This leads to the feature T becoming false, as we now have a known secret node that is being tested. This rule enables the action of testing the highest node that is still untested, as it is the only action that can possibly lead to testing the known secret node when the observation is that the secret is less than the current node. But the true observation on applying this action may not lead to a known secret node being tested (leading to no change in the features). Hence, this rule can be applied repeatedly.
- Finish after discovering the secret: The second rule corresponds to testing a node when there is no node discovered yet but there is a known secret node being tested. This leads to the feature F becoming true, as we have now discovered the secret node and reached the goal state, and D becoming false as we have now attempted to discover a node.

5.2.3 Discussion

The learned policies for the evaluated domains are compact and interpretable, consisting of a small number of selected features that capture the essential conditions for action selection. The policies effectively leverage epistemic features that represent the agent's knowledge about the world, enabling reasoning under uncertainty and guiding exploration in partially observable settings. The generalization capabilities of the learned policies are demonstrated by their successful application to larger, unseen problem instances, indicating that they capture underlying structural properties of the domains rather than overfitting to specific training instances. Overall, the results highlight the potential of combinatorial optimization approaches for learning generalized policies in complex planning domains with partial observability.

A natural advantage of learning policies is that they are executed in an online manner and do not freeze when the underlying problem instance is not solvable, as opposed to traditional contingent or Partially Observable Markov decision process (POMDP) planners that would abort or give infinite costs when the belief state contains dead-end states [1].

6 Conclusion

POD planning extends classical planning with partial observability, where information about the current state can be gathered through sensing. The aim of this thesis is to find generalized policies for POD domains that solve a family of problems once learned, rather than learning a policy for each problem individually. Originally formalized as a search problem in belief space, the POD model can be efficiently translated to an epistemic state space, in which the uncertainty about the current states is compiled away using literals KL and $\neg KL$, thereby transforming it into a fully observable problem at the knowledge level. As sensing outcomes are not known a priori, they are modeled as non-deterministic effects of the actions. We propose to learn general policies for the obtained FOND model using a self-supervised learning approach that utilizes combinatorial optimization, similar to the one proposed in [14, 24]. The idea is that the expanded state space $\mathcal{S}$ of a small training set of problem instances, along with a feature pool $\mathcal{F}$, can be input to an appropriate propositional theory $T(\mathcal{S}, \mathcal{F})$, with the aim of finding a satisfying assignment of 'good' transitions that lead to the goal state, and the most appropriate features distinguishing these transitions from the 'bad' ones that do not lead to the goal state. The policy rules $C \mapsto E$ are then derived from this set of transitions by capturing how the values of the selected features change within them. We evaluated the approach on various benchmark POD domains and demonstrated that the approach can learn policies that generalize to larger problem instances than those used for training, though with varying degrees of success and significant computational requirements across different domains.

This thesis makes several contributions to the field of automated planning under partial observability:

Novel application of generalized policy learning We present the first application of a combinatorial approach to learning generalized policies for POD domains, utilizing the methodology originally developed for FOND domains [24]. This represents a step beyond existing work in POD planning, which has focused primarily on learning plans for individual problem instances.

Translation-Based Approach We demonstrate that the translation of POD problems from belief spaces to epistemic state spaces based on [8], combined with our subsequent treatment as FOND state expansion, provides a viable foundation for learning generalized policies for POD domains of width 1. The approach successfully leverages the knowledge-level representation

with epistemic literals (KL, $\neg KL$, $\neg K \neg L$, and $K \neg L$) to reason about uncertainty and derive effective policies.

Empirical Validation Across Multiple Domains We evaluated our approach on seven benchmark POD domains (Doors, Localize, Wumpus, Kill-Wumpus, Coloured-Balls, BTCS, and Binary Search). The results show that the approach can learn compact policies using 2-4 features that successfully generalize in several domains (notably Binary Search, Wumpus, Kill-Wumpus, BTCS, and Localize), though some domains (Doors and Coloured-Balls) achieved only partial problem coverage due to computational constraints.

6.1 Limitations and Future Work

While our approach demonstrates promising results, several limitations and challenges emerged during this work:

Learning generalized policies for POD domains is substantially more computationally intensive than for typical FOND domains. The epistemic literals introduced during translation encode all possible belief states, leading to significantly larger state spaces. This results in increased time and memory requirements for state expansion, feature evaluation of the states, and SAT solving during the learning process. As a result, our experiments were limited to relatively small problem instances (up to 50 objects), and scaling to larger instances remains a significant challenge.

A Appendix

A.1 PDDL Domain: Binary Search

Listing A.1: PDDL domain for binary search with partial observability.

```
(define (domain binary-search)
    (:requirements :strips :typing :existential-preconditions)
    (:types state)
    (:predicates
        (secret ?p - state)
        (lt ?p ?q - state)
        (less-than ?p - state)
        (discover-not-yet-attempted)
        (finish)
        (testing ?p - state)
    )

    (:state-variable (lt-var ?p ?q - state) (lt ?p ?q))
    (:state-variable (hidden-secret)
                     (forall (?p - state) (secret ?p)))
    (:state-variable (testing-var) (forall (?p - state) (testing ?p)))
    (:obs-variable (obs-test ?p - state) (less-than ?p))

    (:sensing-model
        :parameters (?p - state)
        :model-for (less-than ?p)
        :precondition (and (testing ?p) (discover-not-yet-attempted))
        :such-that (exists (?q - state) (and (secret ?q) (lt ?q ?p)))
    )

    (:sensing-model
        :parameters (?p - state)
        :model-for (not (less-than ?p))
        :precondition (and (testing ?p) (discover-not-yet-attempted))
        :such-that (or (secret ?p)
                       (exists (?q - state)
                          (and (secret ?q) (lt ?p ?q))))
    )

    (:action test
        :parameters (?p - state)
```

```
        :precondition (discover-not-yet-attempted)
        :effect (testing ?p)
    )

    (:action discover
        :parameters (?p - state)
        :precondition (discover-not-yet-attempted)
        :effect (and (not (discover-not-yet-attempted))
                     (when (secret ?p) (finish)))
    )
)
```

A.2 PDDL Domain: Coloured Balls

Listing A.2: PDDL domain for coloured balls with partial observability.

```
(define (domain colorballs)
    (:requirements :strips :typing :existential-preconditions)
    (:types pos obj col gar)
    (:predicates
        (adj ?i ?j - pos)
        (color ?o - obj ?c - col)
        (garbage-at ?t - gar ?p - pos)
        (garbage-color ?t - gar ?c - col)
        (trashed ?o - obj)
        (at ?i - pos)
        (holding ?o - obj)
        (obj-at ?o - obj ?i - pos)
        (obs-obj-at ?o - obj ?i - pos)
        (obs-obj-col ?o - obj ?c -col)
        (need-start)
        (arm-free)
    )

    (:state-variable (adj-var ?p ?q - pos) (adj ?p ?q))
    (:state-variable (garbage-at-var ?t - gar ?p - pos)
                     (garbage-at ?t ?p))
    (:state-variable (garbage-color-var ?t - gar ?c - col)
                    (garbage-color ?t ?c))
    (:state-variable (trashed-var ?o - obj) (trashed ?o))
    (:state-variable (holding-var ?o - obj) (holding ?o))
    (:state-variable (var-agent-at) (forall (?p - pos) (at ?p)))
    (:state-variable (var-obj-at ?o - obj)
                     (or (holding ?o) (trashed ?o))
                     (forall (?p - pos) (obj-at ?o ?p)))
    (:state-variable (var-obj-col ?o - obj)
```

```
                    (forall (?c - col) (color ?o ?c)))
    (:obs-variable (var-obs-at ?o - obj ?p - pos)
                   (obs-obj-at ?o ?p)) ; binary
    (:obs-variable (var-obs-col ?o - obj)
                   (forall (?c - col) (obs-obj-col ?o ?c)))

    (:sensing-model
        :parameters (?o - obj ?p - pos)
        :model-for (obs-obj-at ?o ?p)
        :precondition (and (at ?p) (not (need-start)))
        :such-that (obj-at ?o ?p)
    )

    (:sensing-model
        :parameters (?o - obj ?p - pos)
        :model-for (not (obs-obj-at ?o ?p))
        :precondition (and (at ?p) (not (need-start)))
        :such-that (not (obj-at ?o ?p))
    )

    (:sensing-model
        :parameters (?o - obj ?c - col)
        :model-for (obs-obj-col ?o ?c)
        :precondition (and (holding ?o) (not (need-start)))
        :such-that (color ?o ?c)
    )

    (:action start-action
        :parameters (?p - pos)
        :precondition (and (at ?p) (need-start))
        :effect (not (need-start))
    )

    (:action move
        :parameters (?i ?j - pos)
        :precondition (and (adj ?i ?j) (at ?i) (not (need-start)) )
        :effect (and (not (at ?i)) (at ?j))
    )

    (:action pickup
        :parameters (?o - obj ?i - pos)
        :precondition (and (at ?i) (obj-at ?o ?i)
                    (not (need-start)) (arm-free))
        :effect (and (holding ?o)
                     (not (obj-at ?o ?i)) (not (arm-free)))
    )

```

```
    (:action trash
        :parameters (?o - obj ?c - col ?t - gar ?p - pos)
        :precondition (and (holding ?o) (at ?p) (garbage-at ?t ?p)
                           (color ?o ?c) (garbage-color ?t ?c))
        :effect (and (trashed ?o) (not (holding ?o)) (arm-free))
    )
)
```

A.3 PDDL Domain: Doors

Listing A.3: PDDL domain for doors with partial observability.

```
(define (domain doors)
    (:requirements :strips :typing :existential-preconditions)
    (:types row col pos)
    (:predicates
        (adj ?i ?j - pos)
        (at-col ?c - col ?i - pos)
        (at-row ?r - row ?i - pos)
        (right-col ?c - col ?i - pos)
        ;(even ?c - col)
        (need-start)
        (at ?i - pos)
        (opened ?i - pos)
        (obs-open ?i - pos)
    )

    (:state-variable (adj-var ?i ?j - pos) (adj ?i ?j))
    (:state-variable (at-col-var ?c - col ?i - pos) (at-col ?c ?i))
    (:state-variable (at-row-var ?r - row ?i - pos) (at-row ?r ?i))
    (:state-variable (right-col ?c - col ?i - pos) (right-col ?c ?i))

    (:state-variable (agent-pos) (forall (?i - pos) (at ?i)))
    (:obs-variable (obs-at ?i - pos) (obs-open ?i))

    (:sensing-model
        :parameters (?i - pos)
        :model-for (obs-open ?i)
        :precondition (at ?i)
        :such-that (exists (?p - pos ?c - col ?r - row)
                        (and (at-row ?r ?i)
                             (at-row ?r ?p)
                             (right-col ?c ?i)
                             (at-col ?c ?p)
                             (opened ?p)))
    )
```

```
35
36      (:sensing-model
37          :parameters (?i - pos)
38          :model-for (not (obs-open ?i))
39          :precondition (at ?i)
40          :such-that (exists (?p - pos ?c - col ?r - row)
41                              (and (at-row ?r ?i)
42                                   (at-row ?r ?p)
43                                   (right-col ?c ?i)
44                                   (at-col ?c ?p)
45                                   (not (opened ?p))))
46      )
47
48      (:action start
49          :parameters (?i - pos)
50          :precondition (and (at ?i) (need-start))
51          :effect (not (need-start))
52      )
53
54      (:action move
55          :parameters (?i - pos ?j - pos)
56          :precondition (and (adj ?i ?j) (at ?i)
57                              (opened ?j) (not (need-start)))
58          :effect (and (not (at ?i)) (at ?j))
59      )
60  )
```

A.4 PDDL Domain: BTCS

Listing A.4: PDDL domain for BTCS.

```
1  (define (domain btcs)
2      (:types package bomb toilet)
3
4      (:predicates    (in ?p - package ?b - bomb)
5                      (defused ?b - bomb)
6                      (clogged ?t - toilet)
7                      (sensed ?p - package ?b - bomb)
8                      (testing)
9      )
10
11     (:state-variable (bomb-loc ?b - bomb)
12                 (forall (?p - package) (in ?p ?b)))
13     (:obs-variable (bomb-in-package ?p - package ?b - bomb)
14                 (sensed ?p ?b))
15
```

```
    (:sensing-model
        :parameters (?p - package ?b - bomb)
        :model-for (sensed ?p ?b)
        :precondition (testing)
        :such-that (in ?p ?b)
    )

    (:sensing-model
        :parameters (?p - package ?b - bomb)
        :model-for (not (sensed ?p ?b))
        :precondition (testing)
        :such-that (exists (?q - package) (not (in ?q ?b)))
    )

    (:action test
        :parameters ()
        :precondition (not (testing))
        :effect (testing)
    )

    (:action dunk
        :parameters (?p - package ?b - bomb ?t - toilet)
        :precondition (and (not (clogged ?t)) (in ?p ?b))
        :effect (and (defused ?b) (clogged ?t) (not (testing)))
    )

    (:action flush
        :parameters (?t - toilet)
        :precondition (and )
        :effect (and (not (clogged ?t)) (not (testing)))
    )
)
```

A.5 PDDL Domain: Kill Wumpus

Listing A.5: PDDL domain for Kill Wumpus.

```
(define (domain k-wumpus)
    (:types pos)
    (:predicates
        (adj ?p ?q - pos)
        (need-start)
        (at ?p - pos)
        (wumpus-at ?p - pos)
        (killed)
        (stench ?p - pos)
```

```
10          (alive)
11      )
12
13      (:state-variable (adj-var ?p ?q - pos) (adj ?p ?q))
14      (:state-variable (agent-pos) (forall (?p - pos) (at ?p)))
15      (:state-variable (wumpus-at-var)
16                       (forall (?p - pos) (wumpus-at ?p)))
17      (:obs-variable (stench-var ?p - pos) (stench ?p))
18
19      (:sensing-model
20          :parameters (?j - pos)
21          :model-for (stench ?j)
22          :precondition (at ?j)
23          :such-that (exists (?p - pos)
24                      (and (adj ?j ?p) (wumpus-at ?p)))
25      )
26
27      (:sensing-model
28          :parameters (?j - pos)
29          :model-for (not (stench ?j))
30          :precondition (at ?j)
31          :such-that (exists (?p - pos)
32                      (and (not (adj ?j ?p)) (wumpus-at ?p)))
33      )
34
35      (:action start
36          :parameters (?j - pos)
37          :precondition (and (need-start) (alive) (at ?j))
38          :effect (not (need-start))
39      )
40
41      (:action move
42          :parameters (?i ?j - pos)
43          :precondition (and (adj ?i ?j) (at ?i)
44                             (alive) (not (need-start))
45                             (not (wumpus-at ?j)))
46          :effect (and (not (at ?i)) (at ?j)
47                       (when (wumpus-at ?j) (not (alive)))
48                  )
49      )
50
51      (:action kill
52          :parameters (?p1 ?p2 - pos)
53          :precondition (and (adj ?p1 ?p2) (at ?p1) (wumpus-at ?p2)
54                             (not (need-start)))
55          :effect (killed)
56      )
```

```
)
```

A.6 PDDL Domain: Localize

Listing A.6: PDDL domain for Localize.

```
(define (domain localize)
    (:types pos)
    (:predicates
        (wall-up-of ?i - pos)    ; static predicate
        (wall-right-of ?i - pos) ; static predicate
        (wall-down-of ?i - pos)  ; static predicate
        (wall-left-of ?i - pos)  ; static predicate
        (up-of ?i ?j - pos)      ; static predicate
        (right-of ?i ?j - pos)   ; static predicate
        (down-of ?i ?j - pos)    ; static predicate
        (left-of ?i ?j - pos)    ; static predicate
        (possible ?i - pos)      ; static predicate

        (at ?i - pos)

        (free-up)
        (free-down)
        (free-left)
        (free-right)

        (need-start)
    )

    (:state-variable (wall-up-of-var ?i - pos)
                     (wall-up-of ?i))         ;binary variable
    (:state-variable (wall-right-of-var ?i - pos)
                     (wall-right-of ?i))   ;binary variable
    (:state-variable (wall-down-of-var ?i - pos)
                     (wall-down-of ?i))     ;binary variable
    (:state-variable (wall-left-of-var ?i - pos)
                     (wall-left-of ?i))     ;binary variable

    (:state-variable (up-of-var ?i ?j - pos)
                     (up-of ?i ?j))               ;binary variable
    (:state-variable (right-of-var ?i ?j - pos)
                     (right-of ?i ?j))         ;binary variable
    (:state-variable (down-of-var ?i ?j - pos)
                     (down-of ?i ?j))           ;binary variable
    (:state-variable (left-of-var ?i ?j - pos)
                     (left-of ?i ?j))      ;binary variable
```

```
41
42      (:state-variable (possible-var ?i - pos)
43                       (possible ?i))       ;binary variable
44
45      (:state-variable (position) (not (possible ?i))
46                       (forall (?i - pos) (at ?i)))
47      (:obs-variable (obs-up) (free-up))         ; binary variable
48      (:obs-variable (obs-right) (free-right))   ; binary variable
49      (:obs-variable (obs-down) (free-down))     ; binary variable
50      (:obs-variable (obs-left) (free-left))     ; binary variable
51
52      (:sensing-model
53          :parameters ()
54          :model-for (free-up)
55          :precondition (not (need-start))
56          :such-that (exists (?i - pos)
57                       (and (at ?i) (possible ?i) (not (wall-up-of ?i))))
58      )
59
60      (:sensing-model
61          :parameters ()
62          :model-for (not (free-up))
63          :precondition (not (need-start))
64          :such-that (exists (?i - pos)
65                       (and (at ?i) (possible ?i) (wall-up-of ?i)))
66      )
67
68      (:sensing-model
69          :parameters ()
70          :model-for (free-right)
71          :precondition (not (need-start))
72          :such-that (exists (?i - pos)
73                       (and (at ?i) (possible ?i) (not (wall-right-of ?i))))
74      )
75
76      (:sensing-model
77          :parameters ()
78          :model-for (not (free-right))
79          :precondition (not (need-start))
80          :such-that (exists (?i - pos)
81                       (and (at ?i) (possible ?i) (wall-right-of ?i)))
82      )
83
84      (:sensing-model
85          :parameters ()
86          :model-for (free-down)
87          :precondition (not (need-start))
```

```
88          :such-that (exists (?i - pos)
89                      (and (at ?i) (possible ?i)
90                      (not (wall-down-of ?i))))
91      )
92
93      (:sensing-model
94          :parameters ()
95          :model-for (not (free-down))
96          :precondition (not (need-start))
97          :such-that (exists (?i - pos)
98                      (and (at ?i) (possible ?i) (wall-down-of ?i)))
99      )
100
101     (:sensing-model
102         :parameters ()
103         :model-for (free-left)
104         :precondition (not (need-start))
105         :such-that (exists (?i - pos)
106                     (and (at ?i) (possible ?i)
107                     (not (wall-left-of ?i))))
108     )
109
110     (:sensing-model
111         :parameters ()
112         :model-for (not (free-left))
113         :precondition (not (need-start))
114         :such-that (exists (?i - pos)
115                     (and (at ?i) (possible ?i) (wall-left-of ?i)))
116     )
117
118     (:action start
119         :parameters ()
120         :precondition (need-start)
121         :effect (not (need-start))
122     )
123
124     (:action move-up
125         :parameters ()
126         :precondition (not (need-start))
127         :effect (and (forall (?j - pos ?i - pos)
128                         (when (and (up-of ?i ?j)
129                          (at ?i) (possible ?j))
130                               (and (at ?j) (not (at ?i))))))
131     )
132
133     (:action move-down
134         :parameters ()
```

```
135         :precondition (not (need-start))
136         :effect (and (forall (?j - pos ?i - pos)
137                         (when (and (down-of ?i ?j)
138                         (at ?i) (possible ?j))
139                               (and (at ?j) (not (at ?i))))))
140     )
141
142     (:action move-left
143         :parameters ()
144         :precondition (not (need-start))
145         :effect (and (forall (?j - pos ?i - pos)
146                         (when (and (left-of ?i ?j)
147                         (at ?i) (possible ?j))
148                               (and (at ?j) (not (at ?i))))))
149     )
150
151     (:action move-right
152         :parameters ()
153         :precondition (not (need-start))
154         :effect (and (forall (?j - pos ?i - pos)
155                         (when (and (right-of ?i ?j)
156                         (at ?i) (possible ?j))
157                               (and (at ?j) (not (at ?i))))))
158     )
159 )
```

A.7 PDDL Domain: Wumpus

Listing A.7: PDDL domain for Wumpus.

```
1  (define (domain wumpus)
2      (:types pos)
3      (:predicates
4          (adj ?p ?q - pos)
5          (need-start)
6          (at ?p - pos)
7          (wumpus-at ?p - pos)
8          (pit-at ?p - pos)
9          (gold-at ?p - pos)
10         (got-the-treasure)
11         (stench ?p - pos)
12         (breeze ?p - pos)
13         (glitter ?p - pos)
14         (alive)
15     )
16
```

```
    (:state-variable (adj-var ?p ?q - pos) (adj ?p ?q))
    (:state-variable (agent-pos)
                     (forall (?p - pos) (at ?p)))
    (:state-variable (gold-pos) (got-the-treasure)
                     (forall (?p - pos) (gold-at ?p)))
    (:state-variable (wumpus-at-cell ?p - pos)
                     (wumpus-at ?p))
    (:state-variable (pit-at-cell ?p - pos)
                     (pit-at ?p))
    (:obs-variable (stench-var ?p - pos)
                     (stench ?p))
    (:obs-variable (breeze-var ?p - pos)
                     (breeze ?p))
    (:obs-variable (glitter-var ?p - pos)
                     (glitter ?p))

    (:sensing-model
        :parameters (?j - pos)
        :model-for (stench ?j)
        :precondition (at ?j)
        :such-that (exists (?p - pos)
                     (and (adj ?j ?p) (wumpus-at ?p)))
    )

    (:sensing-model
        :parameters (?j - pos)
        :model-for (not (stench ?j))
        :precondition (at ?j)
        :such-that (exists (?p - pos)
                     (and (not (adj ?j ?p)) (wumpus-at ?p)))
    )

    (:sensing-model
        :parameters (?j - pos)
        :model-for (breeze ?j)
        :precondition (at ?j)
        :such-that (exists (?p - pos)
                     (and (adj ?j ?p) (pit-at ?p)))
    )

    (:sensing-model
        :parameters (?j - pos)
        :model-for (not (breeze ?j))
        :precondition (at ?j)
        :such-that (exists (?p - pos)
                     (and (not (adj ?j ?p)) (pit-at ?p)))
    )
```

```
64
65      (:sensing-model
66          :parameters (?j - pos)
67          :model-for (not (glitter ?j))
68          :precondition (at ?j)
69          :such-that (exists (?p - pos)
70                      (and (not (= ?p ?j)) (gold-at ?p)))
71      )
72
73      (:sensing-model
74          :parameters (?j - pos)
75          :model-for (glitter ?j)
76          :precondition (at ?j)
77          :such-that (gold-at ?j)
78      )
79
80      (:action start
81          :parameters (?j - pos)
82          :precondition (and (need-start) (alive) (at ?j))
83          :effect (not (need-start))
84      )
85
86      (:action move
87          :parameters (?i ?j - pos)
88          :precondition (and (adj ?i ?j) (at ?i) (alive)
89                             (not (need-start)) (not (wumpus-at ?j))
90                             (not (pit-at ?j)))
91          :effect (and (not (at ?i)) (at ?j)
92                       (when (wumpus-at ?j) (not (alive)))
93                       (when (pit-at ?j) (not (alive)))
94                  )
95      )
96
97      (:action grab
98          :parameters (?i - pos)
99          :precondition (and (at ?i) (alive) (gold-at ?i)
100                             (not (need-start)))
101         :effect (and (got-the-treasure) (not (gold-at ?i)))
102     )
103 )
```

List of Acronyms

AI Artificial Intelligence
ASP Answer Set Programming

FOND Fully Observable Non-Deterministic

ICAPS International Conference on Planning and Scheduling

PDDL Planning Domain Definition Language
POD Partially Observable Deterministic
POMDP Partially Observable Markov decision process

List of Figures

List of Tables

List of Algorithms

List of Listings

List of References

[1] A. Albore and H. Geffner, 'Acting in partially observable environments when achievement of the goal cannot be guaranteed', in *Proc. of ICAPS Workshop on Planning and Plan Execution for Real-World Systems*, 2009.

[2] A. Albore, H. Palacios and H. Geffner, 'A translation-based approach to contingent planning.', in *IJCAI*, vol. 9, 2009, pp. 1623–1628.

[3] V. Belle and H. J. Levesque, 'Foundations for generalized planning in unbounded stochastic domains', in *KR*, 2016, pp. 380–389.

[4] B. Bonet, G. Formica and M. Ponte, 'Completeness of online planners for partially observable deterministic tasks', in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 40–48.

[5] B. Bonet and H. Geffner, 'Planning with incomplete information as heuristic search in belief space', in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, 2000, pp. 52–61.

[6] B. Bonet, H. Geffner *et al.*, 'Planning under partial observability by classical replanning: Theory and experiments', in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Citeseer, vol. 22, 2011, p. 1936.

[7] B. Bonet and H. Geffner, 'Belief tracking for planning with sensing: Width, complexity and approximations', *Journal of Artificial Intelligence Research*, vol. 50, pp. 923–970, 2014.

[8] B. Bonet and H. Geffner, 'Flexible and scalable partially observable planning with linear translations', in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[9] B. Bonet and H. Geffner, 'Features, projections, and representation change for generalized planning', *arXiv preprint arXiv:1801.10055*, 2018.

[10] B. Bonet, H. Palacios and H. Geffner, 'Automatic derivation of memoryless policies and finite-state controllers using classical planners', in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 19, 2009, pp. 34–41.

[11] M. Chevalier-Boisvert, D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen and Y. Bengio, 'Babyai: A platform to study the sample efficiency of grounded language learning', in *International Conference on Learning Representations*, 2019.

[12] D. Drexler, G. Francès and J. Seipp, *DLPlan*, 2022. DOI: 10.5281/zenodo.5826139. [Online]. Available: https://doi.org/10.5281/zenodo.5826139.

[13] R. E. 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, ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/0004-3702(71)90010-5.

[14] G. Frances, B. Bonet and H. Geffner, 'Learning general planning policies from small examples without supervision', in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 11 801–11 808.

[15] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub and M. Schneider, 'Potassco: The potsdam answer set solving collection', *Ai Communications*, vol. 24, no. 2, pp. 107–124, 2011.

[16] H. Geffner, 'Non-classical planning with a classical planner: The power of transformations', in *Lecture Notes in Computer Science*, Springer, 2014, pp. 33–47. DOI: 10.1007/978-3-319-11558-0_3.

[17] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[18] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld and D. Wilkins, 'Pddl-the planning domain definition language', 1998.

[19] M. Ghallab, D. Nau and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[20] M. Ghallab, D. Nau and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016. DOI: 10.1017/cbo9781139583923.

[21] E. Groshev, M. Goldstein, A. Tamar, S. Srivastava and P. Abbeel, 'Learning generalized reactive policies using deep neural networks for planning', in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, 2018, pp. 408–412.

[22] P. Haslum and P. Jonsson, 'Some results on the complexity of planning with incomplete information', in *European Conference on Planning*, Springer, 1999, pp. 308–318.

[23] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise, R. Brachman, F. Rossi and P. Stone, *An introduction to the planning domain definition language*. Springer, 2019, vol. 13.

[24] T. Hofmann and H. Geffner, 'Learning generalized policies for fully observable non-deterministic planning domains', in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, K. Larson, Ed., Main Track, International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 6733–6742. DOI: 10.24963/ijcai.2024/744. [Online]. Available: https://doi.org/10.24963/ijcai.2024/744.

[25] L. Illanes and S. A. McIlraith, 'Generalized planning via abstraction: Arbitrary numbers of objects', in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 7610–7618.

[26] S. Jiménez, J. Segovia-Aguas and A. Jonsson, 'A review of generalized planning', *The Knowledge Engineering Review*, vol. 34, e5, 2019.

[27] V. Lifschitz, 'Answer sets and the language of answer set programming', *AI Magazine*, vol. 37, no. 3, pp. 7–12, 2016.

[28] C. M. Marco Favorito Francesco Fuggitti, *PDDL*, 2025. [Online]. Available: `https://github.com/AI-Planning/pddl`.

[29] D. M. McDermott, 'The 1998 ai planning systems competition', *AI magazine*, vol. 21, no. 2, pp. 35–35, 2000.

[30] C. Muise, V. Belle and S. McIlraith, 'Computing contingent plans via fully observable non-deterministic planning', *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, Jun. 2014. DOI: `10.1609/aaai.v28i1.9049`. [Online]. Available: `https://ojs.aaai.org/index.php/AAAI/article/view/9049`.

[31] H. Palacios and H. Geffner, 'Compiling uncertainty away in conformant planning problems with bounded width', in *Journal of Artificial Intelligence Research*, vol. 35, 2009, pp. 623–675.

[32] N. Peter and R. S. A. Intelligence, *A Modern Approach*. Pearson Education, USA, 2021.

[33] J. Rintanen, 'Complexity of planning with partial observability.', in *ICAPS*, vol. 4, 2004, pp. 345–354.

[34] O. Rivlin, T. Hazan and E. Karpas, 'Generalized planning with deep reinforcement learning', in *ICAPS 2020 Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning*, 2020.

[35] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd. Prentice-Hall, Englewood Cliffs, NJ, 2003.

[36] S. Srivastava, N. Immerman and S. Zilberstein, 'Learning generalized plans using abstract counting', in *AAAI*, vol. 8, 2008, pp. 991–997.

[37] S. Ståhlberg, B. Bonet and H. Geffner, 'Learning general policies with graph neural networks', in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 355–363.

[38] S. Ståhlberg, B. Bonet and H. Geffner, 'Learning generalized policies without supervision using gnns', in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 12 111–12 119.