Action Model Learning based on Regular Grammar Induction for AI Planning

Maxence Grand

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Apprentissage de modèle d'actions basé sur l'induction grammaticale régulière pour la planification en intelligence artificielle

Action Model Learning based on Regular Grammar Induction for AI Planning

Présentée par :

Maxence GRAND

Direction de thèse :

Damien PELLIER MAÎTRE DE CONFÉRENCE, Université Grenoble Alpes	Directeur de thèse
Humbert FIORINO MAÎTRE DE CONFÉRENCE, Université Grenoble Alpes	Co-encadrant de thèse

Rapporteurs :

Olivier BUFFET
CHARGÉ DE RECHERCHE, INRIA Nancy Grand Est
Dominique DUHAUT
PROFESSEUR DES UNIVERSITÉS, Université Bretagne Sud

Thèse soutenue publiquement le **14 décembre 2022**, devant le jury composé de :

Olivier BUFFET CHARGÉ DE RECHERCHE, INRIA Nancy Grand Est	Rapporteur
Dominique DUHAUT PROFESSEUR DES UNIVERSITÉS, Université Bretagne Sud	Rapporteur
Gérard BAILLY DIRECTEUR DE RECHERCHE, CNRS	Président
Massih-Reza AMINI PROFESSEUR DES UNIVERSITÉS, Université Grenoble Alpes	Examinateur

Invités :

Humbert Fiorino
MAÎTRE DE CONFÉRENCE, Université Grenoble Alpes
Damien PELLIER
MAÎTRE DE CONFÉRENCE, Université Grenoble Alpes

Résumé

Le domaine de l'intelligence artificielle vise à concevoir des agents autonomes capables de percevoir, d'apprendre et d'agir sans aucune intervention humaine pour accomplir des tâches complexes. Pour accomplir des tâches complexes, l'agent autonome doit planifier les meilleures actions possibles et les exécuter. Pour ce faire, l'agent autonome a besoin d'un modèle d'actions. Un modèle d'actions est une représentation sémantique des actions qu'il peut exécuter. Dans un modèle d'actions, une action est représentée à l'aide (1) d'un ensemble de pré-conditions: l'ensemble des conditions qui doivent être satisfaites pour que l'action puisse être exécutée et (2) d'un ensemble d'effets: l'ensemble des propriétés du monde qui vont être modifiées par l'exécution de l'action. La modélisation STRIPS est une méthode classique pour concevoir ces modèles d'actions. Cependant, les modèles d'actions STRIPS sont généralement trop restrictifs pour être utilisés dans des applications réelles. Il existe d'autres formes de modèles d'actions: les modèles d'actions temporels permettant de représenter des actions pouvant être exécutées en concurrence, les modèles d'actions HTN permettant de représenter les actions sous formes de tâches et de sous tâches, etc. Ces modèles sont moins restrictifs, mais moins les modèles sont restrictifs plus ils sont difficiles à spécifier. Dans cette thèse, nous nous intéressons aux méthodes facilitant l'acquisition de ces modèles d'actions basées sur les techniques d'apprentissage automatique.

Dans cette thèse, nous présentons AMLSI (Action Model Learning with State machine Interaction), une approche d'apprentissage de modèles d'actions basée sur l'induction grammaticale régulière. Dans un premier temps nous montrerons que l'approche AMLSI permet d'apprendre des modèles d'actions STRIPS. Nous montrerons les différentes propriétés de l'approche prouvant son efficacité: robustesse, convergence, requiert peu de données d'apprentissage, qualité des modèles appris. Dans un second temps, nous proposerons deux extensions pour l'apprentissage de modèles d'actions temporels et de modèles d'actions HTN.

Mots clés : Planification automatique, Apprentissage de modèles d'actions, Induction grammaticale régulière.

Abstract

The field of Artificial Intelligence aims to design and build autonomous agents able to perceive, learn and act without any human intervention to perform complex tasks. To perform complex tasks, the autonomous agent must plan the best possible actions and execute them. To do this, the autonomous agent needs an action model. An action model is a semantic representation of the actions it can execute. In an action model, an action is represented using (1) a precondition: the set of conditions that must be satisfied for the action to be executed and (2) the effects: the set of properties of the world that will be modified by the execution of the action. STRIPS planning is a classical method to design these action models. However, STRIPS action models are generally too restrictive to be used in real-world applications. There are other forms of action models: temporal action models allowing to represent actions that can be executed concurrently, HTN action models allowing to represent actions as tasks and subtasks, etc. These models are less restrictive, but the less restrictive the models are the more difficult they are to design. In this thesis, we are interested in approaches facilitating the acquisition of these action models based on machine learning techniques.

In this thesis, we present AMLSI (Action Model Learning with State machine Interaction), an approach for action model learning based on Regular Grammatical Induction. First, we show that the AMLSI approach allows to learn (STRIPS) action models. We will show the different properties of the approach proving its efficiency: robustness, convergence, require few learning data, quality of the learned models. In a second step, we propose two extensions for temporal action model learning and HTN action model learning.

Keywords: Automated Planning, Action Model Learning, Regular Grammar Induction.

Contents

Chapter 1

Introduction

Contents

1.1 Context

The field of Artificial Intelligence (AI) intends to design and build agents able to perceive, learn and act without any human intervention to perform complex tasks. As stated by Russell and Norvig (2021), an agent is an intelligent system that can decide what to do and then do it. An agent perceives its environment, plans the best possible actions corresponding to the task that it has to achieve, and executes it. To plan best actions, the agent must therefore be able to make decisions. There are several approaches allowing agents to make decisions. One of them is *automated planning* (AI planning) (Fikes and Nilsson, 1971; Ghallab et al., 2004). The main advantage of AI planning is its flexibility. Most of the approaches allowing an agent to make decisions are *domain-dependent*, i.e. there is an algorithm for a given task that the agent has to achieve. AI planning is *domain-independent*, i.e. there is an algorithm to achieve a set of tasks. This thesis focuses on this approach.

1.1.1 AI Planning

The objective of AI planning is the resolution of *planning problems*. The description of a planning problem is done declaratively: a planning problem specifies *what to do* rather than *how to do it*. The declaration of a planning problem is composed of: (1) the *initial state* of the environment (the objects to

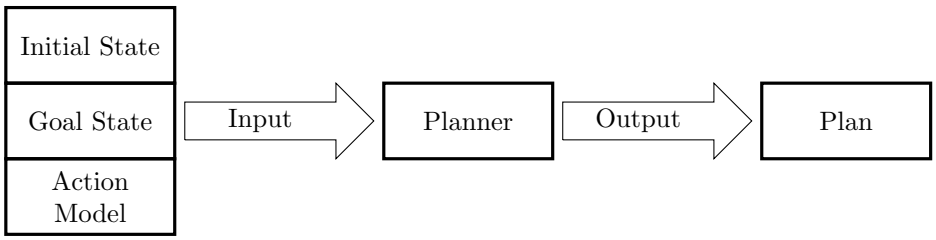


Figure 1.1: Planning Problem resolution Architecture.

be considered and their properties), (2) a *goal*, and (3) the action set that can be executed by the agent to achieve the specified goal. To solve the planning problem, the agent searches for an action sequence, called plan, achieving the goal state from the initial state. The resolution of planning problems is based on descriptive models of actions. They describe which state or set of possible states can result from the execution of an action. More precisely, *action models* are generally used to model actions. Action models define actions in terms of *preconditions* and *effects*. The preconditions express the properties of the environment that must be satisfied in order to apply the action, and the effects express the consequences of executing the action in the environment. In practice, action models are hand-encoded using a declarative language such as PDDL (Planning Domain Description Language) (Ghallab et al., 1998). Finally, Figure 1.1 gives the traditional architecture to solve a planning problem: a solver, called *Planner*, takes as input the initial state, the goal state and the action model, and returns the solution plan. This thesis focuses on action modeling using action models, and more precisely on the acquisition of these action models.

As we mentioned it, action models allow to represent actions in terms of preconditions and effects. A classical way to declare these preconditions and effects is the propositional logic: an action precondition is a set of logical propositions that must be satisfied in the current environment state and effects are sets of logical propositions whose values change after the execution of the action. Declaring preconditions and effects as sets of logical propositions asks that the environment states are modeled using logical propositions, then the initial state and the goal are also modeled as sets of logical propositions. This is the classical STRIPS (STanford Research Institute Problem Solver) (Fikes and Nilsson, 1971) planning formalism. Classical STRIPS planning is based on several assumptions. Among these assumptions, we can cite:

- The environment is deterministic and fully observable. The agent knows at any time the current state and can therefore predict the next state after executing an action.
- The goal is specified using several properties, i.e. logical propositions, that the final environment state reached by the agent has to satisfy.
- The environment is static. Only the actions executed by the agent changes the environment state. The agent is alone and the environment has no

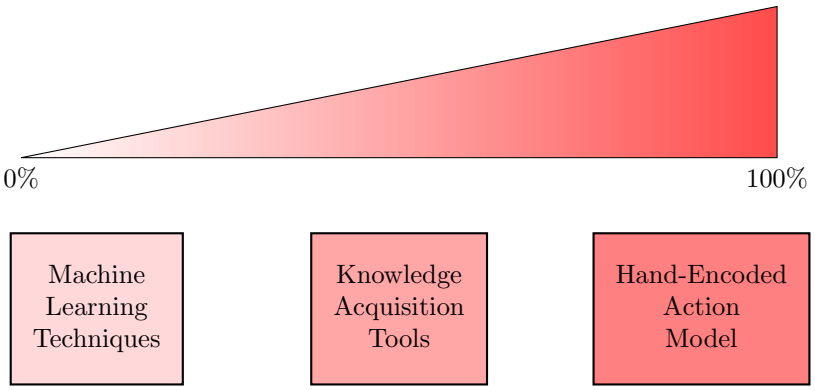


Figure 1.2: Human Effort for Action Model Acquisition.

internal dynamics.

- The execution of the actions is linear and atomic. Actions have no duration, changes in the environment are instantaneous and concurrence between actions is not taken into account.

- Preconditions and effects are sets of propositions.

Several extensions to STRIPS have been proposed to relax some assumptions. For example, temporal planning problems (Fox and Long, 2003) allow to model actions that can be concurrently executed. This can be useful for robots with multiple arms which can handle several objects concurrently. Another example is HTN planning problems (Erol et al., 1994) which declare tasks rather than actions. There are two kinds of tasks. Simple tasks similar to classical actions and complex tasks. Complex tasks are tasks that, when solving the planning problem, are decomposed into sub-tasks (simple and/or complex). Another example is ADL planning problems (Pednault, 1994) which allow to model more complex preconditions and effects with logical quantifier, conditional effects (effects applies when some conditions are satisfied) etc. A last example is multi-agent planning problems (Brenner, 2003) which allow to represent planning problems where several agents interact to achieve the goal. This can be useful if we have several robots interacting in order to achieve a goal.

1.2 Problem Statement

As we have seen before, AI planning requires the declaration of an action model. To declare an action model we use a declarative language such as PDDL. We have briefly presented how declare an action model based on the STRIPS formalism. As we have mentioned, STRIPS planning assumes several restrictive assumptions. On the other hand, AI planning tools have been developed in a wide range of real-world applications such as aerospace (Fisher et al., 2000; Backes et al., 2004; Bresina et al., 2005), autonomous vehicles (Urmson and

Whittaker, 2008), logistics (Cross and Walker, 1994), robotics (Dvorak et al., 2014; Lallement et al., 2018; Liang et al., 2022), industry (Hoffmann et al., 2009), cybersecurity (Edelkamp et al., 2009). Usually, action models used by these tools are hand-encoded. Also, the assumptions assumed by STRIPS are too restrictive for these tools. Modeling real-world applications therefore requires to relax these assumptions. However, if we relax these assumptions the action models will be more complex and therefore more difficult to hand-encode.

Hand-encoding these action models requires a planning expert. Nevertheless, hand-encoding action models remains a difficult task even for a planning expert and requires a lot of human effort (see Figure 1.2). Also, users are generally not planning experts, and planning experts in charge of modelling real-world applications generally have no knowledge of these applications. This a classic issue of software and requirements engineering (Sommerville, 2011) which increases the human effort required to hand-encode real world applications. To facilitate the diffusion of AI planning techniques, it is important to develop approaches that facilitate the acquisition of action models.

Knowledge engineering tools facilitating action model writing have been developed. These tools provide support for consistency and syntax error checking, domain visualisation etc. However, these tools require a lot of AI planning expertise and background in software engineering (Shah et al., 2013). Also, machine learning approaches have been proposed to automatically generate action models (Arora et al., 2018a; Celorrio et al., 2012; Jilani et al., 2014). Generally, machine learning approaches are used for already existing applications: either to automate processes, such as an industrial process for example, or to benefit from the flexibility of AI planning, to facilitate the maintenance of these applications for example. These approaches takes as input a training datasets and learn an action model. A training dataset is a collection of data allowing to learn a model. In the context of AI planning, the training datasets are generally examples of agent executions. These approaches are promising and some of them reduce human effort (Zhuo et al., 2010a). However, these approaches are not efficient enough to be used in real-world applications.

First of all, most of these learning approaches learn classical STRIPS action models, and as we have seen before, classical STRIPS action models are based on assumptions too restrictive to be used in real-world applications. Then, the acquisition of the learning datasets is a difficult and costly process. Indeed, we have to generate the examples, execute them, store the results of the execution. Moreover, some approaches also require to parse these examples to a symbolic representation. There are mainly two kind of datasets: (1) goal oriented execution traces and (2) random walks. Most of the learning approaches use goal oriented execution traces as training datasets (Yang et al., 2007; Aineto et al., 2019; Segura-Muros et al., 2018; Kucera and Barták, 2018) and few approaches use randomly generated execution traces (Rodrigues et al., 2010a; Mourão et al., 2012). Goal oriented execution traces are execution traces achieving a given task. For example, for an autonomous vehicle, a goal oriented execution trace

will be the action sequence allowing to the vehicle to travel from Grenoble to Lyon. The main problem with this kind of execution traces is that they are biased by the task to achieve, and if we have too little data there is a risk of overfitting, i.e. the learned action models are not sufficiently general and several tasks cannot be achieved by these action models. For example, for our autonomous vehicle, if our training datasets contains only travels where the two cities are connected by a highway, it is possible that the learned action model is not able to be used for travels connecting two cities that are not connected by a highway. To limit the risk of overfitting, we therefore need a large number of execution traces acquired from a wide variety of tasks. However, as we mentioned earlier, acquiring these traces is difficult and costly. Random walks are randomly generated action sequences. The advantage of this kind of trace is that it is not biased by a task. By randomly generating sequences, we can cover a large number of execution examples while limiting the risk of overfitting. Also, random walks allow to acquire unfeasible action sequences, i.e. action sequences where one or several actions are not feasible. An action is not feasible if one or several preconditions are not satisfied. Moreover, usually in addition to the action sequences, the execution traces contains observations: i.e. the different states of the environment observed during the execution of the action sequences. These states are sets of logical propositions describing the environment. In practice, obtaining complete and noiseless observations is a difficult task and generally the observations will be partial and noisy. A partial observed state is a state where the values of some propositions are unknown. Also, a noisy observed state is a state where the values of some propositions are erroneous. However, majority of learning approaches are only able to deal with complete or partial and noiseless observations (Wang, 1995; Yang et al., 2007; Aineto et al., 2019). Also, even if some approaches (Mourão et al., 2012; Segura-Muros et al., 2018; Rodrigues et al., 2010a) are able to learn from partial and noisy observations, few approaches are able to handle very high levels of noise and high levels of partial observations as can be encountered in real world applications. Finally, learning approaches are generally not able to learn action models which are usable by planners. A proofreading step by an AI planning expert is generally required.

In this thesis we argue that, to be efficient, a learning approach must tackle the following triple issue:

1. **Output:** As mentioned earlier, STRIPS planning is too restrictive for real-world applications. We therefore have to propose approaches learning less restrictive action models.

2. **Input:** The acquisition of the training dataset must be as simple as possible and requires the least amount of human effort.

3. **Performance:** The learned action model has to be *accurate*. An action model is accurate if the action model can be used by planners to solve planning problems without requiring a proofreading step. Also, even with

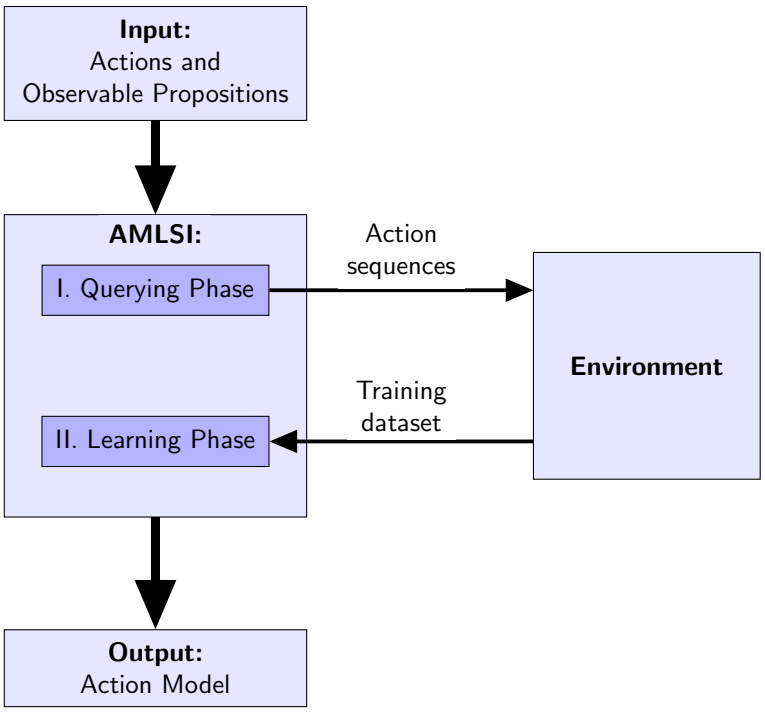


Figure 1.3: Overview of the AMLSI Approach.

inputs requiring not a lot of human effort, acquiring the training dataset is difficult and costly. The learning approach has to require few data in the training dataset to learn action model. Finally, the learning approach has to be robust to partial and noisy observations, i.e. the learning approach has to learn accurate action models even with high level of partiality and noise in the observations.

In this thesis we tackle the following research challenge: *How to automatically learn accurate action models from partial and noisy observations?*

1.3 The AMLSI Approach

This thesis contributes to the field of AI planning, and more specifically to the field of action model acquisition.

In this thesis we propose AMLSI (Action Model Learning with State machine Interaction), a learning approach for action model acquisition. Figure 1.3 gives an overview of this approach. The key idea of the AMLSI approach is to interact with the environment in which the agent will have to solve planning problems to learn the action model: AMSLI tests different actions, observes how the environment evolves when these actions are executed and learns the action model from its observations. This approach is divided into two phases:

I. Querying Phase: AMLSI queries the environment with feasible and unfeasible action sequences and perceives observations. These observation can be partial and noisy. AMLSI tests both feasible and unfeasible action sequences because unfeasible action sequences allow to minimize the amount of (feasible) actions to be executed in the environment and thus minimize the cost of the training dataset acquisition. Indeed, by reducing the number of feasible actions, we reduce the number of actions to execute and therefore the number of observations to acquire and store. Also, the action sequences will be generated randomly to avoid the overfitting issue. This phase takes as input the set of actions that the agent can execute and the set of observable propositions describing the environment. This querying phase will allow AMLSI to build a training dataset that will then be used as input of the learning phase.

II. Learning Phase: It is during this phase that AMLSI learns the action model. To learn the action model, AMLSI will rely on Regular Grammar Induction. As we will see in Chapter 4, planning problems are related to state machines and these state machines are equivalent to regular grammars. Moreover, as we will see in Chapter 3, Regular Grammar Induction is a well-defined problem (Gold, 1967; De La Higuera, 2010), and many algorithms has been proposed to solve it like RPNI (Oncina and Garcia, 1992).

The main contribution of the AMLSI approach is to tackle the triple issue presented before. First of all, we will show that AMLSI approach is able to learn STRIPS action models from partial and noisy observations sufficiently accurate to allow planners to solve new planning problems. We will see that using random walks with unfeasible sequences allows to learn action models with few data. Also, we will show that AMLSI outperforms state-of-the-art approaches. Also, as mentioned above, training dataset acquisition is costly. Also, in practice, the learning data arrives online. Our approach must therefore be able to learn action models incrementally: each time new data is acquired AMLSI will update the action model without starting the learning process from scratch. Then, as STRIPS action models are too restrictive to be applied to real-world applications we will extend AMLSI to learn temporal action models and HTN action models.

1.4 Document Organization

This thesis is organised into two parts (see Figure 1.4)).

The first part consists of theoretical background on AI planning and present a state-of-the-art of action model acquisition techniques. First of all, we will present some background on the AI planning field. Then, we will present a state-of-the-art of action model learning approaches. Our approach being based on regular grammar induction, we will introduce this field. We organised this part into two chapters. In Chapter 2, we introduce the AI planning concepts used in the development of this thesis and the literature related to these concepts. First of all, we will define the classical concepts of AI planning. Next, we

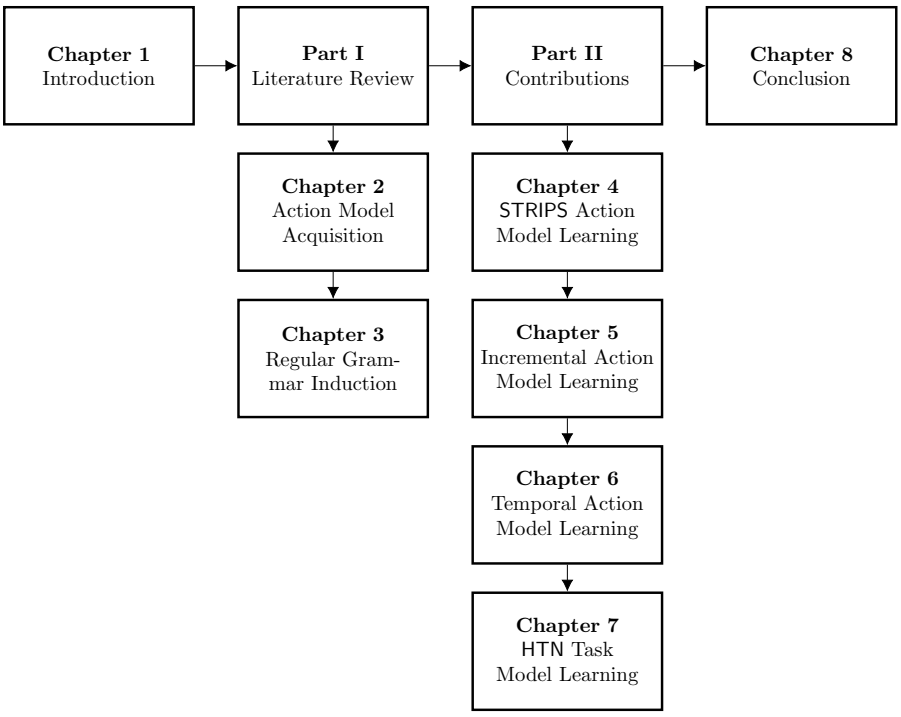


Figure 1.4: Document Organization.

will present three AI planning frameworks: (1) classical STRIPS planning and the PDDL language, (2) hierarchical planning and the HDDL language and finally (3) temporal planning and the PDDL 2.1 language. Finally, we present the state-of-the-art of the action model acquisition approaches. We will first briefly discuss of the knowledge acquisition tools. Then, as this thesis focuses on learning approaches, we will detail the approaches based on machine learning. We will discuss their advantages, disadvantages and open issues. In Chapter 3, we introduce the regular grammar induction concepts used in this thesis. We will start by giving some definitions and concepts of theory of languages and automata. Then we will introduce the regular grammar induction field and present some learning algorithms.

The second part of this thesis presents our contributions, taking into account the research challenge previously presented. We organise this part into four chapters. In Chapter 4 we first show that AMLSI is an accurate STRIPS action model learning algorithm robust to partial and noisy observations. Then, we will show that AMSLI outperforms state-of-the-art approaches. Also, we will present formal properties of AMLSI approach. Then, in Chapter 5 we will present IncrAMLSI, an incremental extension of the AMLSI approach for STRIPS action model learning. After presenting the STRIPS action model learning approach, we will present approaches for less restrictive action models. In Chapter 6 we present TempAMLSI, our temporal extension of AMLSI and show that it is possible to learn temporal features with non-temporal learning techniques. In Chapter 7 we present HierAMLSI, our HTN extension of AMLSI.

Finally, we conclude this thesis in Chapter 8. More precisely, we review the contributions of the thesis, we give concluding remarks and we propose possible perspectives for this work.

Part I

Literature Review

Chapter 2

Action Model Acquisition

Contents

2.1 Introduction

The aim of the AI planning (Fikes and Nilsson, 1971; Ghallab et al., 2004) is to develop solvers, called planners, able to generate action sequences at a symbolic level from an initial state in order to achieve a defined goal. The specification of the problem to be solved is generally based on a high-level language such

as PDDL (Ghallab et al., 1998) (Planning Domain Description Language). The description of a planning problem is done in a declarative way. The specification of a planning problem, whatever the language used, requires the description of three elements: (1) the initial state of the environment (the objects to be considered and their properties), (2) a goal, and (3) the set of actions that can be performed to achieve the goal. Actions are defined in terms of preconditions and effects. The preconditions express the properties of the environment that must be verified in order to apply the action, and the effects express the consequences of executing the action in the environment. The solution of a planning problem is in the general case an ordered sequence of actions, called a plan, which can be executed from the initial state of the problem and which produces a state of the environment entailing the goal.

Classical AI planning is based on several assumptions. Among these assumptions, we can cite:

- The environment is deterministic and fully observable. The agent knows at any time the current state and can therefore predict the next state after executing an action.
- The goal is specified using several properties, i.e. logical propositions, that the environment has to satisfy.
- The environment is static. Only the actions executed by the agent alter the state of the environment. The agent is alone and the environment has no internal dynamics.
- The execution of the actions is linear and atomic. Actions have no duration, changes in the environment are instantaneous and concurrence between actions is not taken into account.
- Preconditions and effects are sets of propositions.

These assumptions are too restrictive to model real-world applications. However, relaxing some of these assumptions has two drawbacks: (1) the models are more difficult to hand-encode and (2) the computational complexity of solving these problems is higher. This thesis focuses on the first drawback.

This chapter is divided into two parts. First, we will see how to declare the action models, we will start with classical AI planning and then we will present several formalism relaxing some assumptions. Then, we will present the methods used to facilitate the acquisition of these action models.

2.2 Action Model Declaration

As mentioned in the introduction, the aim of AI planning is the development of planner solving planning problems. Formally, the solution of a planning problem is expressed as the search for a path representing a plan of actions

in a finite state machine starting from an initial state and choosing which actions should be applied to achieve a given task. The task can be formalized as a goal state, represented by a set of logical propositions, a utility function associated with the states, to be minimized or maximized, or a complex task to be decomposed.

Definition 2.1 *A finite state machine is a tuple* (S, A, γ, s_o, G) *such that4:*

- S *is a set of states.*
- A *is a set of actions.*
- $\gamma : S \times A \rightarrow S$ *is the transition function.*
- s_0 *is the initial state.*
- G *is a set of goal states.*

An action $a \in A$ is applicable in a state s if and only if the transition function $\gamma(s, a)$ is defined. When we apply a in a state s, the state $s' = \gamma(s, a)$ is produced. Finally, a plan is an action sequence $\omega = \langle a_1, \ldots, a_n \rangle$ of size n. The state produced by the application of ω in a state s is the state obtained by the sequential application of each action of the plan ω. Formally, $\gamma(s, \omega)$ is defined as follows:

$$\gamma(s, \omega) = \begin{cases} s & \text{If } n = 0. \\ \gamma(s, a_1) & \text{If } n = 1. \\ \gamma(\gamma(sa_1), \langle a_2 \ldots a_n \rangle) & \text{If } n > 1 \text{ and } a_1 \text{ is applicable in } s. \\ \text{Undefined} & \text{Otherwise.} \end{cases} \tag{2.1}$$

In the rest of this section, we start by presenting the STRIPS formalism (Fikes and Nilsson, 1971) based on the assumptions mentioned above. We present a formal framework and then introduce the PDDL language (Ghallab et al., 1998) declaring STRIPS action models in the form of planning domains. Then, we present two formalisms relaxing some of the assumptions: temporal planning and hierarchical planning. As for STRIPS planning, we first present the formal framework and then we present the language declaring these action models.

We illustrate this chapter using the classical AI planning example Blocksworld. In this example an agent handles blocks: it can pick them up, drop them on the table and stack them. The objective of the agent is to move the blocks in order to reach a given arrangement. The agent perceives its environment to know how the blocks are arranged and has to plan a certain number of actions to achieve its goal. Figure 2.1 shows an example of planning problem for Blocksworld. The initial state is the initial block arrangement: the green block, the blue block and the red block are on the table. The goal is the block arrangement that the agent has to reach: the red block on the green block, the

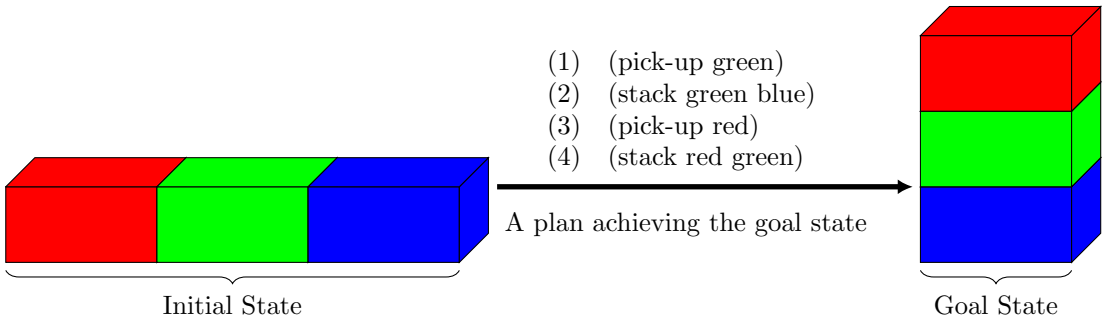


Figure 2.1: A first example of planning problem.

green block on the blue block and the blue block on the table. A plan solving this problem is:

$$\langle (\textit{pick-up green}), (\textit{stack green blue}), (\textit{pick-up red}), (\textit{stack red green}) \rangle$$

In natural language this plan means that, to solve its planning problem, the agent begins by picking the green block (the *(pick-up green)* action) and stacks it on the blue block (the *(stack green blue)* action). Then, the agent picks the red block (the *(pick-up red)* action) and finally stacks it on the green block (the *(stack red green)* action).

2.2.1 Classical Planning

2.2.1.1 Formal Framework

We will give a formal description of classical AI planning, and more precisely, of STRIPS planning. Also, we chose a formal framework inspired by (Höller et al., 2016) in order to define the STRIPS formalism as a propositional language representing the state transition system.

Definition 2.2 *A classical planning problem P is a tuple $(L, A, S, s_0, G, \delta, \tau)$ where:*

- *L is the set of logical propositions describing the environment.*
- *A is the set of action.*
- *$S \in 2^L$ is the set of state.*
- *$s_0 \in S$ is the initial state.*
- *G is the set of goal states.*
- *δ is the action model.*
- *$\tau : S \times A \to \{true, false\}$ is the feasibility function.*

Example 2.1 *Consider the classical Blocksworld example. L is composed of the following propositions:*

- *(handempty): the agent's hand is empty.*
- *(holding red), (holding blue), (holding green): the agent holds the red (resp. blue, green) block.*
- *(ontable red), (ontable blue), (ontable green): the red (resp. blue green) block is on the table.*
- *(clear red), (clear blue), (clear green): the red (resp. blue green) block is clear. A block is clear if there is no block on it and if it is not held by the agent.*
- *(on red green), (on green blue): the red (resp. green) block is on the green (resp. blue) block.*

A possible initial state s_0 *could be* $s_0 = \{$*(ontable red), (ontable blue), (ontable green), (clear red), (clear blue), (clear green), (handempty)*$\}$*. A is composed of the following actions:*

- *(pick-up red), (pick-up blue), (pick-up green): the agent can pick the red (resp. blue, green) block from the table.*
- *(put-down red), (put-down blue), (put-down green): the agent can drop the red (resp. blue, green) block on the table.*
- *(stack red green), (stack green blue): the agent can stack the red (resp. green) block on the green (resp. blue) block.*
- *(unstack red green), (unstack green blue): the agent can unstack the red (resp. green) block from the green (resp. blue) block.*

Actions can only be executed under certain conditions. These conditions are called action *preconditions* and are described using logical propositions.

Example 2.2 *The preconditions of the action (stack green blue) are {(holding green), (clear blue)}. In natural language, this precondition means that the agent can execute the action (stack green blue) if the agent holds the green block and if the blue block is clear.*

Then, once the agent executes an action, the *state* of the environment is altered. These alterations are due to the action *effects*. As preconditions, action effects can be described using logical propositions. Action effects are divided into two subsets:

- *Positive Effects*: the set of logical propositions which are present in the state after the execution of an action.
- *Negative Effects*: the set of logical propositions which are absent in the state after the execution of an action.

Example 2.3 *The positive effects of the action (stack green blue) are {(clear green), (on green blue), (handempty)}. In natural language, these positive effects mean that the green block is now clear and on the blue block and the agent's hand is now empty. The negative effects of the action (stack green blue) are {(clear blue), (holding blue)}. In natural language, these negative effects mean that the blue block is no longer clear and the agent no longer holds the green block. Actions preconditions and effects are usually encoded using an* action model.

Formally, the action model is defined as follows:

Definition 2.3 *An action model is a tuple $\delta = (prec, add, del)$ where:*

- $prec : A \rightarrow 2^L$ *is the function mapping a set of preconditions with* $a \in A$.
- $add : A \rightarrow 2^L$ *is the function mapping a set of positive effects with* $a \in A$.
- $del : A \rightarrow 2^L$ *is the function mapping a set of negative effects with* $a \in A$.

The function $\tau : S \times A \rightarrow \{true, false\}$ returns whether an action is applicable to a state, i.e. $\tau(s, a) \Leftrightarrow prec(a) \subseteq s$. Whenever an action a is applicable in state s_i, the state transition function $\gamma : S \times A \rightarrow S$ returns the resulting state $s_{i+1} = \gamma(s_i, a)$ such that

$$s_{i+1} = \{s_i \cup add(a)\} \setminus del(a). \tag{2.2}$$

Example 2.4 *If the agent applies the action (pick-up g) in s_0, then*

$$\begin{aligned} s_1 &= \{s_0 \cup add(\textit{pick-up green})\} \setminus del(\textit{pick-up green}) \\ &= \{(\textit{ontable red}), (\textit{ontable blue}), (\textit{clear red}), (\textit{clear blue}), (\textit{holding green})\}. \end{aligned}$$

An action sequence $\omega = \langle a_1, \ldots, a_n \rangle$ of actions is applicable to a state s_0 when each action a_i with $1 \leq i \leq n$ is applicable to the state s_{i-1}. Given an applicable sequence $\langle a_1, \ldots, a_n \rangle$ in state s_0, $\gamma(s_0, \langle a_1, \ldots, a_n \rangle) = \gamma(\gamma(s_0, a_1), \langle a_1, \ldots, a_n \rangle) = s_n$. It is important to note that this recursive definition of γ entails the generation of a sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$.

A goal state is a state g such that $g \in G$. An action sequence is a solution plan to a planning problem P if and only if it is applicable to s_0 and achieves a goal state g.

Example 2.5 $g = \{(\textit{ontable blue}), (\textit{clear red}), (\textit{on red green}), (\textit{on green blue}), (\textit{handempty})\}$ *is a goal state and the action sequence*

$$\langle (\textit{pick-up green}), (\textit{stack green blue}), (\textit{pick-up red}), (\textit{stack red green}) \rangle$$

is applicable in s_0, produces the state sequence :

$$\begin{aligned} \langle \; & \{(\textit{ontable red}), (\textit{ontable blue}), (\textit{clear red}), (\textit{clear blue}), (\textit{holding green})\}, \\ & \{(\textit{ontable red}), (\textit{ontable blue}), (\textit{clear red}), (\textit{clear green}), (\textit{on green blue}), (\textit{handempty})\}, \\ & \{(\textit{ontable blue}), (\textit{clear green}), (\textit{on green blue}), (\textit{holding red})\}, \\ & \{(\textit{ontable blue}), (\textit{clear red}), (\textit{on red green}), (\textit{on green blue}), (\textit{handempty})\} \quad \rangle \end{aligned}$$

and achieves the goal state g.

2.2.1.2 The PDDL Language

First proposed in 1998 (Ghallab et al., 1998), the PDDL language is mainly inspired by the STRIPS language (Fikes and Nilsson, 1971) and the ADL language (Pednault, 1994). It expresses planning problems in two files: (1) the planning domain file describing the planning operators and (2) the planning problem file describing the initial state and the goal. A planning operator is an abstraction of an action. Indeed, as we have seen previously, actions are declared using their preconditions and their effects. The preconditions and the effects being a set of logical propositions. Nevertheless, the PDDL language is based on predicate logic. The planning operators are therefore declared using predicates, and these operators will then be instantiated into actions using the objects present in the initial state declared in the problem file. We will see how to define a planning domain and a planning problem using the PDDL language through the Blocksworld example.

PDDL Planning Domain File We will now see how to specify a planning domain using the PDDL language. To start with, let's look at the file header.

```
1 (define (domain blocksworld)
2   (:requirements :strips :typing)
3   (:types block)
```

The domain description always begins with the declaration of its name preceded by the **domain** keyword. Then, domain descriptions define its requirements identified by the keyword **:requirements** and the list of objects types identified by the keyword **:types**.

Then, the domain declaration contains the declaration of all the predicates representing the environment.

```
1 (:predicates
2         (on ?x - block ?y - block)
3         (ontable ?x - block)
4         (clear ?x - block)
5         (handempty)
6         (holding ?x - block)
7 )
```

Finally, the domain declaration contains the declaration of all planning operators. When solving the planning problem, these planning operators will be instantiated in order to obtain the action model. Planning operators are defined as follows:

```
1 (:action pick-up
2  :parameters (?x - block)
3  :precondition (and
4      (clear ?x) (ontable ?x) (handempty))
5  :effect (and
```

```
6       (not (ontable ?x))
7       (not (clear ?x))
8       (not (handempty))
9       (holding ?x))
10  )
```

The declaration of a planning operator contains the name of the operator preceded by the keyword **:action**, the set of parameters with types preceded by the keyword **:parameters** and the precondition and the effects preceded by keywords **:precondition** and **:effect**. In our example, the precondition means that to pick up a block, the block has to be on the table and clear and the agent's hand has to be empty. Then, the effects mean that after having picked the block, the block is no longer clear and on the table, and the agent now holds the block and its hand is no longer empty.

PDDL Planning Problem File Let us now see how to describe a planning problem. For the Blocksworld example we have:

```
1  (define (problem example)
2      (:domain blocksworld)
3      (:objects red green blue - block)
4  (:init
5      (clear red) (clear green) (clear blue)
6      (ontable red) (ontable green) (ontable blue)
7      (handempty))
8  (:goal
9      (on red green) (on green blue)))
```

The example problem starts by listing the different objects of the problem (section identified by the keyword **:objects**) and the initial state (section identified by the keyword **:init**). In this example, the problem has three blocks red, green and blue put on the table. Finally the goal to reach is in the section identified by the keyword **:goal**. In this example, the goal is to reach a state where red is on green and green is on blue.

2.2.2 Temporal Planning

Now suppose that the agent has several hands: a right hand and a left hand. And suppose that it can use both hands simultaneously. Then, the agent is able to produce concurrent plans where several actions are executed simultaneously. Moreover, suppose that each action has a given duration, then Figure 2.2 gives a concurrent plan to solve the Blocksworld problem. Temporal planning allows to model such planning problems where actions can be performed concurrently. More precisely, temporal planning problems are problems allowing to represent *durative actions*, i.e. actions that have a duration, and whose preconditions and effects must be satisfied and applied at different times.

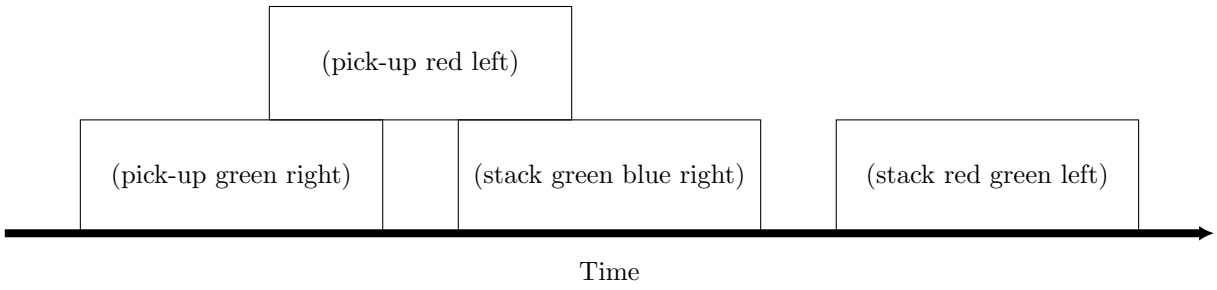


Figure 2.2: A concurrent plan to solve the Blocksworld problem.

2.2.2.1 Formal Framework

We will extends the classical formalization seen in Section 2.2.1.1 to fit with temporal features.

Definition 2.4 *A temporal planning problem P is a tuple $(L, A, S, d, s_0, g, \delta, \tau)$ where:*

- *L is the set of logical propositions describing the environment.*
- *A is the set of durative actions.*
- *$S \in 2^L$ is the set of states.*
- *$d : A \to \mathbb{R}$ is the duration function*
- *$s_0 \in S$ is the initial state.*
- *G is the set of goal states.*
- *δ is the temporal action model.*
- *$\tau : S \times A \to \{true, false\}$ is the feasibility function.*

As for STRIPS problems, L is a set of logical propositions, S is a set of states, $s_0 \in S$ is the initial state, G is the set of goal states. A is a set of *durative actions* and $d : A \to \mathbb{R}$ is the duration function. Unlike STRIPS planning problem, action preconditions, positive and negative effects are labeled with time labels *at-start*, *at-end* and *overall*. Formally, the temporal action model is defined as follows:

Definition 2.5 *A temporal action model is a tuple $\delta = (prec, add, del)$ where:*

- *$prec : A \times \{s, o, e\} \to 2^L$ is the function mapping the set of at start (resp. overall, at end) preconditions with $a \in A$.*
- *$add : A \times \{s, e\} \to 2^L$ is the function mapping the set of at start (resp. at end) positive effects with $a \in A$.*
- *$del : A \times \{s, e\} \to 2^L$ is the function mapping the set of at start (resp. at end) negative effects with $a \in A$.*

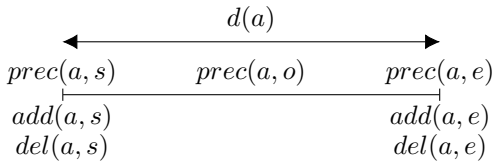


Figure 2.3: Structure of a durative action *a*.

The at-start label means that the precondition (resp. effect) has to be satisfied (resp. applied) when the durative action starts. Then, the at-end label means that the precondition (resp. effect) has to be satisfied (resp. applied) when the durative action ends. Finally, the overall label means that the precondition has to be satisfied during the whole execution of the durative action.

Example 2.6 *For the action (stack green blue right) we have:*

- *Preconditions:*
  - *prec((stack green blue right),o) = {(holding green right)}.*
  - *prec((stack green blue right),s) = ∅.*
  - *prec((stack green blue right),e) = {(clear blue)}.*
- *Positive effects:*
  - *add((stack green blue right),e) = {(on green blue), (clear green)}.*
  - *add((stack green blue right),s) = ∅*
- *Negative effects:*
  - *del((stack green blue right),e) = {(clear blue), (holding green right)}.*
  - *del((stack green blue right),s) = ∅*

The agent can execute the durative action (stack green blue right) if the agent holds the green block with its right hand during the whole execution and if the blue block is clear at the end of the execution. At the end of the execution the green block is now clear and on the blue block and the agent no longer holds the green block and the blue block is no longer clear.

The semantics of durative actions is defined in terms of two discrete events, start-a and end-a, each of which is naturally expressed as a STRIPS action. Starting a durative action *a* in state *s* is equivalent to applying the STRIPS action start-a in *s*, first verifying that prec(start-a) holds in *s*. Ending *a* in state s' is equivalent to applying end-a in s', first by verifying that prec(end-a) holds in s'. start-a and end-a are defined as follows:

- start-a:
  - prec(start-a) = prec(a,s).

- add(start-a) = add(a, s).
- del(start-a) = del(a, s).

- end-a:
  - prec(end-a) = prec(a,e).
  - add(end-a) = add(e, s).
  - del(end-a) = del(a, e).

Example 2.7 *For the durative action (stack blue green right) we have:*
add(end-stack green blue right) = add((stack green blue right),e) = { (on green blue), (clear green)}

Actions start-a and end-a are constrained by the duration of a, denoted $d(a)$ and the overall precondition: end-a has to occur exactly $d(a)$ time units after start-a, and the over all precondition has to hold in all states between start-a and end-a. Although a has a duration, its effects apply instantaneously at the start and the end of a, respectively. The preconditions $prec(a, s)$ and $prec(a, e)$ are also checked instantaneously, but $prec(a, o)$ has to hold for the entire duration of a. The structure of a durative action is summarized in Figure 2.3.

A *temporal action sequence* ω is a sequence of action-time pairs:

$$\omega = \langle (a_1, t_1), \ldots, (a_n, t_n) \rangle$$

Each action-time pair (a, t) is composed of a durative action $a \in A$ and a scheduled start timestamp $t \in \mathbb{R}$ of a, and induces two events start-a and end-a with associated timestamps t and $t + d(a)$, respectively. Events start-a (resp. end-a) is applied in the state s_t (resp. $s_{t+d(a)}$), s_t (resp. $s_{t+d(a)}$) being a state time-stamped with t (resp. $t + d(a)$). Then, the temporal transition function γ can be rewritten as: $\gamma(s, a, t) = (\gamma(s_t, \textit{start-a}), \gamma(s_{t+d(a)}, \textit{end-a}))$. The transition function $\gamma(s, a, t)$ is defined if and only if: $prec(a, s) \subseteq s_t$, $prec(a, e) \subseteq s_{t+d(a)}$ and $\forall t'$ such that $t \leq t' \leq t + d(a)$ $prec(a, o) \subseteq s_{t'}$.

Example 2.8 *Let's suppose that* $d(\textit{stack green blue right}) = 1$ *and suppose that the agent execute this durative action at* $t = 1$*, then the agent can execute (stack green blue right) iff:*

1. $prec(\textit{(stack green blue right)}, s) \subseteq s_1$.
2. $prec(\textit{(stack green blue right)}, e) \subseteq s_2$.
3. $\forall 1 \leq t' \leq 2, prec(\textit{(stack green blue right)}, o) \subseteq s_{t'}$.

2.2.2.2 The PDDL 2.1 Language

Proposed by (Fox and Long, 2003), the PDDL 2.1 language is an extension of the PDDL language modeling temporal features. As for PDDL, it expresses planning problems in two files: (1) the planning domain file describing the operators and (2) the planning problem file describing the initial state and the goal. We will see how to define a planning domain and a planning problem using the PDDL 2.1 language through the Blocksworld example.

PDDL 2.1 Planning Domain File As for the PDDL, to declare a planning domain we have to declare the name of the domain, the requirements, the types, and predicates:

```
(define (domain blocksworld_temporal)
    (:requirements :durative-actions :typing)
    (:types block hand)
(:predicates
    (on ?x - block ?y - block)
    (ontable ?x - block)
    (clear ?x - block)
    (handempty ?h - hand)
    (holding ?x - block ?h - hand)
 )
```

Finally, the domain declaration contains the declaration of all planning operators:

```
(:durative-action stack
 :parameters (?x - block ?y - block ?h - hand)
 :duration (= ?duration 1)
 :condition (and
    (overall (holding ?x ?h))
    (at end (clear ?y)))
 :effect (and
    (at end (not (holding ?x ?h)))
    (at end (not (clear ?y)))
    (at end (clear ?x))
    (at end (handempty ?a))
    (at end (on ?x ?y))))
```

The declaration of a planning operator contains the name of the operator preceded by the keyword **:durative-action**, the duration preceded by the keyword **duration** the set of parameters with types preceded by the keyword **:parameters** and the precondition and the effects preceded by keywords **:condition** and **:effect**. In our example, the condition means that to stack a block x on a block y using the hand h, the agent has to hold the block x with its hand h during the whole execution and the block y has to be clear at the end of the

execution. At the end of the execution the block x is now clear and on the block y and the agent no longer holds the block x and the block y is no longer clear.

PDDL 2.1 Planning Problem File Planning problems are declared in the same way as for the PDDL language:

```
(define (problem example_temporal)
    (:domain blocksworld)
    (:objects
        red green blue - block
        left right - hand)
(:init
    (clear red) (clear green) (clear blue)
    (ontable red) (ontable green) (ontable blue)
    (handempty right) (handempty left))
(:goal
    (on red green) (on green blue))
```

2.2.3 Hierarchical Planning

Until now, the goal of planning problems was declared using a set of propositions. For example, for Blocksworld we had: *{(on red green), (on green blue)}*. In addition, the agent could have a task to achieve. Unlike a goal defined by a set of propositions, a task is defined by its name and its parameters. For example, suppose there is a task "put the red block on the green block", then the agent should find an action sequence to solve this task. One way to do this would be to decompose this task into subtasks. It is HTN (*Hierarchical Task Network*) planning that allows to declare these tasks and decompositions.

The HTN formalism is very expressive and used to express a wide variety of planning problems. This formalism allows planners to exploit domain knowledge to solve problems more efficiently (Nau et al., 2005) when planning problems can be naturally decomposed hierarchically in terms of tasks and task decompositions. In contrast to the classical STRIPS problem in which only the action model needs to be specified, HTN problems require to specify the task model. A task model can be primitive or compound. Primitive tasks are described by classical actions. Compound tasks are described by HTN methods. An HTN method describes the set of primitive and/or compound task required to decompose a specific compound task.

2.2.3.1 Formal Framework

We will extends the classical formalization seen in Section 2.2.1.1 to fit with HTN features. This extension is based on the notation of (Höller, 2021). Finally, we consider only Totally Ordered task models.

Definition 2.6 *An* HTN *planning problem* P *is a tuple* $(L, C, A, S, M, s_0, \omega_I, G, \delta, \sigma, \zeta)$ *where:*

- L *is the set of logical propositions describing the environment.*
- $S \in 2^L$ *is the set of states.*
- C *is the set of compound tasks.*
- A *is the set of actions (or primitive tasks).*
- M *is the set of* HTN *methods.*
- $s_0 \in S$ *is the initial state.*
- $\omega_I \in \{A \cup C\}^*$ *is the initial task network.*
- G *is the set of goal states.*
- δ *is the task model.*
- $\sigma : M \rightarrow C \times \{A \cup C\}^*$ *is the method decomposition function*[1].
- $\zeta : \{A \cup C\}^* \times S \rightarrow \{A \cup C\}^*$ *is the decomposition function.*

As for STRIPS problems, L is a set of logical propositions describing the environment states, S is a set of states, $s_0 \in S$ is the initial state, $G \subseteq S$ is the set of goal states, and preconditions, positive and negative effects are given by the functions *prec*, *add* and *del* included in δ. A is the set of actions (or primitive tasks) and C is a set of compound (or non primitive) tasks, with $C \cap A = \emptyset$.

Example 2.9 *For Blocksworld, A is composed of the following primitive tasks: {(pick-up red),(pick-up green),(pick-up blue),(stack red green),...} and C is composed of the following compound tasks:*

- *(do-clear red), (do-clear blue), (do-clear green): the agent has to clear the red (resp. blue green) block.*
- *(do-on-table red), (do-on-table blue), (do-on-table green): the agent has to put on the table the red (resp. blue green) block.*
- ...

Tasks are maintained in *task networks*. A task network is a sequence of tasks. A task network is an element out of $\{A \cup C\}^*$. Compound tasks can be decomposed by methods. The set M contains all method labels. Methods are defined by the function $\sigma : M \rightarrow C \times \{A \cup C\}^*$. A method $m \in M$ is relevant for a task $c \in C$ if the method m allows to decompose the task c.

[1] * is the Kleene operator

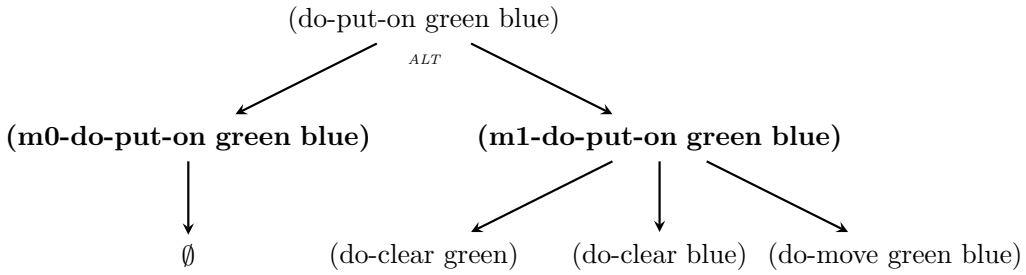


Figure 2.4: An example of HTN method decomposition.

Example 2.10 *As example, Figure 2.4 gives the methods decomposition of the task (do-put-on green blue). There are two relevant methods: (m0-do-put-on green blue) and (m1-do-put-on green blue). The first one decomposes the task into an empty task network:* σ*(m0-do-put-on green blue)* $=$ *((do-put-on green blue),*$\emptyset$*). The second one decomposes the task into three subtasks:* σ*(m1-do-put-on green blue)* $=$ *((do-put-on green blue),* $\langle$*(do-clear green),(do-clear blue),(do-move green blue)*$\rangle$*).*

A compound task c is decomposable in a state s if and only if there exists a relevant method $m \in M$ such that: $\sigma(m) = (c, \omega')$ and $prec(m) \in s$. The function $\zeta : \{A \cup C\}^* \times S \rightarrow \{A \cup C\}^*$ gives the decomposition function. For a totally ordered task network $\omega = \omega_1 t \omega_2$, ζ is defined as follows:

$$\zeta(\omega_1 t \omega_2, s) = \begin{cases} \omega_1 t \omega_2 & \text{if } t \text{ is a primitive task.} \\ \omega_1 \omega' \omega_2 & \text{if } t \text{ is a compound task and } t \text{ is decomposable in } \gamma(s, \omega_1). \\ \emptyset & \text{Otherwise.} \end{cases}$$

Example 2.11 *For Blocksworld, we have:*

$$\begin{array}{rcl} prec(\text{m0-do-put-on green blue}) & = & \{(\text{on green blue}), (\text{handempty})\} \\ prec(\text{m1-do-put-on green blue}) & = & \{(\text{not(on green blue)}), (\text{handempty})\} \end{array}$$

Let's take our initial state s_0 *and suppose the agent tries to decompose the compound task (do-put-on green blue), we have* $prec(\text{m0-do-put-on green blue}) \notin s_0$ *and* $prec(\text{m1-do-put-on green blue}) \in s_0$*, then* ζ *cannot decompose the task with the method (m0-do-put-on green blue) but can decompose the task with the method (m1-do-put-on green blue). Finally, we have:*

$$\zeta((\text{do-put-on green blue}), s_0) = \{(\text{do-clear blue}), (\text{do-clear green}), (\text{do-move green blue})\}$$

As $\omega_1 t \omega_2$ is a totally ordered task network, ω_1 contains only primitive tasks. Indeed, as the network is totally ordered, the compound tasks are decomposed from left to right and therefore if we have to decompose t then ω_1 contains only primitive task.

We denote $\omega \rightarrow^* \omega^*$ that ω can be decomposed into ω^* by 0 or more method applications. Finally, ω_I is the initial task network. More precisely, ω_I is the task network that must be decomposed to solve the planning problem. In our example, $\omega_I = \{(\text{do-put-on green blue}), (\text{do-put-on red green})\}$.

A solution to an HTN planning problem is a task network ω with:

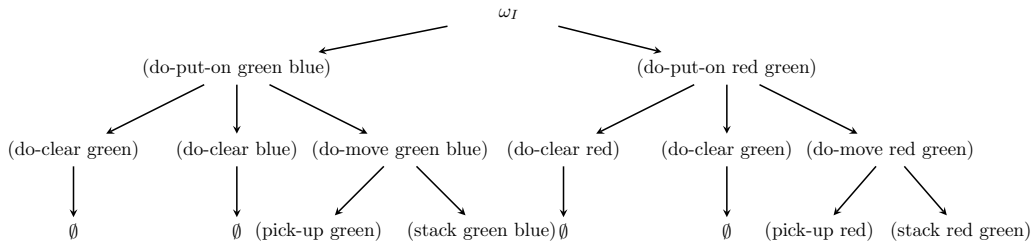


Figure 2.5: A solution task network for the Blocksworld example.

1. $\omega_I \rightarrow^* \omega$, i.e. it can be reached by decomposing ω_I.
2. $\omega \in A^*$, i.e. all tasks are primitive.
3. $\gamma(s_0, \omega) \models g$, i.e. ω is applicable in s_o and results in a goal state.

Figure 2.5 gives a solution task network for our example.

2.2.3.2 The HDDL Language

Proposed by Höller et al. (2020), the HDDL language is an extension of the PDDL language modeling HTN features. As for PDDL, it expresses planning problems in two files: (1) the planning domain file describing the operators and decomposition methods, and (2) the planning problem file describing the initial state, the initial task network and the goal. We will see how to define a planning domain and a planning problem using the HDDL language through the Blocksworld example.

HDDL Planning Domain File As for PDDL, to declare a planning domain we have to declare the name of the domain, the requirements, the types, and predicates:

```
(define (domain blocksworld_htn)
  (:requirements :typing :hierarchy :method-preconditions)
  (:types block)
  (:predicates
        (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (handempty)
        (holding ?x - block)
  )
```

Then, we declare all compound and primitive tasks with their parameters:

```
(:task do-put-on
 :parameters (?x - block ?y - block))
(:task do-on-table
```

```
 :parameters (?x - block))
(:task do-move
 :parameters (?x - block ?y - block))
(:task do-clear
 :parameters (?x - block))
(:task pick-up
 :parameters (?x - block))
(:task put-down
 :parameters (?x - block))
(:task stack
 :parameters (?x - block ?y - blocks))
(:task unstack
 :parameters (?x - block ?y - blocks))
```

Then, we declare HTN methods decomposing compound tasks:

```
(:method m1-do-put-on
 :parameters (?x - block  ?y - block)
 :task (do_put_on ?x ?y)
 :precondition (handempty)
 :ordered-subtasks(and
     (do-clear ?x)
     (do-clear ?y)
     (do-move ?x ?y))
)
```

To declare an HTN method we declare its name preceded by the keyword **:method**, its parameters preceded by the keyword **:parameters**, the task to decompose preceded by the keyword **:task**, the precondition preceded by the keyword **:precondition** and its subtasks preceded by the keyword **:ordered-subtasks**.

Finally primitive tasks are declared as classical PDDL operators:

```
(:action pick-up
 :parameters (?x - block)
 :precondition (and
      (clear ?x) (ontable ?x) (handempty))
 :effect (and
     (not (ontable ?x))
     (not (clear ?x))
     (not (handempty))
     (holding ?x))
)
```

HDDL Planning Problem File Let's now see how to describe a planning problem. For the Blocksworld example we have:

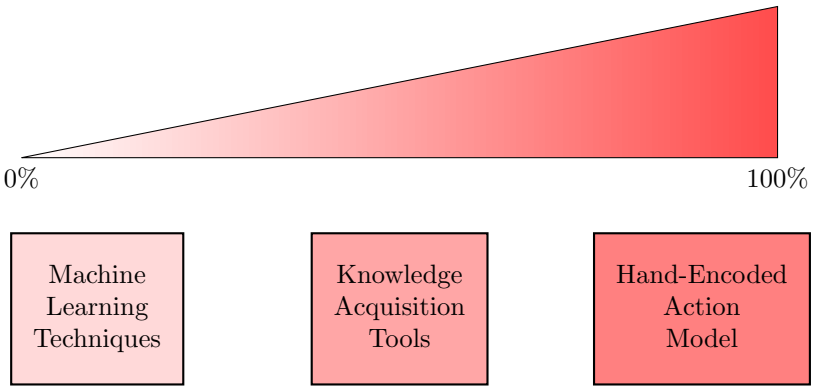


Figure 2.6: Human Effort for Action Model Acquisition.

```
(define (problem example_htn)
    (:domain blocksworld)
    (:objects red green blue - block)
(:htn
 :parameters ()
 :ordered-subtasks (and
     (task1 (do-put-on green blue))
     (task2 (do-put-on red green)))
)
(:init
    (clear red) (clear green) (clear blue)
    (ontable red) (ontable green) (ontable blue)
    (handempty)
)
(:goal
    (on red green) (on green blue)
)
```

As for PDDL, to declare the example problem we start by listing the different objects of the problem (section identified by the keyword **:objects**). Then, we declare the initial task network (section identified by the keyword **:htn**). Finally, as for PDDL, we declare the initial state and the goal state.

2.3 Action Model Acquisition

As we have just seen, AI planning requires the declaration of an action model in the form of a planning domain. To declare an action model we use a declarative language such as PDDL. The classical way to declare action models is the STRIPS formalism. As we have mentioned, STRIPS planning assumes several assumptions. On the other hand, AI planning tools have been developed in a wide range of real-world applications such as aerospace (Fisher et al., 2000;

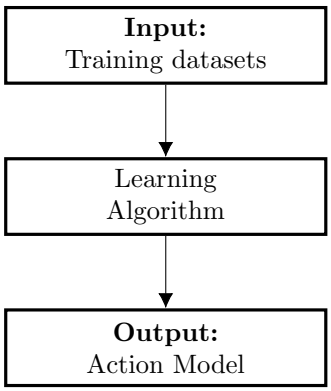


Figure 2.7: Machine Learning based Approach Architecture.

Backes et al., 2004; Bresina et al., 2005), autonomous vehicles (Urmson and Whittaker, 2008), logistics (Cross and Walker, 1994), robotics (Dvorak et al., 2014; Lallement et al., 2018; Liang et al., 2022), industry (Hoffmann et al., 2009). The assumptions made by STRIPS are too restrictive for these tools. Modeling real-world applications therefore requires to relax these assumptions and declare action models with more expressive declarative language such as PDDL 2.1 or HDDL. However, the more expressive a declarative language is, the more difficult it will be to hand-encode the planning domain. We must therefore develop tools to facilitate action model acquisition (AMA). Figure 2.6 shows the human efforts required for AMA.

A first approach to facilitate action model acquisition is the development of knowledge engineering tools (KET). These tools provide support for consistency and syntax error checking, domain visualization etc. Among them we can cite: GIPO (Simpson, 2005), itSimple (Vaquero et al., 2007, 2013), JABBAH (González-Ferrer et al., 2009), VIZ (Vodrázka and Chrpa, 2010), and EUROPA (Barreiro et al., 2012). However, these tools require a lot of AI planning expertise and background in software engineering (Shah et al., 2013).

The second approach for AMA consists in using machine learning (ML) techniques. The rest of this section focuses on this approach.

2.3.1 Action Model Acquisition based on Machine Learning

Figure 2.7 shows the architecture of ML-based AMA approaches. A learning algorithm takes as input a training dataset and returns an action model. This training dataset can contain a set of execution traces, structured data, human annotations and instructions, video etc. There are 3 main types of training datasets (see Figure 2.8):

- **Symbolic Execution Traces:** Symbolic Execution Traces (SET) are action sequences executed by the agent and are used by the majority of approaches. Among them we can cite ARMS (Yang et al., 2007), FAMA (Aineto et al., 2019), LSONIO (Mourão et al., 2012), OBSERVER (Wang,

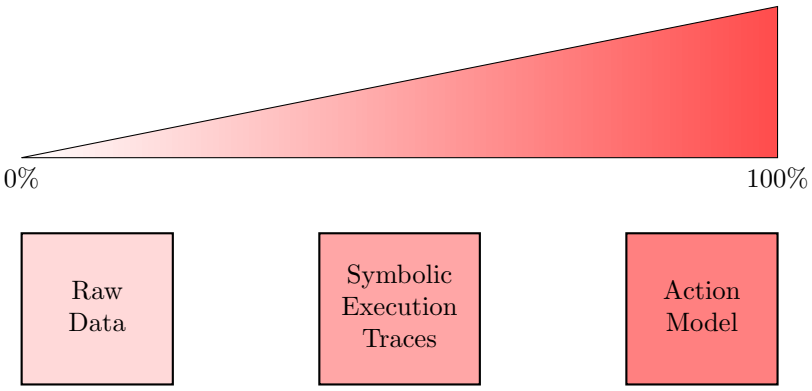


Figure 2.8: Effort for Training Dataset Acquisition.

1995) etc. These traces are called *symbolic* because we have the name of the action and its parameters, for example the following plan is a SET:

$$\langle (\textit{pick-up green}), (\textit{stack green blue}), (\textit{pick-up red}), (\textit{stack red green}) \rangle$$

In addition to the action sequences, SET also contain the states of the environment observed during the execution of the actions. Again we have a symbolic representation of these states, usually in the form of a set of logical propositions. Two reasons make the SET acquisition difficult: (1) it is necessary to generate, execute the action sequences and store the execution traces (2) it is necessary to parse the execution traces in order to obtain SET.

- **Action Model:** Some approaches take as input an action model. Among them, we can cite OpMaker (McCluskey et al., 2002), RIM (Zhuo et al., 2013), LIVE (Shen, 1993), EXPO (Gil, 1994), PELA (Celorrio et al., 2008). In addition to the action model, these approach takes as input symbolic execution traces. For these approaches, the ML techniques aim to correct and refine this action model. The input action model is usually obtained in two different ways, (1) it is assumed that the action model has been hand-encoded and the approach should only allow maintaining this action model, this is for example the case for the RIM algorithm which refines the action model, (2) it is assumed that the action model have been acquired using KET, and then ML techniques are used to correct and improve the action model. This is for example the case for the OpMaker approach: a STRIPS action model is generated using GIPO, and then OpMaker corrects the preconditions and effects and adds HTN methods.

- **Raw Data:** Raw data are execution traces which are not parsed into SET. This data can be verbal instructions (Miglani and Yorke-Smith, 2020), images (Asai, 2019), graph (Bonet and Geffner, 2020), human annotations (Zhuo, 2015).

Although SET are not the easiest inputs to acquire, most of learning algorithms take SET as input and we focus on these approaches.

2.3.1.1 ML-based AMA approaches using Symbolic Execution Traces

We are interested in algorithms for learning action models. We start by presenting the different criteria to categorize them.

Input There mainly exist two kinds of SET:

- **Goal Oriented:** Goal Oriented execution traces are plan traces, i.e. action sequences solving a given task. The main drawback of this kind of execution trace is that Goal Oriented execution traces are biased by the task solved. As execution traces are biased, there is a significant risk of overfitting.
- **Random Walk:** Random Walks are randomly generated action sequences. Random walks are generally generated either by directly testing actions in the environment or by querying an oracle. This approach has two advantages: (1) it limits the learning bias since the learning set is randomly generated, and (2) allows to exploit not only information about feasible sequences of actions but also information about infeasible sequences of actions.

In addition to the action sequences, execution traces usually contain observations of the different states of the environment. As seen previously, these states are sets of logical propositions. For example, the initial state of the Blocksworld problem will be represented as follows:

$$\{(\textit{ontable red}), (\textit{ontable green}), (\textit{ontable blue}), (\textit{clear red}),\\ (\textit{clear green}), (\textit{clear blue}), (\textit{handempty})\}$$

These observations can be partial and noisy. A partial observation is an observation where some propositions are missing:

$$\{(\textit{ontable red}), (\textit{clear green}), (\textit{clear blue}), (\textit{handempty})\}$$

and a noisy observation is an observation where some propositions are misjudged:

$$\{(\textit{ontable red}), (\textit{clear green}), (\textit{clear blue}), (\textit{handempty}), \textbf{(on green red)}\}$$

Finally, let us note that there are some algorithms having no state observation.

Machine Learning Techniques We will review the different ML techniques used for the action models acquisition.

- **Induction:** The learning approach takes as input a hypothesis space H and a training dataset Ω. The desired output is an hypothesis h from the hypothesis space H compatible with the dataset. Inductive techniques allow to identify patterns and generalize examples (Genesereth and Nilsson, 1988).

- **Max-Sat:** A weighted Max-Sat (maximum satisfiability) (Kautz and Selman, 1996; Borchers and Furman, 1998) problem can be stated as follows: given a collection of m clauses C: $(C_1, \ldots, C_m)$ each clause being a disjunction of logical variables, with a weight w_i of each clause, the learning approach tries to find the value of the logical variables maximizing the total weight of the satisfied clauses in C.

- **Probabilistic graphical models:** Probabilistic graphical models (PGM) (Pearl, 2014) are directed or undirected graphs. Each vertex represents a random variable and each edge represents a dependency of these variables.

- **Markov Logical Network:** A Markov Logical Network (MLN) (Richardson and Domingos, 2006)) is a combination of first-order logic and probabilistic graphical models. It is a first-order knowledge base composed of weighted formulas. It can be assimilated to a model used to build Markov networks.

- **Kernel Trick:** The kernel trick allows to use a linear classifier to solve a nonlinear problem. The key idea is to transform the representation space of the input data into a higher dimensional space, where a linear classifier can be used (Boser et al., 1992).

- **Genetic Algorithm:** A genetic algorithm is an iterative algorithm that will evolve a population of individuals, i.e. a set of candidates, until finding the optimal individual, i.e. the individual maximizing the fitness score (Mitchell, 1998).

- **Recurrent Neural Network:** Recurrent neural networks (RNN) (Jordan, 1997) allows to solve the sequence labeling problem efficiently. These neural networks are trained to associate each term of a sequence with a set of grammatical labels.

- **AI Planning:** The model to learn is represented by a planning problem with conditional effects. This planning problem is then solved by a planner and the model is built from the solution plan (Bonet et al., 2009).

- **Reinforcement Learning:** Reinforcement learning (RL) consists, for an agent, to learn actions from experiences in order to optimize a reward. The agent chooses the next action according to its current state. In return, the environment provides a positive or negative reward. The agent's goal is to find the policy that maximizes the reward function (Sutton, 1988).

- **Constraint Satisfaction Problems:** Constraint Satisfaction Problems (CSP) are optimization problems where we look for states satisfying several given constraints (Tsang, 1993).

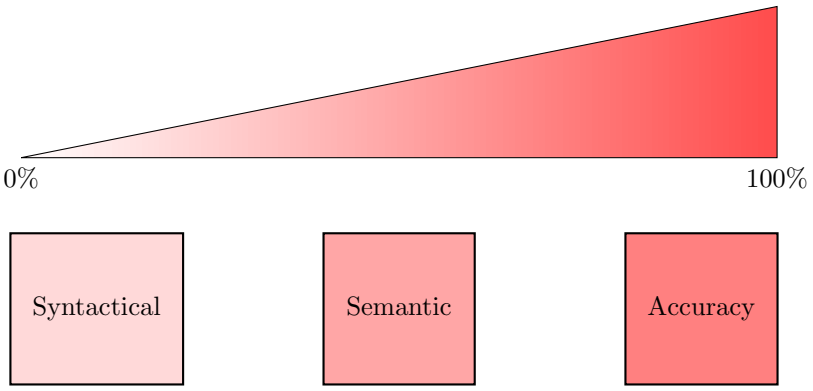


Figure 2.9: Performance Evaluation Quality.

Output The output is the action model. As we have seen in the previous section, there are several planning frameworks. The classical framework is STRIPS. STRIPS planning is based on restrictive assumptions. Some assumptions assumed by STRIPS can be relaxed. For example, HTN planning allows to declare, in addition to a goal state, a set of tasks to achieve. Temporal planning allows to declare actions whose execution can be concurrent. Although this thesis focuses on STRIPS, temporal and HTN planning, we can note that other formalisms exist, such as ADL (Pednault, 1994), allowing to declare more expressive action models: logical quantifier, conditional effects etc and multi-agent planning (Brenner, 2003) allowing to declare models where several agents interact.

Performances In practice, to learn action models, learning datasets are generated using action models from the International Planning Competition (IPC)[2]. This allows to test the approaches on a wide variety of actions models. Moreover, this action model benchmark is also used to test the quality of the learned action models. There are three main ways to measure the performance of the learning approach (see Figure 2.9):

- **Syntactical:** A syntactical evaluation simply compares the "ground truth" action model, i.e. the IPC action model with the learned action model. Usually by comparing the preconditions and effects of these action models.
- **Semantic:** A semantic evaluation allows to check some properties of an action model. Generally, a semantic evaluation is done using a testing dataset and will allow to check that there is no extra precondition or effect, that feasible (resp. infeasible) sequences with the ground truth action model are also feasible (resp. infeasible) with the learned action model etc.
- **Accuracy:** The accuracy evaluates the ability of the learned action model to be used by a planner to solve new planning problems, without a

[2] https://www.icaps-conference.org/competitions/

Algorithm	Input			ML Techniques	Eval		
	Traces	Environment	Noise		Syn	Sem	Acc
OBSERVER Wang (1995)	GO	FO		Induction			•
OLAM Lamanna et al. (2021a)	GO	FO		Induction	•		
AIA Verma et al. (2021a,b)	GO	FO		Induction	•		
SRMLearn Arora et al. (2017)	GO	FO		Max-Sat	•		
SLAF Shahaf and Amir (2006)	GO	PO		Induction	•		
ARMS Yang et al. (2007)	GO	PO		Max-Sat	•	•	
AMAN Zhuo and Kambhampati (2013)	GO	PO		PGM	•	•	
PDeepLearn Arora et al. (2018b)	GO	PO		RNN	•		
Louga Kucera and Barták (2018)	GO	PO		Genetic		•	
FAMA Aineto et al. (2019)	GO	PO		AI Planning	•		
Plan-Milner Segura-Muros et al. (2018)	GO	PO	•	Induction			•
AMDN Zhuo et al. (2019)	GO	FO	•	Max-Sat	•	•	
LOCM Cresswell et al. (2013)	GO	NO		Induction		•	
LOCM2 Cresswell and Gregory (2011)	GO	NO		Induction		•	
LOP Gregory and Cresswell (2015)	GO	NO		Induction		•	
ERA Balac et al. (2000a)	RW	FO		RL			•
LOPE Garcia-Martinez and Borrajo (2000)	RW	FO		RL			•
MARLIE Croonenborghs et al. (2007)	RW	FO		RL			•
IRALe Rodrigues et al. (2010a,b)	RW	FO	•	RL	•		•
LSO-NIO Mourão et al. (2012)	RW	PO	•	Kernel	•		

(a) STRIPS learning algorithms

Algorithm	Input			ML Techniques	Output			Eval		
	Traces	Environment	Noise		Prim	Methods	Prec	Syn	Sem	Acc
HTN-Maker Hogg et al. (2008)	GO	FO		Induction		•			•	
HTN-MakerND Hogg et al. (2009)	GO	FO		Induction		•			•	
Xiaoa et al. (2019)	GO	FO		Induction		•				•
Hogg et al. (2010)	GO	FO		Induction		•				•
LHTNDT Nargesian and Ghassem-Sani (2008)	GO	FO		Induction		•		•		
CAMEL Ilghami et al. (2002, 2005)	GO	FO		Induction		•	•			•
HDL Ilghami et al. (2006)	GO	FO		Induction		•	•			•
Garland and Lesh (2003)	GO	PO		Induction		•				•
HTN Learner Zhuo et al. (2009)	GO	PO		Max-Sat	•	•	•	•		

(b) HTN learning algorithms. This Table also present the output: Primitive Task Model, the set of HTN Methods and the HTN Methods preconditions.

Table 2.1: State-of-the-art of ML-based PDA approaches. From left to right: the kind of trace: **G**oal **O**riented or **R**andom **W**alk, the environment: **F**ully **O**bservable, **P**artially **O**bservable or **N**on **O**bservable, the robustness to noise in observations, the ML techniques used, the evaluation method used: **Syn**tactical, **Sem**antic or **Acc**uracy.

proofreading step by a planning expert. In this thesis, we argue that the accuracy is the most important evaluation method.

2.3.1.2 State-of-the-art of the ML based AMA approaches

We now present the different approaches according to their output, this section is summarized by Table 2.1. Since this thesis focuses on the learning of STRIPS, HTN and temporal action models, we focus on these action models.

STRIPS **Action Model Learning** A first group of approaches takes GO execution traces as input. Most of them deals with partial observations (except Observer (Wang, 1995) and OLAM (Lamanna et al., 2021b) that needs complete observations). Among these approaches are ARMS (Yang et al., 2007), Louga

(Kucera and Barták, 2018), SRMLearn (Arora et al., 2017), AIA (Verma et al., 2021a,b) or FAMA (Aineto et al., 2019). Among these works, the ARMS system is the most known. It gathers knowledge on the statistical distribution of frequent sets of actions in the GO execution traces. Then, it forms a weighted propositional satisfiability problem (weighted SAT) and solves it with a weighted MAX-SAT solver. Unlike ARMS, SLAF is able to learn actions with conditional effects, i.e. effects applied if several conditions are satisfied. To that end, SLAF relies on building logical constraint formula based on a direct acyclic graph representation. FAMA takes as input partial GO execution traces, i.e. GO execution traces where some actions are missing, and observations are partial. This algorithm turns the task of learning into a planning problem, and it resolves it by using a classical planner. The AIA algorithm learns action models by using a query system. It generates plans and queries a black-box AI system with these plans to test them and updates its action model from the black-box responses. Then, The SRMLearn algorithm uses an alternative representation of input state-action relationships to learn an output action model. It represents a set of dependencies, intra-action and inter action, in the form of constraints of a weighted maximum satisfiability problem. These constraints are then solved with a weighted MAX-SAT solver. Then, the PDeepLearn algorithm is divided into three parts (1) PDeepLearn enumerates all possible candidate action patterns, (2) PDeepLearn identifies frequent action pairs and (3) PDeepLearn labels the action sequences in order to identify the ideal pattern. Then, the Louga (Kucera and Barták, 2018) algorithm uses a genetic algorithm to learn the effects, and an ad-hoc algorithm to learn the preconditions. Finally, the LOCM family of action model learning approaches (Cresswell et al., 2013; Cresswell and Gregory, 2011; Gregory and Cresswell, 2015) works without information about initial, intermediate and final states. These algorithms extract, from GO execution traces, parameterized automata representing the behaviour of each object of the planning problems. Then, preconditions and effects are generated from these automata.

The second group of approaches takes as input random walks, i.e. sets of randomly generated action sequences. Random walk approaches like ERA (Balac et al., 2000a,b) and MARLIE (Croonenborghs et al., 2007) deal with complete and noiseless observations and use reinforcement learning techniques. Also, approaches like IRALe (Rodrigues et al., 2010a) deal with complete but noisy observations. IRALe is based on an online active algorithm to explore and to learn incrementally the action model with noisy observations. Other approaches such as LSONIO (Mourão et al., 2012) deal with both partial and noisy observations. LSONIO uses a classifier based on a kernel trick method to learn action models. It consists of two steps: (1) it learns a state transition function as a set of classifiers, and (2) it derives the action model from the parameters of the classifiers.

Temporal Action Model Learning Some approaches have been proposed to learn temporal features (Gabel and Su, 2010; Neider and Gavran, 2018; Gaglione

et al., 2021; Shah et al., 2018), only Garrido and Jiménez (2020) proposed an approach learning temporal action models.

HTN **Task Model Learning:** These approaches can be classified according to the output data of the learning process. The output can be the primitive task model, the set of HTN Methods and HTN Methods preconditions. Also, all these approaches take as input GO execution traces. A first group of approaches only learns the set of HTN Methods. First of all, Xiaoa et al. (2019) take as input a set of GO execution traces and HTN Methods and propose an algorithm to update incomplete HTN Methods by task insertions. HTN-Maker (Hogg et al., 2008) and HTN-MakerND (Hogg et al., 2009) takes as input plan trace generated from STRIPS planner and annoted task provided by a planning expert. An annoted task is a triplet (n, Pre, Eff) where n is a task, Pre is a set of propositions known as the preconditions and Eff is a set of atoms known as the effects. Then, Hogg et al. (2010) proposed an algorithm based on reinforcement learning. Then, Li et al. (2014) proposed an algorithm taking as input only GO execution traces. This algorithm builds, from GO execution traces, a context free grammar (CFG) allowing to regenerate all plans. Then, methods are generated using CFG: one method for each production rule in the CFG. Then, Garland and Lesh (2003) and Nargesian and Ghassem-Sani (2008) proposed to learn HTN Methods from annotated plan. Annotated plans are plans segmented with the different tasks solved. However, obtaining these annotated examples is difficult and needs a lot of human effort.

A second group of approach learns HTN Methods preconditions. First of all, the CAMEL algorithm (Ilghami et al., 2002) learns HTN Methods and the preconditions of HTN Methods from observations of GO execution traces, using the version space algorithm. This approach uses annotated task to build incrementally HTN Methods with preconditions. Then, the HDL algorithm (Ilghami et al., 2006) takes as input GO execution traces. For each decomposition in GO execution traces, HDL checks if there exist a method responsible of this decomposition. If not, HDL adds a new method and initializes a new version space to capture its preconditions. Preconditions are learned in the same way as in the CAMEL algorithm.

Only HTN-Learner proposes to learn both action model and HTN Methods from decomposition trees. A decomposition tree is a tree corresponding to the decomposition of a method.

Although we focus on algorithms learning STRIPS, HTN and temporal action models, we can note that some approaches have also been proposed to learn ADL action models (Shahaf and Amir, 2006; Zhuo et al., 2010a), numerical features (Martínez et al., 2015; Gregory and Lindsay, 2016; Segura-Muros et al., 2018) and multi-agent action models (Zhuo et al., 2011).

2.4 Conclusion

In this chapter, we introduced the AI planning concepts used in the development of this thesis and the literature related to these concepts. First of all, we presented the classical concepts of AI planning. Next, we presented three AI planning frameworks: (1) classical STRIPS planning and the PDDL language, (2) temporal planning and the PDDL 2.1 language and finally (3) hierarchical planning and the HDDL language. We have seen that these languages allow to declare the action models in the form planning domains. We then showed how to obtain these action models and we focused on ML-based approaches.

The ML-based approaches presented in this chapter have several drawbacks. The main drawback of these algorithms is that the learned action models are usually not correct enough to be usable by planners. The majority of the learning algorithms proposed so far do not provide any results on the ability of the learned action models to solve new problems, and a proofreading step of the learned action models is almost always required to correct errors. Moreover, most of the presented algorithms require GO execution traces, while we have seen that this kind of traces has the disadvantage to be biased. Finally, it is important that the learning algorithms are able to learn action models with partial and noisy observations. Indeed, in most cases these observations will be obtained using sensors, it is therefore possible that these different sensors cannot observe the environment in its entirety and it is also possible that there is noise in the observations due to the imprecision of these sensors.

As we have mentioned in this chapter, a planning problem can be seen as the search for a path in a state machine. Moreover, state machines are equivalent to automata. As we will see in the next chapter, an automaton allows to represent a grammar. We will see in the second part of this thesis that STRIPS planning problems can be represented using regular grammars. It is therefore possible to learn a regular grammar representing the state machine related to a planning problem. This is the basis of our approach: learning the state machine related to the planning problem using Regular Grammar Induction algorithms (see Chapter 3) and inducing the action model from this grammar. Finally, as planning problems are declared using a planning domain, our approach will have to represent the action model in the form of a planning domain, and more precisely, in the form of a set of planning operators. We will also see that it is possible to extend this approach for planning problems relaxing some assumptions made by STRIPS planning.

Before presenting our contributions in more details, we will give in the next chapter an introduction to the regular grammar induction field.

Chapter 3

Regular Grammar Induction

Contents

3.1 Introduction

We are interested in learning structural patterns that can be described by a grammar. A grammar describes how to form strings, i.e. sequences of symbols, belonging to a language. This problem belongs to the grammatical induction field. Grammatical induction is the process of discovering an acceptable

grammar for a language from a training dataset. This dataset consists of at least one *positive sample*, i.e. a finite subset of a language. Grammatical induction is therefore the process of discovering a grammar from a language containing at least all the elements of the positive sample. We can also have a *negative sample*, i.e. a finite set of sequences not belonging to the language. In this case, grammatical induction is the process of discovering a grammar from a language containing at least all the positive sample and containing no element of the negative sample.

Grammatical induction has been studied since the development of the theory of formal grammars. Besides its theoretical interest, it offers a set of potential applications, in particular in the fields of *Syntactic and Structural Pattern Recognition*, *Natural Language Processing* (Adriaans and van Zaanen, 2004; Dupont et al., 2008; Boström, 1996; Boström, 1998; Cruz-Alcázar and Vidal, 1998; Bex et al., 2006; Cruz-Alcázar and Vidal, 2008; Stein et al., 2006; Bréhélin et al., 2001; Raffelt and Steffen, 2006; Berg et al., 2006). We will limit ourselves to regular induction, i.e. the learning of a grammar representing a regular language. Indeed, as we will see in the next part, planning problems can be represented by regular grammars.

In this chapter we present the different notions of automata theory useful for grammatical induction. Once these notions are presented we give the formal framework of grammatical induction. More precisely, we will present the notion of *identification in the limit* and the search space explored by the regular grammar induction algorithms. Finally we present several approaches to grammatical induction. And more precisely, we are interested in regular grammar induction algorithms using positive and negative examples, because they allow us to learn regular languages in the limit.

Please note that for the sake of coherence and consistency with the rest of this thesis, we will not use traditional notations.

3.2 Definitions and Notations

We start by introducing the basic notions of automata theory. These notions will be useful to give the formal framework of grammatical induction.

3.2.1 Basic Notions of Automata Theory

We denote A a finite non-empty alphabet, i.e. the set of symbols present in the language. Then, we denote u, v, w the elements of A^*, i.e. the sequences of finite length on A. Finally, we denote $|u|$ the length of the sequence u and ϵ the empty sequence.

Definition 3.1 *u is a prefix of v if there exists w such that $uw = v$.*

Definition 3.2 *u is a suffix of v if there exists w such that $wu = v$.*

Example 3.1 *Let's take the alphabet $A = \{a, b\}$, with the sequence $v = aababbba$. The sequence $u = aaba$ is a prefix of v and $w = bbba$ is a suffix of v.*

Definition 3.3 *A language $\mathcal{L}$ is a subset of A^*.*

Example 3.2 *Let $\mathcal{L} = (a^*ba^*b)^*$ be the language accepting an even number of b, all sequences u such that u contains an even number of b belong to the language $\mathcal{L}$ (denoted $u \in \mathcal{L}$). Every sequence u such that u contains an odd number of b do not belong to the language L (denoted $u \notin \mathcal{L}$). In our example, we have $abbbab \in \mathcal{L}$ and $abbab \notin \mathcal{L}$.*

In the rest of this chapter, we denote $\mathcal{L}^{even}$ the language accepting an even number of b.

Definition 3.4 *Let $Pr(\mathcal{L}) = \{u | \exists v, uv \in \mathcal{L}\}$ be the prefix set of $\mathcal{L}$ and $\mathcal{L}/u = \{v | uv \in \mathcal{L}\}$ be the right quotient of $\mathcal{L}$. We have $\mathcal{L}/u \neq \emptyset$ if and only if $u \in Pr(\mathcal{L})$.*

Example 3.3 *Let's continue with our example, we have $v = abbaaabba \in \mathcal{L}^{even}$, $u = ab$ is a prefix of v and $w = ba$ is a suffix of v. Also, we have $ab \in Pr(\mathcal{L}^{even})$ and the right quotient is: $\mathcal{L}^{even}/ab = a^*b(a^*ba^*b)^*$.*

3.2.2 Regular Grammar

A regular grammar (RG) is a grammar describing regular language (Yu, 1997). An RG can be *right* or *left*:

- A right RG has the form:
  - $S \to aS'$
  - $S \to a$
  - $S' \to a$
  - $S' \to aS$
- A left RG has the form:
  - $S \to S'a$
  - $S \to a$
  - $S' \to a$
  - $S' \to Sa$

where S and S' are non-terminal symbols, and a is a terminal symbol. A terminal symbol is a symbol belonging to the alphabet A and a non-terminal symbol is a which can be replaced according to the above rules.

Property 3.1 *An RG can be represented as a Deterministic Finite Automaton (DFA) where states are non-terminal symbols and transitions are terminal symbols.*

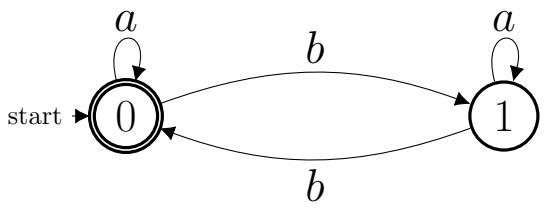


Figure 3.1: A first example of automaton: Σ^{even} accepting the even language $\mathcal{L}^{even}$.

The notion of DFA will be discussed in Section 3.2.3.

Example 3.4 *Let's take as example the following right RG:*

- $S_0 \rightarrow \epsilon$
- $S_0 \rightarrow a$
- $S_0 \rightarrow aS_0$
- $S_0 \rightarrow bS_1$
- $S_1 \rightarrow bS_0$
- $S_1 \rightarrow b$
- $S_1 \rightarrow aS_1$

This right RG accepts the even language $\mathcal{L}^{even}$ and can be represented by the DFA in Figure 3.1.

In the rest of this chapter, we denote Σ^{even} this DFA accepting the even language $\mathcal{L}^{even}$.

3.2.3 Automata and Partitions

3.2.3.1 Automata

Definition 3.5 *An automaton is a tuple $\Sigma = (S, A, \gamma, s_0, G)$ where S is a finite state set, A is a finite alphabet, γ is a transition function such that $S \times A \rightarrow S$, $s_0 \in S$ is the initial state and $G \subseteq S^*$ is the goal state set.*

Informally, an automaton is a set of states, linked together by transitions labeled by symbols. Given a sequence as input, the automaton reads the symbols of the sequence one by one and goes from state to state according to the transitions. The sequence read is either accepted by the automaton or rejected. A sequence is accepted if the automaton can read all the symbols of the sequence, and if the last state reached is a goal state.

Definition 3.6 *A DFA is a finite automaton whose transitions are deterministic, i.e. for each state, there is a unique transition for a read symbol.*

Definition 3.7 *Let $\gamma : S \times A^* \rightarrow S$ be the transition function extended to sequences. This function returns the last reached state by the automaton after reading the sequence from a given state. This function is defined as follows:*

$$\forall s \in S, \forall u \in A^*, \gamma(s,u) = \begin{cases} s & \text{if } u = \epsilon. \\ \gamma(\gamma(s,u'),a) & \text{if } u = u'a. \end{cases}$$

We can formally define the acceptation of a sequence u by an automaton Σ: it accepts a sequence u if and only if $\gamma(s_0,u) \in G$. Finally, the set of sequences accepted by an automaton Σ is denoted $\mathcal{L}(\Sigma)$ and the language accepted by Σ is defined as follows:

$$\mathcal{L}(\Sigma) = \{u \in A^* \; s.t. \; \gamma(s_0,u) \in G\}.$$

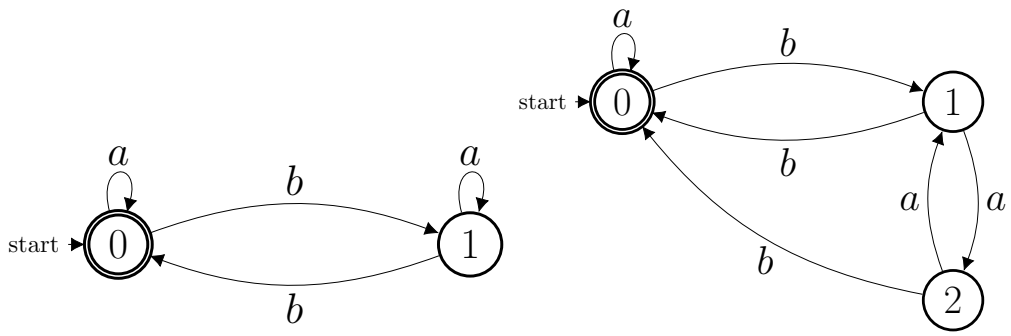
Example 3.5 *Let's take the automaton in Figure 3.1. In this example, the automaton states are $S = \{0,1\}$. Then, the alphabet is $A = \{a,b\}$. Then, the transition function is represented by arcs connecting two states. For example, we have $\gamma(0,b) = 1$. Then, the goal state is: $G = \{0\}$ and the initial state is $s_0 = 0$. This automaton accepts the sequence abba because $\gamma(0,abba) = 0 \in G$ and rejects, i.e. does not accept, the sequence abbaba because $\gamma(0,abbaba) = 1 \notin G$. Finally, we have $abba \in \mathcal{L}(\Sigma)$, $abbaba \notin \mathcal{L}(\Sigma)$.*

A same language can be accepted by several automata. Nevertheless, in the field of grammatical induction, there is a form of automaton of particular interest to us: the *minimal canonical automaton* $A(\mathcal{L})$[1]. Informally, $A(\mathcal{L})$ is the smallest automaton accepting the language $\mathcal{L}$. For example, let's take the even language $\mathcal{L}^{even}$. The automaton in Figure 3.2a gives its minimal canonical automaton. And Figure 3.2b shows another automaton, non canonical, accepting this language. We note that if we merge states 1 and 2 of the automaton in Figure 3.2b we obtain the automaton in Figure 3.2a. This merge is a *derivation* of the automaton3.2b.

3.2.3.2 Quotient Automata

Let's go back to the automata in Figures 3.2a and 3.2b. Let's assume that we have $S = \{0,1,2\}$, we denote $\Pi_i = \{\{0\},\{1,2\}\}$ and $\Pi_j = \{\{0\},\{1\},\{2\}\}$ two sets of subsets of S. We call these sets *partitions*. Π_i is the partition of S for the automaton in Figure 3.2a and Π_j is the partition of S for the automaton in Figure 3.2b. Both automata share the same state set S but Figure 3.2a considers that states 1 and 2 are the same state, while Figure 3.2b considers them as distinct states. The subsets of these partitions are called *blocks*. For Π_i blocks are: $\{0\}$ and $\{1,2\}$. And for Π_j blocks are: $\{0\}$, $\{1\}$ and $\{2\}$. We can note that Π_i is smaller than Π_j. Indeed, Π_i contains two blocks while Π_j contains three blocks. This is

[1] We will give a formal definition in Section 3.2.3.3



(a) The minimal canonical automaton accepting an even number of b.

(b) An automaton accepting an even number of b.

Figure 3.2: Several automata accepting an even number of b.

due to Π_i being *finer* than Π_j. A partition Π_i is finer than a partition Π_j when all blocks of Π_i are either a block or the union of several blocks of Π_j. Finally, we can note that, when we merge the blocks $\{1\}$ and $\{2\}$ of Π_j, we obtain the automaton 3.2a. The automaton 3.2a is the quotient automaton of 3.2b w.r.t. the partition Π_i.

We will now formally define these different notions.

Definition 3.8 *For any set S, a partition Π is a set of subsets of S, non-empty and disjoint two by two whose union is S. If $s \in S$ then $B(s, \Pi)$ is the single block of Π containing s.*

Definition 3.9 *A partition Π_i is finer than a partition Π_j if each block of Π_j is either a block of Π_i or the union of several blocks of Π_j.*

Definition 3.10 *Let $\Sigma = (S, A, \gamma, s_0, G)$ be an automaton. The automaton $\Sigma/\Pi = (S', A, \gamma', B(s_0, \Pi), G')$ is derived from Σ w.r.t. the partition Π, and is called the quotient automaton Σ/Π. This automaton is defined as follows:*

$$\begin{aligned} S' = &\quad S/\Pi = \{B(s, \Pi) | s \in S\}, \\ G' = &\quad G/\Pi = \{B(g, \Pi) | g \in G\}, \\ \gamma' = &\quad S' \times A \to 2^{S'} : \forall s'_i, s'_j \in S', \forall a \in A, s'_j \in \gamma'(s'_i, a) \text{ iff} \\ &\quad \exists s_i, s_j \in S \text{ s.t. } s'_i \in B(s_i, \Pi) \wedge s'_j \in B(s_j, \Pi) \wedge s_j \in \gamma(s_i, a). \end{aligned}$$

3.2.3.3 Samples of a Language and Associated Automata

Before giving the formal framework of regular grammatical induction (RGI), we will define what a sample is and give some remarkable automata used in the RGI field.

Definition 3.11 *A positive sample, denoted I_+, is a finite subset of sequences of the language $\mathcal{L}$.*

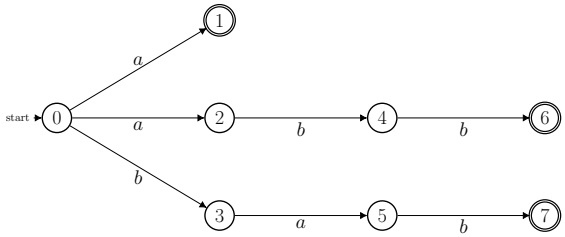


Figure 3.3: The maximal canonical automaton with $I_+ = \{a, abb, bab\}$.

Definition 3.12 *A negative sample, denoted I_-, is a finite subset of sequences of the language $\overline{\mathcal{L}}$, the* complement *of $\mathcal{L}$: $\overline{\mathcal{L}} = A^* \setminus \mathcal{L}$.*

Example 3.6 *Let's go back to the even language $\mathcal{L}^{even}$. The complement of this language is $\overline{\mathcal{L}} = \{a^*ba^*\{a^*ba^*ba^*\}^*\}$; the language accepting an odd number of b. A positive sample of $\mathcal{L}^{even}$ could be $I_+ = \{a, abb, abbabb, \ldots\}$ and a negative sample could be $I_- = \{b, babb, bbb, \ldots\}$.*

Definition 3.13 *A positive sample I_+ is structurally complete (Fu and Booth, 1975) to the automaton Σ accepting the language $\mathcal{L}$ if there exists an acceptation of I_+ such that all automaton's transitions are used and all goal states of G are used as goal states.*

Example 3.7 *Let's go back to the automaton Σ^{even} (see Figure 3.1) accepting the even language $\mathcal{L}^{even}$. The positive sample $I_+ = \{\epsilon, a, abb, abab\}$ is structurally complete but the positive sample $I_+ = \{\epsilon, a, abb\}$ is not structurally complete because the transition $\gamma(1, a)$ is not used.*

Now that we have given the definitions related to sampling, we will give some remarkable automata used for regular grammar induction.

Definition 3.14 *The maximal canonical automaton w.r.t I_+, denoted $MCA(I_+)$, is defined as follows:*

$$\begin{array}{lcl} S & = & \{S_{i,j} | 1 \leq i \leq |I_+|, 1 \leq j \leq |v_i|, v_{i,j} = a_{i,1}, a_{i,2}, \ldots, a_{i,j}\} \cup \{\epsilon\}, \\ s_0 & = & \epsilon, \\ G & = & I_+, \\ \gamma\epsilon, a & = & \{a_{i,1} | a_{i,1} = a, 1 \leq i \leq |I_+|\}, \\ \gamma(v_{i,j}, a) & = & \{v_{i,j+1} | v_{i,j+1} = v_{i,j}a, 1 \leq i \leq |I_+|, 1 \leq j \leq |v_i| - 1\}. \end{array}$$

Example 3.8 *Suppose we have $I_+ = \{a, abb, bab\}$. The alphabet is $A = \{a, b\}$. The state set is:*

$$\begin{array}{rl} S & = \{S_{i,j} | 1 \leq i \leq |I_+|, 1 \leq j \leq |v_i|, v_{i,j} = a_{i,1}, a_{i,2}, \ldots, a_{i,j}\} \cup \{\epsilon\}, \\ & = \{S_{i,j} | 1 \leq i \leq |\{a, abb, bab\}|, 1 \leq j \leq |v_i|, v_{i,j} = a_{i,1}, a_{i,2}, \ldots, a_{i,j}\} \cup \{0\}, \\ & = \{1, 2, 3\} \cup \{4, 5\} \cup \{6, 7\} \cup \{0\} = \{0, 1, 2, 3, 4, 5, 6, 7\}. \end{array}$$

Then, $s_0 = 0$. Then, goal states are states defined by the last symbol of each example then $G = \{1, 6, 7\}$. Then, we build transition starting from the initial state s_0: $\gamma(0, a) =$

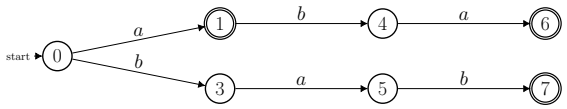


Figure 3.4: The prefix tree acceptor with $I_+ = \{a, abb, bab\}$.

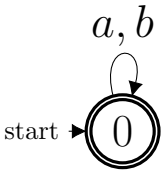


Figure 3.5: The universal automaton for $A = \{a, b\}$.

$\{1, 2\}$, $\gamma(0, b) = \{3\}$. *Finally, we add all transitions :* $\gamma(2, b) = 4$, $\gamma(3, b) = 5$ *etc. Figure 3.3 gives the* $MCA(\{a, abb, bab\})$. *Therefore,* $\mathcal{L}(MCA(I_+)) = I_+$ *and* $MCA(I_+)$ *is the bigger automaton (possibly non-deterministic), i.e. the automaton with the maximal number of states, for which* I_+ *is structurally complete. We can note that* $MCA(I_+)$ *is possibly non-deterministic.*

Definition 3.15 *The prefix tree acceptor of* I_+, *denoted* $PTA(I_+)$ *is the automaton* $MCA(I_+)/\Pi$ *where the partition* Π *is defined as follow:*

$$\forall s, s' \in S, B(s, \Pi) = B(s', \Pi) \; iff \; Pr(s) = Pr(s').$$

Less formally, the $PTA(I_+)$ is the determinization of $MCA(I_+)$. Then, to build $PTA(I_+)$ we merge all states sharing the same prefix. Therefore, $\mathcal{L}(PTA(I_+)) = I_+$ and $PTA(I_+)$ is the biggest DFA, i.e. the DFA with the maximal number of states, for which I_+ is structurally complete.

Example 3.9 *Suppose we have* $I_+ = \{a, bab, abb\}$. *The set state is* $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ *and the partition after merging is* $\Pi = \{\{0\}, \{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$. *Figure 3.4 gives the* $PTA(\{a, abb, bab\})$.

Definition 3.16 *The universal automaton of the alphabet* A, *denoted* UA, *is the automaton accepting the language* A^*.

Example 3.10 *Figure 3.5 gives the universal automaton for* $A = \{a, b\}$.

Definition 3.17 *The automaton* $A(\mathcal{L})$ *is the smallest automaton for the language* $\mathcal{L}$. $A(\mathcal{L})$ *is called the minimal canonical automaton of* $\mathcal{L}$ *and is defined as follows:*

$$\begin{aligned} &S = \{\mathcal{L}/u | u \in Pr(\mathcal{L})\}, \\ &s_0 = \mathcal{L}/\epsilon, \\ &G = \{\mathcal{L}/u | u \in \mathcal{L}\}, \\ &\gamma(\mathcal{L}/u, a) = \mathcal{L}/ua, \; ua \in Pr(\mathcal{L}). \end{aligned}$$

Example 3.11 *Let's take the the even language* $\mathcal{L}^{even}$. *The alphabet is* $A = \{a, b\}$, *the state set is build using prefixes :* $S = \{0, 1\}$, *we have* $s_0 = 0$ *and* $G = \{0\}$. *Finally, we build the transition function:* $\gamma(0, a) = 0$, $\gamma(0, b) = 1$ *etc. Figure 3.6 gives* Σ^{even}, *the minimal canonical automaton accepting* $\mathcal{L}^{even}$.

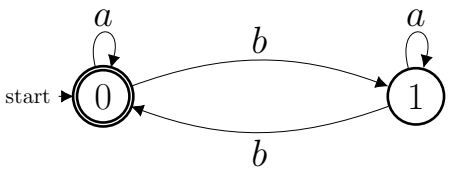


Figure 3.6: Σ^{even}: The minimal canonical automaton accepting $\mathcal{L}^{even}$.

3.3 Regular Grammatical Induction

Now that we have seen the different notions around the theory of automata used for regular grammatical induction (RGI), we will give the formal framework of RGI. First of all, we will describe the notion of *identification in the limit* (Gold, 1967). An RGI algorithm identifies a language $\mathcal{L}$ in the limit, if, from a finite set of training data, this algorithm identifies $\mathcal{L}$, and, if we add more data, continues to identify $\mathcal{L}$. We will show in this section that all regular languages are identifiable in the limit when the training datasets contain both positive and negative examples. Then, we will introduce the search space used by RGI algorithms dealing with both positive and negative examples.

3.3.1 Identification in the limit

We will introduce the notion of identification in the limit.

Suppose we have an infinite ordered set of positive examples, including any element of our language $\mathcal{L}$, and an infinite set of negative examples including any element of the complement language. And let us denote ω_i the i^{th} training data, with ω_i either a positive or a negative example. Let us also assume that we have H the hypotheses space, i.e. the set of hypotheses. An hypothesis is a possible concept or model for the training datasets. For example, in the context of grammar induction, the set of hypotheses could the set of automata accepting the positive sample. Also, we denote $H(\omega_i)$ the set of correct hypotheses after reading the first i training data, i.e. the set of hypotheses covering all the first i training data.

Let's step out of the RGI framework for a moment. Suppose we want to identify a set of numbers with the following training dataset: $\Omega = \{\{0,1\},\{1,1\},\{2,1\},\ \{3,1\},\{4,0\},\{5,1\},\{6,0\},\{7,0\},\{8,1\}\ldots\}$ with $H = \{\mathbb{N}, even, odd, Fibonacci\}$ our hypotheses space.

- $\omega_0 = 0$ is a positive example: Several hypotheses can identify a set beginning by 0, the set of natural integers $\mathbb{N}$, the Fibonacci sequence, the set of even integers. Only the odd hypothesis is incorrect.
- $\omega_1 = 1$ is a positive example: We have $H(\omega_1) = \{\mathbb{N}, Fibonacci\}$. Both even and odd hypotheses are incorrect.
- $\omega_2 = 2$ is a positive example: We have $H(\omega_2) = \{\mathbb{N}, Fibonacci\}$. Both even and odd hypotheses are incorrect.

- $\omega_3 = 3$ is a positive example: We have $H(\omega_3) = \{\mathbb{N}, Fibonacci\}$. Both even and odd hypotheses are incorrect.
- $\omega_4 = 4$ is a negative example: We have $H(\omega_4) = \{Fibonacci\}$. The Fibonacci sequence in the only correct hypothesis.
- $\omega_5 = 5$ is a positive example: We have $H(\omega_5) = \{Fibonacci\}$. The Fibonacci sequence in the only correct hypothesis.
- $\omega_6 = 6$ is a negative example: We have $H(\omega_6) = \{Fibonacci\}$. The Fibonacci sequence in the only correct hypothesis.
- $\omega_7 = 7$ is a negative example: We have $H(\omega_7) = \{Fibonacci\}$. The Fibonacci sequence in the only correct hypothesis.
- $\omega_8 = 8$ is a positive example: We have $H(\omega_8) = \{Fibonacci\}$. The Fibonacci sequence in the only correct hypothesis.

We notice that from ω_4 only the Fibonacci hypothesis is correct, and this hypothesis remains correct after adding all positive and negative examples. This is the identification in the limit. Now that we have seen an introductory example about the notion of identification in the limit, we will formally introduce this notion.

Definition 3.18 *A positive presentation of a language $\mathcal{L}$ is an infinite set of examples with at least one occurrence of each element of $\mathcal{L}$. These elements are the positive examples of the language $\mathcal{L}$.*

Definition 3.19 *A negative presentation of a language $\mathcal{L}$ is an infinite set of examples with at least one occurrence of each element of the complement language $\overline{\mathcal{L}}$. These elements are the negative examples (or counter examples) of the language $\mathcal{L}$.*

Definition 3.20 *A complete presentation of a language $\mathcal{L}$ is an infinite and ordered set $\Omega = \{(\omega_i, d_i) : A^* \times \{0,1\}\}$ with:*

$$d = \begin{cases} 1 & \text{If } \omega_i \in L. \\ 0 & \text{Otherwise.} \end{cases}$$

A positive sample is a finite set of elements of a language. We can therefore consider a positive presentation as an infinite sequence of positive samples whose size increases. Such a presentation is called *admissible*.

Notions of identification in the limit and class of languages have been introduced by Gold (1967) and are defined as follows:

Definition 3.21 *Let M be an inference method and Ω an admissible presentation of a language $\mathcal{L}$. We denote ω_i the i^{th} element of Ω. The inference method M has to propose a new solution in the search space for each new example. Let $H(\omega_i)$ be the hypotheses proposed by M after reading the i first elements of Ω. A method M identifies in the limit the language $\mathcal{L}$ if and only if there exists a finite index j such that:*

1. $\forall i \geq j, H(\omega_i) = H(\omega_j)$,

2. $H(\omega_j) = \{\mathcal{L}\}$.

The first condition ensures that M converges and the second condition ensures that M converges to a correct representation of $\mathcal{L}$.

Definition 3.22 *An inference method M identifies a class $\mathcal{C}$ of languages in the limit if M identifies in the limit all languages $\mathcal{L} \in \mathcal{C}$.*

Theorem 3.1 *The class of regular languages is identifiable in the limit in a polynomial time for an inference method M using an admissible presentation containing both positive and negative samples.*

In this chapter we focus on the notion of identification in the limit because it is an exact identification criterion. Nevertheless, many other identification criteria have been proposed. For example, the *BC-identification* (Behaviorally Correct Identification) (Barzdinš, 1974; Case and Smith, 1983) does not require that the inference method converge to a single hypothesis in the space of hypotheses, but that, from the point of convergence, all the hypotheses proposed constitute a possible description of the correct solution.

3.3.2 Search Space

We will now focus on the search space of RGI algorithms using both positive and negative samples. First we will present, in a general framework, the notion of lattice. Then, we will show how to use the lattice in the RGI field.

A lattice is a partially ordered set with a greatest upper and a smallest lower bound. Each pair of elements of a lattice has a smallest lower bound and a greatest upper bound. We can formally define a lattice as follows:

Definition 3.23 *A lattice is a pair $(S, \leq)$ such that:*

- *$\leq$ is an ordering relation of S such that:*
  - $\forall x \in S, x \leq x$,
  - $\forall x, y \in S, x \leq y \wedge y \leq x \implies x = y$,
  - $\forall x, y, z \in S, x \leq y \wedge y \leq z \implies x \leq z$.
- *All pairs $\{x, y\}$ of S has a greatest upper and a smallest lower bound:*
  - *the smallest lower bound of x and y, denoted $x \top y$ is the unique maximal element of the set of predecessors of x and y,*
  - *the greatest upper bound of x and y, denoted $x \bot y$ is the unique maximal element of the set of successors of x and y.*

Property 3.2 *If a partition Π_i is finer than a partition Π_j, denoted $\Pi_i \leq \Pi_j$, then quotient automata follow the same relation: $\Sigma/\Pi_i \leq \Sigma/\Pi_j$, and $\mathcal{L}(\Sigma/\Pi_i) \subseteq \mathcal{L}(\Sigma/\Pi_j)$.*

Definition 3.24 *The set of quotient automata of Σ with the partial order $\leq$ is an automata lattice, denoted $Lat(\Sigma)$, for whom Σ and UA are, respectively, the smallest lower and the greater upper bound.*

Theorem 3.2 *Let I_+ be a positive sample of the language $\mathcal{L}$ and Σ an automaton such that $\mathcal{L}(\Sigma) = \mathcal{L}$. If I_+ is structurally complete w.r.t Σ, then $\Sigma \in Lat(MCA(I_+))$.*

Theorem 3.3 *Let I_+ be a positive sample of the language $\mathcal{L}$ and $C(\mathcal{L})$ the canonical automaton of $\mathcal{L}$. If I_+ is structurally complete w.r.t $A(\mathcal{L})$, then we have $A(\mathcal{L}) \in Lat(PTA(I_+))$.*

We can consider the grammatical induction by the search of an automaton compatible with I_+ and I_-. Formally, we search an automaton Σ such that (1) $I_+ \subseteq \mathcal{L}(\Sigma)$ and (2) $I_- \cap \mathcal{L}(\Sigma) = \emptyset$. There exist several automata checking both conditions. For example, $MCA(I_+)$ checks both conditions. However, this automaton does not generalize the positive sample. So, we have to search the most general solution w.r.t the negative sample I_-. If we choose the simplicity of the inferred automaton as a generality criterion and restrict the search to DFA, then the solution we are looking for is the compatible DFA with the minimum number of states. This is the so-called *minimal DFA consistency problem* which is NP-hard (Angluin, 1978; Gold, 1978). We are therefore looking for an element of the *border set* of our automata lattice. The border set (Dupont et al., 1994) is the limit of the possible generalization from a positive sample and a negative sample.

Example 3.12 *Suppose we have $I_+ = \{a, bab\}$ and $I_- = \{ababb, b, baa\}$. Figure 3.7 shows the automata lattice whose smallest lower bound is $MCA(I_+)$ and greatest upper bound is UA. Automata in the green area are automata compatible with I_- and automata in the red area are automata incompatible with I_-. The border set, denoted $BS_{MCA}(I_+, I_-)$, is the border between the green area and the red area.*

3.4 Regular Grammar Induction Algorithms

We focus on RGI algorithms taking as input both positive and negative samples. Unlike algorithms using only a positive sample, algorithms that take both positive and negative samples as input can identify the class of regular languages in the limit. Also, this identification is done in a polynomial time with samples of polynomial size w.r.t the number of states. These properties are particularly interesting in applications where the dataset acquisition is a difficult and costly task. Let us note however that there are several inference algorithms using only positive samples e.g. ID (Angluin, 1981), IID (Parekh et al., 1998),

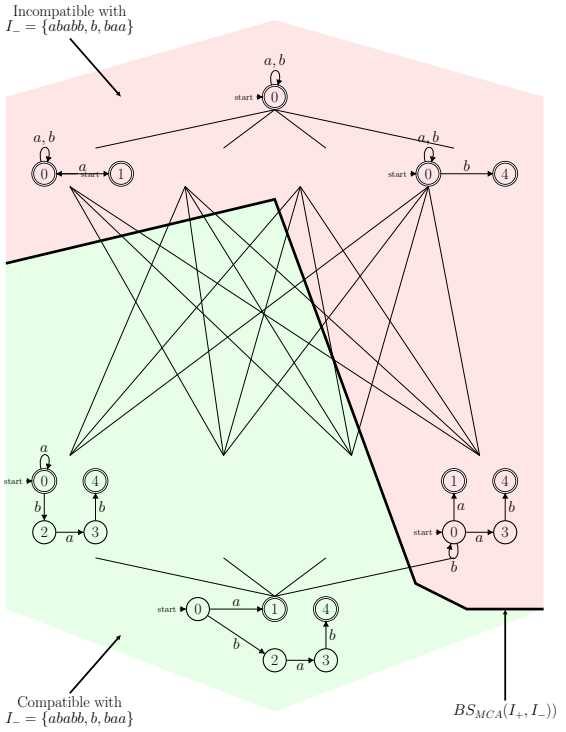


Figure 3.7: The automata lattice with $I_+ = \{a, bab\}$ and $I_- = \{ababb, b, baa\}$.

k-TSSI (Garcia and Vidal, 1990), MGGI (Garcia et al., 1990), ECGI (Kudo and Shimbo, 1988) etc.

A first method of inference is the use of genetic algorithm (Goldberg, 1989; Mitchell, 1998). For example, the GIG algorithm evolves in the lattice $Lat(MCA(I_+))$ a population obtained by deriving $MCA(I_+)$ using structural mutation and crossover.

Then, another approach is to build the border set and returns the smallest automaton belonging to the border set. For example, the RIG algorithm (**R**egular **I**nference of **G**rammar) (Miclet and de Gentile, 1994) builds the border set $BS_{MCA}(I_+, I_-)$. It proceeds by enumerating the automata derived from $MCA(I_+)$. This is a breadth-first search of $Lat(MCA(I_+))$ keeping only compatible automata at each depth in the lattice. At each step, the algorithm performs a hierarchical pruning, i.e. it eliminates all automata at depth i+1 that derive from at least one incompatible automaton. Finally, the border set is obtained by storing all the compatible automata that have no compatible derivatives.

The last approach is to test block merging until there are no more possible merge. The EDSM algorithm (**E**vidence **D**riven **S**tate **M**erging) (Lang, 1998) begins by building the prefix tree acceptor $PTA(I_+)$. Then, the algorithm tests all block mergings compatible with I_-. To find blocks to merge, EDSM uses a heuristic that computes a score for each state of the automaton. Then, it tests the pairs of blocks with the highest score. Then, the RPNI algorithm (**R**egular **P**ositive and **N**egative **I**nference) (Oncina and Garcia, 1992) performs a deep

Algorithm 1: The RPNI Algorithm.

input : I_+, I_-: Positive and Negative samples
output : Σ: A DFA

```
1  Π ← {{0}, {1}, ..., {N − 1}};
2  Σ ← PTA(I+);
3  for i ← 1 : |Π| − 1 do
4      for j ← 0 : i − 1 do
5          Π' ← Π/{Bj, Bi} ∪ {Bi ∪ Bj};
6          Σ/Π' ← derive(Σ/Π');
7          Σ/Π'' ← deterministic_merge(Σ/Π'');
8          if compatible(Σ/Π'', I−) then
9              Σ ← Σ/Π'';
10             Π ← Π'';
11             break;
12 return Σ, Π
```

search in $Lat(PTA(I_+))$ and finds a locally optimal solution to the minimal DFA consistency problem. This algorithm relies on the structur completeness of the positive sample I_+. RPNI starts by creating a partition of the set of states of $PTA(I_+)$. RPNI then proceeds to merge the blocks that do not cause compatibility problems with I_- according to the partial order induced by the lattice.

3.4.1 Focus on the RPNI Algorithm

Among all RGI algorithms, RPNI is very efficient and has all the right properties: (1) it is able to identify in the limit the class of the regular languages; (2) RPNI is locally optimal.

3.4.1.1 The RPNI Algorithm

The RPNI algorithm is described by Algorithm 1 and Figure 3.8 gives an example of execution. RPNI takes as input a positive sample I_+ and a negative sample I_-.

Example 3.13 *For example, consider the following sample:*

$$\begin{aligned} I_+ &= \{\epsilon, a, baaaba, bab, bb, bba\}, \\ I_- &= \{ba, b, ab\}. \end{aligned}$$

RPNI starts by initializing the partition (line 1) where each block contains a single state: $\Pi = \{\{0\}, \{1\}, \ldots, \{10\}\}$. *Then, RPNI builds the prefix tree acceptor and initializes the automaton* Σ *(line 2). Then, RPNI tests all state merges and*

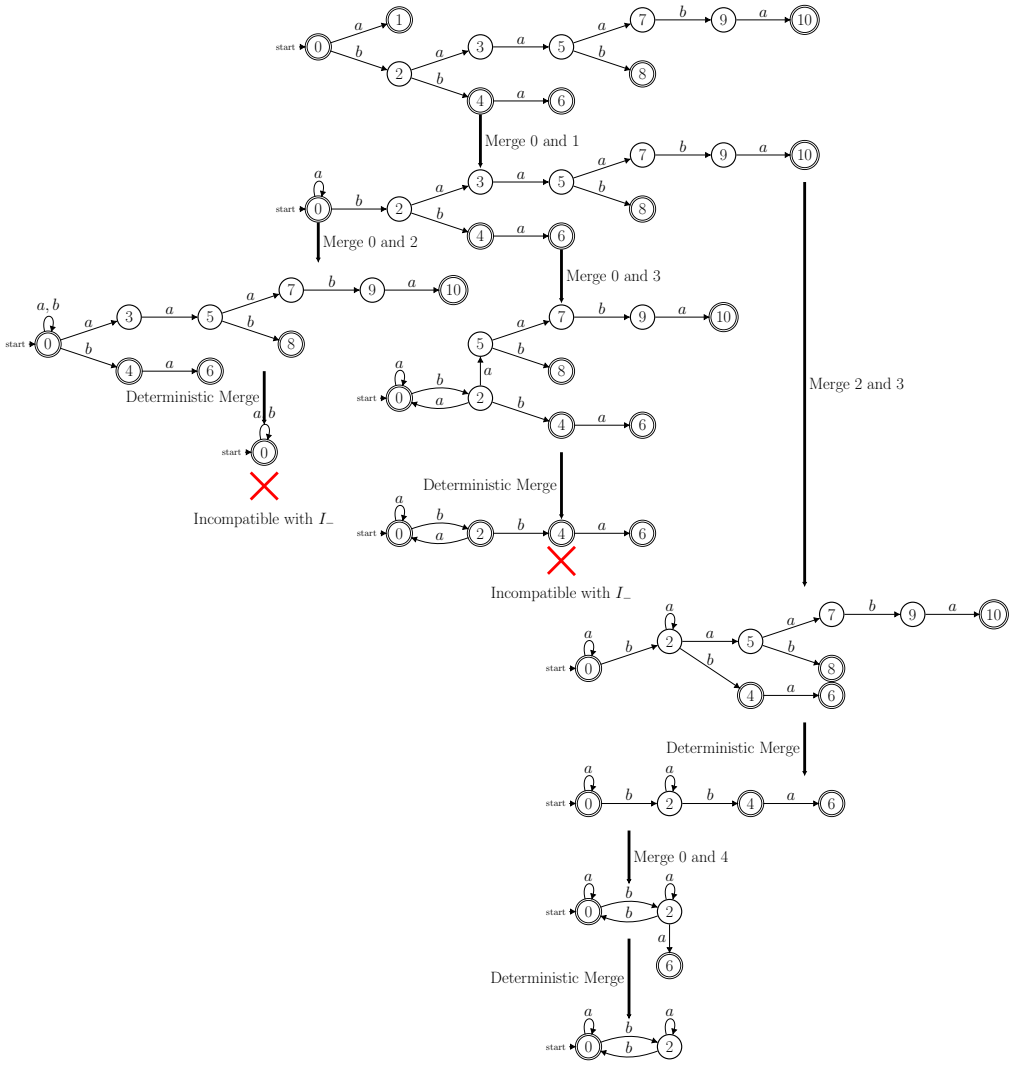


Figure 3.8: RPNI Execution with $I_+ = \{\epsilon, a, baaaba, bab, bb, bba\}$ and $I_- = \{ba, b, ab\}$.

retains only merges which do not cause the acceptation of a negative example (line 3-11). RPNI begins by testing the merge between states 0 *and* 1*. RPNI updates the partition by merging blocks where states* 0 *and* 1 *appear (line 5):* $\Pi' = \{\{0,1\},\{2\},\{3\},\ldots,\{10\}\}$*. Then, RPNI computes the quotient automaton* Σ'/Π' *(line 6). As the quotient automaton is deterministic and rejects all negative examples, the merge is retained (line 8-10). Now, RPNI tries to merge states* 0 *and* 2*. We can observe that the automaton is non-deterministic. Indeed, state* 0 *has now two outgoing transitions a. To remove the non-deterministic transitions, RPNI performs a deterministic merge (line 7). Figure 3.9 shows an execution of the deterministic merge. RPNI begins by merging states* 0 *and* 3*. As the automaton is always non-deterministic (the automaton has two outgoing transitions a and b in the state* 0*), RPNI merges states* 0 *and* 4*, and states* 0 *and* 5*. The automaton obtained after these mergings is now deterministic. However, it accepts all negative examples, then RPNI*

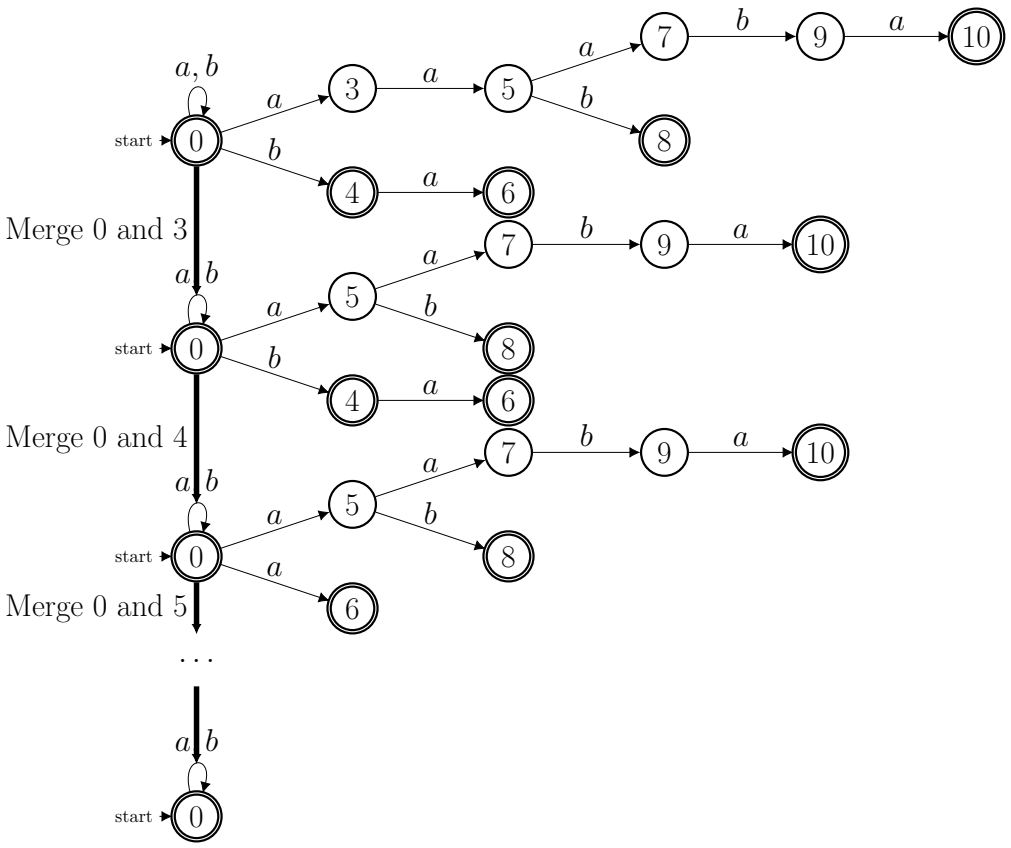


Figure 3.9: Deterministic Merge.

goes back to the previous successful merge and tries to merge the next pair of sates 0 *and* 3*. As automaton is non-deterministic, RPNI performs the deterministic merge. After determinization, the automaton accepts the negative example ba, the merge is therefore not retained. After having testes all state merges, the final partition is* $\Pi = \{\{0,1,4,6,8,9,10\},\{2,3,5,7\}\}$.

We can now study the complexity of the RPNI algorithm. We have $|I_+|$ the number of states in $PTA(I_+)$, and we denote $|I_-| = \sum_{\forall u \in I_-} |u|$ the size of the negative sample. The complexity of the deterministic merge of the quotient automaton is $\mathcal{O}(|I_+|)$ because the quotient automaton contains at most $|I_+|$ states, therefore, at most $\mathcal{O}(|I_+|)$ merges are necessary. Then, the complexity to test the compatibility with I_- is $\mathcal{O}(|I_-|)$. Finally, we have:

$$C_{RPNI}(I_+, I_-) = O(|I_+|^2.(|I_+| + |I_-|))$$

3.4.1.2 Identification in the Limit and Characteristic Sample Specification

The RPNI algorithm identifies in the limit the class of regular languages. In other words, when the sample is characteristic of a language $\mathcal{L}$, then RPNI learns the minimal canonical automaton of $A(\mathcal{L})$. We will define notion of characteristic

sample in this section. The notion of characteristic sample $\Omega^c = (I^c_+, I^c_-)$ was introduced by Dupont (1996).

The RPNI algorithm identifies the class of regular languages in the limit. Then, for each regular language $\mathcal{L}$ there exists a sample $\Omega^c = \{I^c_+, I^c_-\}$ such that:

- $RPNI(I^c_+, I^c_-) \rightarrow \Sigma$ with $\Sigma = A(\mathcal{L})$.
- $\forall I'_+, I'_-$ such that $\forall \omega \in I'_+, \omega \in \mathcal{L}$ and $\forall \omega \in I'_-, \omega \notin \mathcal{L}(P)$ then $RPNI(I^c_+, I^c_-) = RPNI(\{I^c_+ \cup I'_+\}, \{I^c_- \cup I'_-\})$.

The sample Ω^c is the *characteristic* sample of $\mathcal{L}(P)$. We give the conditions that the samples must satisfy to be characteristic for the language $\mathcal{L}$ and the RPNI algorithm. To this end, we shall present the notions of the *set of short prefixes* and the *kernel* of a language (Oncina and Garcia, 1992).

Definition 3.25 *The set of short prefixes $S_P(L)$ for a language $\mathcal{L}$ is defined as follows:*

$$S_P(\mathcal{L}) = \{x \in Pr(\mathcal{L}) | \neg \exists u \in A^* \text{ s.t. } \mathcal{L}/u = \mathcal{L}/x \wedge u < x\}.$$

Less formally, the set of short prefixes $S_P(\mathcal{L})$ is the set of prefixes reaching to a state of the canonical automaton of $\mathcal{L}$.

Definition 3.26 *The kernel $N(\mathcal{L})$ for a language $\mathcal{L}$ is defined as follows:*

$$N(\mathcal{L}) = \{\epsilon\} \cup \{xa | x \in S_P(\mathcal{L}) \wedge a \in A \wedge xa \in Pr(L)\}.$$

Example 3.14 *For example, consider the DFA Σ^{even}, the minimal canonical automaton of the even language $\mathcal{L}^{even}$. For this DFA $S_P(\mathcal{L}) = \{\epsilon, b\}$. And the kernel of a language is the set of elements of $S_P(\mathcal{L})$ to which each transition of the minimal canonical automaton has been added. Let us take again the DFA Σ^{even}, for this DFA we have $N(\mathcal{L}) = \{\epsilon, a, b, ba, bb\}$.*

Finally, we can build the characteristic sample for the language $\mathcal{L}$:

Definition 3.27 *The sample $\Omega^c = (I^c_+, I^c_-)$ is characteristic for the language $\mathcal{L}$ if and only if:*

1. $\forall x \in N(\mathcal{L})$ *if* $x \in \mathcal{L}$ *then* $x \in I^c_+ \vee \exists u \in A^* | xu \in I^c_+$,
2. $\forall x \in S_P(\mathcal{L}), \forall y \in N(\mathcal{L})$ *if* $\mathcal{L}/x \neq \mathcal{L}/y$ *then* $\exists u \in A^* | (xu \in I^c_+ \wedge yu \in I^c_-) \vee (xu \in I^c_- \wedge yu \in I^c_+)$.

The first condition requires that all the kernel elements are present in our positive sample. When an element of the kernel belongs to $\mathcal{L}$, then it belongs to our positive sample, but when it does not belong to $\mathcal{L}$ then this element is the prefix of an element of the positive sample. And the second condition requires that for each pair of elements of $S_P(\mathcal{L})$ and $N(\mathcal{L})$ leading to distinct states, there exists a suffix u which distinguishes the two states reached in our sampling.

(x, y)	u	I_+^c	I_-^c
(ϵ, a)	-	$-$	$-$
(ϵ, b)	a	a	ba
(ϵ, ba)	b	bab	b
(ϵ, bb)	$-$	$-$	$-$
(b, ϵ)	a	a	ba
(b, a)	b	bb	ab
(b, ba)	$-$	$-$	$-$
(b, bb)	a	bba	ba

Table 3.1: Characteristic sample specification for the even language

Example 3.15 *Let's take again the DFA Σ^{even}, we have:*

$$\begin{aligned} S_P(\mathcal{L}) &= \{\epsilon, b\}, \\ N(\mathcal{L}) &= \{\epsilon, a, b, ba, bb\}. \end{aligned}$$

To build a characteristic sample for this DFA we have to build a sample respecting the two conditions seen previously. The first condition requires that all the elements of the kernel are either positive examples or are prefixes of positive examples. We start by recovering the elements of the kernel which are sequences accepted by the automaton. So we have: $N(\mathcal{L}) \cap \mathcal{L} = \{\epsilon, a, bb\}$. Now we have to find positive examples whose suffixes would be b and ba. We can use bb whose prefix is b, and baaaba whose prefix is ba. So the first condition gives us the following positive sample:

$$I_+^c = \{\epsilon, a, bb, baaaba\}.$$

We must now build the negative sample, and complete the positive sample to satisfy the second condition. The second condition requires that for each pair of elements of $N(\mathcal{L})$ and $S_P(\mathcal{L})$ leading to two distinct states, there is a positive example, and a negative example distinguishing them. For example, if we take the pair (ϵ, a), both sequences reach the same state. On the other hand if we take (ϵ, b), ϵ reaches the state 0, while b reaches the state 1, so we have to add a positive and a negative example for this pair. We can use $u = a$, which implies that we have $a \in I_+^c$ and $ba \in I_-^c$. Table 3.1 gives in detail the whole specification of the positive and negative samples. Finally, the characteristic sample for the even language is:

$$\begin{aligned} I_+^c &= \{\epsilon, a, baaaba, bab, bb, bba\}. \\ I_-^c &= \{ab, b, ba\}. \end{aligned}$$

Please note that a characteristic sample is not unique, indeed, several suffix u can be used to satisfy the first or the second condition. Finally, we can now compute the size of the characteristic sample:

$$\begin{aligned} &|S_P(\mathcal{L})| = |S|. \\ &|N(\mathcal{L})| = \mathcal{O}(1 + |S| + |A|). \\ &|I_+^c| = \mathcal{O}(|S|^2.|A|). \\ &|I_-^c| = \mathcal{O}(|S|^2.|A|). \end{aligned}$$

3.5 Conclusion

To conclude, we have seen in this chapter the theoretical framework of RGI as well as a state of the art on RGI using both positive and negative samples. Note that grammatical induction techniques have been proposed to learn other forms of grammars such as transducers (Oncina et al., 1993; Vilar, 2000), probabilistic grammars (Angluin, 1988; Stolcke and Omohundro, 1994; Carrasco and Oncina, 1994; Guttman et al., 2006), context-free grammars (Sakakibara, 1992; Nevill-Manning and Witten, 1997; Koshiba et al., 2000; Eyraud et al., 2007) etc. We focused on RGI techniques because, as we will see later, planning problems are related to state machines that represent regular languages.

We have focused on algorithms using both positive and negative samples because these algorithms can identify in the limit the class of regular languages in a polynomial time. In addition, we specified the sampling characterization for the RPNI algorithm. Although there are other identification criteria, identification in the limit is an exact identification criterion. This is of interest to us because we will use RGI techniques to learn action models. As we will see next, we will use RGI techniques, and more precisely the RPNI algorithm, to learn a language representing a planning problem and then we will induce the action model from the language. Using RGI techniques with an exact identification criterion will allow us to avoid learning errors in the learned action model.

Part II

Contributions

Chapter 4

STRIPS Action Model Learning

Contents

4.1 Introduction

As mentioned in Chapter 2, a planning problem can be seen as the search for a path in a state machine. Moreover, state machines are equivalent to automata. As we have seen in Chapter 3, an automaton allows to represent a grammar. As we will see in this chapter, these grammars are regular. It is therefore possible to identify in the limit the regular grammar related to a given planning problem using algorithm taking as input both positive and negative samples such as

RPNI (Oncina and Garcia, 1992). In this chapter, we will exploit this property to learn action models.

As mentioned in Chapter 2, hand-encoding action models is a difficult task which makes it difficult to use AI planning techniques in real-world applications. To mitigate this difficulty, the AI planning community have proposed approaches using machine learning techniques to compute action models from execution traces such as, for example, ARMS (Yang et al., 2007), SLAF (Shahaf and Amir, 2006), LSONIO (Mourão et al., 2012), LOCM (Cresswell et al., 2013). These approaches have three main drawbacks:

1. Most of these approaches require a lot of data to perform the learning of action models and in many real world applications, acquiring training datasets is difficult and costly, e.g., Mars Exploration Rover operations (Bresina et al., 2005) or fleet of Autonomous Underwater Vehicles for offshore missions (Carreno et al., 2020; Lesire et al., 2016)).

2. The learned domains are not enough *accurate* to be used "as is" in a planner: a step of expert proofreading is still necessary to correct them. Even small syntactical errors can make sometime the learned domains useless for planning.

3. Even if some approaches, e.g., (Mourão et al., 2012; Segura-Muros et al., 2018; Rodrigues et al., 2010a) are able to learn from noisy and/or partially observable data, few approaches are able to handle very high levels of noise and high levels of partial observations as can be encountered in real world applications.

In this chapter, we present AMLSI (*Action Model Learning with State machine Interaction*) (Grand et al., 2020b,a), a novel approach learning STRIPS action models based on grammar induction techniques (see Chapter 3). The idea is to learn the state machine related to the planning problem using RGI algorithms and inducing the action model from this state machine. Also, as planning problems are declared using a planning domain, our approach will have to represent the action model in the form of a planning domain.

The key idea of the AMLSI approach is to interact with the environment in which the agent will have to solve planning problems to learn the action model. AMLSI tests different randomly choosen actions, observes how the environment evolves when these actions are executed, learns the DFA and induces the action model from its observations. AMLSI does not require any prior knowledge regarding the feasibility of actions in a given state, and state observations can be partial and noisy. AMLSI is highly accurate even with highly partial and noisy state observations. Our contribution is threefold:

1. AMLSI maximizes data usability by exploiting both feasible and infeasible sequences of actions. Also, as action sequences are randomly generated, the AMLSI approach avoid the overfitting issue.

2. AMLSI is highly accurate and requires few data.

3. AMLSI is robust to partial and noisy observations.

In the rest of this chapter, we start by giving the problem statement of the STRIPS Action Model Learning. More precisely, we extend the formal framework proposed in Chapter 2 to the learning problem. Then, we detail the AMLSI approach. We present the different steps of our approach, and we give the formal properties of our approach. Finally, we propose an evaluation of our approach. We compare AMLSI with state-of-the-art approaches and show that AMLSI outperforms these approaches.

4.2 Problem statement

Without loss of generality, we propose a formal framework inspired by (Höller et al., 2016) in order to define the STRIPS learning problem as the lifting of a state transition system into a propositional language. This formal framework extends the formal framework proposed in Chapter 2 to the learning problem.

Definition 4.1 *A classical planning problem P extended to the learning problem is a tuple* $(L, A, S, s_0, G, \delta, \tau, \lambda)$ *where:*

- L *is the set of logical propositions describing the environment.*
- A *is the set of actions.*
- S *is the set of state labels.*
- $s_0 \in S$ *is the initial state label.*
- $G \in S$ *is the set of goal state labels.*
- δ *is the action model.*
- $\tau : S \times A \rightarrow \{true, false\}$ *is the feasibility function.*
- $\lambda : S \rightarrow 2^L$ *is the observation function.*

Example 4.1 *Consider the Blocksworld example. L is composed of the following propositions: {(ontable red),(ontable green),(ontable blue),(on green blue),...}. A possible observed initial state* s_0 *could be* $\lambda(s_0) = \{$*(ontable red), (ontable green), (ontable blue), (clear red), (clear green), (clear blue), (handempty)*$\}$.

We assume that an observation could be partial and/or noisy. A partial observation is a state where some logical propositions are missing and a noisy observation is a state where the truth value of some propositions is erroneous.

Example 4.2 *For example, let take the following noiseless and complete observation:*

$$\lambda(s_0) = \{\textit{(ontable red), (ontable green), (ontable blue), (clear red), (clear green), (clear blue), (handempty)}\}$$

A partial observation could be:

$$\lambda(s_0) = \{\textit{(ontable green), (ontable blue), (clear green), (handempty)}\}$$

And a noisy observation could be:

$\lambda(s_0) = \{$*(ontable green), (ontable blue), (clear green), (handempty),* **(on red green)**$\}$

A is a set of action labels. Action preconditions, positive and negative effects are given by the functions *prec*, *add* and *del* that are included in $\delta = (prec, add, del)$. *prec* is defined as $prec : A \rightarrow 2^L$. The functions *add* and *del* are defined in the same way.

Example 4.3 *As example, consider the action (pick-up red):*

- *prec(pick-up red) = {(handempty), (ontable red), (clear red)}*
- *add(pick-up red) = {(holding red)}*
- *del(pick-up red) = {(handempty), (ontable red), (clear red)}*

The function $\tau : S \times A \rightarrow \{true, false\}$ returns whether an action is applicable to a state, i.e. $\tau(s, a) \Leftrightarrow prec(a) \subseteq \lambda(s)$. Whenever an action a is applicable in state s_i, the state transition function $\gamma : S \times A \rightarrow S$ returns the resulting state $s_{i+1} = \gamma(s_i, a)$ such that $\lambda(s_{i+1}) = [\lambda(s_i) \setminus del(a)] \cup add(a)$. A sequence $\langle a_1 \dots a_n \rangle$ of actions is applicable to a state s_0 when each action a_i with $1 \leq i \leq n$ is applicable to the state s_{i-1}. Given an applicable sequence $\langle a_1 \dots a_n \rangle$ in state s_0, $\gamma(s_0, \langle a_1 \dots a_n \rangle) = \gamma(\gamma(s_0, a_1), \langle a_2 \dots a_n \rangle) = s_n$. It is important to note that this recursive definition of γ entails the generation of a sequence of states $\langle s_0 s_1 \dots s_n \rangle$. A goal state is a state g such that $g \in G$. An action sequence is a solution plan to a planning problem P if and only if it is applicable to s_0 and entails a goal state.

As we have seen in Chapter 3 a formal language is, for a given alphabet A, a subset of A^*, i.e. a possibly infinite set of sequences of elements of A. It is therefore possible to represent a STRIPS planning problem as a language such that all the elements of this language is a solution plan. More precisely, we denote $\mathcal{L}(P)$ the language generated by the STRIPS planning problem P, this language is defined as follows:

$$\mathcal{L}(P) = \{\omega = \langle a_1 \dots a_n \rangle | a_i \in A, \gamma(s_0, \omega) \models g\}.$$

We know that the set of languages generated by STRIPS planning problems are regular languages (Höller et al., 2016). In other words, a STRIPS planning

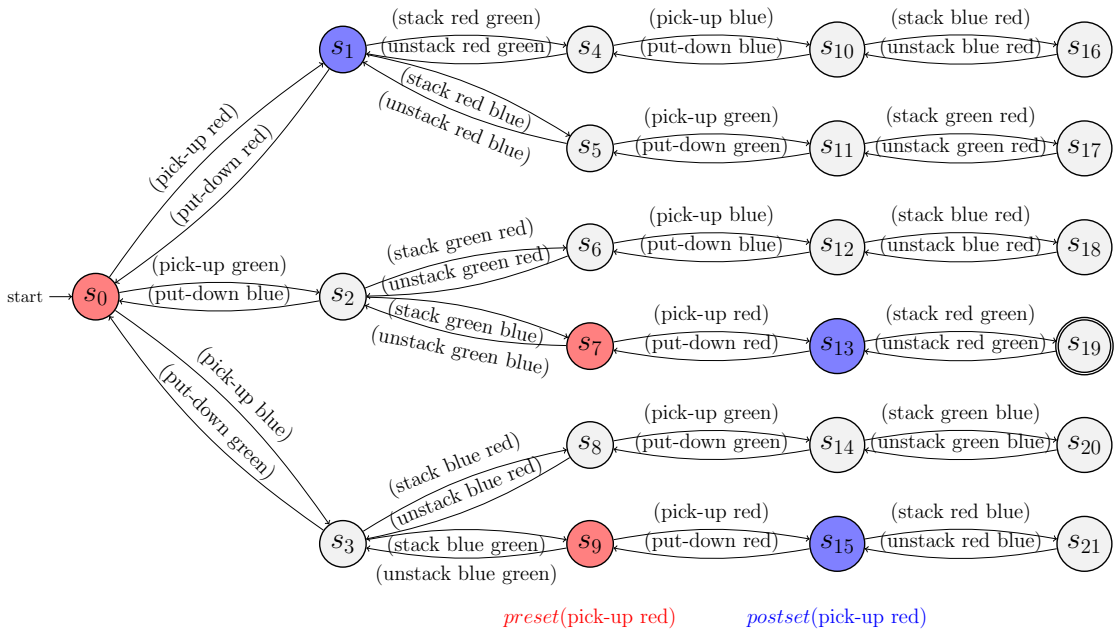


Figure 4.1: An example of DFA with pre-states and post-states.

problem P generates a language $\mathcal{L}(P)$ that is equivalent to a DFA $\Sigma = (S, A, \gamma, s_0, G)$. S and A are respectively the states and the arcs of the DFA, γ is the transition function, s_0 is the initial state and G is the set of goal state labels. Figure 4.1 gives the DFA for the Blocksworld example.

For any arc $a \in A$, we call *pre-set* of a the set $preset(a) = \{s \in S \mid \gamma(s, a) = s'\}$ and *post-set* of a the set $postset(a) = \{s' \in S \mid \gamma(s, a) = s'\}$. In our example we have $\{s_0, s_7, s_9\} \in preset(pick\text{-}up\ red)$ and $\{s_1, s_{13}, s_{15}\} \in postset(pick\text{-}up\ red)$ (see Figure 4.1).

A STRIPS learning problem is defined as follows: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn the DFA Σ, and then infer the action model δ and generalize it into a planning domain Δ from the learned DFA Σ?

For instance, suppose

$$\begin{aligned}\Omega = \{\ & \langle (pick\text{-}up\ red), (put\text{-}down\ red), (pick\text{-}up\ red), (stack\ red\ green), (unstack\ red\ green), \\ & (stack\ red\ green)\rangle \\ & \langle (pick\text{-}up\ blue), (put\text{-}down\ blue), (pick\text{-}up\ red), (stack\ res\ blue), (unstack\ red\ blue), \\ & (put\text{-}down\ red)\rangle \\ & \ldots\}\end{aligned}$$

The key idea of our approach is to learn Σ with the action sequences and infers the action model

$$\begin{aligned}\delta:\ & prec(pick\text{-}up\ red) = \{(ontable\ red), (clear\ red), (handempty)\} \\ & prec(pick\text{-}up\ blue) = \{(ontable\ blue), (clear\ blue), (handempty)\} \\ & \ldots\end{aligned}$$

from Σ and the partial and noisy observation function λ and generalizes it into the following PDDL planning domain:

```
(:action pick-up
 :parameters (?x - block)
 :precondition (and (clear ?x) (ontable ?x) (handempty))
 :effect
 (and (not (ontable ?x))
   (not (clear ?x))
   (not (handempty))
   (holding ?x)))
```

4.3 STRIPS Learning

The main idea of AMLSI is that it is possible to learn a state machine by testing these transitions and by observing the states resulting from the action executions and to represent it as an action model. AMLSI assumes that it knows the names of the actions, i.e., the names of the transition of the state machine, it can test and it is able to observe the state resulting from their application as a set of logical propositions whose predicates are also known.

Based on these assumptions, AMLSI produces a set of observations Ω by using a random walk and learns as output an action model modeling these observations. The action model learned is expressed as a classical planning domain using the PDDL language. To perform this learning, AMLSI learns first the transition function expressed as a set of actions δ of a particular planning problem P and then, it generalizes δ as a set of planning operators Δ. AMLSI assumes that L, A, S, and s_0 are known, and the observation function λ is possibly partial and noisy. No knowledge of the goal states G is required. As we have no knowledge on G, we consider that the language $\mathcal{L}(P)$ is defined for all feasible action sequences ω, even if ω is not a solution plan. Formally,

$$\mathcal{L}(P) = \{\omega = \langle a_1 \dots a_n \rangle | a_i \in A, \gamma(s_0, \omega) \text{ is defined.}\}$$

AMLSI algorithm consists of 4 steps (see Figure 4.2):

1. *Observation Generation.* AMLSI produces a set of observations Ω by using a random walk. In Section 4.3.1, we will present how AMLSI is able to efficiently exploit these observations by taking into account not only the fact that some actions are applicable in certain states but also that others are not.

2. *DFA Learning.* AMLSI learns the DFA Σ representing the observed state transition function γ using an alternative version of the RPNI algorithm (Oncina and Garcia, 1992) (see Section 4.3.2).

3. *PDDL Operator Generation.* Once the DFA Σ is learned, AMLSI infers the action precondition, positive and negative effect functions in δ from the DFA Σ and the observation function λ. Finally, the set of PDDL Operators Δ are induced from δ (see Section 4.3.3).

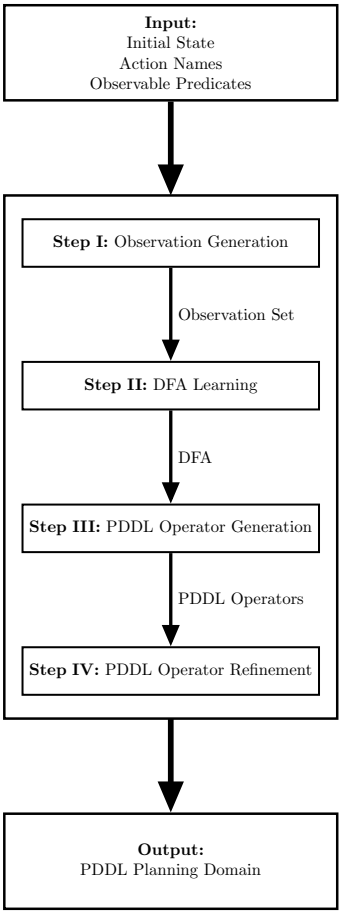


Figure 4.2: STRIPS learning process.

4. *PDDL Operator Refinement.* AMLSI refines PDDL operators in order to deal with partial and noisy observations (see Section 4.3.4).

4.3.1 Observation Generation

Figure 4.3 gives an overview of this step. To generate the observations in Ω, AMLSI uses random walks by applying a randomly selected action to the initial state of the problem. If this action is feasible, it is appended to the current action sequence. Otherwise, the feasible prefix plus the infeasible action are added to the set of negative samples I_-. The procedure is repeated until the feasible prefix achieves an arbitrary size and added to the set of positive samples I_+. Random walks are repeated until I_+ and I_- achieve an arbitrary size. As an example, consider below the sets of feasible sequences I_+ and not feasible sequences I_- in the Blocksworld domain.

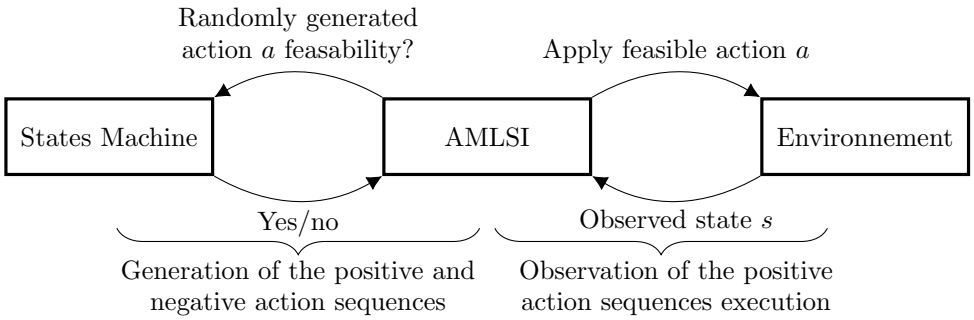


Figure 4.3: Observation Generation Overview.

$$
\begin{aligned}
I_+ = \{ \quad & \langle (\textit{pick-up red}), (\textit{put-down red}), (\textit{pick-up red}), (\textit{stack red blue}), \\
& (\textit{unstack red blue}), (\textit{stack red blue}) \rangle \\
& \langle (\textit{pick-up blue}), (\textit{put-down blue}), (\textit{pick-up red}), (\textit{stack red blue}), \\
& (\textit{unstack red blue}), (\textit{put-down reed}) \rangle \\
& \langle (\textit{pick-up red}), (\textit{put-down red}), (\textit{pick-up blue}), (\textit{stack blue red}), \\
& (\textit{unstack blue red}), (\textit{stack blue red}) \rangle \\
& \langle (\textit{pick-up blue}), (\textit{put-down blue}), (\textit{pick-up blue}), (\textit{stack blue red}), \\
& (\textit{unstack blue red}), (\textit{put-down blue}) \rangle \}
\end{aligned}
$$

$$
\begin{aligned}
I_- = \{ \quad & \langle (\textit{pick-up red}), (\textit{put-down red}), (\textit{pick-up red}), (\textit{stack red blue}), \\
& (\textit{unstack red blue}), (\textit{pick-up blue}) \\
& \langle (\textit{pick-up red}), (\textit{put-down red}), (\textit{pick-up red}), (\textit{stack red blue}), \\
& (\textit{pick-up blue}) \rangle \\
& \langle (\textit{pick-up red}), (\textit{stack red blue}), (\textit{put-down red}) \rangle \\
& \langle (\textit{pick-up blue}), (\textit{pick-up red}) \rangle \}
\end{aligned}
$$

4.3.2 DFA Learning

The goal of this step is to learn the state machine related to the planning problem. As we have seen in Section 4.2, this state machine can be represented by a regular grammar. Moreover, since we have both positive and negative samples, it is possible to identify this grammar in the limit using RGI algorithms. Among RGI algorithms, we will use the RPNI[1] algorithm (Oncina and Garcia, 1992). RPNI is very efficient and has all the right properties: (1) it is able to identify in the limit the class of the regular languages; (2) RPNI is locally optimal.

As we have seen before, to identify a regular grammar, we need our sample to be characteristic. It is not possible to construct such a sample a priori. However, if we have a very large sample size, then it is likely that our sample is characteristic; however, one of the conditions for our approach to be effective and to be used in practice is that it requires little training data. We are confronted

[1]We give a complete presentation of this algorithm in Chapter 3.

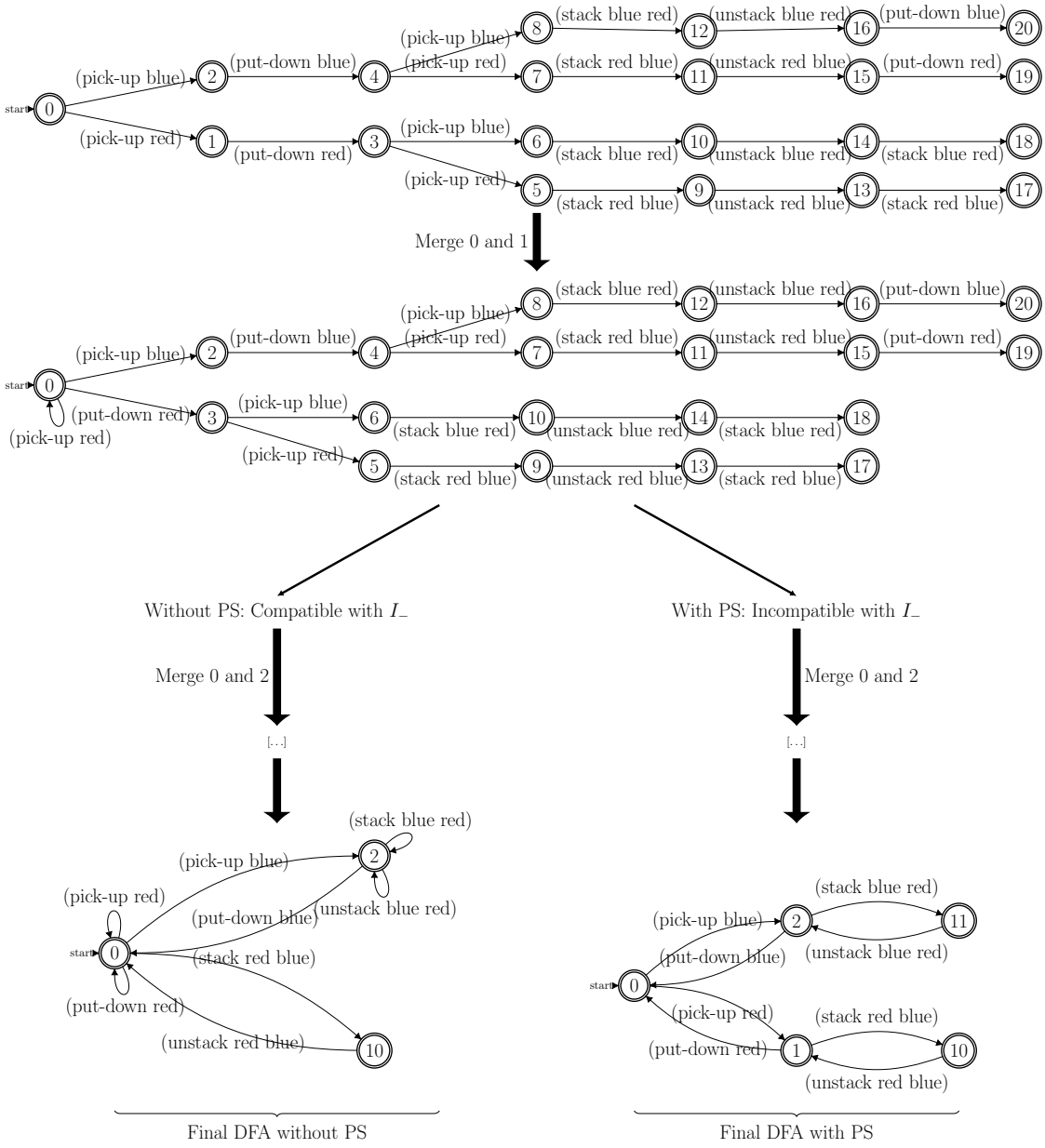


Figure 4.4: Comparison of RPNI execution without and with pairwise sequences.

with a contradiction. To overcome this contraction, we propose to extend the RPNI algorithm using a heuristic. The goal of this heuristic is to allow us to reduce the amount of negative examples. This will allow us to bias the learning of the DFA and to prevent that the DFA from being too general. Indeed, we assume that it is better to have a restrictive DFA, i.e. a DFA that does not recognize all feasible action sequences but limits the recognition of infeasible action sequences, allowing to decrease the risk to learn an action model generating incorrect plans.

Let's consider the definition of an action model. An action is defined using preconditions and effects. These preconditions and effects are defined using a set of logical propositions. These preconditions and effects can be contradictory, i.e. the preconditions of one action will be inconsistent with the effects of another action and vice versa.

Example 4.4 *For example, we have (clear red) $\in$ prec(pick-up red) and (clear red) $\in$ del(stack green red), so after executing (stack green red), the logical proposition (clear red) will always be absent and the action (pick-up red) will never be feasible, no matter what action precedes action (stack green red). In practice, very few actions follow each other. For our Blocksworld problem, the action (stack green red) can only be followed by the actions (unstack green red) and (pick-up blue).*

We will use this property to reduce our negative sample size. More precisely, we propose to force the DFA learned to accept only the sequences defined in I_+ and reject all other unobserved sequences. A simple way to achieve this result is to compute all unobserved pairwise sequences (PS) $(a_i a_j)$ and add them to the set of negative examples I_-. The computation of the set of unobserved PS consists in computing all the possible action pairs from the actions A of the DFA and to subtract the paris present in I_+. Formally, the set of unobserved PS added to I_- is defined as follows:

$$\{(a_i, a_j) \mid (a_i, a_j) \in A^2 \text{ and } \nexists \omega \in I_+ \text{ s.t. } \omega = \langle \omega_1, a_i, a_j, \omega_2 \rangle\} \tag{4.1}$$

Example 4.5 *For example, the pairwise sequence ((pick-up red), (pick-up blue)) will be added to I_- because ((pick-up red), (pick-up b)) is not a subsequence contained in I_+. In contrast, the pairwise sequence ((pick-up red), (put-down red)) will not be added to I_- because ((pick-up red), (put-down red)) is a subsequence of the sequence $\langle$(pick-up red), (put-down red), (pick-up red), (stack red blue), (unstack red blue), (stack red blue)$\rangle$ contained in I_+.*

Example 4.6 *Figure 4.4 compares the execution of the vanilla RPNI algorithm and RPNI improved with PS. We can note that the unobserved sequences prevent the merging of some states. For instance, the merge between states 0 and 1 is no more possible because the sequence $\langle$(pick-up red), (pick-up blue)$\rangle$ have been added to I_- based on Equation 4.1. Therefore, the DFA learned with this improvement accepts only observed sequences decreasing the risk to learn an action model generating incorrect plans.*

Proposition 4.1 *AMLSI adds a polynomial number of sequences.*

Proof 4.1 *In the worst case, $|A|^2$ pairwise sequences are unobserved. Then, AMLSI adds $\mathcal{O}(|A|^2.|I_+|)$ sequences in I_-.*

Note that the unobserved sequences are only used for improving the DFA learning step. For the next steps, we assume that I_- contains only infeasible observed sequences.

4.3.3 Operator generation

Operator generation consists in generating $\delta = (prec, add, del)$ from the learned DFA and generalizes it into a set of PDDL operators Δ. Operator generation is based on three steps:

Precondition generation To learn the preconditions $prec(a)$ of the action a, AMLSI computes the logical propositions that appear in all the states preceding a in Σ:

$$prec(a) = \cap_{s \in preset(a)} \lambda(s) \tag{4.2}$$

Example 4.7 *Let take the DFA in Figure 4.1. We have $\{s_0, s_7, s_9\} \in$ preset(pick-up red). Suppose we have the following, possibly partial and/or noisy, observations:*

- $\lambda(s_0) =$ *{(ontable red), (clear red), (handempty) }*
- $\lambda(s_7) =$ *{(ontable red), (clear red) }*
- $\lambda(s_9) =$ *{(ontable red), (handempty) }*

Then, the precondition is computed as follows: prec(pick-up red) = {(ontable red)}.

Effect generation To learn the positive effects $add(a)$ and the negative effects $del(a)$ of an action a, AMLSI computes the logical propositions that never appear in states before the execution of a, and always present after a execution:

$$add(a) = \cap_{s \in postset(a)} \lambda(s) \setminus prec(a) \tag{4.3}$$

Symmetrically,

$$del(a) = prec(a) \setminus \cap_{s \notin postset(a)} \lambda(s) \tag{4.4}$$

Example 4.8 *Let take the DFA in Figure 4.1. We have $\{s_1, s_{13}, s_{15}\} \in$ postset(pick-up red). Suppose we have the following, possibly partial and/or noisy, observations:*

- $\lambda(s_1) =$ *{(holding red) }*
- $\lambda(s_{13}) =$ *{(holding red) }*
- $\lambda(s_{15}) =$ *{(holding red) }*

Then, the effects is computed as follows: add(pick-up red) = {(holding red)} and del(pick-up red) = {(ontable red)}.

Action generalization Once actions' preconditions and effects have been learned, AMLSI generalizes actions to operators. To do this, AMLSI breaks down the DFA actions into sets of actions that have the same name, number and type of parameters. Each of these sets of actions can be generalized into an operator.

Example 4.9 *For example, assumes that we have two actions (pick-up red) and (pick-up green) that have the same name, number and type of parameters in the learned DFA. The action (pick-up red) is defined as follows:*

$$\begin{array}{ll} prec(\text{pick-up red}) = & \{(ontable\ red)\}. \\ add(\text{pick-up red}) = & \{(holding\ red)\}. \\ del(\text{pick-up red}) = & \{(ontable\ red)\}. \end{array} \tag{4.5}$$

and the action (pick-up green) is defined as follows:

$$\begin{array}{ll} prec(\text{pick-up green}) = & \{(ontable\ green), (clear\ green)\}. \\ add(\text{pick-up green}) = & \{(holding\ green)\}. \\ del(\text{pick-up green}) = & \{(ontable\ green), (clear\ green)\}. \end{array} \tag{4.6}$$

These two actions can be generalized into one operator (pick-up ?x) by using the OI-subsumption (subsumption under Object Identity) (Esposito et al., 2000). The OI-subsumption consists in substituting constants in preconditions and effects by variables by respecting the type of the parameters. In our example, the subsumption of the preconditions of (pick-up red) is {(ontable ?x)} and the subsumption of the preconditions of (pick-up green) is {(clear ?x), (ontable ?x)}.

Then, to compute operators preconditions and effects AMLSI computes the less general preconditions and effects satisfied for all subsumptions. In the end, the operator *(pick-up ?x)* obtained is :

```
(:action pick-up
  :parameters (?x - block)
  :precondition (and (ontable ?x))
  :effect (and (holding ?x)
               (not(ontable ?x)))
```

This generalization method applied for each set of actions that have the same name, number and type of parameters in the learned DFA allows us to ensure that all the necessary preconditions, i.e., the preconditions allowing to differentiate the states where an action is feasible from a state where an action is infeasible, to be all encoded.

We can note that the learned operator is not correct. Indeed, some preconditions and effects are missing. This due to some observations being partial. To deal with it, we perform a refinement step.

4.3.4 Operator refinement

Due to the partial and noisy observations, AMLSI performs a refinement step. This refinement is carried out in three substeps. The first two substeps are

to refine the preconditions and effects to deal with the partial observations and ensure that the learned operators are able to regenerate the induced DFA. Specifically, these two steps ensure that operators can regenerate each DFA transition. The last substep is to refine the operators to deal with noisy observations using a Tabu Search.

Effect refinement This step ensures that the generated operators allow to regenerate the induced regular grammar. We use the observation function to check that for each pair of consecutive actions a and a' in the DFA, the effects of action a applied in the state s satisfy the preconditions of action a'. If it is not the case, we add in the effects of a the propositions satisfying the preconditions of a'.

Example 4.10 *For example, let take the action (stack red green) is an outcoming edge of the state s_1 (see Figure 4.1) and an incoming edge of the state s_4, and the action (unstack red green) is an outcoming edge of the state s_1. Likewise suppose (on red green) $\notin \lambda(s_1)$ and (on red green) $\in$ prec(unstack red green), we put (on ?x ?y) $\in$ add(stack ?x ?y) to ensure that the action (unstack red green) is feasible in state s_4 after applying (stack red green) in state s_1.*

Precondition refinement In this step, we assume like (Yang et al., 2007) that the propositions of the negative effects of an operator must be in its preconditions. Thus, for each negative effect in an operator, we add the corresponding predicate in its preconditions.

Example 4.11 *For example, if (holding ?x) $\in$ del(stack ?x ?y) then we put (holding ?x) $\in$ prec(stack ?x ?y).*

Moreover, since effect refinements depend on preconditions and precondition refinements depend on effects, we repeat these two steps until convergence, i.e., no more precondition or effect is added. The refinement process converges because (1) the preconditions and the effects can be added only once during the whole refinement process of an operator and (2) the number of preconditions and effects that could be added is limited by the number of effects and preconditions of the next action in the DFA.

Proposition 4.2 *Effects and preconditions refinement steps converge in a polynomial number of iterations.*

Proof 4.2 *These two steps add effects and preconditions without removing any. In the worst case, all possible preconditions and effects are added. Then, these steps converge in $\mathcal{O}(|A|.|S|)$ iterations.*

Tabu Search The refinement step previously described is able to find most of the preconditions and the effects of the operators even with partial observations. However, this refinement does not prevent to remove relevant or add irrelevant preconditions and effects when observations are noisy. For example, suppose that the action *(stack green blue)* is applicable in two states s and s' in the DFA, i.e., s and $s' \in$ *preset(stack green blue)*. Now, suppose that the observation function $\lambda(s)$ returns *(clear blue)* as $true$ and $\lambda(s')$ returns *(clear blue)* as $false$ due to the noise. In that case, *(clear blue)* is not included in the preconditions of *(stack green blue)* even if it has to. Thus, after generalisation, the operator *(stack ?x ?y)* will not have as precondition *(clear ?y)*.

To deal with this problem, we propose to use Tabu Search (Glover and Laguna, 1997). Tabu Search is a classical meta-heuristic search method employing local search methods used for mathematical optimization. The idea is to explore variants of the operators set learned in the previous step by adding and removing preconditions and effects. At each steps only variants improving the set of learned operators are kept until a local minimum is found. To determine whether one variant is better than another it is necessary to define an evaluation function. This evaluation function is called a fitness function. In practice, a variant Δ of an operators set is better than an other one given a positive and a negative set of observations if : (1) Δ accepts more positive observations, (2) Δ rejects more negative observations and (3) the state sequences produced by applying the transition function γ_Δ on the positive actions sequences observed in I_+ violate fewer preconditions and effects than the sequences produced with the other. Formally, the fitness function used by AMLSI to evaluate a candidate variant Δ given the observations sets I_+ and I_- is defined as follows:

$$f(\Delta, I_+, I_-) = \sum_{\omega \in I_+} accept(\Delta, w) + \sum_{\omega \in I_-} reject(\Delta, w) + \sum_{w \in I_+} \sum_{s \in \gamma_\Delta(s_0, \omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)| \quad (4.7)$$

where:

$$accept(\Delta, w) = 1 \text{ if } \forall s \in \gamma_\Delta(s_0, w)\ s \in \lambda(s)\ 0\ otherwise. \quad (4.8)$$

$$reject(\Delta, w) = 1 \text{ if } \exists s \in \gamma_\Delta(s_0, w)\ s \notin \lambda(s)\ 0\ otherwise. \quad (4.9)$$

Proposition 4.3 *The complexity to compute the fitness score for all candidate variants Δ is polynomial.*

Proof 4.3 *First of all, at each step of the Tabu Search, AMLSI tests $\mathcal{O}(|A|.|S|)$ candidates. Then, the complexity to compute the acceptation (resp. rejection) of positive (resp. negative) samples I_+ (resp. I_-) is $\mathcal{O}(|I_+|.|S|)$ (resp. $\mathcal{O}(|I_-|.|S|)$). Finally, the complexity to compute the number of preconditions and effects violated is $\mathcal{O}(|I_+|.|S|)$. To conclude, the complexity to compute the fitness score for all candidate variants Δ is $\mathcal{O}(|A|.|S|^2.(|I_+| + |I_-|))$.*

Once the Tabu Search is done, i.e. a local optimum of the fitness score $f(\Delta, I_+, I_-)$ is reached, we repeat all the three refinement steps (effect and precondition refinement plus Tabu Search) until convergence (see Figure 4.5).

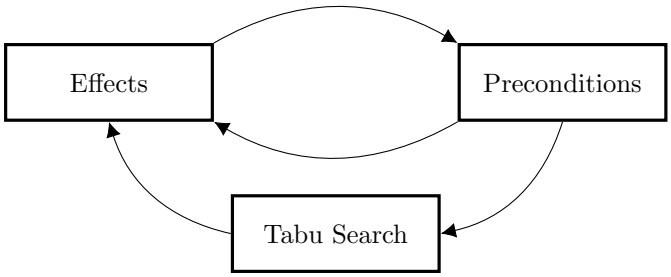


Figure 4.5: Refinement Steps.

4.4 Formal properties

In this section we will give the formal properties of the AMLSI algorithm. First, we will show that the DFA Learning algorithm identifies the language $\mathcal{L}(P)$ in the limit.

Then, we will show that STRIPS Domain Learning is sound. STRIPS domain learning is sound means that, for a given observation set Ω, AMLSI learns a STRIPS Domain generating the given observation set. We will prove the STRIPS Domain Learning soundness when observations are complete and noiseless. When observations are partial and/or noisy it is possible that the learning is not sound. However, we will show experimentally in Section 4.5 that AMLSI keeps high performances when observations are partial and/or noisy despite the lack of soundness.

4.4.1 DFA Learning Formal Properties

We know that the RPNI algorithm identifies the class of regular language in the limit (Dupont, 1996). As the language $\mathcal{L}(P)$ is regular, RPNI algorithm identifies this language. So, there exists a sample $\Omega^c = \{I^c_+, I^c_-\}$ such that:

- $RPNI(I^c_+, I^c_-) \rightarrow \Sigma$ with $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$.
- $\forall I'_+, I'_-$ such that $\forall \omega \in I'_+\omega \in \mathcal{L}(P)$ and $\forall \omega \in I'_-\omega \notin \mathcal{L}(P)$ then $RPNI(I^c_+, I^c_-) \equiv RPNI(\{I^c_+ \cup I'_+\}, \{I^c_- \cup I'_-\})$.

The sample Ω^c is the *characteristic* sample of $\mathcal{L}(P)$. See Chapter 3 for more details.

As we have seen in Section 4.3.2, the RPNI algorithm is improved with unobserved sequences. More precisely, AMLSI adds in the negative sample I_- all unobserved pairwise sequences $(a_i a_j)$. Let I^{PS}_- be the set of unobserved pairwise sequences.

Property 4.1 *$RPNI(I^c_+, I^c_-) \equiv RPNI(I^c_+, \{I^c_- \cup I^{PS}_-\})$ if and only if $\forall \omega \in I^{PS}_-\omega \notin \mathcal{L}(P)$*

I^{PS}_- contains all pairwise sequences $(a_i a_j)$ such that: $(a_i, a_j) \in A^2$ *and* $\nexists \omega \in I^c_+ \; s.t. \; \omega = (\omega_1, a_i, a_j, \omega_2)\}$. So, to ensure that the property 4.1 is checked we

need that all pairwise sequences present in the language $\mathcal{L}(P)$ are present in the positive sample I_+^c. Formally:

$$\forall (a_i, a_j) \in A^2 \; s.t. \; \exists \omega = (\omega_1, a_i, a_j, \omega_2) \in \mathcal{L}(P) \implies \exists \omega' = (\omega'_1, a_i, a_j, \omega'_2) \in I_+^c \quad (4.10)$$

Theorem 4.1 *DFA Learning algorithm identifies the language $\mathcal{L}(P)$ in the limit.*

Proof 4.4 *AMLSI learns Σ using the RPNI algorithm improved with unobserved sequence. We know that RPNI identifies int the limit $\mathcal{L}(P)$. Also, thanks the property 4.1, we know that RPNI improved with unobserved sequence identifies in the limit $\mathcal{L}(P)$ if and only if the positive sample satisfies the constraint given by Equation 4.10. As there exists a characteristic sample for the language $\mathcal{L}(P)$ then the DFA Learning algorithm identifies the language $\mathcal{L}(P)$ in the limit.*

4.4.2 Domain Learning Soundness

We will now show that the PDDL domain learned by AMLSI, when $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$ and when observations are complete and noiseless, is a PDDL domain Δ_P able to generate $\mathcal{L}(P)$ and the observations function λ. AMLSI returns the PDDL domain refined by the refinement steps (see Section 4.3.4). We must therefore show that the different refinement steps allow to learn a PDDL Domain Δ such that Δ is able to generate $\mathcal{L}(P)$ and the observations function λ. More precisely, we will prove that when the Tabu Search reaches the global maxima, AMLSI refinement stops and returns a PDDL domain able to generate $\mathcal{L}(P)$ and the observations function λ.

Let's us assume there exists a PDDL Domain Δ_P such that Δ_P is able to generate the regular language $\mathcal{L}(P)$ and the observation function λ.

Lemma 4.1 *If $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$, when observations are complete and noiseless, Δ_P maximizes the fitness function $f(\Delta, I_+, I_-)$ of the Tabu Search.*

Proof 4.5 *First of all, as Δ_P generates the regular language $\mathcal{L}(P)$ we have:*

$$\Delta_P = argmax_\Delta(\sum_{\omega \in I_+} accept(\Delta, w)) \quad (4.11)$$

$$\Delta_P = argmax_\Delta(\sum_{\omega \in I_-} reject(\Delta, w)) \quad (4.12)$$

Indeed, if Δ_P generates the regular language $\mathcal{L}(P)$ then $\forall \omega \in I_+$, Δ_P can generate ω. So, $\sum_{\omega \in I_+} accept(\Delta, w) = |I_+| = Max_\Delta(\sum_{\omega \in I_+} accept(\Delta, w))$. In the same way, if Δ_P generates the regular language $\mathcal{L}(P)$ then $\forall \omega \in I_-$, Δ_P cannot generate ω. So, $\sum_{\omega \in I_-} reject(\Delta, w) = |I_-| = Max_\Delta(\sum_{\omega \in I_-} reject(\Delta, w))$.

Then, as Δ_P generates the observation function λ we have:

$$\Delta_P = argmax_\Delta(\sum_{w \in I_+} \sum_{s \in \gamma_\Delta(s_0, \omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)|) \quad (4.13)$$

Indeed, if Δ_P generates the observation function λ then $\sum_{w \in I_+} \sum_{s \in \gamma_{\Delta_P}(s_0,\omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)| = \sum_{w \in I_+} \sum_{s \in \gamma_{\Delta_P}(s_0,\omega)} |\lambda(s)| - 0 = Max_\Delta(\sum_{w \in I_+} \sum_{s \in \gamma_\Delta(s_0,\omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)|)$

Finally, we have

$$\begin{aligned} Max_\Delta(f(\Delta, I_+, I_-)) &= Max_\Delta(\sum_{\omega \in I_+} accept(\Delta, w) + \sum_{\omega \in I_-} reject(\Delta, w) + \sum_{w \in I_+} \sum_{s \in \gamma_\Delta(s_0,\omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)|) \\ &= Max_\Delta(\sum_{\omega \in I_+} accept(\Delta, w)) + Max_\Delta(\sum_{\omega \in I_-} reject(\Delta, w)) + Max_\Delta(\sum_{w \in I_+} \sum_{s \in \gamma_\Delta(s_0,\omega)} |s \cap \lambda(s)| - |s \backslash \lambda(s)|)) \end{aligned} \tag{4.14}$$

And thanks to Equations 4.11, 4.12 and 4.13 we know that $f(\Delta_P, I_+, I_-) = Max_\Delta(f(\Delta, I_+, I_-))$. So, $\Delta*$ maximizes the fitness function of the Tabu Search when observations are complete and noiseless.

Lemma 4.2 *When the Tabu Search reaches the global maxima, the refinement step converges if $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$ and observations are complete and noiseless.*

Proof 4.6 *When the Tabu Search reaches the global maxima, the function $f(\Delta, I_+, I_-)$ is maximized. So, all transitions in the DFA Σ are feasible, so the Effect Refinement Step (see Section 4.3.4) does not add new effects. Also, all preconditions are encoded in Δ, so the Precondition Refinement Step (see Section 4.3.4) does not add new preconditions. So, when Tabu Search reaches the global maxima, the AMLSI refinement step converges.*

Theorem 4.2 *If $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$, when observations are complete and noiseless, the refined PDDL domain Δ learned by AMLSI is able to generate $\mathcal{L}(P)$ and the observations function λ.*

Proof 4.7 *Thanks the Lemma 4.1, we know that the global maxima of the Tabu Search is a PDDL domain able to generate $\mathcal{L}(P)$ and the observations function λ when $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$ and when observations are complete and noiseless. Also, thanks the Lemma 4.2 we know that when the Tabu Search reaches the global maxima, then the AMLSI refinement step converges. So, if $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$, when observations are complete and noiseless, the refined PDDL domain Δ learned by AMLSI is able to generate $\mathcal{L}(P)$ and the observations function λ.*

Finally, we can state the following Theorem:

Theorem 4.3 *Soundness. When the observation set Ω is characteristic and when observations are complete and noiseless, AMLSI learns a PDDL domain able to generate the language $\mathcal{L}(P)$ and the observation function λ.*

Proof 4.8 *Thanks the Theorem 4.1 we know that AMLSI learns a DFA Σ such that $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$ when the observation set Ω is characteristic. Then, if $\mathcal{L}(\Sigma) \equiv \mathcal{L}(P)$, when observations are complete and noiseless AMLSI learns a PDDL domain Δ which is able to generate the regular language $\mathcal{L}(P)$ and the observation function λ.*

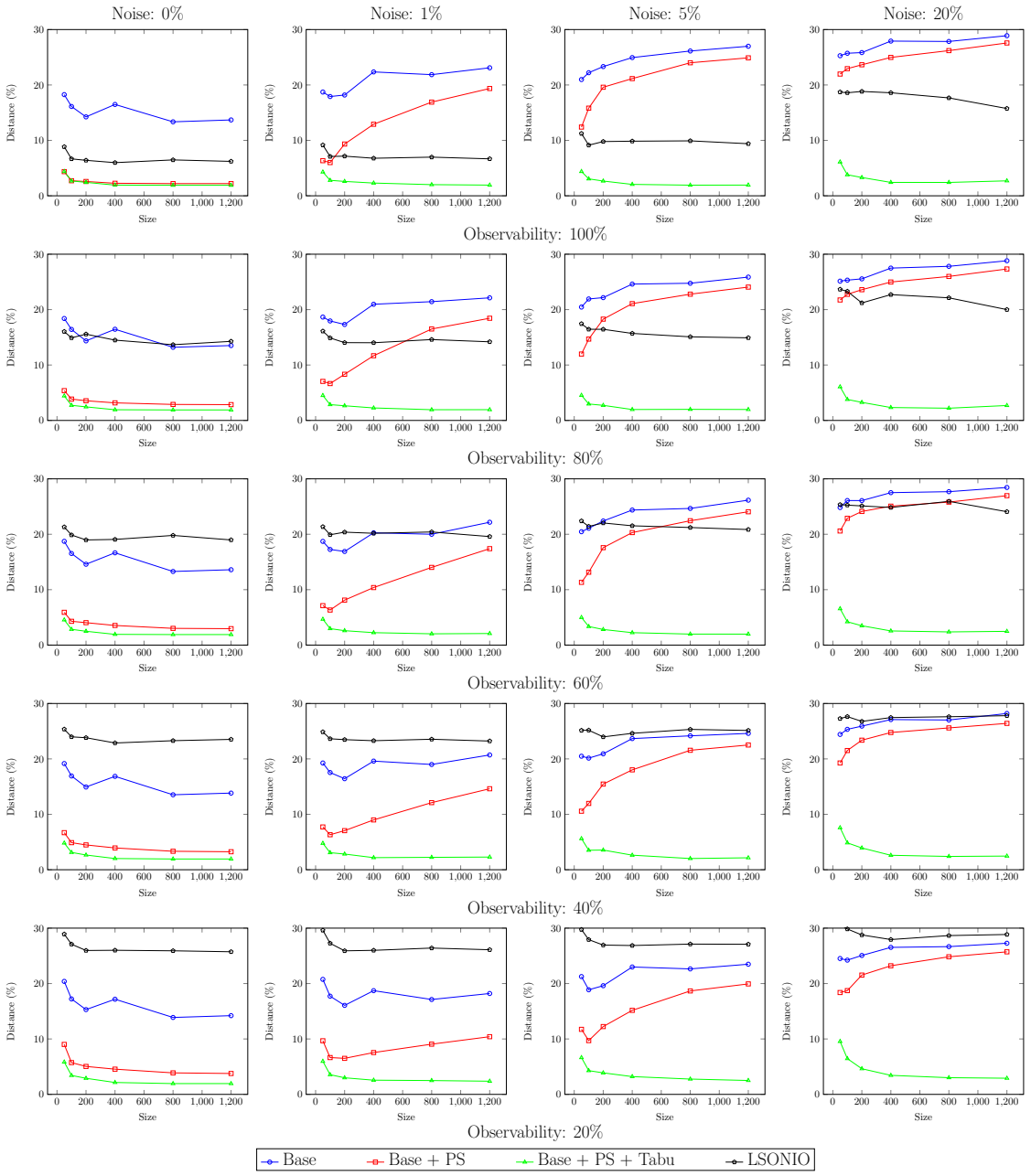


Figure 4.6: Average performances in terms of syntactical distance of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

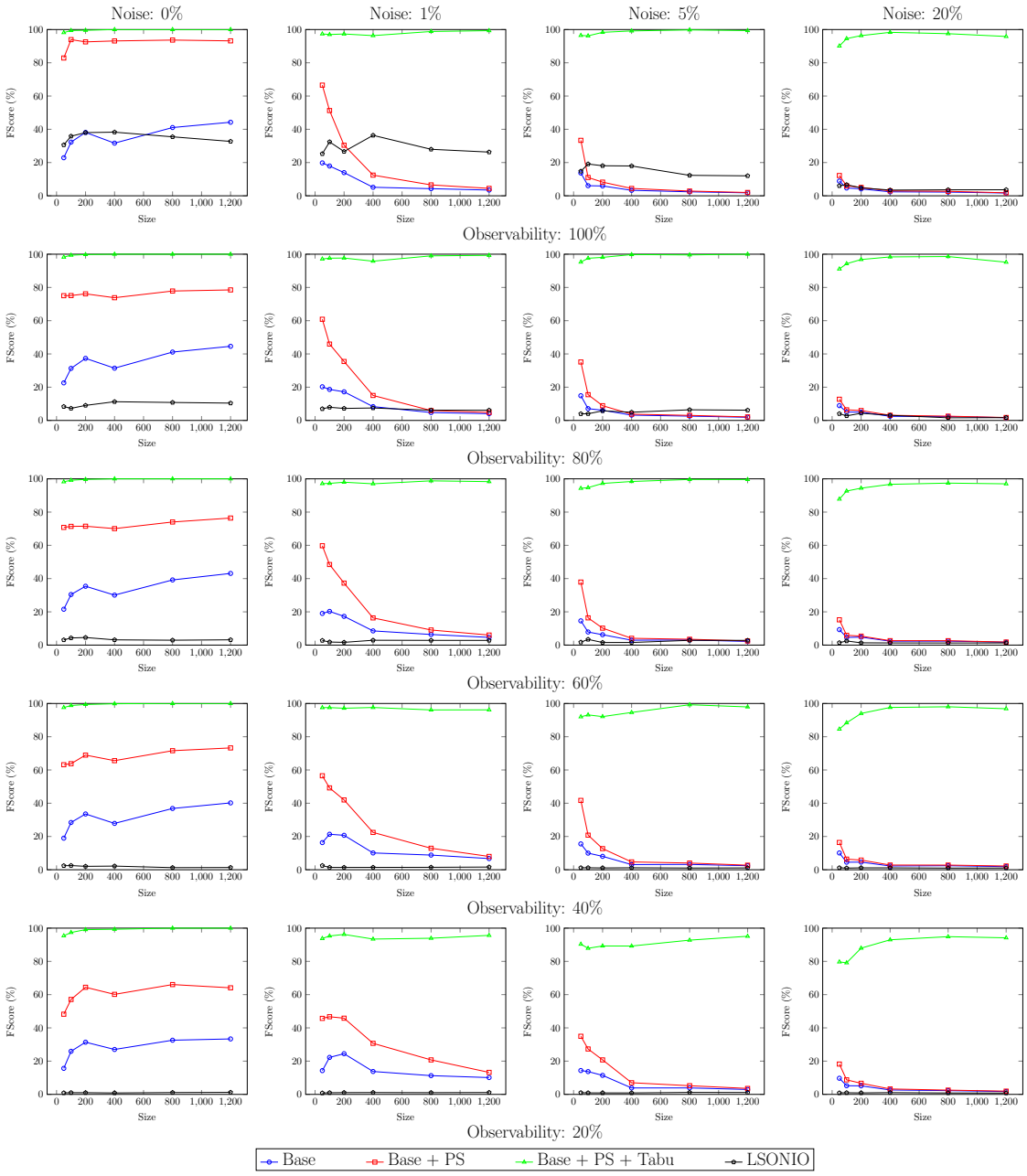


Figure 4.7: Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

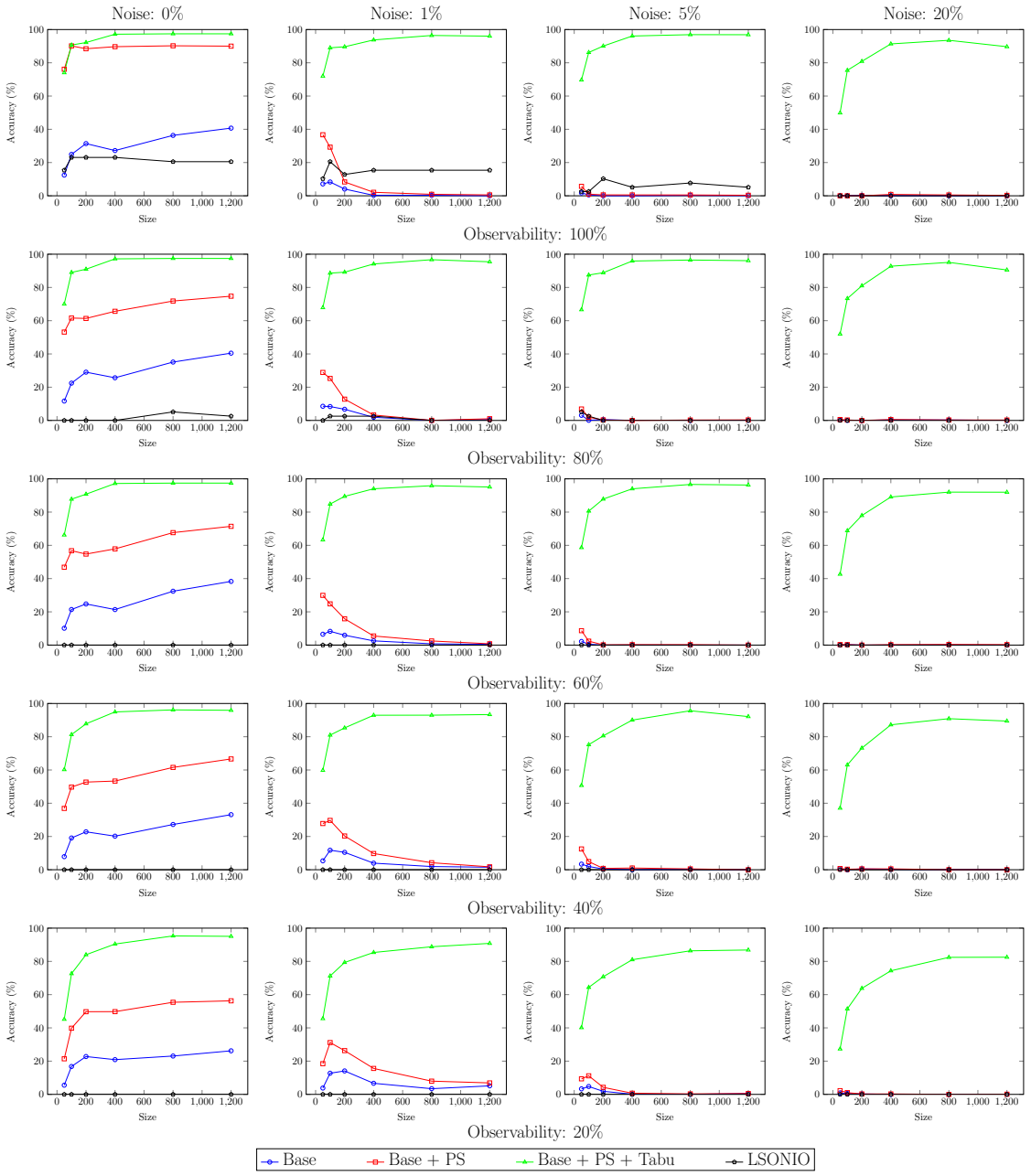


Figure 4.8: Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

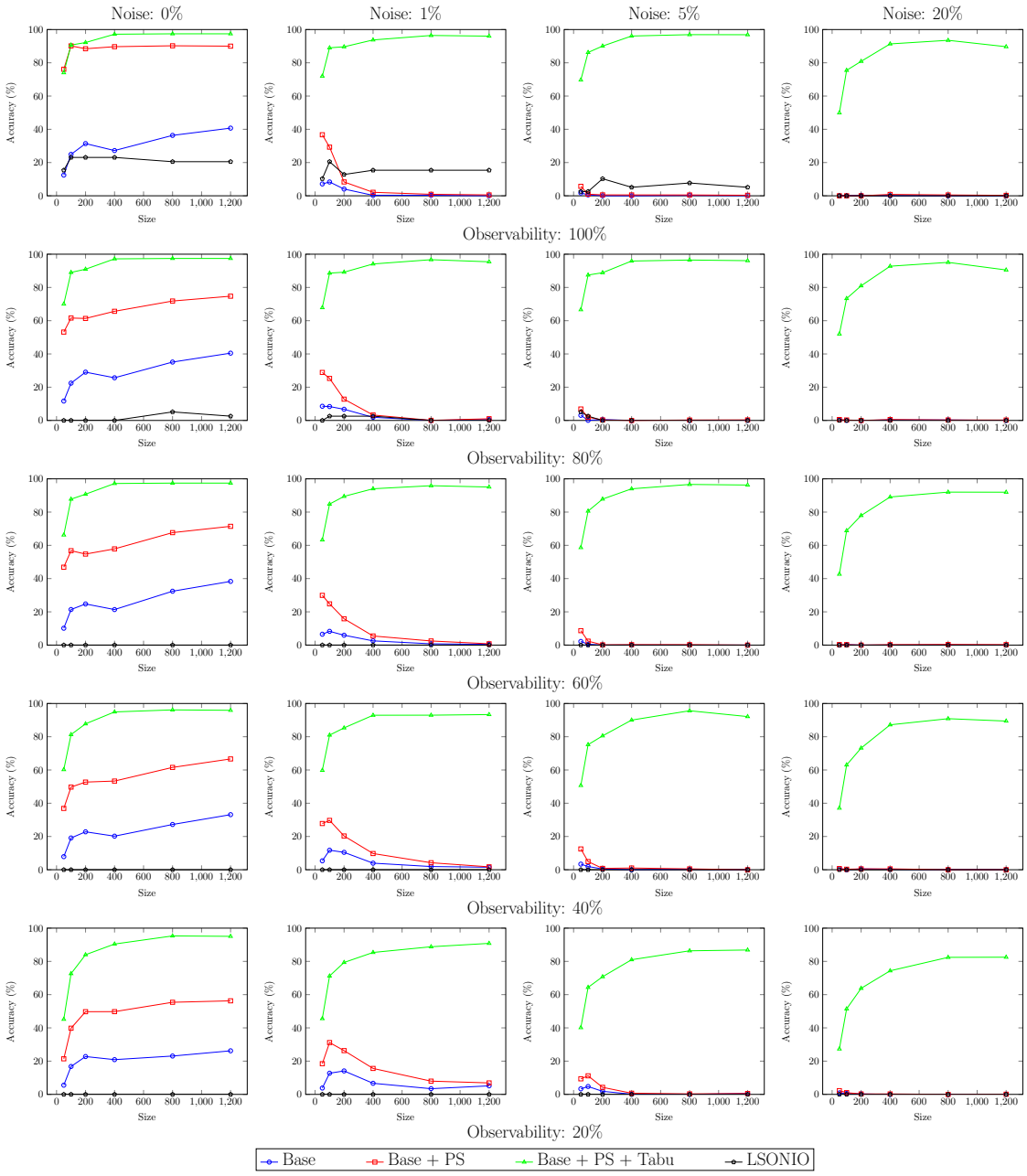


Figure 4.9: Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of actions.

Domain	*#Operators*	*#Predicates*	E_+	E_-	ω_+	ω_-
Blocksworld	4	5	100	32546	49.8	33.8
Gripper	3	4	100	13163	51.3	33.9
Hanoi	4	7	100	34600	50.3	33.7
N-Puzzle	1	3	100	36626	49.9	33.7
Peg-Solitaire	3	4	100	14508	6.9	5.3
Parking	4	5	100	64963	50.6	34.0
Zenotravel	5	5	100	18154	50.4	33.9
Sokoban	2	4	100	40302	50.2	33.8
Visit All	4	7	100	16702	50.9	35.7
Elevator	4	6	100	13122	51.0	35.7
Spanner	4	6	100	4628	7.0	5.1
Logistics	6	3	100	31622	49.7	32.9
Floortile	6	10	100	48773	51.0	37.1

Table 4.1: Benchmark characteristics (from left to right): the number of operators, the number of predicates, the average size of the E_+ and the E_- testing dataset, and the average length of the positive (resp. negative) testing sequences $\omega_+ \in E_+$ (resp. $\omega_- \in E_-$).

4.5 Experiments

4.5.1 Experimental setup

Our experiments are based on 13 STRIPS-Compliant IPC[2] benchmarks: Blocksworld, Gripper, Hanoi, N-Puzzle, Peg Solitaire, Parking, Zenotravel, Sokoban, Visit All, Elevator, Spanner, Logistics and Floortile. Table 4.1 shows our experimental setup[3].

We test each IPC action model with 3 different instances over ten runs, and we use ten randomly generated seeds for each run. Also, we generate partial observations by randomly removing a fraction of the propositions of the states, and we generate noise by changing the value of a fraction of the observable propositions. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino, 2018) was used to generate the benchmark data.

4.5.2 Evaluation metrics

Three metrics are used for the evaluation: the *syntactical error* (Zhuo et al., 2010b) that computes the distance between the original action model and the action model learned, the *accuracy* (Zhuo et al., 2013) that expresses the capability

[2] `https://www.icaps-conference.org/competitions/`

[3] Experimental setup are publicly available at `https://github.com/maxencegrand/AMLSI`

of the action model learned to solve new problems (without proofreading). Even though the syntactical error is the most used metric in the literature, we argue that the accuracy is the most important metric *in practice* for planning because it measures to what extent a learned action model is useful. Indeed, it often happens that one missing precondition or effect, which amounts to a small syntactical error, makes the learned action model unable to solve planning problems. Finally, the last metric is the *FScore* (van Rijsbergen, 1979) that expresses the capability of the learned action model to generate the grammar related to the planning problem.

Formally, the syntactical error $error(o)$ for an operator is the Hamming distance between the learned operator and the ground truth operator, i.e. the number of extra or missing predicates in the preconditions $prec(o)$, the positive effects $add(o)$ and the negative effects $del(o)$ divided by the total number of possible predicates. By extension, the syntactical error for a action model composed of a set of operator O is:

$$E_\sigma = \frac{1}{|O|} \sum_{o \in O} error(o)$$

Then, FScore $= \frac{2.P.R}{P+R}$ where R is the recall, i.e. the rate of sequences e accepted by the original IPC action model that are successfully accepted by the learned action model, computed as $R = \frac{|\{e \in E^+ \mid accept(\delta, e)\}|}{|E^+|}$, and P is the precision, i.e. the rate of sequences e accepted by the learned action model that are also accepted by the original IPC action model, computed as $P = \frac{|\{e \in E^+ \mid accept(\delta, e)\}|}{|\{e \in E^+ \mid accept(\delta, e)\} \cup \{e \in E^- \mid accept(\delta, e)\}|}$. The test sets E^+ and E^- used to compute the FScore are generated by random walks.

Finally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learned action model, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. STRIPS problems are solved with Fast Downward v19.06 (Helmert, 2006). For each experiment, plan validation is done with VAL (Howey and Long, 2003), which is used in the IPC competitions. In addition to the Accuracy we report the IPC score in order to compute the quality of generated plans. The score of a action model on a solved problem is the ratio between the length of a reference plan, i.e. a plan generated by the reference IPC action model, and the length of the plan generated by the learned action model. The score on an unsolved problem is 0. The score of a learned action model is the sum of its scores for all problems.

4.5.3 Discussion

Figures 4.6, 4.7, 4.8 shows the average performance of AMLSI and LSONIO obtained on the 13 action models of our benchmarks when varying the training dataset size. The size of the training set is indicated in number of actions. AMLSI

and LSONIO are tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%. We have three variants of AMLSI: (B) *Base:* DFA learning is done without Pairwise Sequences (PS) and without Tabu Search, (B+PS) *Base + PS:* DFA learning is done with PS but without Tabu Search, and (B+PS+Tabu) *Base + PS + Tabu:* DFA learning is done with PS and with Tabu Search during the refinement step.

Comparison with LSONIO We observe that AMLSI outperforms LSONIO whatever the size of the learning dataset in terms of accuracy or in terms of syntactical distance. We also observe that AMLSI needs very little data to obtain a relatively large accuracy (almost 70% with only a learning dataset of 400 actions) in the most difficult scenario.

Ablation study The Base+PS variant is more robust to partial observations than the Base variant of AMLSI. This is due to the fact that DFA learned with PS are generally better than action model learned without PS. More precisely, DFA learned with PS generally have fewer states and fewer transitions. This allows for fewer false transitions which makes it easier to learn effects and preconditions. However, when observations are noisy, the Base+PS variant is not able to learn action models accurate enough to be used for planning whatever the level of observability. Only the Base+PS+Tabu variant is both robust to partial and noisy observations. Our ablation study confirms that adding unobserved Pairwise Sequences improves the learning of the DFA, and makes AMLSI more robust to partial observations while refining the preconditions and the effects by using a Tabu Search allows AMLSI to learn accurate action models with a high level of noise.

Finally, we observe that the plans generated with the learned action models are generally longer than plans generated with the IPC action models, even with optimal accuracy (ipc < 20). When the accuracy is not optimal, we notice that the IPC score is close to the ratio between the IPC score with optimal accuracy and the rate of solved problems, this implies that even when the accuracy is not optimal the plans are not much longer than the original plans.

4.6 Conclusion

In this chapter, we have presented the AMLSI approach. The AMLSI approach learns action models from interactions with the environment in which the agent will have to solve planning problems. The AMLSI approach is divided in four steps: (1) AMLSI generates a set of observations, (2) AMLSI learns the DFA related to the planning problem using a variant the RPNI algorithm, (3) AMLSI learns from the DFA the action model and express it as a PDDL planning domain and (4) AMLSI refines the action model. Also, we have given the formal properties of our approach. More precisely, we have shown that our variant

of the RPNI algorithm preserves the identification in the limit property of the vanilla RPNI algorithm. Also, we have shown the soundness of the AMLSI approach.

Moreover, we can draw several lessons from our experimentation. First, we show that the AMLSI approach is highly accurate whatever the level of partiality and noise of the observations. Moreover, we have seen that AMLSI requires little training dataset. Also, we have shown that AMLSI outperforms LSONIO, the closest state-of-the-art approach. Finally, thanks to the ablation study, we were able to show the usefulness of each step of our approach.

While these results are encouraging, the AMLSI approach has several limitations. First of all, we can observe that AMLSI requires little training dataset to be accurate. Also, the larger the data the better the results. However, as we have seen in the previous part, acquiring data is a difficult and costly process. Although AMLSI requires little training data, it is not possible to know a priori how much training data is required to learn a model. It is therefore important to be able to incrementally learn the action model and to have a criterion to know that the action model is learned and to stop the learning process. This is what we will see in the next chapter. Also, the AMLSI approach learns STRIPS action models, but we have seen in the previous part that STRIPS action models are based on assumptions that are too restrictive to be used in real-world applications. In the following chapters, we will propose two extensions for AMLSI to learn less restrictive action models.

Chapter 5

IncrAMLSI: Incremental Action Model Learning

Contents

5.1 Introduction

In the previous chapter, we have presented the AMLSI approach. We have seen that AMLSI successfully address the accuracy issue even when observations are partial and/or noisy. Also, we have seen that AMLSI requires few data to be accurate. However, in practise, data acquisition is a long term evolutive process: in real world applications, training data become available gradually over time, are difficult and costly to obtain, as for instance, Mars Exploration Rover operations (Bresina et al., 2005) or robot fleets for offshore missions (Carreno et al., 2020). Moreover, in practice, it is important to be able to update learned action models to new incoming data without restarting the learning process from scratch. Finally, it is also important to know when to stop learning in order to know when to stop the data acquisition process and minimize the amount of data acquired: a convergence criterion is required. In this chapter, we

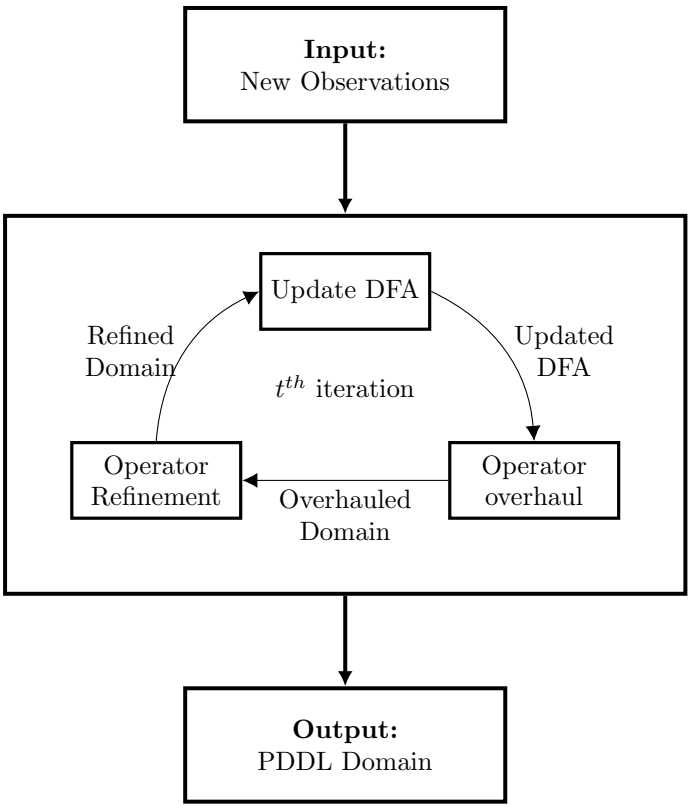


Figure 5.1: Incremental Learning Process.

propose an incremental version of AMLSI called IncrAMLSI (Grand et al., 2021) to overcome these issues.

The rest of this chapter is organized as follows. In Section 5.2 we detail the incremental learning process and give the formal properties of this incremental extension. In Section 5.3 we show that our incremental allows to efficiently consume training data. The problem statement being the same as in the previous chapter, we will not present it in this chapter.

5.2 Incremental Learning

An overview of IncrAMLSI is shown in Figure 5.1. IncrAMLSI learns incrementally the action model from incoming data and does not restart from scratch when new data become available at each iteration t. More precisely, IncrAMLSI consists in incrementally updating the PDDL domain Δ^t with the new incoming training datasets Ω^{t1} available at iteration t to produce the PDDL domain Δ^{t+1}. This algorithm is made up of three steps:

1. *Update of the DFA* with the modified version of RPNI algorithm in order to accept I^t_+ and to reject I^t_- at iteration t (see Section - 4.3.2).

[1] Ω^t contains all observations since the beginning of the learning process.

Algorithm 2: Overhaul_Operator(Δ^t, I^t_+, I^t_-).

```
1  Δ^t_new ← operatorGeneration();
2  backtrackInfeasibleEffects(Δ^t);
3  backtrackInfeasiblePreconditions(Δ^t);
4  repeat
5  |  repeat
6  |  |  backtrackPreconditions(Δ^t);
7  |  |  backtrackNegativeEffects(Δ^t);
8  |  until acceptAll(Δ^t, I^t_+);
9  |  backtrackEffects(Δ^t)
10 until acceptAll(Δ^t, I^t_+);
11 Δ^{t+1} ← merge(Δ^t, Δ^t_new);
12 return Δ^{t+1}
```

2. *Overhaul of the PDDL operators* in order to add new operators and remove preconditions and effects that are no longer compatible with I^t_+ and I^t_-, and the updated DFA at iteration t (see Section 5.2.1).

3. *Refinement of the PDDL operators* as in AMLSI to deal with noisy and partial states in I^t_+ and I^t_-, and to produce the new domain Δ^{t+1} (see Section 4.3.4).

The incremental process is operated each time new training datasets are input and until convergence of the PDDL domain.

5.2.1 Operator Overhaul

When at iteration t new positive and negative datasets are integrated and the DFA is updated, it is possible that the domain previously learned Δ^t is no longer compatible with it. There are two possibilities:

1. Some operators may not have been generated in the previous datasets, so IncrAMLSI have to add them

2. Some effects and preconditions in Δ^t have to be removed to match the updated DFA

Algorithm 2 describes the procedure to compute Δ^{t+1} from Δ^t, I^t_+, and I^t_- at iteration t. First of all, IncrAMLSI generates new operators in a PDDL domain Δ^t_{new} (line 1). Then, IncrAMLSI removes all extra preconditions and effects in Δ^t (line 2 - 10). Finally, IncrAMLSI merges Δ^t_{new} with Δ^t (line 11) by making the union sets of the preconditions and the effects of the operators.

First of all, functions *backtrackInfeasibleEffects* and *backtrackInfeasiblePreconditions* (line 2, 3) remove the effects and preconditions infeasible in the DFA. More precisely, *backtrackInfeasibleEffects* removes positive

and negative effects that does not respect the conditions given by Equations 4.3 and 4.4 in Section - 4.3.3.

Example 5.1 *For example, suppose we have (holding ?x) $\in$ add(unstack ?x ?y), and there are no states in the current DFA where this effect appears, then (holding ?x) is removed.*

Likewise, *backtrackInfeasiblePreconditions* removes extra preconditions that does not validate Equation 4.2.

Example 5.2 *For example, suppose we have (holding ?y) $\in$ prec(unstack ?x ?y) and $\exists s \in \mu_{preset}$(unstack green blue) such that (holding blue) $\notin \lambda(s)$, then (holding ?y) is removed.*

Then, as observed states are noisy and partial, it is possible that some extra preconditions and effects are not present in the current DFA. Therefore lines 4 - 10 remove extra preconditions and effects independently of this DFA. First of all, *backtrackPreconditions* (line 6) removes all the preconditions that are not compatible with I^t_+.

Example 5.3 *For instance, suppose we have this positive sample:*

$$\langle \textit{(pick-up green), (put-down green), (pick-up green), (stack green blue), (unstack green blue)} \rangle \in I^t_+$$

and, when we execute it with the current domain Δ^t we have:

$$\textit{(holding green)} \notin \lambda(\gamma(s_0, \langle \textit{(pick-up green), (put-down green), (pick-up green), (stack green blue)} \rangle))$$

i.e. in the state before the last action (unstack green blue), the proposition (holding green) is absent and suppose that (holding ?x) $\in$ prec(unstack ?x ?y). Thus we remove this unsatisfied precondition (holding ?x) from the operator (unstack ?x ?y).

Then, with *backtrackNegativeEffects* (line 7), we remove all the negative effects that does not satisfy PDDL syntactical constraints (Yang et al., 2007) (e.g., negative effects must be in the preconditions etc.) We repeat these two functions until the domain accepts all the positive samples (line 8), i.e. the domain is able to regenerate all the positive samples I^t_+.

The next step (line 9) in the algorithm is to remove all the effects that are not compatible with the negative samples I^t_- with function *backtrackEffects*.

Example 5.4 *Suppose that two samples share the same prefix (unstack green blue):*

$$\langle \textit{(unstack green blue), (put-down green)} \rangle \in I^t_+.$$

and

$$\langle \textit{(unstack green blue), (pick-up green)} \rangle \in I^t_-.$$

Also, suppose that, wrongly, (holding ?y) $\in$ add(unstack ?x ?y), and, correctly, (holding ?x) $\in$ prec(put-down ?x). And suppose that the current domain

Δ^t can generate $\langle$(unstack green blue)$\rangle$. The extra effect (holding green) $\in$ add(unstack green blue) allows to the precondition of the unfeasible action (put-down blue) to be satisfied, and (holding blue) is not in the preconditions of the feasible action (put-down green). Thus, we remove the effect (holding ?y) from the operator (unstack ?x ?y).

As the *backtrackEffects* function removes effects, it is possible that some positive samples can no longer be generated by the current domain Δ^t. We therefore have to repeat these three backtracking functions until the domain accepts all the positive samples (line 10). Indeed, repeating these functions until the domain accepts all the positive samples allows to ensure that the last removed effects do not impact the acceptation of the positive samples (termination is ensured by only allowing precondition and/or effect withdrawals).

5.2.2 Convergence

IncrAMLSI stops when, after a given number T of iterations, the domain Δ^t checks three conditions: (1) Δ^t accepts all the positive samples I^t_+, (2) Δ^t rejects all the negative samples I^t_-, and (3) $\Delta^t = \Delta^{t+1}$ during T iterations. We will show experimentally (see section 5.3) that IncrAMLSI converges with partial and noisy observations. However, it is possible to show formally that it is true when state observations are complete and noiseless and $T \to \infty$. More precisely, we can prove that IncrAMLSI converges to a domain Δ_P able to generate the language $\mathcal{L}(P)$ and the observation function λ (see Chapter 4).

Lemma 5.1 *When observation are complete and noiseless and when the observation set $\Omega^t = \{I^t_+, I^t_-\}$ is characteristic, then the overhaul operators step returns a domain Δ^{t+1} with no extra preconditions and effects w.r.t the domain Δ_P.*

Proof 5.1 *First of all, as the observation set is characteristic we know, thanks to Theorem 4.1 (see Chapter 4), that $\mathcal{L}(\Sigma^t) = \mathcal{L}(P)$. Also, as observations are complete and noiseless, Δ^t_{new} does not contain any extra preconditions and effects. Then, as functions backtrackInfeasibleEffects and backtrackInfeasiblePreconditions remove all preconditions and effects incompatible with equations of the operators generation steps, then all extra preconditions and effects are removes. Finally, as all extra preconditions and effects are removed and as Δ^t_{new} does not contain any extra preconditions and effects then Δ^{t+1} does not contain any extra preconditions and effects*

Theorem 5.1 *When observation are complete and noiseless and when $T \to \infty$, IncrAMLSI converges to Δ_P.*

Proof 5.2 *First of all, if $T \to \infty$ then the observation set Ω^T is characteristic. Also, thanks the Lemma 5.1 we know that, before refinement, the domain Δ^T does not contain any extra preconditions and effects. Finally, thanks the Theorem 4.2 (see Chapter 4) we know that the domain returned by the refinement step is Δ_P the observation set Ω^T*

Domain	*#Operators*	*#Predicates*
Blocksworld	4	5
Gripper	3	4
Hanoi	4	7
N-Puzzle	1	3
Peg-Solitaire	3	4
Parking	4	5
Zenotravel	5	5
Sokoban	2	4
Visit All	4	7
Elevator	4	6
Spanner	4	6
Logistics	6	3
Floortile	6	10

Table 5.1: Benchmark characteristics.

is characteristic and when observations are complete and noiseless. So, we can conclude that IncrAMLSI converges to Δ_P when observation are complete and noiseless and when $T \to \infty$.

5.3 Experiments

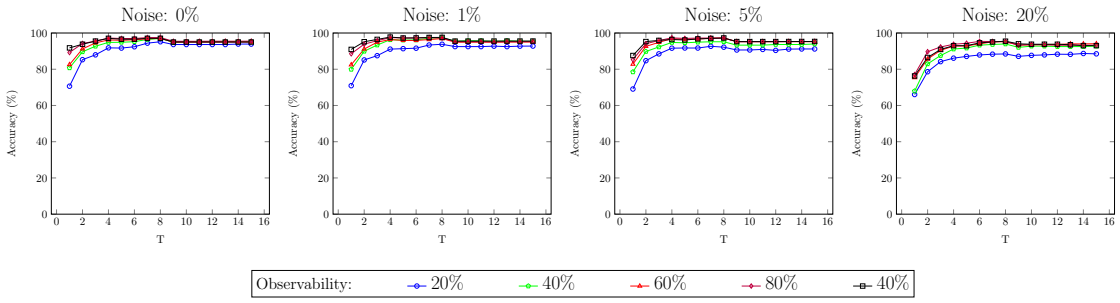
5.3.1 Experimental setup

Our experiments are based on 13 STRIPS-Compliant IPC[2] benchmarks: Blocksworld, Gripper, Hanoi, N-Puzzle, Peg Solitaire, Parking, Zenotravel, Sokoban, Visit All, Elevator, Spanner, Logistics and Floortile. Table - 5.1 shows our experimental setup[3].

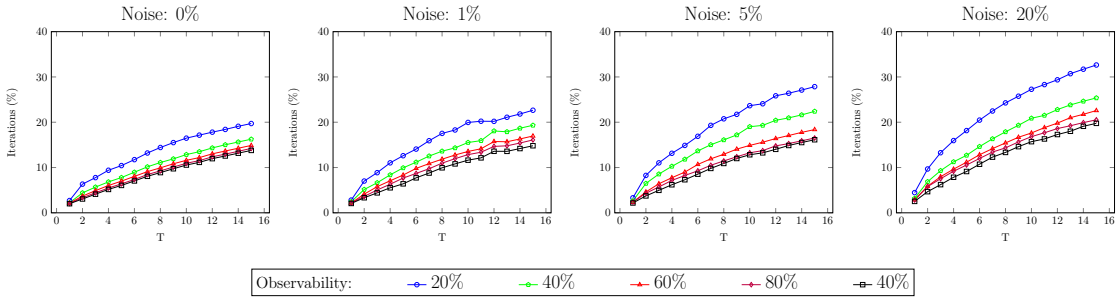
We test each IPC domain with 3 different initial states over ten runs, and we use randomly generated seeds for each run. Also, we generate partial observations by randomly removing a fraction of the propositions of the states, and we generate noise by changing the value of a fraction of the observable propositions. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino, 2018) was used to generate the benchmark data.

[2] `https://www.icaps-conference.org/competitions/`

[3] Experimental setup are publicly available at `https://github.com/maxencegrand/AMLSI`



(a) Accuracy.



(b) Iterations.

Figure 5.2: Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.

5.3.2 Evaluation metrics

The metrics used for this evaluation is the *accuracy* (Zhuo et al., 2013) that measures the learned domain performance to solve new problems. Formally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learned domain, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. The problems are solved with Fast Downward v19.06 (Helmert, 2006). Plan validation is done with VAL (Howey and Long, 2003), which is used in the IPC competitions.[4]

5.3.3 Discussion

Figure 5.2 shows the average performance of IncrAMLSI obtained on the 13 domains of our benchmarks when varying the convergence criterion T. IncrAMLSI is tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%

Whatever the experimental scenario, IncrAMLSI learns from accurate

[4]In the previous chapter we have also used the syntactical distance and the fscore. However, we could observe that the most significant metric was the accuracy, so for this and the following chapters we will use only this metric.

models. Also, increasing T generally leads to better results; Finally, we can observe that whatever the value of T and the experimental scenario, IncrAMLSI converges in less than 35 iterations.

5.4 Conclusion

In this chapter, we have presented the incremental extension of the AMLSI approach: IncrAMLSI. IncrAMLSI learns incrementally action models. The incremental learning process is divided in 3 steps: each time new training data is received, (1) IncrAMLSI updates the DFA, (2) removes extra preconditions and effects and (3) refines the action model. IncrAMLSI stops when the convergence criterion is reached. In addition, we have shown the convergence of the IncrAMLSI extension.

The AMLSI approach, and its incremental extension, learns STRIPS action models, but we have seen in the previous part that STRIPS action models are based on assumptions that are too restrictive to be used in real-world applications. In the following chapters, we will propose two extensions for AMLSI to learn less restrictive action models. More precisely, we will extend the AMLSI approach to learn temporal action model (see Chapter 6) and HTN action models (see chapter 7).

Chapter 6

TempAMLSI: Temporal Action Model Learning

Contents

6.1 Introduction

As seen previously, hand-encoding and proofreading STRIPS action models is difficult, and this is even harder with less restrictive action models such as temporal action models. It is therefore essential to develop tools allowing to automatically learn temporal action models. As we have seen in Chapter 2 several approaches have been proposed to automatically learn STRIPS action models: ARMS (Yang et al., 2007), SLAF (Shahaf and Amir, 2006), Louga (Kucera and Barták, 2018), LSONIO (Mourão et al., 2012), LOCM (Cresswell et al., 2013), IRale (Rodrigues et al., 2010a), PlanMilner (Segura-Muros et al., 2018).

A major open issue is to learn temporal action models (Fox and Long, 2003). Temporal action models are action models allowing to represent *durative actions*, i.e. actions that have a duration, and whose preconditions and effects must be satisfied and applied at different times. An important property of durative actions is that they can be executed concurrently. Temporal action models have different levels of action concurrency (Cushing et al., 2007). Some are sequential, which means that all the plan parts containing overlapping durative actions can be rescheduled into a completely sequential succession of durative actions: each durative action starts after the previous durative action is terminated. One important property of sequential temporal action models is that they can be rewritten as non-temporal action models, and therefore used by classical planners. Some temporal action models require other forms of action concurrency such as Single Hard Envelope (SHE) (Coles et al., 2009). SHE is a form of action concurrency where a durative action can be executed only if another durative action called the envelope extends over it. This is due to the need by the enveloped durative action of a resource, all along its execution, added at the start of the envelope and deleted at the end of the envelope. One important property of SHE temporal action models is that they cannot be sequentially rescheduled. Although some approaches have been proposed to learn temporal features (Gabel and Su, 2010; Neider and Gavran, 2018; Gaglione et al., 2021; Shah et al., 2018), only (Garrido and Jiménez, 2020) proposed an approach learning temporal action models. However this approach is limited to sequential temporal action models. To our best knowledge, there is no learning approach for both SHE and sequential temporal action models.

In this chapter, we present TempAMLSI (Grand et al., 2022b), an accurate learning algorithm for both SHE and sequential temporal action models. TempAMLSI is built on AMLSI (see Chapter 4). Some planners (Fox and Long, 2002a; Halsey et al., 2004; Celorrio et al., 2015; Furelos Blanco et al., 2018) solve temporal planning problems by using non-temporal planners and translation techniques. The key idea of the TempAMLSI approach is to reuse these translation techniques for the learning problem. Like AMLSI, TempAMLSI interacts with the environment to generate input feasible and infeasible action sequences to frame what is allowed by the targeted action model. Then, the temporal learning consists of three steps: (1) TempAMLSI translates temporal sequences into STRIPS sequences, (2) TempAMLSI learns a non-temporal action model with AMLSI, and then (3) translates it into a temporal action model.

TempAMLSI contributions in temporal action model learning are threefold:

- Concurrency: TempAMLSI is able to learn both sequential and SHE temporal action models,
- Partial and noisy observations: TempAMLSI is able to learn temporal action models with both partial and noisy observations.
- Accuracy: TempAMLSI is accurate even with highly partial and noisy learning datasets: thus, it minimises proofreading for AI planning experts.

We show that in many temporal benchmarks TempAMLSI does not require any correction of the learned domains at all.

The rest of this chapter is organized as follows. In Section 6.2 we present a problem statement. In Section 6.3 we give some backgrounds on STRIPS translation techniques, in Section 6.4, we detail TempAMLSI steps. Finally, Section 6.5 evaluates the performance of TempAMLSI on IPC temporal benchmarks.

6.2 Problem Statement

We propose a formal framework inspired by (Höller, 2021) in order to define the temporal learning problem. This formal framework extends the formal framework proposed in Chapter 2 to the learning problem.

Definition 6.1 *A temporal planning problem P is a tuple $(L, A, S, d, s_0, G, \delta, \tau, \lambda)$ where:*

- *L is the set of logical propositions describing the environment.*
- *A is the set of durative actions.*
- *S is the set of state labels.*
- *$d : A \rightarrow \mathbb{R}$ is the duration function.*
- *$s_0 \in S$ is the initial state label.*
- *$G \in S$ is the set of goal state labels.*
- *δ is the temporal action model.*
- *$\tau : S \times A \rightarrow \{true, false\}$ is the feasibility function.*
- *$\lambda : S \rightarrow 2^L$ is the observation function.*

As for STRIPS problems, L is a set of logical propositions, S is a set of states, $s_0 \in S$ is the initial state, G is the set of goal states, and λ is the observation function. A is a set of *durative actions* and $d : A \rightarrow \mathbb{R}$ is the duration function. Unlike STRIPS planning problems, action preconditions, positive and negative effects are labeled with time labels *at-start*, *at-end* and *overall*. More precisely, δ includes:

- $prec : A \times \{s, o, e\} \rightarrow 2^L$: preconditions of $a \in A$ at start, over all, and at end, respectively.
- $add : A \times \{s, e\} \rightarrow 2^L$: positive effects of $a \in A$ at start and at end, respectively.

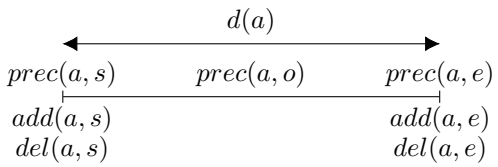


Figure 6.1: Structure of a durative action.

- $del : A \times \{s, e\} \rightarrow 2^L$: negative effects of $a \in A$ at start and at end, respectively.

Example 6.1 *For example, for the the action (stack green blue right) we have:*

- *Preconditions:*
  - *prec((stack green blue right),o) = {(holding green right)}.*
  - *prec((stack green blue right),s) = ∅.*
  - *prec((stack green blue right),e) = {(clear blue)}.*
- *Positive effects:*
  - *add((stack green blue right),e) = {(on green blue), (clear green)}.*
  - *add((stack green blue right),s) = ∅*
- *Negative effects:*
  - *del((stack green blue right),e) = {(clear blue), (holding green right)}.*
  - *del((stack green blue right),s) = ∅*

The semantics of durative actions is defined in terms of two discrete events start-a and end-a, each of which is naturally expressed as a STRIPS action. Starting a durative action a in state s is equivalent to applying the STRIPS action start-a in s, first verifying that prec(start-a) holds in s. Ending a in state s' is equivalent to applying end-a in s', first by verifying that prec(end-a) holds in s'. start-a and end-a are defined as follows:

- start-a:
  - prec(start-a) = prec(a,s).
  - add(start-a) = add(a, s).
  - del(start-a) = del(a, s).
- end-a:
  - prec(end-a) = prec(a,e).
  - add(end-a) = add(e, s).
  - del(end-a) = del(a, e).

Example 6.2 *For the durative action (stack blue green right) we have:*
add(end-stack green blue right) = add((stack green blue right),e) = { (on green blue), (clear green)}

start-a and end-a are constrained by the duration of a, denoted $d(a)$ and the overall precondition: end-a has to occur exactly $d(a)$ time units after start-a, and the overall preconditions have to hold in all states between start-a and end-a. Although a has a duration, its effects apply instantaneously at the start and the end of a, respectively. The preconditions $prec(a,s)$ and $prec(a,e)$ are also checked instantaneously, but $prec(a,o)$ has to hold for the entire duration of a. The structure of a durative action is summarized in Figure 6.1.

A *temporal action sequence* is a set of action-time pairs $\langle(a_1,t_1),\ldots,(a_n,t_n)\rangle$. Each action-time pair (a,t) is composed of a durative action $a \in A$ and a scheduled start timestamp $t \in \mathbb{R}$ of a, and induces two events start-a and end-a with associated timestamps t and $t+d(a)$, respectively. Events start-a (resp. end-a) is applied in the state s_t (resp. $s_{t+d(a)}$), s_t (resp. $s_{t+d(a)}$) being a state time-stamped with t (resp. $t+d(a)$). Then, the temporal transition function γ can be rewritten as: $\gamma(s,a,t) = (\gamma(s_t, \textit{start-a}), \gamma(s_{t+d(a)}, \textit{end-a}))$. The transition function $\gamma(s,a,t)$ is defined if and only if: $prec(a,s) \subseteq \lambda(s_t)$, $prec(a,e) \subseteq \lambda(s_{t+d(a)})$ and $\forall t'$ such that $t \leq t' \leq t+d(a)$ $prec(a,o) \subseteq \lambda(s_{t'})$.

Finally, we can define a temporal planning problem P as a formal language:

$$\mathcal{L}(P) = \{\omega = ((a_1,t_1)\ldots(a_n,t_n)) | a_i \in A, t_i \in T, g \in G, \gamma(s_0,\omega,t_0) \models g\}.$$

Unlike a STRIPS problem, the alphabet of the $\mathcal{L}(P)$ language does not only contain actions but also timestamps. However, as we will see in Section 6.3, it is possible to represent a temporal problem in the form of a STRIPS problem, so it is possible to represent a temporal problem in the form of a regular language $\mathcal{L}_{\mathcal{STRIPS}}(P)$.

A Temporal learning problem is as follow: given a set of temporal observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn the temporal action model and express it into a PDDL 2.1 domain?

The key idea of our approach is to translate the temporal observations Ω into non-temporal observations Ω_{STRIPS}, learns the DFA $\Sigma = (S,A,\gamma)$ corresponding to the regular language $\mathcal{L}_{\mathcal{STRIPS}}(P)$, infers a STRIPS action model from Σ and the partial and noisy observation function λ, generalizes it into PDDL planning domain and translates it into the following PDDL 2.1 domain:

```
(:durative-action stack
 :parameters (?x ?y - block ?h - hand)
 :duration (= ?duration 1)
 :condition (and
  (overall (holding ?a ?h))
  (at end (clear ?y)))
 :effect (and
```

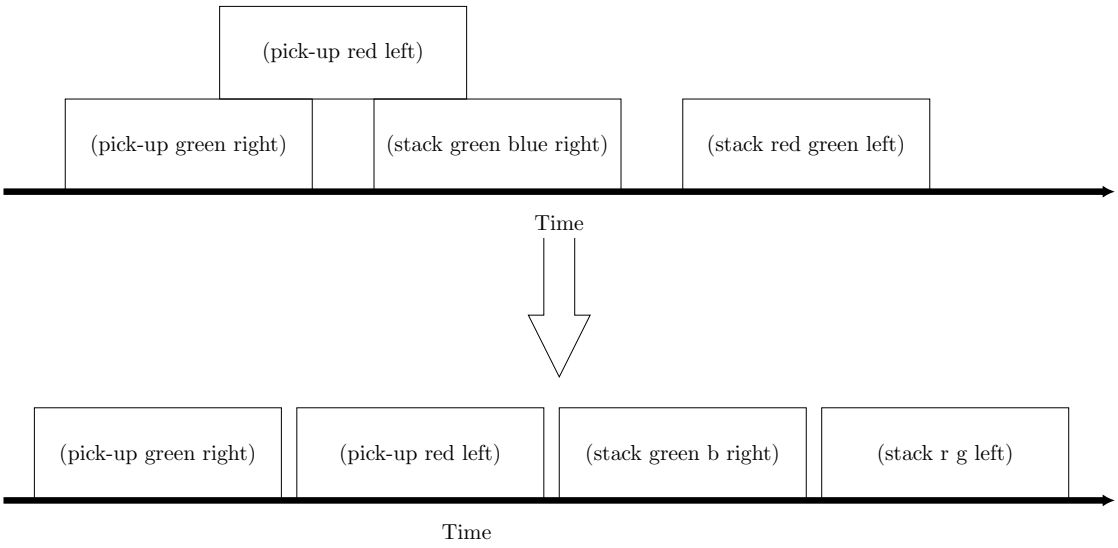


Figure 6.2: Sequential Domains: An example of a concurrent plan rescheduled into a sequential plan.

```
(at end (not (holding ?x ?h)))
(at end (not (clear ?y)))
(at end (clear ?x))
(at end (handempty ?h))
(at end (on ?x ?y))))
```

6.3 Background on STRIPS Translation based Planning

Some planners (Fox and Long, 2002a; Halsey et al., 2004; Celorrio et al., 2015; Furelos Blanco et al., 2018) solve temporal problems by using non-temporal planners. To that end, they convert temporal problems into classical non-temporal STRIPS problems, solve them with a non-temporal planner. Then they convert the classical plan into a temporal plan with rescheduling techniques.

Temporal action models have different levels of required action concurrency (Cushing et al., 2007). Some of them are *sequential*, which means that all the plan parts containing overlapping durative actions can be rescheduled into a completely sequential succession of durative actions: each durative action starts after the previous durative action is terminated. For example, the Blocksworld domain is a sequential domain (see Figure 6.2).

One important property of sequential temporal domains is that they can be rewritten as classical domains, and therefore used by classical non-temporal planners. To solve a sequential temporal problem, we can translate each durative actions $a \in A$ to a compressed STRIPS action C_a that simulates all of a at once (Coles et al., 2009). The precondition of C_a is the union of the preconditions *at-*

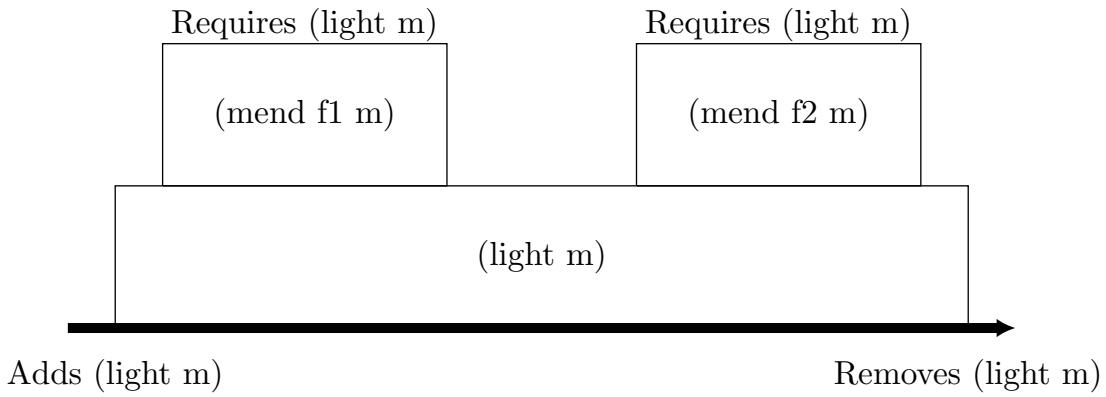


Figure 6.3: An example of SHE temporal domain.

start of a with the preconditions *overall* and *at-end* not achieved by the add effect *at-start*. The effect of C_a is the effect *at-start* of a followed immediately by its effect *at-end*. Formally, the compressed action C_a is defined as follows:

- $prec(C_a) = prec(a,s) \cup \{\{prec(a,o) \cup prec(a,e)\} \setminus del(a,s)\}$
- $add(C_a) = \{add(a,s) \setminus del(a,e)\} \cup add(a,e)\}$
- $del(C_a) = \{del(a,s) \setminus add(a,e)\} \cup del(a,e)\}$

Example 6.3 *For example, for Blocksworld and the action (stack green blue right), we have:*

- $prec(C_{(stack\ green\ blue\ right)}) = \{$*(holding right green), (clear blue)*$\}$
- $add(C_{(stack\ green\ blue\ right)}) = \{$*(handempty right), (on green blue), (clear green)*$\}$
- $del(C_{(stack\ green\ blue\ right)}) = \{$*(holding right green), (clear blue)*$\}$

Once the durative actions are translated, the temporal problem becomes a STRIPS problem that can be solved using a classical planner. When the STRIPS problem is solved, the plan containing compressed actions is translated into a plan with durative actions executed one after another.

Some temporal domains require different forms of action concurrence such as *Single Hard Envelope* (SHE) (Coles et al., 2009). SHE is a form of action concurrency where the execution of a durative action a is required for the execution of a second durative action a'. Formally, a SHE is a durative action a' that adds a proposition p at-start and deletes it at-end while p is an overall preconditions of a durative action a. Contrary to sequential temporal domains, for temporal domains containing SHE there exists temporal action sequences that cannot be sequentially rescheduled.

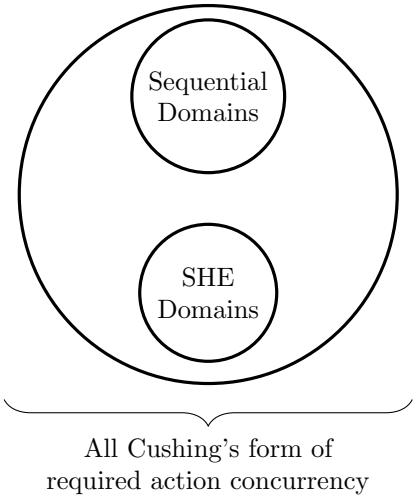


Figure 6.4: Different forms of required action concurrency.

Example 6.4 *For example, see the IPC Match domain*[1] *(see Figure 6.3) and the following durative actions:*

- *(mend f1 m) such that (light m)* $\in$ *prec*$\big($*(mend f1 m)*$, o\big)$
- *(mend f2 m) such that (light m)* $\in$ *prec*$\big($*(mend f2 m)*$, o\big)$
- *(light m) such that (light m)* $\in$ *add*$\big($*(light m)*$, s\big)$ *and (light m)* $\in$ *del*$\big($*(light m)*$, e\big)$

The durative action (mend f m) cannot start before the start of the durative action (light m) and (mend f m) cannot end after the end of (light m), so (mend f m) has to start after the start of (light m) and to end before the end of (light m): it is therefore impossible to sequentially reschedule such temporal action sequences.

Generally, to solve SHE Temporal planning problems, planners start by translating durative actions into STRIPS actions. For instance, the CRICKEY planner (Coles et al., 2009) translates each durative action *a* into three STRIPS actions start-a, inv-a and end-a. Then classical planners are used to solve the problem. Finally, scheduling techniques are used to translate the plans. For instance, the CRICKEY planner builds a set of partially ordered plans with the STRIPS actions. Then, a Simple Temporal Network is used to translate the set of partially ordered plans into a temporal plan.

In addition, it should be noted that there are other forms of required action concurrency besides SHE (Cushing et al., 2007) (see Figure 6.4).

6.4 Temporal Learning

The main idea of TempAMLSI is that it is possible to learn a state machine by testing durative actions and by observing the states resulting from the

[1] An agent needs to repair fuses. To repair a fuse the agent needs a lighted match, see Annexe C for more details.

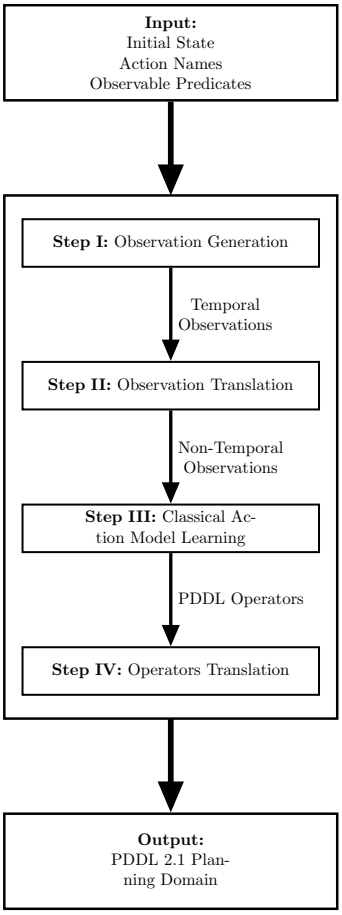


Figure 6.5: Overview of the TempAMLSI approach.

executions of the durative actions and to represent it as a PDDL 2.1 planning domain. TempAMLSI assumes that it knows the names of the durative actions, i.e., the names of the transitions of the state machine, that it can test them and it is able to observe the state resulting from their applications as a set of logical propositions whose predicates are also known.

Based on these assumptions, TempAMLSI produces a set of observations Ω by using a random walk and learns as output a planning domain modeling these observations. The planning domain learned is expressed using the PDDL 2.1 language. To perform this learning, TempAMLSI translates Ω into a non temporal observation set Ω_{STRIPS}, learns a STRIPS action model, and more precisely a set of PDDL operators Δ_{STRIPS} and translates it into a temporal action model, and more precisely a set of temporal PDDL 2.1 operators Δ. TempAMLSI assumes that L, A, C, S, s_0, d are known and the observation function λ is possibly partial and noisy. No knowledge of the goal states G is required.

The TempAMLSI approach (see Figure 6.5) consists of 4 steps:

1. *Observation Generation.* TempAMLSI produces a set of observations Ω by using a random walk. In Section 6.4.1, we will present how TempAMLSI is able to efficiently exploit these observations by taking into account not

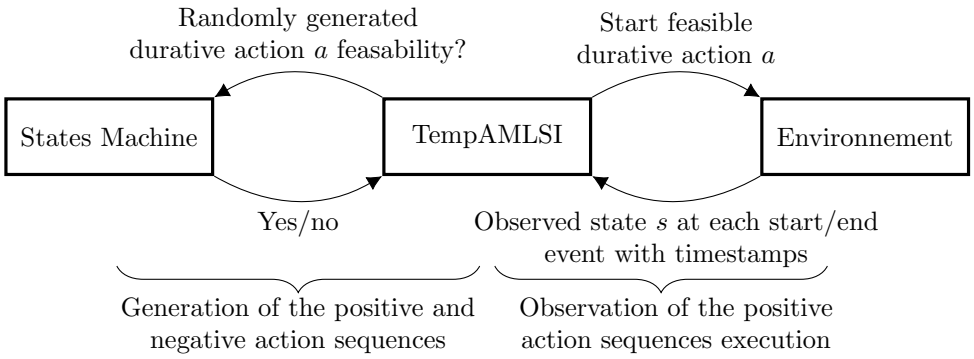


Figure 6.6: Observation Generation Overview.

only the fact that some actions are deasible in certain states.

2. *Observations Translation.* After having generated the samples of *temporal* sequences (including both feasible and infeasible sequences), TempAMLSI translates them into non-temporal sequences (see Section - 6.4.2).

3. *Classical Action Model Learning.* TempAMLSI learns a classical STRIPS action model, and more precisely a set of PDDL operators, from the translated samples using the AMLSI algorithm (see Chapter 4).

4. *Operators Translation.* TempAMLSI translates PDDL operators them into Temporal PDDL 2.1 operators (see Section - 6.4.3)

6.4.1 Observation Generation

Figure 6.6 gives an overview of this step. To generate the observations in Ω, TempAMLSI uses random walks by applying a randomly selected durative action to the initial state of the problem. If this action is feasible, the start event with the corresponding timestamp is appended to the current action sequence. Otherwise, the feasible prefix plus the infeasible start event are added to the set of negative samples I_-. The procedure is repeated until the feasible prefix achieves an arbitrary size and added to the set of positive samples I_+. Random walks are repeated until I_+ and I_- achieves an arbitrary size.

Example 6.5 *As an example, consider the following feasible sequence ω in the Blocksworld problem:*

$$\omega = \langle \quad (0, start(\textit{pick-up green right})), (0.5, start(\textit{pick-up red left})), (1, end(\textit{pick-up green right})), \\ (1.1, start(\textit{stack green blue right})), (1.5, end(\textit{pick-up red left})), (2.1, end(\textit{stack green blue right})) \quad \rangle$$

6.4.2 Observations Translation

In practice, the temporal sequences generated by TempAMLSI are timestamped start and end event sequences (see Example 6.5). This means that durative action (*pick-up green right*) starts at 0 and finishes at 1, (*pick-up red left*) starts at 0.5 and

finishes at 1.5 and (*stack green blue right*) starts at 1.1 and finishes at 2.1. In the rest of this section we focus on the sample and operator translation steps. We will present two variants for these translations:

2-Operators Translation : The STRIPS action sequences contain, for each durative action a, the start action start-a and the end action end-a corresponding to the events of a durative action (see Section - 6.2). This method only translates the events observed in the temporal sequences. But, it does not directly represent the *overall* preconditions that constrain the "life cycle" of a durative action. Indeed, for a durative action a to be executed the *at-start* preconditions must be checked at the start event, and the *at-end* preconditions must be checked at the end event, but it is also necessary that the *overall* preconditions are satisfied on all the duration of action a.

3-Operators Translation : The STRIPS action sequences contain, for each durative action a, as for the 2-Operators translation, the start action start-a and the end action end-a corresponding to the events of a durative action. However, they also contains inv-a: an invariant action. This invariant action is added at each new event (a durative action starts or ends) and allows to represent the *overall* preconditions.

Example 6.6 *Let ω be a temporal sequence such that:*

$$\omega = \langle \ (0, start(\textit{pick-up green right})), (0.5, start(\textit{pick-up red left})), (1, end(\textit{pick-up green right})), \\ (1.1, start(\textit{stack green blue right})), (1.5, end(\textit{pick-up red left})), (2.1, end(\textit{stack green blue right})) \ \rangle$$

- 2-Operators: *Each durative action a is converted into two event actions start-a and end-a. After conversion:*

$$\omega_{STRIPS_{2OP}} = \langle \ (\textit{start-pick-up green right}), (\textit{start-pick-up red left}), (\textit{end-pick-up green right}), \\ (\textit{start-stack green blue right}), (\textit{end-pick-up red left}), (\textit{end-stack green blue right}) \ \rangle$$

- 3-Operators: *In this case, each durative action a is converted into three event actions start-a, inv-a and end-a. After conversion, we have the following sequence:*

$$\omega_{STRIPS_{3OP}} = \langle \ (\textit{start-pick-up green right}), (\textit{inv-pick-up green right}), (\textit{start-pick-up red left}), \\ (\textit{inv-pick-up green right}), (\textit{inv-pick-up red left}), (\textit{end-pick-up green right}), \\ (\textit{inv-pick-up red left}), (\textit{start-stack green blue right}), (\textit{inv-pick-up red left}), \\ (\textit{inv-stack green blue right}), (\textit{end-pick-up red left}), (\textit{inv-stack green blue right}), \\ (\textit{end-stack green blue right}) \ \rangle$$

6.4.3 Operators Translation

After having learned the STRIPS domain with AMLSI, TempAMLSI converts STRIPS operators into temporal operators.

2-Operators *at-start* (resp. *at-end*) effects are the effects of start (resp. end) STRIPS operators. *overall* preconditions are the intersection of preconditions of start and end STRIPS operators. And, *at-start* (resp. *at-end*) preconditions are the preconditions of the start (resp. end) STRIPS operators excluding end (resp. start) preconditions. Formally, 2-Operators translation is as follows:

- prec(a,s)= prec(start-a) $\setminus$ prec(end-a)
- add(a,s)=add(start-a)
- del(a,s)=del(start-a)
- prec(a,e)= prec(end-a) $\setminus$ prec(start-a)
- add(a,e)=add(end-a)
- del(a,e)=del(end-a)
- prec(a,o)= prec(start-a) $\cap$ prec(end-a)

3-Operators *at-start* (resp. *at-end*) effects are the effects of start (resp. end) STRIPS operators. *overall* preconditions are the preconditions of *inv* STRIPS operators. And, *at-start* (resp. *at-end*) preconditions are the preconditions of start (resp. end) STRIPS operators excluding inv preconditions. Formally, 3-Operators translation is as follows:

- prec(a,s)= prec(start-a) $\setminus$ prec(inv-a)
- add(a,s)=add(start-a)
- del(a,s)=del(start-a)
- prec(a,e)= prec(end-a) $\setminus$ prec(inv-a)
- add(a,e)=add(end-a)
- del(a,e)=del(end-a)
- prec(a,o)= prec(inv-a)

Example 6.7 *Let's go back to the Blocksworld example:*

- 2-Operators: *Let Δ_{2OP} be the set of PDDL operators learned such that:*
  - *prec(start-stack ?x ?y ?h) = { (holding ?x ?h) }, add(start-stack ?x ?y ?h) = $\emptyset$, del(start-stack ?x ?y ?h) = $\emptyset$*
  - *prec(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }, add(end-stack ?x ?y ?h) = { (handempty ?h),(clear ?x),(on ?x ?y) }, del(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }*

After translation, we have the following set of Temporal PDDL 2.1 operators Δ *such that:*

- *prec((stack ?x ?y ?h),s) = prec(start-stack ?x ?y ?h)* $\setminus$ *prec(end-stack ?x ?y ?h) =* $\emptyset$
- *prec((stack ?x ?y ?h),e) = prec(end-stack ?x ?y ?h)* $\setminus$ *prec(start-stack ?x ?y ?h) = { (clear ?y) }*
- *prec((stack ?x ?y ?h),o) = prec(start-stack ?x ?y ?h)* $\cap$ *prec(end-stack ?x ?y ?h) = { (holding ?x ?h) }*
- *add((stack ?x ?y ?h),s) = add(start-stack ?x ?y ?h) =* $\emptyset$
- *del((stack ?x ?y ?h),s) = del(start-stack ?x ?y ?h) =* $\emptyset$
- *add((stack ?x ?y ?h),e) = add(end-stack ?x ?y ?h) = { (handempty ?h),(clear ?x),(on ?x ?y) }*
- *del((stack ?x ?y ?h),e) = del(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }*

- 3-Operators: *Let* Δ_{3OP} *be the set of PDDL operators learned such that:*
  - *prec(start-stack ?x ?y ?h) = { (holding ?x ?h) }, add(start-stack ?x ?y ?h) =* $\emptyset$*, del(start-stack ?x ?y ?h) =* $\emptyset$
  - *prec(inv-stack ?x ?y ?h) = { (holding ?x ?h) }, add(inv-stack ?x ?y ?h) =* $\emptyset$*, del(inv-stack ?x ?y ?h) =* $\emptyset$
  - *prec(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }, add(end-stack ?x ?y ?h) = { (handempty ?h),(clear ?x),(on ?x ?y) }, del(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }*

  After translation, we have the following set of Temporal PDDL 2.1 operators Δ *such that:*

  - *prec((stack ?x ?y ?h),s) = prec(start-stack ?x ?y ?h)* $\setminus$ *prec(inv-stack ?x ?y ?h) =* $\emptyset$
  - *prec((stack ?x ?y ?h),e) = prec(end-stack ?x ?y ?h)* $\setminus$ *prec(inv-stack ?x ?y ?h) = { (clear ?y) }*
  - *prec((stack ?x ?y ?h),o) = prec(inv-stack ?x ?y ?h) = { (holding ?x ?h) }*
  - *add((stack ?x ?y ?h),s) = add(start-stack ?x ?y ?h) =* $\emptyset$
  - *del((stack ?x ?y ?h),s) = del(start-stack ?x ?y ?h) =* $\emptyset$
  - *add((stack ?x ?y ?h),e) = add(end-stack ?x ?y ?h) = { (handempty ?h),(clear ?x),(on ?x ?y) }*
  - *del((stack ?x ?y ?h),e) = del(end-stack ?x ?y ?h) = { (holding ?x ?h),(clear ?y) }*

Domain	Operators	Predicates	Type
Peg Solitaire	1	3	Sequential
Sokoban	2	3	Sequential
Zenotravel	5	4	Sequential
Match	2	4	SHE
Turn and Open	5	8	SHE

Table 6.1: Benchmark domain characteristics.

6.5 Experiments and Evaluations

The evaluation consists in the comparison of the performance of the 2-Operators and 3-Operators variants of TempAMLSI.

6.5.1 Experimental Setup

Our experiments are based on 5 temporal IPC action models[2] (see Table 6.1)[3]. More precisely we test TempAMLSI with three sequential action models (Peg Solitaire, Sokoban, Zenotravel), and two SHE action models (Match, Turn and Open). We test each IPC action model with 3 different initial states over ten runs, and we use ten randomly generated seeds for each run. Finally we generate partial observations by randomly removing a fraction of the propositions of the states, and we generate noise by changing the value of a fraction of the observable propositions. All the tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino, 2018) was used to generate the benchmark data.

6.5.2 Evaluation Metrics

TempAMLSI is evaluated using the *accuracy* (Zhuo et al., 2013) that measures the learned action model performance to solve new problems.

Formally, $Accuracy = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learned action model, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. We also report in our results the ratio of (possibly incorrectly) solved problems. A problem is incorrectly solved when a solution plan is found with the learned action model that is not correct with respect to the original action model. In the experiments, we solve the benchmark problems with the TP-SHE (Celorrio et al., 2015) planner. Plan validation is done with VAL, the IPC competition validation tool (Howey and Long, 2003).

[2] `https://www.icaps-conference.org/competitions/`

[3] Experimental setup are publicly available at `https://github.com/maxencegrand/AMLSI`

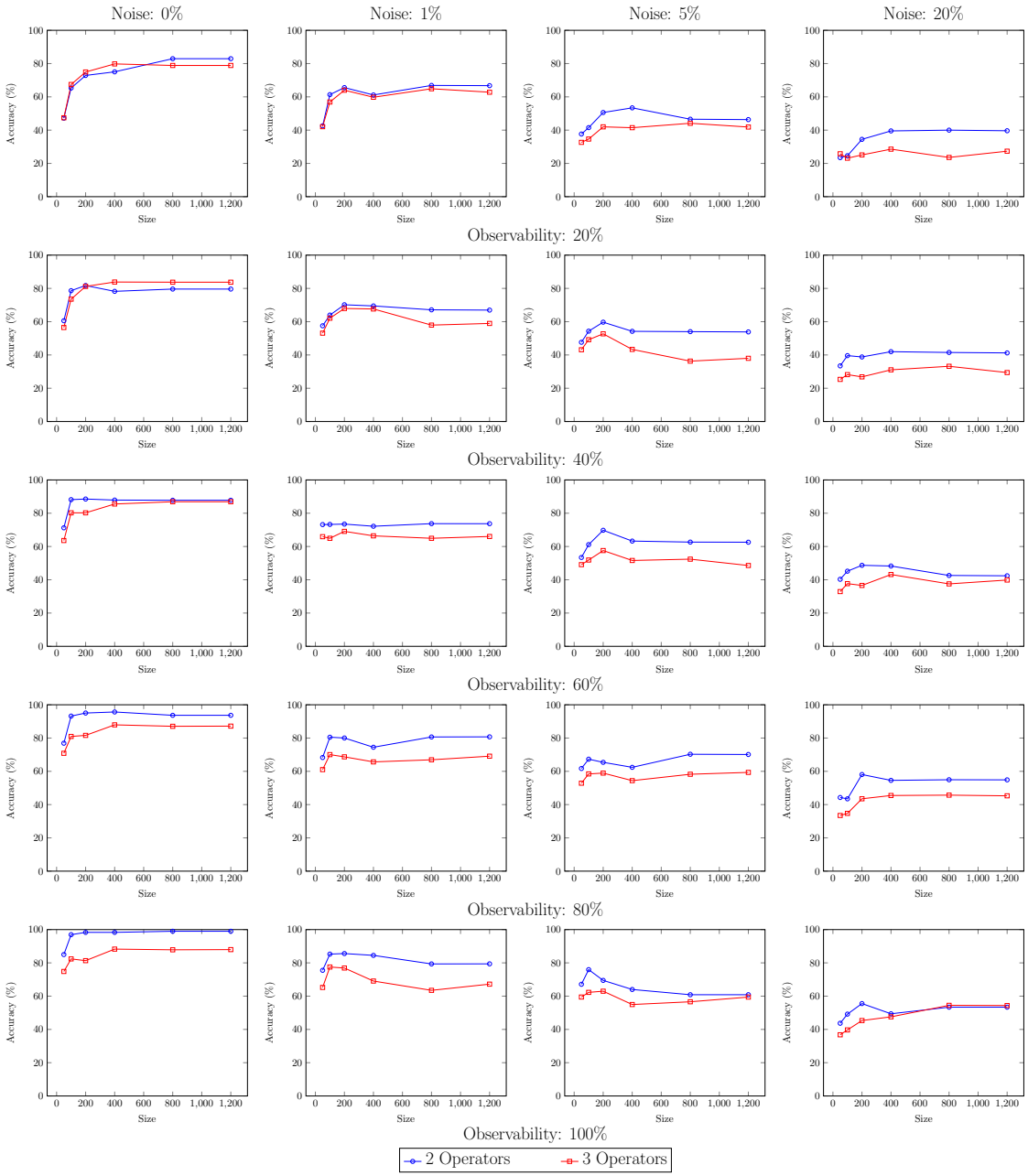


Figure 6.7: Average performance of TempAMLSI when the training data set size increases in terms of number of durative actions in terms of Accuracy.

6.5.3 Discussion

Figure 6.7 shows the average performance of TempAMLSI obtained on the 5 action models of our benchmarks when varying the training data set size. The size of the training set is indicated in number of durative actions. TempAMLSI is tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%.

We observe that 2-Operators and 3-Operators variants are accurate when the

level of noise is not high ($< 20\%$) whatever the level of observability. Only the 2-Operators variant is accurate with high level of noise. Also, we observe that the 2-Operators variant is generally more robust than the 3-Operators variant. The fact that the 2-Operators variant is more robust than the 3-Operators variant can be explained in different ways. First of all, the fact that action sequences of 2-Operators variants are shorter than action sequences of 3-Operators variants makes DFAs easier to learn since they have fewer states. The better the DFA learning, the better the operator learning. Moreover, it is easier for 2-Operators variants than for 3-Operators variants because 2-Operators variants have less operators.

6.6 Conclusion

In this chapter we have presented TempAMLSI, a novel algorithm to learn temporal action models. TempAMLSI is built on the AMLSI approach and the idea to use classical STRIPS translation techniques: after generated a set of temporal action sequences, TempAMLSI converts temporal action sequences into non-temporal sequences. Then TempAMLSI uses AMLSI algorithm to learn a classical action model and converts it into a temporal action.

Our experimental results show that TempAMLSI is able to learn accurately both sequential and SHE temporal action models from partial and noisy datasets. However, SHE are not the only form of required action concurrency. Indeed, there exist different levels of required action concurrency for each Allen's interval algebra. So in future works, TempAMLSI could be extended to encompass more temporal relations. Also, the learned action models are STRIPS-compliant, i.e. preconditions and effects are sets of logical propositions, it would be interesting to learn more complex preconditions and effects including logical quantifiers.

In the next chapter, we will present another extension of the AMSLI approach learning HTN task models.

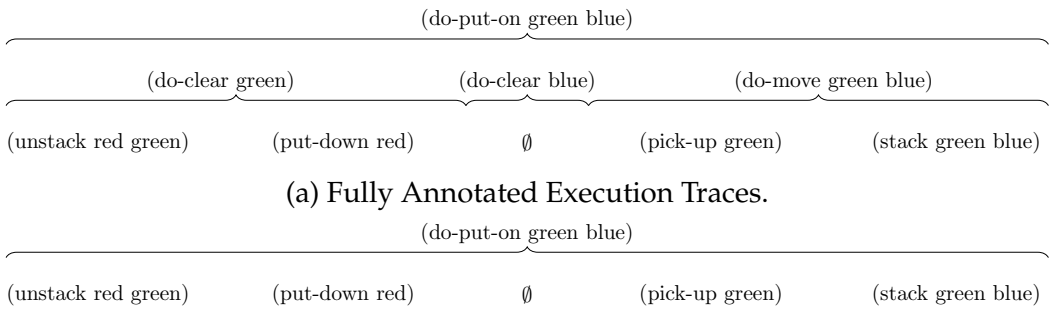
Chapter 7

HierAMLSI: HTN Task Model Learning

Contents

7.1 Introduction

As we have seen it in Chapter 2, the Hierarchical Task Network (HTN) formalism (Erol et al., 1994) is very expressive and used to express a wide variety of planning problems. This formalism allows planners to exploit domain knowledge to solve problems more efficiently (Nau et al., 2005) when planning problems can be naturally decomposed hierarchically in terms of tasks and task decompositions. In contrast to the classical STRIPS formalism in which only the action model needs to be specified, the HTN formalism requires to specify the task model. A task model can be primitive and compound. A primitive tasks model is described by classical actions. A compound tasks model is described using HTN methods. A HTN method describes the set of primitive



(a) Fully Annotated Execution Traces.

(b) Partially Annotated Execution Traces.

Figure 7.1: Examples of Task Annotated Execution Traces.

and/or compound tasks requires to decompose a specific compound task. For this reason, hand-encoding HTN task models is considered more difficult and more error-prone than classical STRIPS action model. This makes it all the more necessarily to develop techniques to learn HTN task models.

In Chapter 2, we have seen that some approaches have been proposed to learn HTN task models, e.g. CAMEL (Ilghami et al., 2002), HTN-Maker (Hogg et al., 2008, 2009), LHTNDT (Nargesian and Ghassem-Sani, 2008) or HTN-Learner (Zhuo et al., 2009). These approaches have several drawbacks. First, they only learn compound task models, except HTN-Learner, i.e., they consider that the primitive task model is known a priori. Also, although the majority of approaches learning only compound task models are accurate, this is not the case for methods learning both primitive and compound task models. Finally, usually these approaches take as input execution traces containing task annotations, i.e. execution traces are fully annotated with the compound tasks and their decompositions (see Figure 7.1a). Obtaining these annotation is difficult and needs a lot of human effort.

In this chapter, we present HierAMLSI (Grand et al., 2022a), an accurate learning algorithm for both compound and primitive task models robust to partial and noisy observations. HierAMLSI is built on AMLSI (see Chapter 4). Like AMLSI, HierAMLSI interacts with the environment to generate input feasible and infeasible task sequences to frame what is allowed by the targeted task model. In order to reduce the difficulty to obtain execution traces, execution traces are only partially annotated, i.e. intermediate task decomposition are unknown. Figure 7.1 gives a comparison between a fully and a partially annotated trace.

HierAMLSI contributions in HTN task model learning are fourfold:

- Output: HierAMLSI is able to learn primitive task model or compound task model or both.
- Partial and noisy observations: HierAMLSI is able to learn task models with both partial and noisy observations.
- Annotation: HierAMLSI takes as input only partially annotated execution

traces that allows to reduce the difficulty to obtain annotated execution traces.

- Accuracy: HierAMLSI is accurate even with highly partial and noisy learning datasets: thus, it minimises proofreading for AI Planning experts. We show that in many HTN benchmarks HierAMLSI does not require any correction of the learned action models at all.

This chapter is organized as follows. In Section 7.2 we present the problem statement. In Section 7.3, we detail the HierAMLSI steps. Then, Section 7.4 evaluates the performance of HierAMLSI on IPC benchmarks. Although HierAMLSI is able to learn the primitive task model or the compound task model or both, the primitive task model learning being done using AMLSI, this chapter focus on the compound task model learning.

7.2 Problem Statement

We propose a formal framework inspired by (Höller, 2021) in order to define the HTN learning problem. This formal framework extends the formal framework proposed in Chapter 2 to the learning problem.

Definition 7.1 *An* HTN *planning problem* P *is a tuple* $(L, C, A, S, M, s_0, \omega_I, G, \delta, \lambda, \sigma, \zeta)$ *where:*

- L *is the set of logical propositions describing the environment.*
- S *is the set of state labels.*
- C *is the set of compound tasks.*
- A *is the set of actions (or primitive tasks).*
- M *is the set of* HTN *method labels.*
- $s_0 \in S$ *is the initial state.*
- $\omega_I \in \{A \cup C\}^*$ *is the initial task network.*
- $G \in S$ *is the set of goal state labels.*
- δ *is the task model.*
- $\lambda : S \rightarrow 2^L$ *is the observation function.*
- $\sigma : M \rightarrow C \times \{A \cup C\}^*$ *is the method decomposition function*[1].
- $\zeta : \{A \cup C\}^* \times S \rightarrow \{A \cup C\}^*$ *is the decomposition function.*

[1] * is the Kleene operator

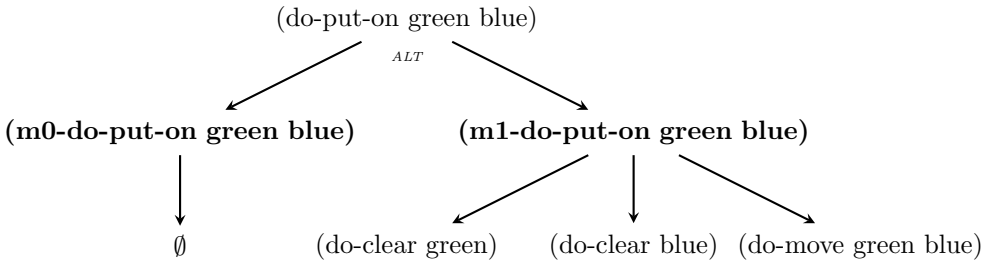


Figure 7.2: Method decomposition of the task *(do-put-on green blue)*.

As for STRIPS problems, L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, $G \subseteq S$ is the set of goal label, λ is the observation function and preconditions, positive and negative effects are given by the functions *prec*, *add* and *del* included in δ.

A is the set of action (or primitive task) labels and C is a set of compound (or non primitive) task labels, with $C \cap A = \emptyset$.

Example 7.1 *In the Blocksworld example, A is composed of the following primitive tasks: {(pick-up red),(pick-up blue),(pick-up green),(stack green blue),...} and C is composed of the following compound tasks: {(do-clear green),(do-clear blue),(do-clear red),(do-put-on green blue),...}.*

Tasks are maintained in *task networks*. A task network is a sequence of tasks. Compound tasks can be decomposed by methods. The set M contains all method labels. Methods are defined by the function $\sigma : M \rightarrow C \times \{C \bigcup A\}^*$.

Example 7.2 *As example, Figure 7.2 gives the method decomposition of the task (do-put-on green blue). There are two relevant methods, i.e. methods allowing to decompose this task: (m0-do-put-on green blue) and (m1-do-put-on green blue). The first one decomposes the task into an empty task network: σ(m0-do-put-on green blue) = ((do-put-on green blue), $\emptyset$). The second one decomposes the task into three subtasks: σ(m1-do-put-on green blue) = ((do-put-on green blue), $\langle$(do-clear green),(do-clear blue),(do-move green blue)$\rangle$).*

A compound task c is decomposable in a state s if and only if there exists a relevant method $m \in M$ such that: $\sigma(m) = (c, \omega)$ and $prec(m) \in \lambda(s)$. The function $\zeta : \{C \bigcup A\}^* \times S \rightarrow \{C \bigcup A\}^*$ gives the decomposition function. For a totally ordered task network $\omega = \omega_1 t \omega_2$, ζ is defined as follows:

$$\zeta(\omega_1 t \omega_2, s) = \begin{cases} \omega_1 t \omega_2 & \text{if } t \text{ is a primitive task.} \\ \omega_1 \omega' \omega_2 & \text{if } t \text{ is a compound task} \\ & \text{and } t \text{ is decomposable in } \gamma(\omega_1, s). \\ \emptyset & \text{Otherwise.} \end{cases}$$

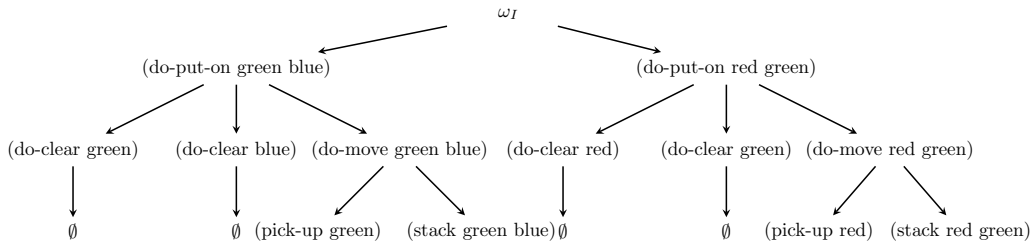


Figure 7.3: A Solution Task Network.

Example 7.3 *In our example, we have:*

$$\begin{aligned} \textit{prec(m0-do-put-on green blue)} &= \{\textit{(on green blue)}, \textit{(handempty)}\} \\ \textit{prec(m1-do-put-on green blue)} &= \{\textit{(not(on green blue))}, \textit{(handempty)}\} \end{aligned}$$

Let's take our initial state s_0, then

$$\zeta(\textit{(m0-do-put-on green blue)}, s_0) = \{\textit{(do-clear blue)}, \textit{(do-clear green)}, \textit{(do-move green blue)}\}$$

Indeed, we have prec(m0-do-put-on green blue) $\notin \lambda(s_0)$ *and prec(m1-do-put-on green blue))* $\in \lambda(s_0)$*, then ζ cannot decompose the task with the method (m0-do-put-on green blue) but can decompose the task with the method (m1-do-put-on green blue).*

As $\omega_1 t \omega_2$ is a totally ordered task network, ω_1 contains only primitive tasks. Indeed, as the network is totally ordered, the compound tasks are decomposed from left to right and therefore if we have to decompose t then ω_1 contains only primitive task.

We denote $\omega \rightarrow^* \omega^*$ that ω can be decomposed into ω^* by 0 or more method applications. Finally, ω_I is the initial task network.

Example 7.4 *In our example:*

$$\omega_I = \{\textit{(do-put-on green blue)}, \textit{(do-put-on green blue)}\}$$

A solution to an HTN planning problem is a task network ω with:

1. $\omega_I \rightarrow^* \omega$, i.e. it can be reached by decomposing ω_I.
2. $\omega \in A^*$, i.e. all tasks are primitive.
3. $\gamma(s_0, \omega) \models g$, i.e. ω is applicable in s_o and results in a goal state.

Figure 7.3 gives a solution task network for our example.

Finally, we can define an HTN planning problem P as a formal language:

$$\mathcal{L}(P) = \{\omega = \langle t_1 \dots t_n \rangle | t_i \in A, \gamma(s_0, \omega) \models g, \omega_I \rightarrow^* \omega\}.$$

Unlike STRIPS planning problems, the language $\mathcal{L}(P)$ is not necessarily regular (Höller et al., 2014) and cannot be represented as a DFA. As mentioned by (Höller et al., 2014; Höller, 2021), $\mathcal{L}(P)$ is the intersection of two languages:

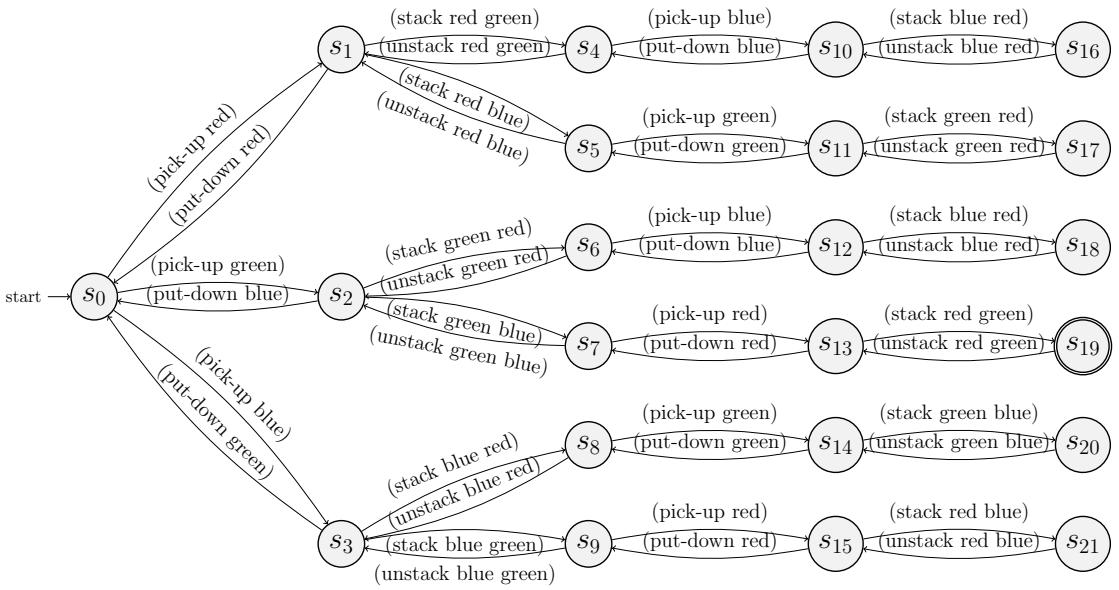


Figure 7.4: The DFA accepting the regular language $\mathcal{L}_{\mathcal{C}}(P)$ for the Blocksworld example.

1. $\mathcal{L}_{\mathcal{C}}(P) = \{\omega \in A^* | \gamma(s_0, \omega) \in g\}$, which is defined by the state transition system defined by the preconditions and effects of the primitive tasks. This language is regular. (see Figure 4.1 in Chapter 4)
2. $\mathcal{L}_{\mathcal{H}}(P) = \{\omega \in A^* | w_I \rightarrow^* \omega\}$, which is defined by the decomposition hierarchy, i.e. by the compound tasks and methods. In our example, $\mathcal{L}_{\mathcal{H}}(P)$ is defined as follows:
   - $\omega_I \rightarrow$ *(do-put-on green blue) (do-put-on red green)*
   - *(do-put-on green blue)* $\rightarrow \emptyset |$ *(do-clear green) (do-clear blue) (move green blue)*
   - *(do-clear green)* $\rightarrow \emptyset |$ *(do-clear blue) (unstack blue green) (put-down blue)* $|$ *(do-clear red) (unstack red green) (put-down red)*
   - ...

A HTN learning problem is as follow: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn the decomposition method function σ and express it into a HDDL domain?

The key idea of our approach is to learn the DFA $\Sigma_C = (S, A, \gamma)$ corresponding to the regular language $\mathcal{L}_{\mathcal{C}}(P)$, and modify the DFA by adding compound task transitions in order to encode the rules of $\mathcal{L}_{\mathcal{H}}(P)$ and approximate the language $\mathcal{L}(P)$ with the DFA $\Sigma = (S, \{A \cup C\}, \gamma)$, infer the decomposition function σ:

$$\begin{aligned} \sigma(\textit{m0-do-put-on b c}) &= (\textit{(m0-do-put-on b c)}, \emptyset) \\ \sigma(\textit{m1-do-put-on b c}) &= (\textit{(m0-do-put-on b c)}, \langle \textit{(do-clear b), (do-clear c), (do-move b c)} \rangle) \\ &\dots \end{aligned}$$

and the HTN methods preconditions

$$\begin{aligned} \textit{prec(m0-do-put-on b c)} &= \{\textit{(on b c)}, \textit{(handempty)}\} \\ \textit{prec(m1-do-put-on b c)} &= \{\textit{(not(on b c))}, \textit{(handempty)}\} \\ &\ldots \end{aligned}$$

from ζ, Σ and the partial and noisy observation function λ and express them as the following HDDL domain:

```
(:method m0_do_put_on
 :parameters ( ?x - block  ?y - block )
 :task (do_put_on ?x ?y)
 :precondition (and (on ?x ?y) (handempty))
 :ordered-subtasks(and ))

(:method m1_do_put_on
 :parameters ( ?x - block  ?y - block )
 :task (do_put_on ?x ?y)
 :precondition (and (handempty))
 :ordered-subtasks(and
                   (do_clear ?x)
                   (do_clear ?y)
                   (do_move ?x ?y)) )
```

7.3 The HierAMLSI approach

The main idea of HierAMLSI is that it is possible to learn a state machine by testing decompositions and transitions and by observing the states resulting from the executions of the decompositions and to represent it as a HDDL planning domain. HierAMLSI assumes that it knows the names of the primitive tasks, i.e., the names of the transitions of the state machine and the names of the compound tasks, that it can test and it is able to observe their decompositions and the state resulting from their applications as a set of logical propositions whose predicates are also known.

Based on these assumptions, HierAMLSI produces a set of observations Ω by using a random walk and learns as output a planning domain modeling these observations. The planning domain learned is expressed using the HDDL language. To perform this learning, HierAMLSI learns first the transition function expressed as a set of actions δ and the method decomposition function σ of a particular problem P and generalizes δ and σ as a HDDL domain Δ. HierAMLSI assumes that it knows L, A, C, S, s_0, the decomposition function ζ is partially annotated (no knowledge about intermediate task decompositions) and the observation function λ is possibly partial and noisy. No knowledge of the goal states G is required.

The HierAMLSI approach (see Figure 7.5) consists of 4 steps:

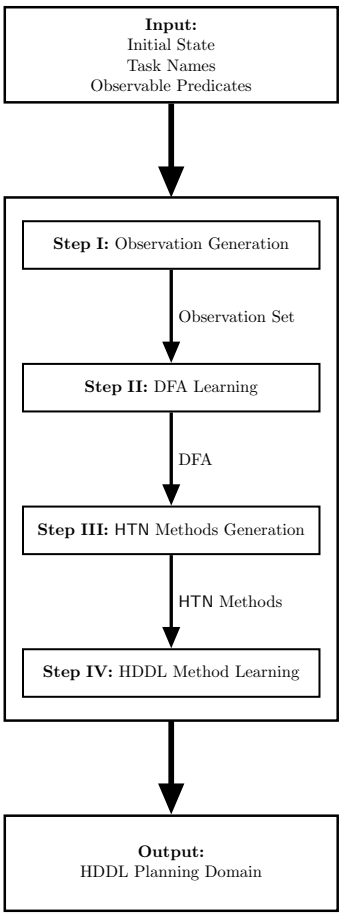


Figure 7.5: HTN Learning Overview.

1. *Observation Generation.* HierAMLSI produces a set of observations Ω by using a random walk. In Section 7.3.1, we will present how HierAMLSI is able to efficiently exploit these observations by taking into account not only the fact that some task are decomposable in certain states and their decomposition but also that others are not.

2. *DFA Learning.* HierAMLSI learns a DFA approximating the language $\mathcal{L}(P)$ (see Section 7.3.2).

3. HTN *Methods Generation.* HierAMLSI generates from the DFA learned previously a set of HTN Methods allowing to decompose all tasks observed in Ω (see Section 7.3.3).

4. *HDDL Methods Learning.* Once HTN Methods have been learned, HierAMLSI has to learn the HTN Methods preconditions. To do this, HierAMLSI treats HTN Methods as primitive tasks and learn an action model containing all methods using the learning and refinement techniques described in Chapter 4.

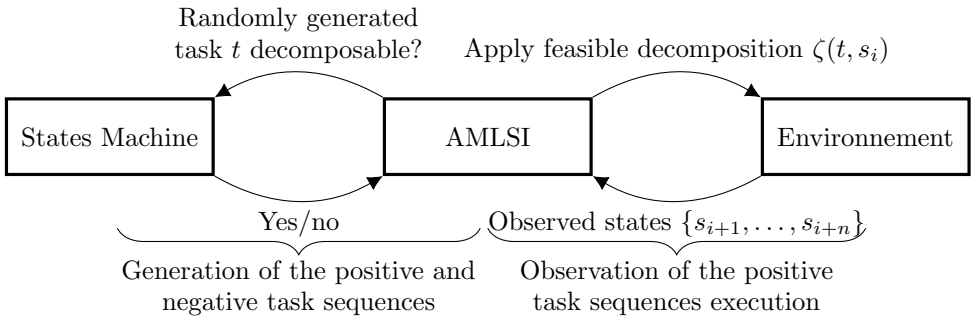


Figure 7.6: HTN Observation Generation Overview.

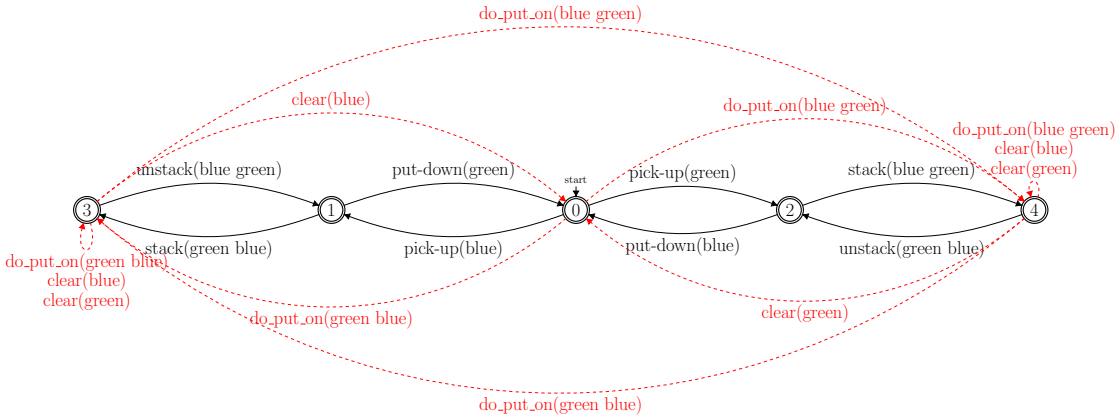


Figure 7.7: DFA Learning Step. The primitive task DFA is the DFA containing only primitive tasks, i.e. black transitions, and the task DFA contains compound tasks, i.e. dashed red transitions, in addition.

7.3.1 Observation Generation

The observation generation process (see Figure 7.6) is similar to the generation method described in Chapter 4. To generate the observations in Ω, HierAMLSI uses random walks by querying a State Machine. HierAMLSI chooses randomly a (primitive or compound) task t. If the task t is decomposable in a state s, HierAMLSI adds $\zeta(t, s)$, the final decomposition containing only primitive task to the current primitive task sequence. Otherwise, the feasible prefix plus the infeasible task is added to set of negative samples I_-. The procedure is repeated until the feasible prefix achieves an arbitrary size and added to the set of positive samples I_+. Random walks are repeated until I_+ and I_- achieve an arbitrary size.

7.3.2 DFA Learning

As mentioned in Section 7.2 the language $\mathcal{L}(P)$ is not necessarily regular, then the purpose of this step is to learn a DFA approximating this language. More

precisely, the DFA learning step is divided in 2 steps: (1) HierAMLSI learns a DFA corresponding to the language $\mathcal{L}_{\mathcal{C}}(P)$ which is defined by the state transition system defined by the preconditions and effects of the primitive tasks and (2) HierAMLSI adds transitions to represent compound tasks in the DFA to allow to approximate the language $\mathcal{L}(P)$.

Step 1: Primitive task DFA Learning AMLSI starts by using the DFA Learning algorithm described in Chapter 4 to learn the DFA containing only primitive tasks.

Step 2: Task DFA Induction Once the primitive task DFA has been learned, HierAMLSI induces the task DFA by adding compound task transitions in the DFA, i.e. by adding transitions whose labels are compound task labels.

Example 7.5 *Figure 7.7 gives an example of task DFA. Suppose we have the compound task (do-put-on green blue) has been decomposed by primitive tasks {(pick-up green), (stack green blue)} in state* 0 *and reached the state* 3*. Then we add the following transitions in the DFA* $\gamma(0, \textit{(do-put-on green blue)}) \rightarrow 3$*.*

7.3.3 HTN Methods Learning

Once the task DFA is induced HierAMLSI can directly extract HTN Methods from the task DFA. However, it is possible that a large number of HTN Methods has been generated.

Example 7.6 *Let's take the task DFA in Figure 7.7. For the compound task (do-put-on green blue), HierAMLSI can generate several methods:*

$$
\begin{aligned}
\omega_1 &= \langle\rangle \\
\omega_2 &= \langle \textit{(pick-up green)}, \textit{(stack green blue)} \rangle \\
\omega_3 &= \langle \textit{(unstack blue green)}, \textit{(put-down blue)}, \textit{(pick-up green)(stack green blue)} \rangle \\
\omega_4 &= \langle \textit{(do-clear green)}, \textit{(pick-up green)(stack green blue)} \rangle \\
\omega_5 &= \langle \textit{(do-clear green)}, \textit{(do-clear blue)}, \textit{(pick-up green)(stack green blue)} \rangle \\
\omega_6 &= \langle \textit{(do-clear green)}, \textit{(do-clear blue)}, \textit{(do-put-on green blue)} \rangle \\
&\dots
\end{aligned}
$$

Some of these decompositions are redundant. To facilitate proof reading we want a more compact description of the HTN Methods. More precisely, we want minimizing the set of methods allowing to decompose observed compound tasks. Then, the HTN Methods learning problem can be reduce to a variant of the set cover problem (Karp, 1972) which is NP-Complete. The Greedy Approximation (GA) (Chvátal, 1979) is a classical way to approximate the solution in a polynomial time. GA is an iterative process which, at each stage, adds the method covering the largest number of decompositions. GA stops once all decompositions are covered by the method set. The main drawback of this approach is that it does not take into account the dependencies between tasks.

(do-clear green) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack blue green), (put-down blue)*$\rangle$
(do-clear blue) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack green blue), (put-down green)*$\rangle$
(do-put-on green blue) : $\omega_1 = \langle\rangle$
$\omega_2 = \langle$*(pick-up green), (stack green blue)*$\rangle$
$\omega_3 = \langle$*(unstack blue green), (put-down blue), (pick-up green)(stack green blue)*$\rangle$
(do-put-on blue green) : $\omega_1 = \langle\rangle$
$\omega_2 = \langle$*(pick-up blue), (stack blue green)*$\rangle$
$\omega_3 = \langle$*(unstack green blue), (put-down green), (pick-up blue)(stack blue green)*$\rangle$

(a) **Step 0:** Initialization with no compound task dependency.

(do-clear green) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack blue green), (put-down blue*$\rangle$
(do-clear blue) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack green blue), (put-down green*$\rangle$
(do-put-on green blue) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(do-clear green), (pick-up green), (stack green blue)*$\rangle$
(do-put-on blue green) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(do-clear blue), (pick-up blue), (stack blue green)*$\rangle$

(b) **Step 1:** Induction with 1 compound task dependency.

(do-clear green) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack blue green), (put-down blue*$\rangle$
(do-clear blue) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(unstack green blue), (put-down green*$\rangle$
(do-put-on green blue) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(do-clear green), (do-clear blue), (pick-up green), (stack green blue)*$\rangle$
(do-put-on blue green) : $\omega_1 = \langle\rangle, \omega_2 = \langle$*(do-clear blue), (do-clear green), (pick-up blue), (stack blue green)*$\rangle$

(c) **Step n:** Induction with n compound task dependencies.

Figure 7.8: HTN Methods Generation Example.

Example 7.7 *For example, the optimal way to decompose the compound task (do-put-on green blue) is* $\omega_1 = \langle\rangle$, $\omega_5 = \langle$*(do-clear green), (do-clear blue), (pick-up green), (stack green blue)* $\rangle$. *So the compound task (do-put-on green blue) depends of the compound tasks (do-clear green) and (do-clear blue). So, as long as all methods for the compound tasks (do-clear green) and (do-clear blue) have been generated, GA will always prioritize* ω_3 *to* ω_5. *Indeed, the decomposition* ω_5 *can be added to the current solution of GA if and only if all methods for the compound tasks (do-clear green) and (do-clear blue) have been added.*

Heuristic Approach We propose a sound, complete and polynomial heuristic approach taking into account dependencies between tasks. Figure 7.8 gives an example for the IPC Blocksworld domain[2].

AMLSI starts by initializing the set of HTN methods using the decomposition function ζ observed during the observation generation step (see Section 7.3.1). For each compound task we have therefore a set of HTN Methods containing only primitive tasks and no compound task dependencies.

Example 7.8 *For the compound task (do-put-on green blue) we have the three following*

[2]Please note that for the sake of readability the example is deliberately incomplete.

decomposition:

$$\begin{aligned}\omega_1 &= \langle\rangle \\ \omega_2 &= \langle \textit{(pick-up green)}, \textit{(stack green blue)}\rangle \\ \omega_3 &= \langle \textit{(unstack blue green)}, \textit{(put-down blue)}, \textit{(pick-up green)(stack green blue)}\rangle\end{aligned}$$

Then, at each iteration HierAMLSI uses GA to compute a new set of HTN Methods with an additional compound task dependency. Finally, if the new HTN Method set is smaller than the one learned in the previous iteration, then it is retained.

Example 7.9 *Suppose we have the two following decompositions for the compound task (do-clear blue):*

$$\begin{aligned}\omega_1 &= \langle\rangle \\ \omega_2 &= \langle \textit{(unstack green blue)}, \textit{(put-down green}\rangle\end{aligned}$$

Then, the Greedy Search return only two decompositions for the compound task (do-put-on a blue):

$$\begin{aligned}\omega_1 &= \langle\rangle \\ \omega_2 &= \langle \textit{(do-clear blue)}, \textit{(pick-up green)(stack green blue)}\rangle\end{aligned}$$

As adding a dependency reduces the number of methods required to decompose the task (do-clear blue), then these new decompositions are retained and the previous are removed.

HierAMLSI, repeats this step until it can no longer add new dependencies.

Lemma 7.1 *The Heuristic approach is sound and complete. The heuristic approach generates a set of* HTN *Methods M able to decompose all observed compound tasks in the observation set* Ω*.*

Proof 7.1 *During the observation generation step (see Section 7.3.1), for each generated compound task t, we have its final decomposition* ζ*. So at the initialization step of the Heuristic approach, there are at least one method able to decompose each observed task. The initialization is therefore sound and complete. Moreover the following steps of the Heuristic approach generates methods decomposing as many tasks as the previous steps, then the Heuristic approach is sound and complete.*

Lemma 7.2 *The Heuristic approach is polynomial.*

Proof 7.2 *First of all, we have* $\mathcal{O}(|I_+|)^3$ *states in the DFA. Then, in the worst case, we have a possible* HTN *Method for each state pair, then we have* $\mathcal{O}(|I_+|^2)$ *possible* HTN *Methods in the DFA. Then, the complexity of GA is* $\mathcal{O}(|I_+|^3)$ *in term of tested decomposition. Moreover, GA is repeated* $|C|^2$ *times. Finally, the complexity of the Heuristic approach is* $\mathcal{O}(|C|^2.|I_+|^3)$*.*

[3] $|I_+|$ denote the number of primitive tasks in the positive sample

Domain	# Primitive Task	# Compound Task	# Methods	# Predicates
Blocksworld	4	4	8	5
Gripper	3	3	4	4
Zenotravel	4	2	5	7
Transport	3	4	6	5
Childsnack	6	1	2	12

Table 7.1: Benchmark domain characteristics. From left to right, the number of Primitive Tasks, the number of Compound Tasks, the number of Methods and the number of Predicates for each IPC domain.

7.4 Experiments and evaluations

The purpose of this evaluation is to evaluate the performance of HierAMLSI through two variants: (1) we evaluate the performance of HierAMLSI when it learns separately action or methods, and (2) we evaluate the performance of HierAMLSI when it learns both methods and the action model. We use several experimental scenarios: the level of noise varies between 0% and 20% and the level of observable propositions varies between 20% and 100%.

7.4.1 Experimental setup

Our experiments are based on 5 HDDL (Höller et al., 2020; Höller et al., 2019) action models (see Table - 7.1) from the IPC 2020 competition[4]: Blocksworld, Childsnack, Transport, Zenotravel and Gripper[5].

HierAMLSI learns HTN action models from one instance. To avoid performances being biased by the initial state, HierAMLSI is evaluated with different instances. Also, for each instance, to avoid performances being biased by the generated observations, experiments are repeated ten times. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino, 2018) was used to generate the benchmark data.

7.4.2 Evaluation Metrics

HierAMLSI is evaluated using the *accuracy* (Zhuo et al., 2013) that measures the learned action model performance to solve new problems. As for the previous chapters, the accuracy is computed over 20 problems. The problems are solved with the TFD (Totally Ordered Fast Downward) planner (Pellier and Fiorino, 2020) provided by the PDDL4J library. Plan validation is done with VAL, the IPC competition validation tool (Howey and Long, 2003).

[4] `https://www.icaps-conference.org/competitions/`

[5] Experimental setup are publicly available at `https://github.com/maxencegrand/AMLSI`

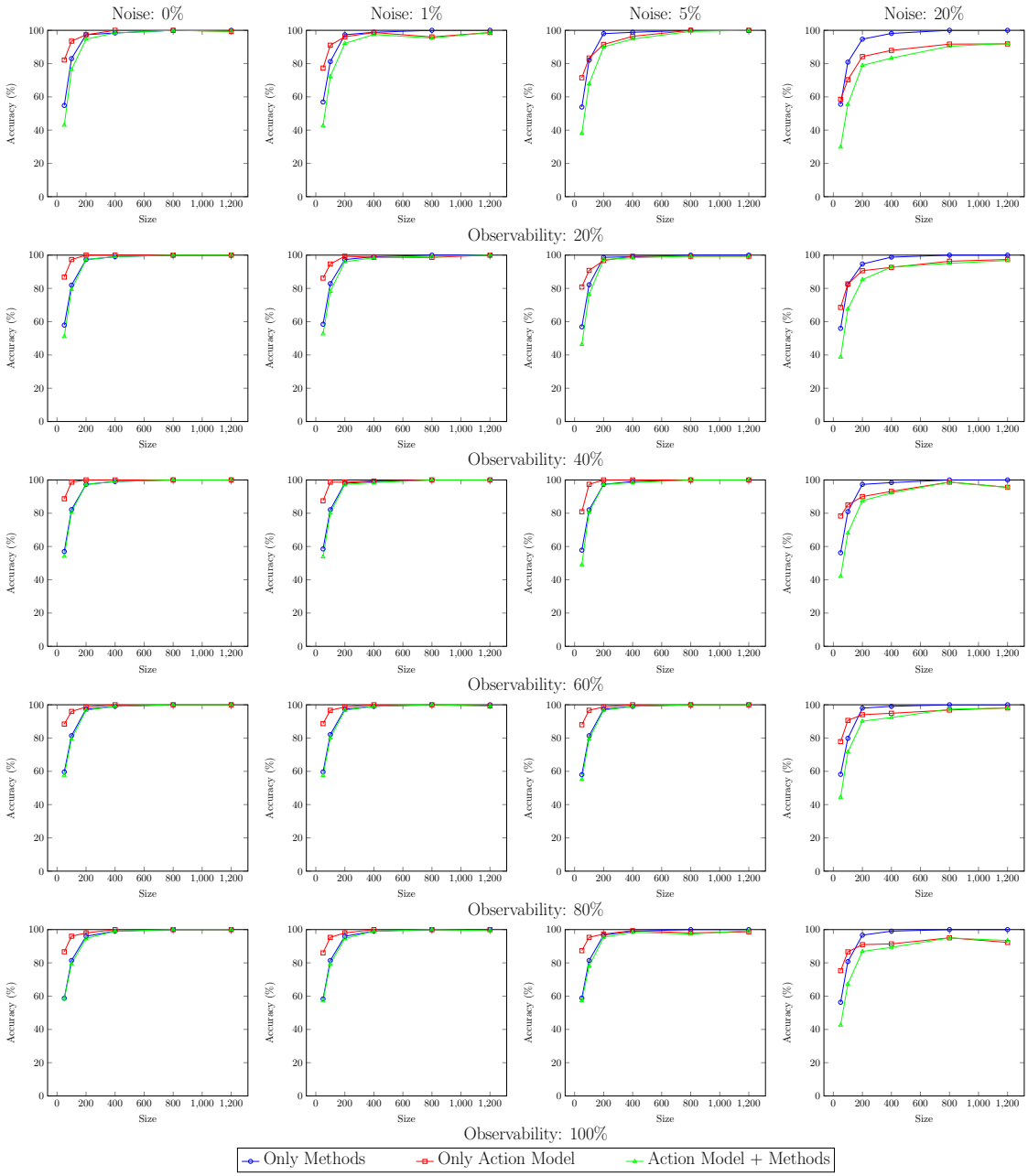


Figure 7.9: Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

7.4.3 Discussion

Figure 7.9 shows the average performance of HierAMLSI obtained on the 5 action models of our benchmarks when varying the training data set size. The size of the training set is indicated in number of tasks. HierAMLSI is tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%.

First of all, we observe that when HierAMLSI learns only the set of HTN

Methods, learned action models are generally optimal (Accuracy = 100%) with 600 tasks whatever the experimental scenario. Also, 100 tasks are generally sufficient to learn accurate action models (Accuracy $>$ 50%). Then, when HierAMLSI learns both action model and HTN Methods performances are similar when observations are noiseless. However, when observations are noisy, performances are downgraded. This is due to learning errors in the primitive task model learned. However, learned action models remain accurate when there are at least 300 tasks in the training dataset.

To conclude, we have shown experimentally that HierAMLSI learns accurate action models. More precisely, when the action model is known, HierAMLSI generally learns optimal task models. Also the performances are downgraded when AMLSI has to learn the primitive task model in addition to the set of HTN methods, but the learned task models remain accurate. Performance degradation are due to learning errors in the primitive task model.

7.5 Conclusion

In this chapter we have addressed the problem of learning HTN task models from traces with noisy and partial observations. To deal with this problem, we have we have presented HierAMLSI, an HTN extension of the AMLSI approach. This extension is composed of four steps. The first step consists in building two training sets of feasible and infeasible action sequences. In the second step, our extension induces a DFA. The third step is the generation of the HTN Methods, and the last step learns HDDL planning domain. Our experimental results show that our extension is able to learn accurately both primitive and compound task models from partial and noisy observations.

As for temporal action model learning, the learned action models are STRIPS-compliant, i.e. preconditions and effects are sets of logical propositions, it would be interesting to learn more complex preconditions and effects including logical quantifiers. Also, we have restricted ourselves to Totally Ordered HTN planning, a possible extension would be to generalize the HierAMLSI approach for Partially Ordered HTN planning.

Chapter 8

Conclusion & Perspectives

Contents

8.1 Introduction

The field of artificial intelligence aims to design and build agents able to perceive, learn and act without any human intervention to perform complex tasks. To perform complex tasks, the agent must plan the best possible actions and execute them. To do this, the agent needs an action model. An action model is a semantic representation of the actions it can execute. In an action model, an action is represented using (1) a precondition: the set of conditions that must be satisfied to execute an action, and (2) the effects: the set of properties of the world that will be altered by executing an action. STRIPS planning is a classical method to design these action models. However, STRIPS action models are generally too restrictive to be used in real-world applications. There are other forms of action models: temporal action models allowing to represent actions that can be executed concurrently, HTN action models allowing to represent actions as tasks and subtasks, etc. These models are less restrictive, but the less restrictive the models are the more difficult they are to design. In this thesis, we are interested in approaches facilitating the acquisition of these action models based on machine learning techniques.

In this thesis, we claim that to be efficient, an action model learning approach has to be able to learn action models less restrictive than STRIPS action model. Also, learning approaches require training datasets. For these approaches to

be usable in practice, the data acquisition must not be too costly. Moreover, it is imperative that the action models are sufficiently accurate to be used by planners without planning experts proofreading step. Also, the learning approach has to be robust to both partial and noisy observations, i.e. the learned action models must be accurate even if observations are partial and/or noisy. Finally, the learning approach must require few training data while avoiding the overfitting issues.

8.2 Contributions – The AMLSI Approach

In this thesis we have introduced AMLSI (Action Model Learning with State machine Interaction), a learning approach for action model acquisition.

In a first step we have shown that our approach was able to learn accurate STRIPS action models. More precisely, in Chapter 4, we presented AMLSI, a learning approach for STRIPS action models. This approach is based on the fact that STRIPS planning problems are related to state machines equivalent to regular grammars. The key idea of our approach is to learn the state machine related to the planning problem using RGI algorithms and inducing the action model from this state machine. Also, as planning problems are declared using a planning domain, our approach has to represent the STRIPS action model in the form of a PDDL planning domain. The AMLSI approach tests different actions in the environment in which the agent has to solve planning problems, observes how the environment evolves when these actions are executed, and learns the action model from its observations. We have shown experimentally that the AMLSI approach learns accurate action models even with high level of partiality and/or noise in the observations. Also, as the AMLSI approach uses both feasible and infeasible action sequences, the AMLSI approach requires few training data to learn accurate action models. Finally, we have shown that the AMLSI approach outperforms state-of-the-art approaches. Then, in Chapter 5, we have presented IncrAMLSI, an incremental extension of the AMLSI approach. This incremental extension allows us to take into account that training data acquisition beging a long term evolutive process.

In a second step we have extended the AMLSI approach to learn less restrictive action models. First of all, in Chapter 6, we have presented TempAMLSI, an accurate temporal extension of the AMLSI approach for both sequential and SHE temporal action models robust to partial and noisy observations. The key idea of the TempAMLSI approach is to use translation techniques used by some planners Fox and Long (2002a); Halsey et al. (2004); Celorrio et al. (2015); Furelos Blanco et al. (2018) for the learning problem. Like AMLSI, TempAMLSI interacts with the environment to generate input feasible and infeasible action sequences to frame what is allowed by the targeted action model. Then, the temporal learning consists of three steps: (1) TempAMLSI translates temporal sequences into STRIPS sequences, (2) TempAMLSI learns a non-temporal action model with AMLSI, and then (3) translates it into a

temporal action model. Then, in Chapter 7, we have presented HierAMLSI, an accurate HTN extension of the AMLSI approach for both compound and primitive task models robust to partial and noisy observations. HierAMLSI is built on AMLSI (see Chapter 4). Like AMLSI, HierAMLSI interacts with the environment to generate input feasible and infeasible action sequences to frame what is allowed by the targeted task model. Then, HierAMLSI learns the state machine, generates HTN methods and induces HTN task models.

8.3 Perspectives

Our work has many perspectives. We develop here the main ones.

8.3.1 AMLSI Extension

A first perspective is to extend the AMLSI approach.

The main limitation of the AMLSI approach is the expressiveness of the learned action models. Although that the AMLSI approach learns less restrictive models than STRIPS, the preconditions and effects of actions are STRIPS-compliant, i.e. preconditions and effects are sets of logical propositions. However, it is possible to have more complex preconditions and effects. First of all, the ADL Pednault (1994) formalism allows to include logical quantifier in preconditions and effects. During this thesis, we tested an extension of AMLSI for ADL action models. However, this extension was not able to learn accurate action models when the observations were noisy and/or partial. Also, the AMLSI approach does not learn any numerical features. However, some description languages and AI planning formalism allow to take into account numerical features such as probabilistic effects Younes and Littman (2004); Sanner (2010), numerical function and fluent Fox and Long (2003, 2002b, 2006). A first perspective for our approach will be to extend it to respond to these limitations.

As we have stated in Chapter 2, planning problems less restrictive than STRIPS problems are more complex to solve. ML-based techniques have been proposed to facilitate the resolution of these problems such as macro actions Dawson and Siklóssy (1977); Korf (1985); Botea et al. (2005); Castellanos-Paez et al. (2018), generalized policies Minton (2012); Borrajo and Veloso (1997); de la Rosa et al. (2007, 2008) and heuristics De La Rosa et al. (2009); Yoon et al. (2006) learning. The objective of these methods is to learn macro, policy or heuristics to facilitate the resolution of planning problems from execution traces. For example, macro actions are actions composed of several actions. For example, for Blocksworld, we could have the macro action *pick-up-stack* composed of the actions *pick-up* and *stack*. To learn these macros, the learning approaches typically take as input a set of solution plans and return the macros that have been observed often. A possible perspective could be to reuse the key idea of AMLSI to learn macros, policies and heuristics: learn the state machine related

to the planning problem using grammar induction algorithms and induce, from this state machine, macros, policies and heuristics. More generally, we could extend the AMLSI approach to be able to both learn action models and learn structured knowledge in order to efficiently solve planning problems using the learned action models.

8.3.2 Applications

As we have seen before, the main interest of approaches learning action models is to facilitate the acquisition of these models in order to use them in real-world applications such as aerospace Fisher et al. (2000); Backes et al. (2004); Bresina et al. (2005), autonomous vehicles Urmson and Whittaker (2008), logistics Cross and Walker (1994), robotics Dvorak et al. (2014); Lallement et al. (2018); Liang et al. (2022), industry Hoffmann et al. (2009), cybersecurity Edelkamp et al. (2009). A natural perspective for our work would therefore be to use the AMLSI approach to facilitate the action model acquisition in these real-world applications.

A second application would be to take advantage of the interactive aspect of the AMLSI approach to facilitate the development of tools using end-user interactions. For example the programming of robots, and more precisely, the Programming by Demonstration (PbD) Billard et al. (2008) of robots. PbD is an end-user programming technique for teaching a robot new skills by demonstrating them. AMLSI can be adapted to this context to learn action models. As we have seen before, the first step of the AMLSI approach is a query phase. In the context of PbD this query phase could be the interaction between the robot and the user: the robot could ask the user about feasible and infeasible actions or tasks. In this context, the demonstrations would be the execution of the generated action sequences.

Finally, in this thesis we took advantage of the fact that planning problems are equivalent to grammars to learn action models. Also, we have seen in Chapter 3 that grammatical induction has several applications, e.g. *Syntactic and Structural Pattern Recognition*, *Natural Language Processing* Adriaans and van Zaanen (2004); Dupont et al. (2008); Boström (1996); Boström (1998); Cruz-Alcázar and Vidal (1998); Bex et al. (2006); Cruz-Alcázar and Vidal (2008); Stein et al. (2006); Bréhélin et al. (2001); Raffelt and Steffen (2006); Berg et al. (2006). A final application could be to use AMLSI in these application settings when a representation in the form of action models can have benefits. For example, in the field of System Behavior Modeling, grammar induction algorithms are used to discover the behavior of a system, software, industrial process etc. The behavior of these systems are represented as a grammar and can then be analyzed to automatize them Dupont et al. (2008), detect intrusions Su and Wassermann (2006); Godefroid et al. (2008) etc. In this context, AMLSI could be used to learn action models representing the behavior of these systems. The advantage would be to have a more compact and readable representation of these systems. Also, the learned action models could be directly used to

automatize these systems, to detect intrusions Edelkamp et al. (2009) etc.

Annexes

Nomenclature

ω	A sequence.
δ	Action model.
γ	Transition function.
σ	HTN method decomposition function.
τ	Feasabili function.
ζ	Decomposition function.
A	Set of actions/Alphabet.
add	Set of positive effects.
C	The set of compound tasks.
d	Duration function.
del	Set of negative effects.
G	Set of goal states.
L	Set of logical propositions.
M	Set of HTN methods.
P	A planning problem.
prec	Set of preconditions.
S	Set of states.
s_0	Initial state.
at-end	Time label for the end of a durative action.
at-start	Time label for the start of a durative action.

end-a End event of a durative action a.

overall Time label for the whole duration of a durative action.

start-a Start event of a durative action a.

Δ A set of planning operators.

Δ_P The set of planning operators accepting $\mathcal{L}(P)$ and generating λ.

λ Observation function.

$postset(a)$ The set of states where a is an ingoing transition.

$preset(a)$ The set of states where a is an outgoing transition.

$\mathcal{L}_{\mathcal{C}}(P)$ The language defined by the state transition system.

$\mathcal{L}_{\mathcal{H}}(P)$ The language defined by the decomposition hierarchy.

$\mathcal{L}(P)$ The language accepted the set of solution plans of the planning problem P.

Ω The training dataset.

$\overline{\mathcal{L}}$ The complement language of $\mathcal{L}$.

Π A Partition.

Σ An automaton.

$BS_{MCA}(I_+, I_-)$ The border set.

I_+ A positive sample.

I_- A negative sample.

Lat A lattice.

$MCA(I_+)$ The Maximal Canonical Automaton.

$Pr(\mathcal{L})$ The prefix set of $\mathcal{L}$.

$PTA(I_+)$ The Prefix Tree Acceptor.

$\mathcal{L}$ A language.

$\mathcal{L}(\Sigma)$ The language accepted by Σ.

$\mathcal{L}(H(\omega_j))$ The language induced after reading the j first element of Ω.

$\mathcal{L}/u$ the right quotient of $\mathcal{L}$.

List of Figures

List of Tables

Appendix A

Publications and Submissions

Here is the list of papers that have been published or submitted during the Ph.D.:

A.1 Conference

1. M. Grand, H. Fiorino, D. Pellier. Retro-engineering state machines into PDDL domains. In Proc. of the International Conference on Tools in Artificial Intelligence, 2020, pages 1186–1193.

2. M. Grand, H. Fiorino and D. Pellier. IncrAMLSI: Incremental Learning of Accurate Planning Domains from Partial and Noisy Observations. In Proc. of the International Conference on Tools in Artificial Intelligence, 2021, pages 121–128.

3. M. Grand, H. Fiorino and D. Pellier. TempAMLSI : Temporal Action Model Learning based on Grammar Induction. In Proc. of the International Conférence on Planning and Scheduling, 2022, pages 597–605.

4. M. Grand, D. Pellier, H. Fiorino. An Accurate PDDL Domain Learning Algorithm from Partial and Noisy Observations. In Proc. of the International Conference on Tools in Artificial Intelligence, 2022, To appear.

A.2 Workshop

1. M. Grand, H. Fiorino, D. Pellier. AMLSI: A Novel and Accurate Action Model Learning Algorithm. In Proc. of the International Workshop on Knowledge Engineering for Planning and Scheduling, ICAPS 2020.

2. M. Grand, H. Fiorino and D. Pellier. TempAMLSI : Temporal Action Model Learning based on Grammar Induction. In Proc. of the International workshop of Knowledge Engineering for Planning and Scheduling, 2021

3. M. Grand, H. Fiorino and D. Pellier. An Accurate HDDL Domain Learning Algorithm from Partial and Noisy Observations. In Proc. of the International workshop of Knowledge Engineering for Planning and Scheduling, 2022

4. M. Grand, H. Fiorino and D. Pellier. An Accurate HDDL Domain Learning Algorithm from Partial and Noisy Observations. In Proc. of the Hierarchical Planning Workshop, 2022

5. M. Grand. Action Model Learning based on Grammar Induction. In Proc. of the Doctoral Consortium of the International Conférence on Planning and Scheduling, 2022

Appendix B

Experiments – Detailed Results

Contents

B.1 Introduction

In this chapter, we present all the experiments performed in the framework of this thesis. Table B.1 shows benchmarks action model characteristics. A complete description of all action models in the IPC benchmark is given in annexes C.

STRIPS **Learning:** Our experiments are based on 13 STRIPS-Compliant IPC benchmarks: Blocksworld, Gripper, Hanoi, N-Puzzle, Peg Solitaire, Parking, Zenotravel, Sokoban, Visit All, Elevator, Spanner, Logistics and Floortile. Table - B.1a shows our experimental setup.

Temporal Learning: Our experiments are based on 5 temporal IPC action models (see Table B.1b). More precisely we test TempAMLSI with three Sequential action models (Peg Solitaire, Sokoban, Zenotravel), and two SHE action models (Match, Turn and Open).

HTN: **Learning:** Our experiments are based on 5 HDDL (Höller et al., 2020; Höller et al., 2019) action models (see Table - B.1c) from the IPC 2020 competition: Blocksworld, Childsnack, Transport, Zenotravel and Gripper.

Domain	*#Operators*	*#Predicates*	E_+	E_-	ω_+	ω_-
Blocksworld	4	5	100	32546	49.8	33.8
Gripper	3	4	100	13163	51.3	33.9
Hanoi	4	7	100	34600	50.3	33.7
N-Puzzle	1	3	100	36626	49.9	33.7
Peg-Solitaire	3	4	100	14508	6.9	5.3
Parking	4	5	100	64963	50.6	34.0
Zenotravel	5	5	100	18154	50.4	33.9
Sokoban	2	4	100	40302	50.2	33.8
Visit All	4	7	100	16702	50.9	35.7
Elevator	4	6	100	13122	51.0	35.7
Spanner	4	6	100	4628	7.0	5.1
Logistics	6	3	100	31622	49.7	32.9
Floortile	6	10	100	48773	51.0	37.1

(a) From left to right: the number of operators, the number of predicates, the average size of the E_+ and the E_- testing dataset, and the average length of the positive (resp. negative) testing sequences $\omega_+ \in E_+$ (resp. $\omega_- \in E_-$).

Domain	# Operators	# Predicates	Type	$\lvert E_+ \rvert$	$\lvert E_- \rvert$	$\lvert \omega_+ \rvert$	$\lvert \omega_- \rvert$
Peg Solitaire	1	3	Sequential	100	4309	3.9	3.7
Sokoban	2	3	Sequential	100	57165	25.0	13.4
Zenotravel	5	4	Sequential 100	22711	25.0	13.5	
Match	2	4	SHE	100	3259	4.0	4.3
Turn and Open	5	8	SHE	100	23148	25.0	13.1

(b) From left to right: the number of operators and predicates, the temporal action model type, the average size of the E_+ and the E_- testing dataset, and the average length of the positive (resp. negative) testing sequences $\omega_+ \in E_+$ (resp. $\omega_- \in E_-$).

Domain	# Primitive Task	# Compound Task	# Methods	# Predicates	E_+	E_-	ω_+	ω_-
Blocksworld	4	4	8	5	100	5804	45.5	30.1
Gripper	3	3	4	4	100	8336	38.4	26.5
Zenotravel	4	2	5	7	100	3575	32.4	22.3
Transport	3	4	6	5	100	5710	42.0	28.0
Childsnack	6	1	2	12	100	25256	29.6	23.7

(c) From left to right, the number of Primitive Tasks, the number of Compound Tasks, the number of Methods, the number of Predicates for each IPC action model, the average size of the E_+ and the E_- testing dataset, and the average length of the positive (resp. negative) testing sequences $\omega_+ \in E_+$ (resp. $\omega_- \in E_-$).

Table B.1: IPC Benchmark Domain Characteristics.

For each experiment, we test each IPC action model with 3 different initial states over ten runs, and we use ten randomly generated seeds for each run. Also, we generate partial observations by randomly removing a fraction of the propositions of the states, and we generate noise by changing the value of a fraction of the observable propositions. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz

with 16GB of memory. PDDL4J library (Pellier and Fiorino, 2018) was used to generate the benchmark data.

B.2 Evaluation metrics

We evaluate AMLSI and all its extensions with three different metrics: the *syntactical error* (Zhuo et al., 2010b) that computes the distance between the original action model and the learned model, the *accuracy* (Zhuo et al., 2013) that expresses the capability of the learned action model to solve new problems (without proofreading). Even though the syntactical error is the most used metric in the literature, we argue that the accuracy is the most important metric *in practice* for planning because it measures to what extent a learned action model is useful. Indeed, it often happens that one missing precondition or effect, which amounts to a small syntactical error, makes the learned action model unable to solve planning problems. Finally, the last metric is the *FScore* that expresses the capability of the learned action model to generate the grammar related to the planning problem.

Formally, the syntactical error $error(o)$ for an operator is the Hamming distance between the learned operator and the ground truth operator, i.e. the number of extra or missing predicates in the preconditions $prec(o)$, the positive effects $add(o)$ and the negative effects $del(o)$ divided by the total number of possible predicates. By extension, the syntactical error for an action model composed of a set of operator O is:

$$E_\sigma = \frac{1}{|O|} \sum_{o \in O} error(o)$$

Then, FScore $= \frac{2.P.R}{P+R}$ where R is the recall, i.e. the rate of sequences e accepted by the original IPC action model that are successfully accepted by the learned action model, computed as $R = \frac{|\{e \in E^+ \mid accept(\delta, e)\}|}{|E^+|}$, and P is the precision, i.e. the rate of sequences e accepted by the learned action model that are also accepted by the original IPC action model, computed as $P = \frac{|\{e \in E^+ \mid accept(\delta, e)\}|}{|\{e \in E^+ \mid accept(\delta, e)\} \cup \{e \in E^- \mid accept(\delta, e)\}|}$. The test sets E^+ and E^- used to compute the FScore are generated by random walks.

Finally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learned action model, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. STRIPS problems are solved with Fast Downward v19.06 (Helmert, 2006), Temporal problems are solved with the TP-SHE (Celorrio et al., 2015) planner and HTN problems are solved with the TFD planner (Pellier and Fiorino, 2020) provided by the PDDL4J library. For each experiment, plan validation is done with VAL (Howey and Long, 2003), which is used in the IPC competitions. In addition to the Accuracy we report the IPC score in order to compute the quality of generated plans. The score of an action model on a solved

problem is the ratio between the length of a reference plan, i.e. a plan generated by the reference IPC action model, and the length of the plan generated by the learned action model. The score on an unsolved problem is 0. The score of a learned action model is the sum of its scores for all problems.

B.3 Discussion

B.3.1 STRIPS Learning

Figures B.1 – B.48 show the average performance of AMLSI and LSONIO obtained on the 13 action models of our benchmarks when varying the training dataset size. The size of the training set is indicated in number of actions. AMLSI and LSONIO are tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%. We have three variants of AMLSI: (B) *Base:* DFA learning is done without Pairwise Sequences (PS) and without Tabu Search, (B+PS) *Base + PS:* DFA learning is done with PS but without Tabu Search, and (B+PS+Tabu) *Base + PS + Tabu:* DFA learning is done with PS and with Tabu Search during the refinement step.

Comparison with LSONIO We observe that AMLSI outperforms LSONIO whatever the size of the learning dataset in terms of accuracy or in terms of syntactical distance. We also observe that AMLSI needs very little data to obtain a relatively large accuracy (almost 90% with only a learning dataset of 200 actions) in the most difficult scenario.

Ablation study The Base+PS variant is more robust to partial observations than the Base variant of AMLSI. This is due to the fact that DFA learned with PS are generally better that action models learned without PS. More precisely, DFA learned with PS generally have fewer states and fewer transitions. This allows for fewer false transitions which makes it easier to learn effects and preconditions. However, when observations are noisy, the Base+PS variant is not able to learn action models accurate enough to be used for planning whatever the level of observability. Only the Base+PS+Tabu variant is both robust to partial and noisy observations. Our ablation study confirms that adding unobserved Pairwise Sequences improves the learning of the DFA, and makes AMLSI more robust to partial observations while refining the preconditions and the effects by using a Tabu Search allows AMLSI to learn accurate action models with a high level of noise.

Convergent Learning Figures B.53a – B.63d shows the average performance of IncrAMLSI obtained on the 13 action models of our benchmarks when varying the convergence criterion T. IncrAMLSI is tested with 20 experimental scenarios:

the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%. Whatever the experimental scenario, IncrAMLSI learns from accurate models. Also, increasing T generally leads to better results. Finally, we can observe that whatever the value of T and the experimental scenario, IncrAMLSI converges in less than 35 iterations.

Finally, we observe that the plans generated with the learned action models are generally a little longer than plansg generated with the IPC action models, even with optimal accuracy (ipc < 20). When the accuracy is not optimal, we notice that the IPC score is close to the ratio between the ipc score with optimal accuracy and the rate of solved problems, this implies that even when the accuracy is not optimal the plans are not much longer than the original plans.

B.3.2 Temporal Learning

Figures B.65 – B.84 shows the average performance of HierAMLSI obtained on the 5 action models of our benchmarks when varying the training data set size. The size of the training set is indicated in number of tasks. HierAMLSI is tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%.

When observations are complete and noiseless we observe that we observe that both variants learn optimal action models. Indeed, FScore and accuracy are optimal. However, we can observe that the syntactical distance is not optimal for some action models.this is due to the fact that some at start and at end effect preconditions are encoded as overall preconditions and vice versa.

Also, We observe that 2-Operators and 3-Operators variants are accurate when the level of noise is not high ($<$ 20%) whatever the level of observability. Only the 2-Operators variant is accurate with high level of noise. Also, we observe that the 2-Operators variant is generally more robust than the 3-Operators variant. The fact that the 2-Operators variant is more robust than the 3-Operators variant can be explained in different ways. First of all, the fact that action sequences of 2-Operators variants are shorter than action sequences of 3-Operators variants makes DFAs easier to learn since they have fewer states. The better the DFA learning, the better the operator learning. Moreover, it is easier for 2-Operators variants than for 3-Operators variants because 2-Operators variants have less operators.

Finally, as for STRIPS learning, we observe that the plans generated with the learned action models are generally a little longer than plans generated with the IPC action models. However, we notice that the IPC score is close to the ratio between the IPC score with optimal accuracy and the rate of solved problems, this implies that even when the accuracy is not optimal the plans are not much longer than the original plans.

B.3.3 HTN Learning

The purpose of this evaluation is to evaluate the performance of HierAMLSI though two variants: (1) we evaluate the performance of HierAMLSI when only HTN Methods are learned, i.e. the action model is known and (2) we evaluate the performance of HierAMLSI when both HTN Methods are learned and the action model is unknown. We use several experimental scenarios: the level of noise varies between 0% and 20% and the level of observable propositions varies between 20% and 100%.

Figures B.85 – B.99 show the average performance of HierAMLSI obtained on the 5 action models of our benchmarks when varying the training data set size. The size of the training set is indicated in number of tasks. HierAMLSI is tested with 20 experimental scenarios: the level of observability varies between 20% and 100% and the level of noise varies between 0% and 20%.

First of all, we observe that when HierAMLSI learns only the set of HTN Methods, learned action models are generally optimal (Accuracy $=$ 100%) with 600 tasks whatever the experimental scenario. Also, 100 tasks are generally sufficient to learn accurate action models (Accuracy $>$ 50%). Then, when HierAMLSI learns both action model and HTN Methods performances are similar when observations are noiseless. However, when observations are noisy, performances are downgraded. This is due to the fact that there are learning error in the primitive task model learned. However, learned action models remain accurate when there are at least 300 tasks in the training dataset.

Finally, as for STRIPS learning, we observe that the plans generated with the learned action models are generally a little longer than plans generated with the IPC action models. However, we notice that the IPC score is close to the ratio between the IPC score with optimal accuracy and the rate of solved problems, this implies that even when the accuracy is not optimal the plans are not much longer than the original plans.

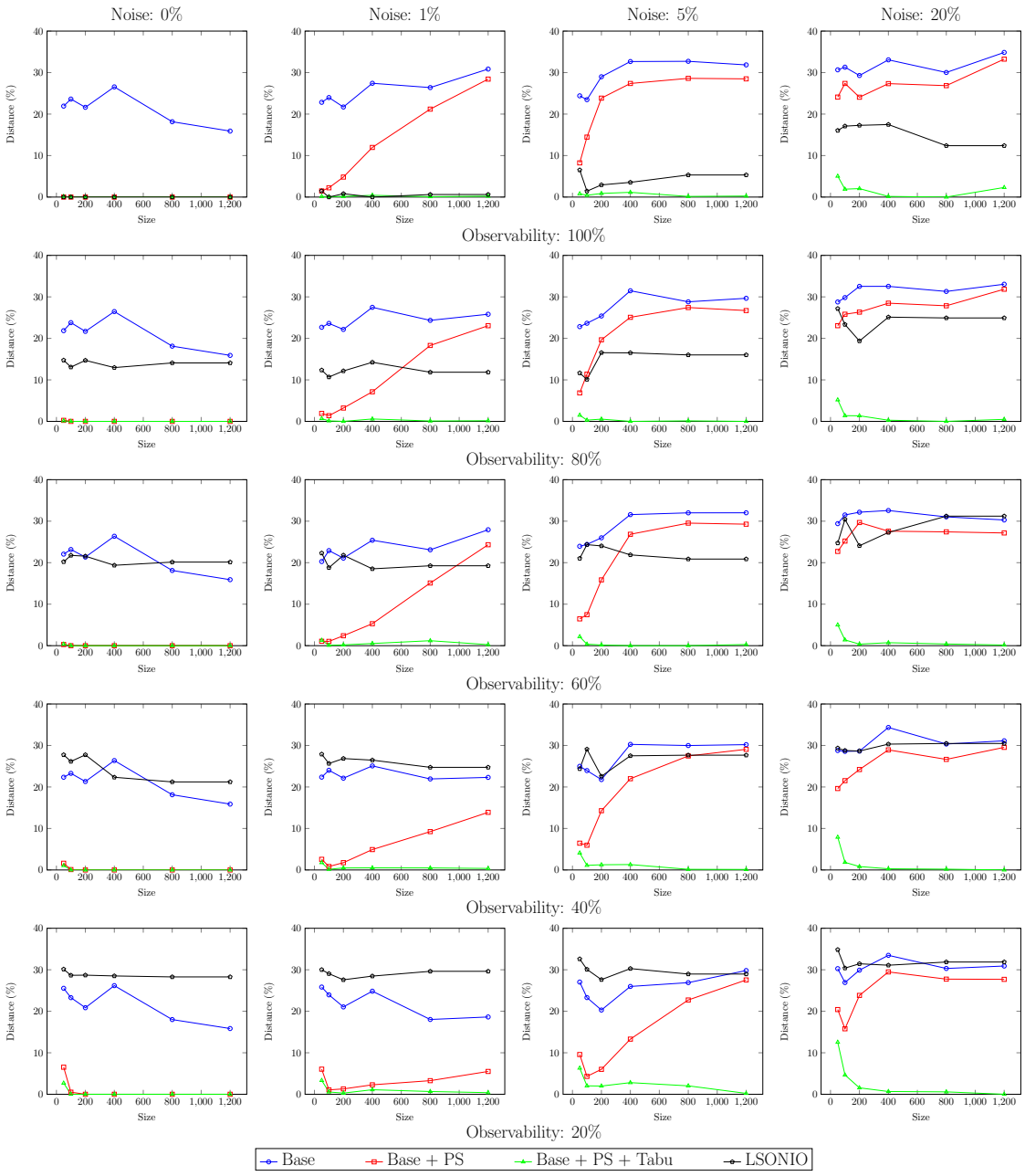


Figure B.1: Blocksworld – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

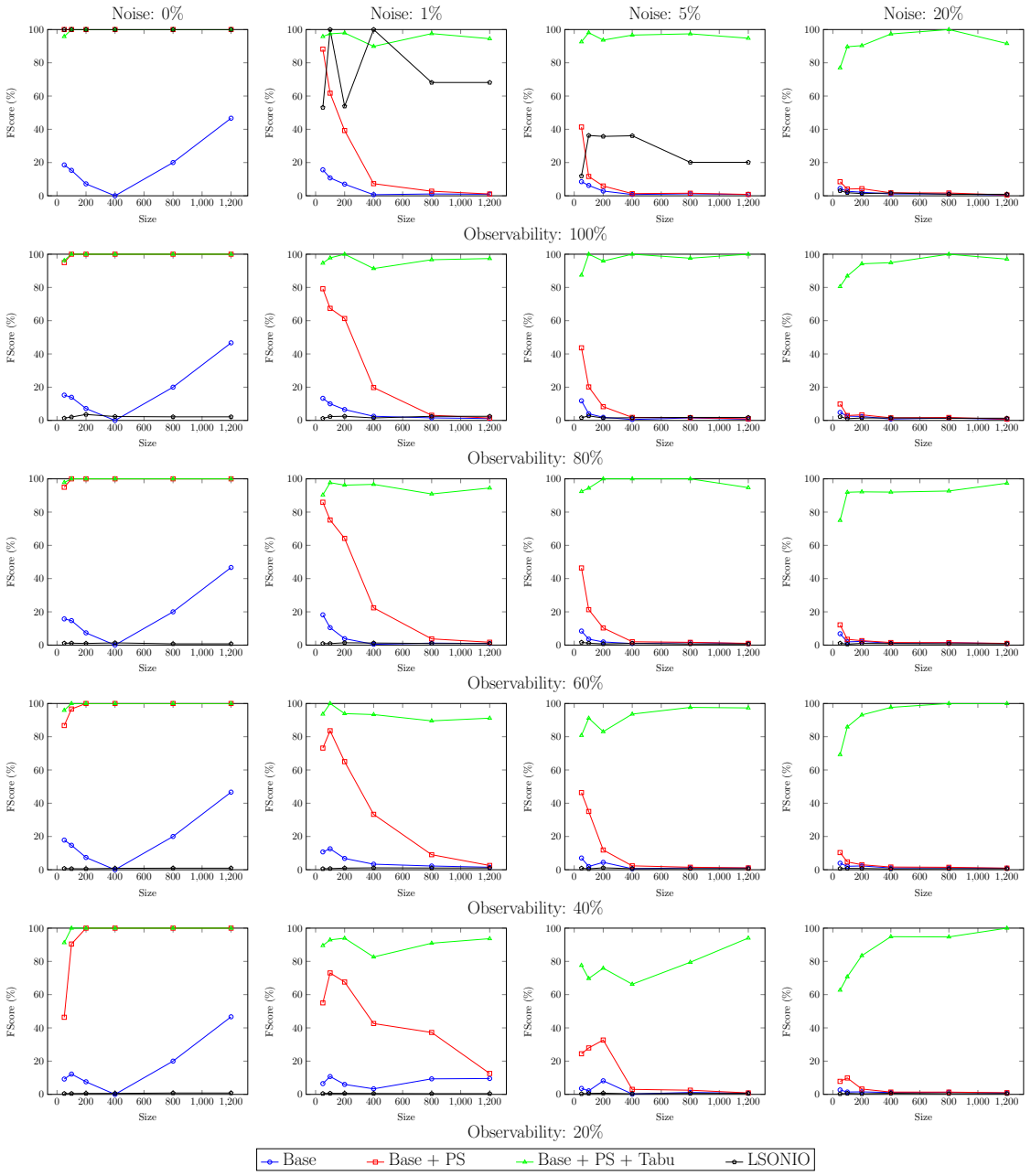


Figure B.2: Blocksworld – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

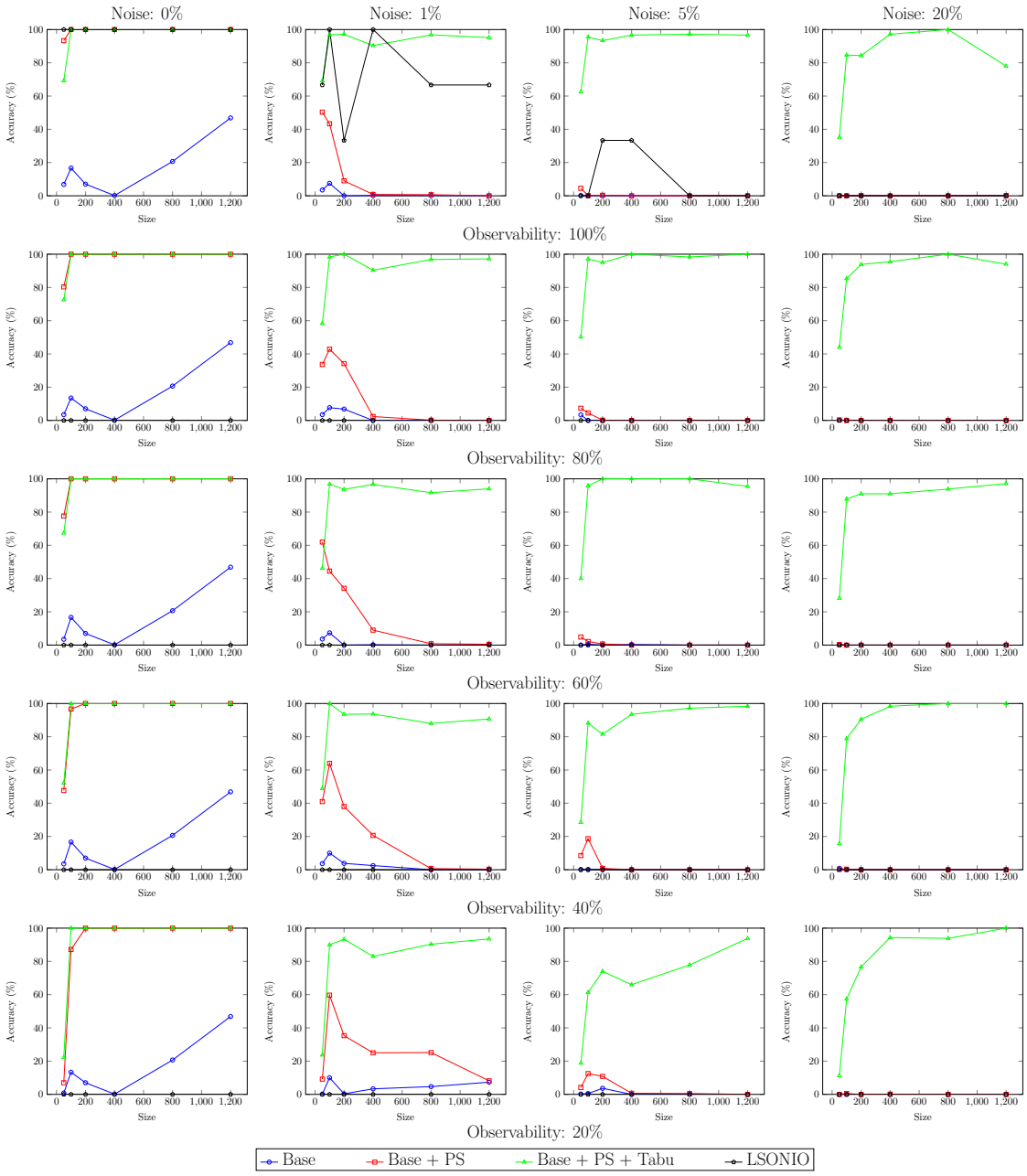


Figure B.3: Blocksworld – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

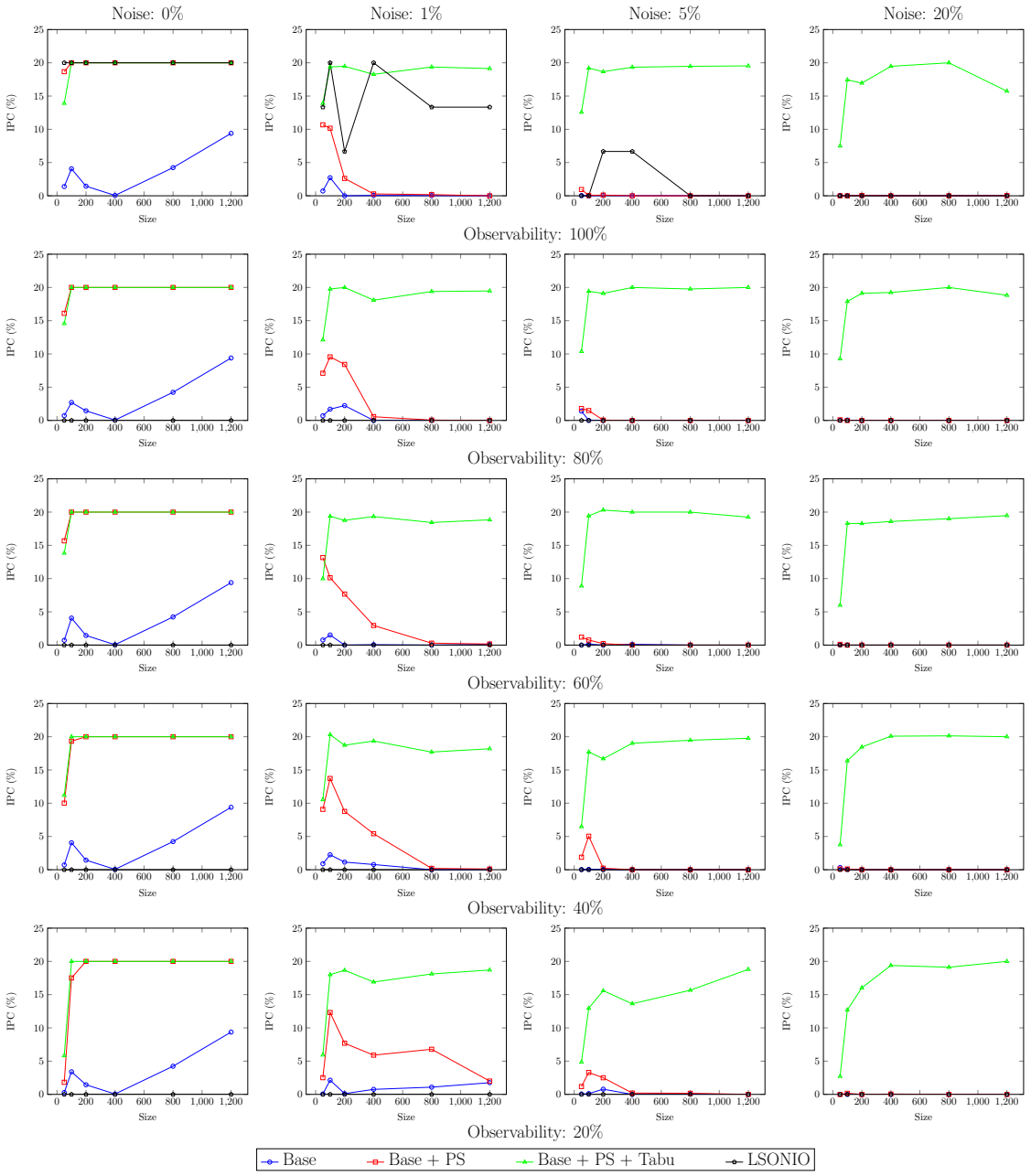


Figure B.4: Blocksworld – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

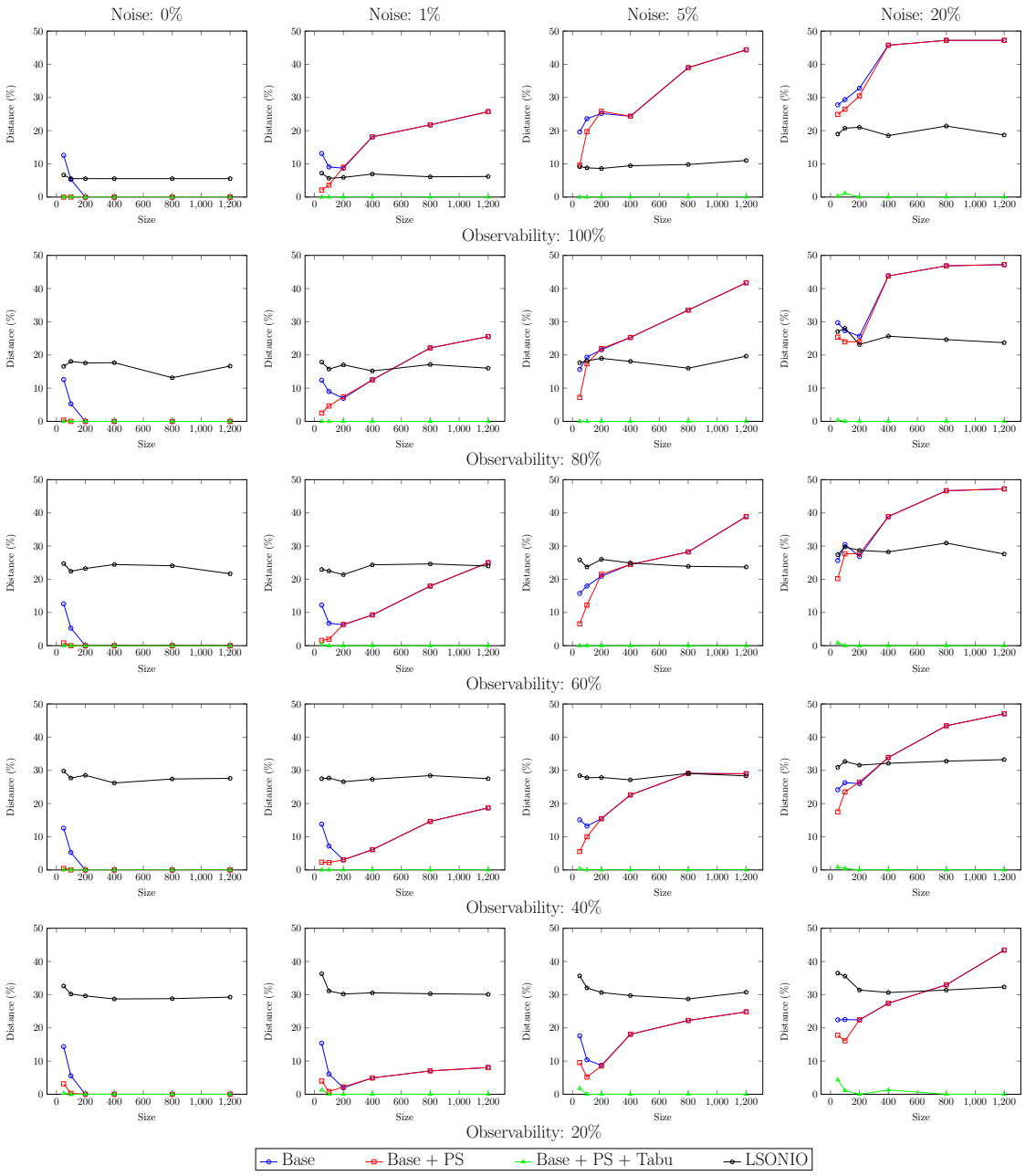


Figure B.5: Gripper – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

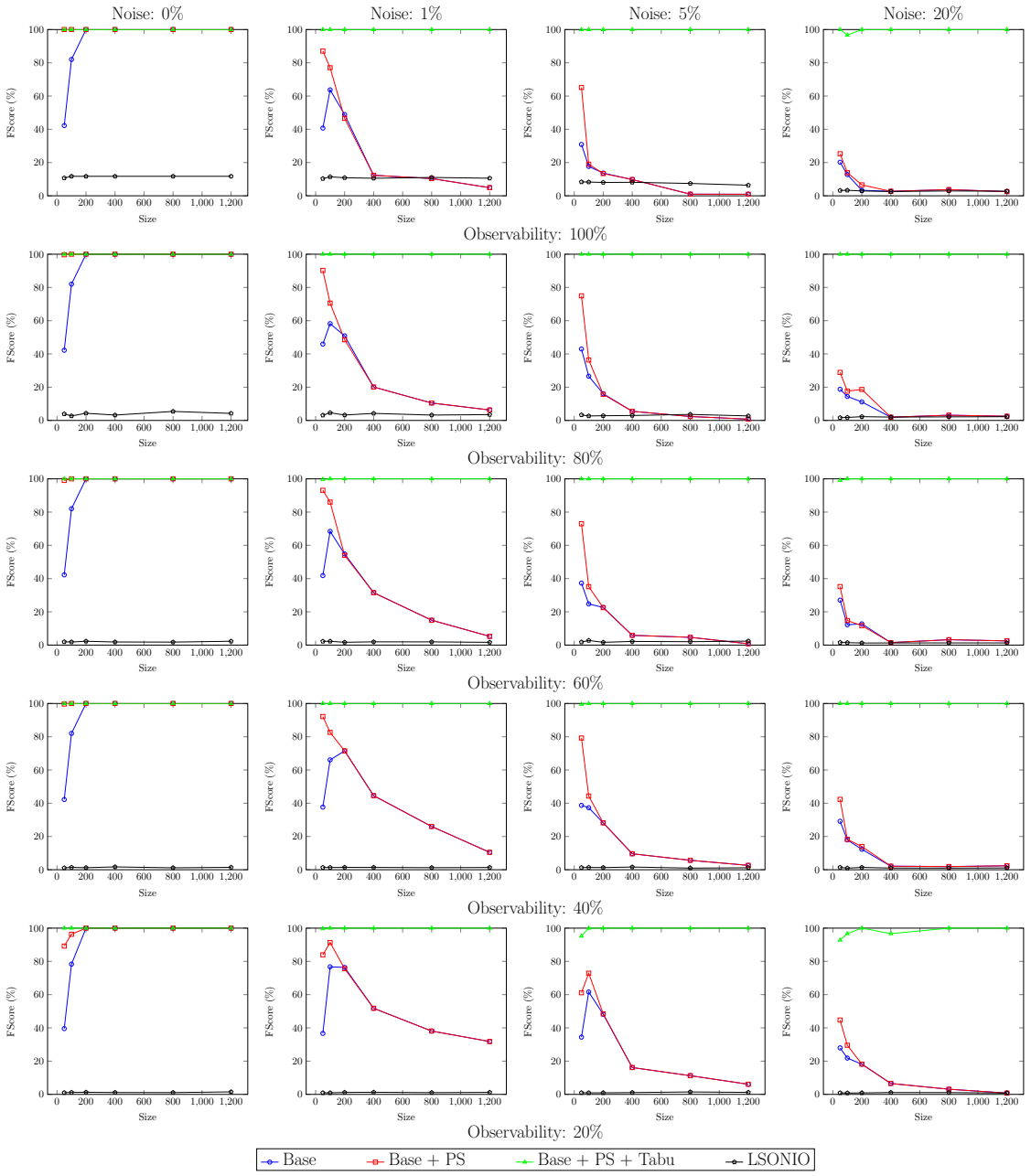


Figure B.6: Gripper – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

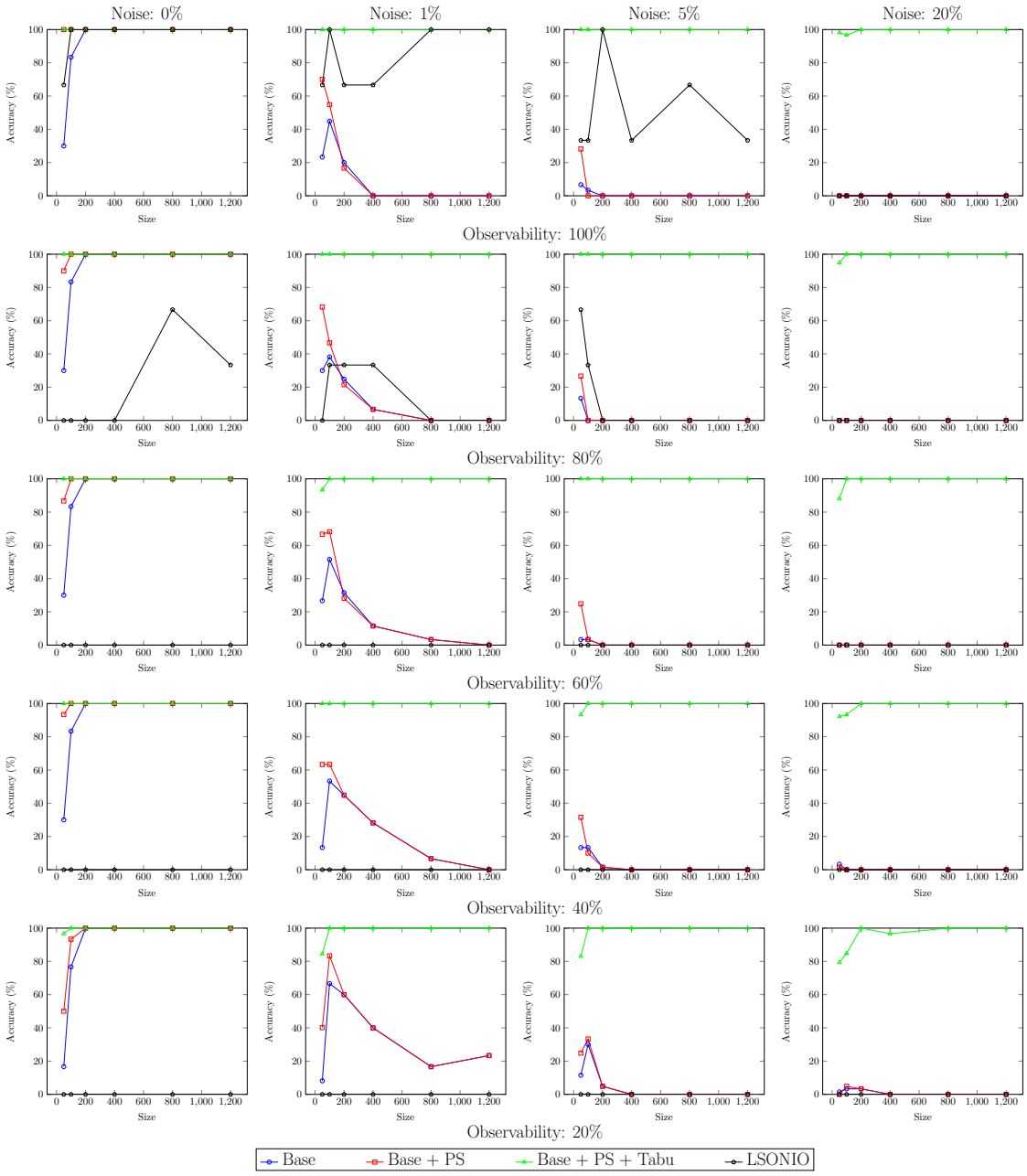


Figure B.7: Gripper – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

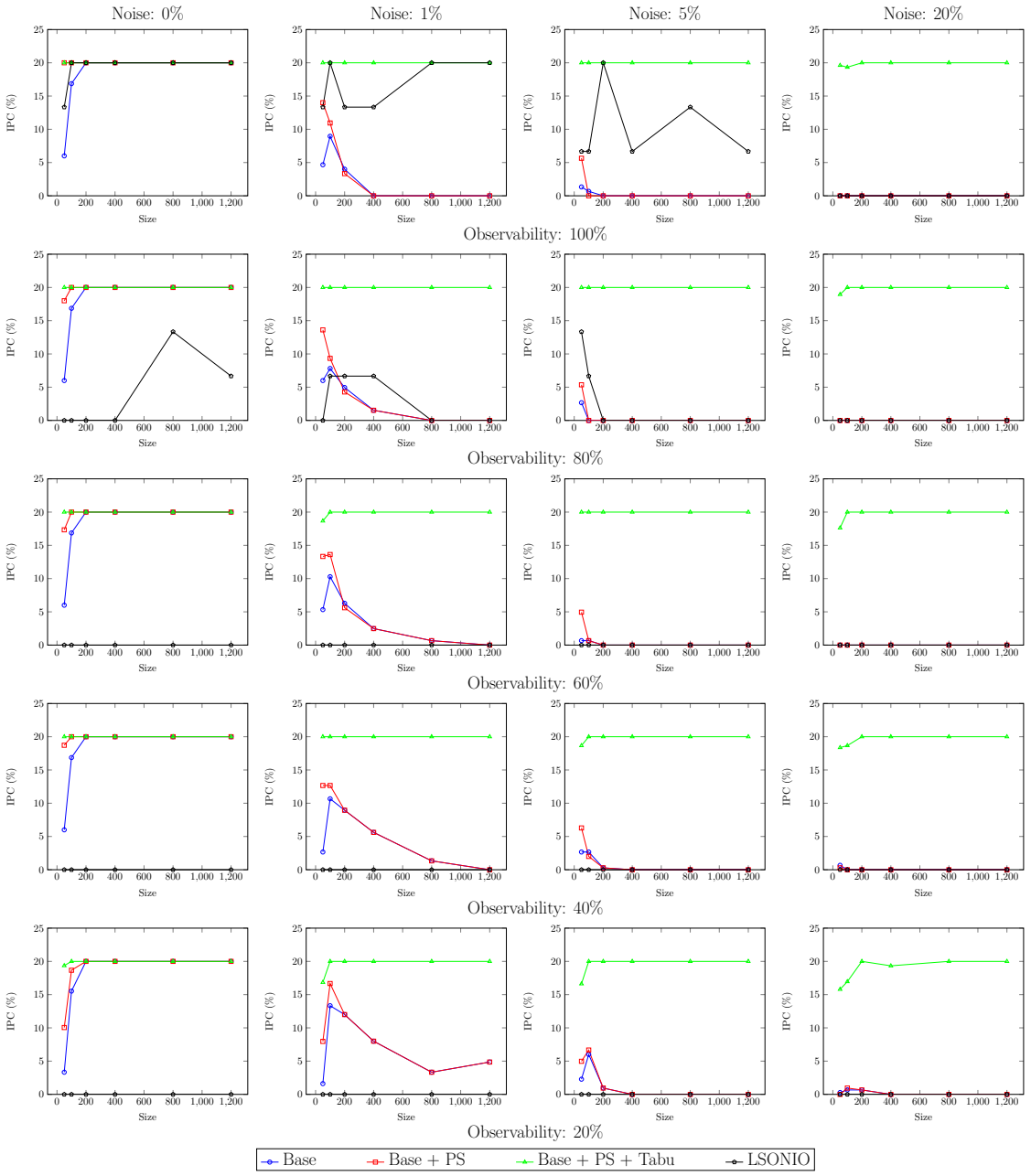


Figure B.8: Gripper – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

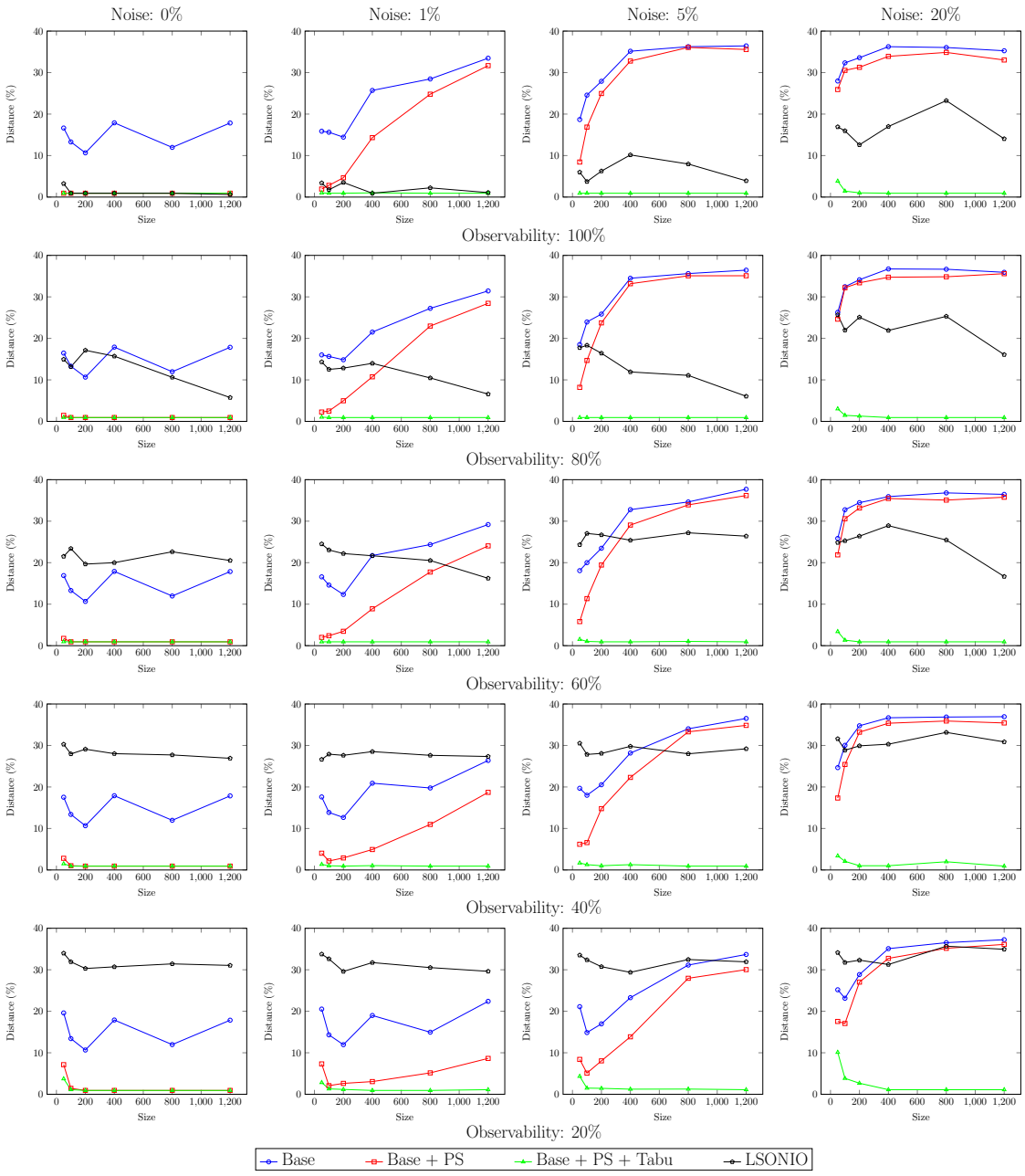


Figure B.9: Hanoi – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

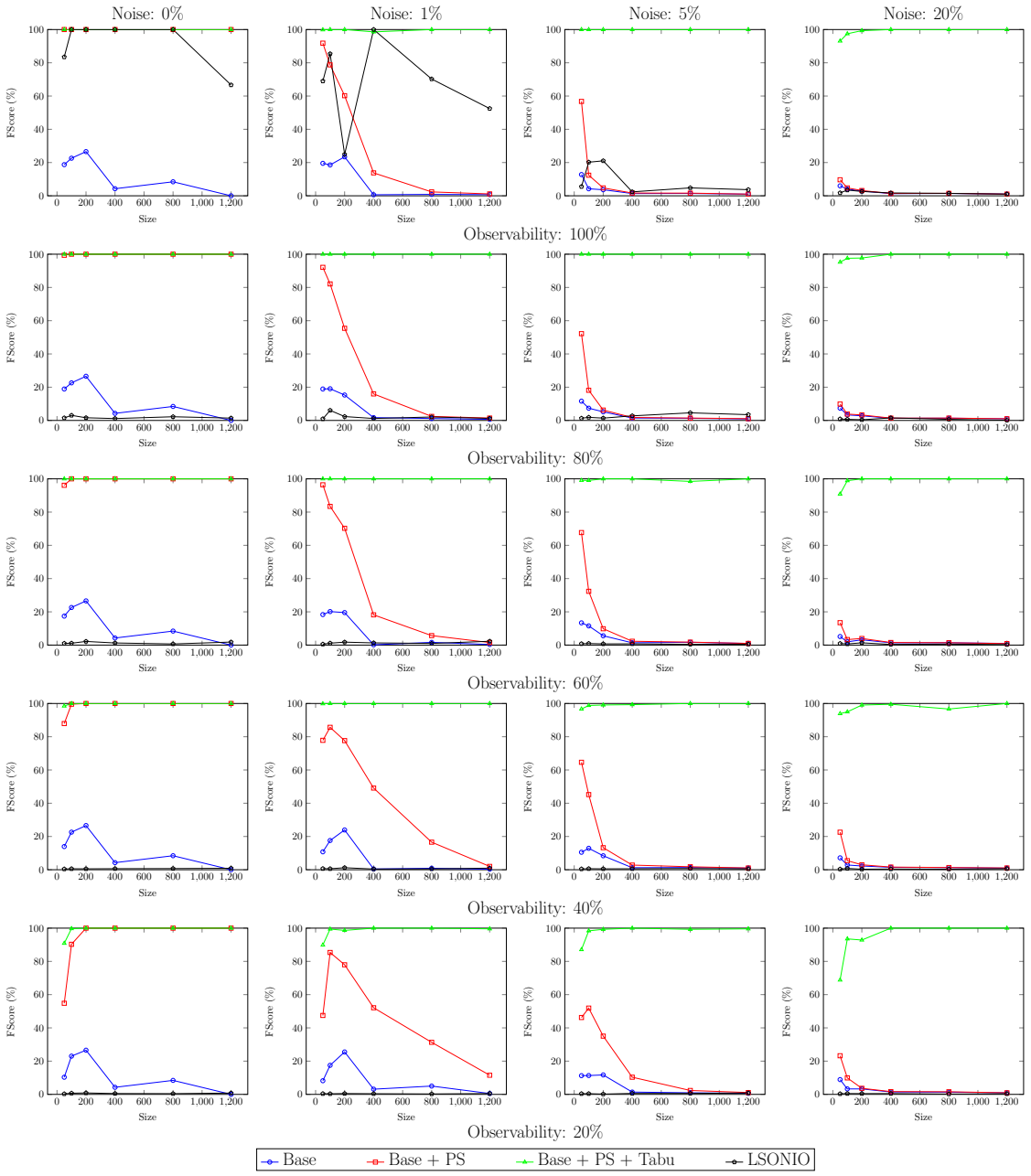


Figure B.10: Hanoi – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

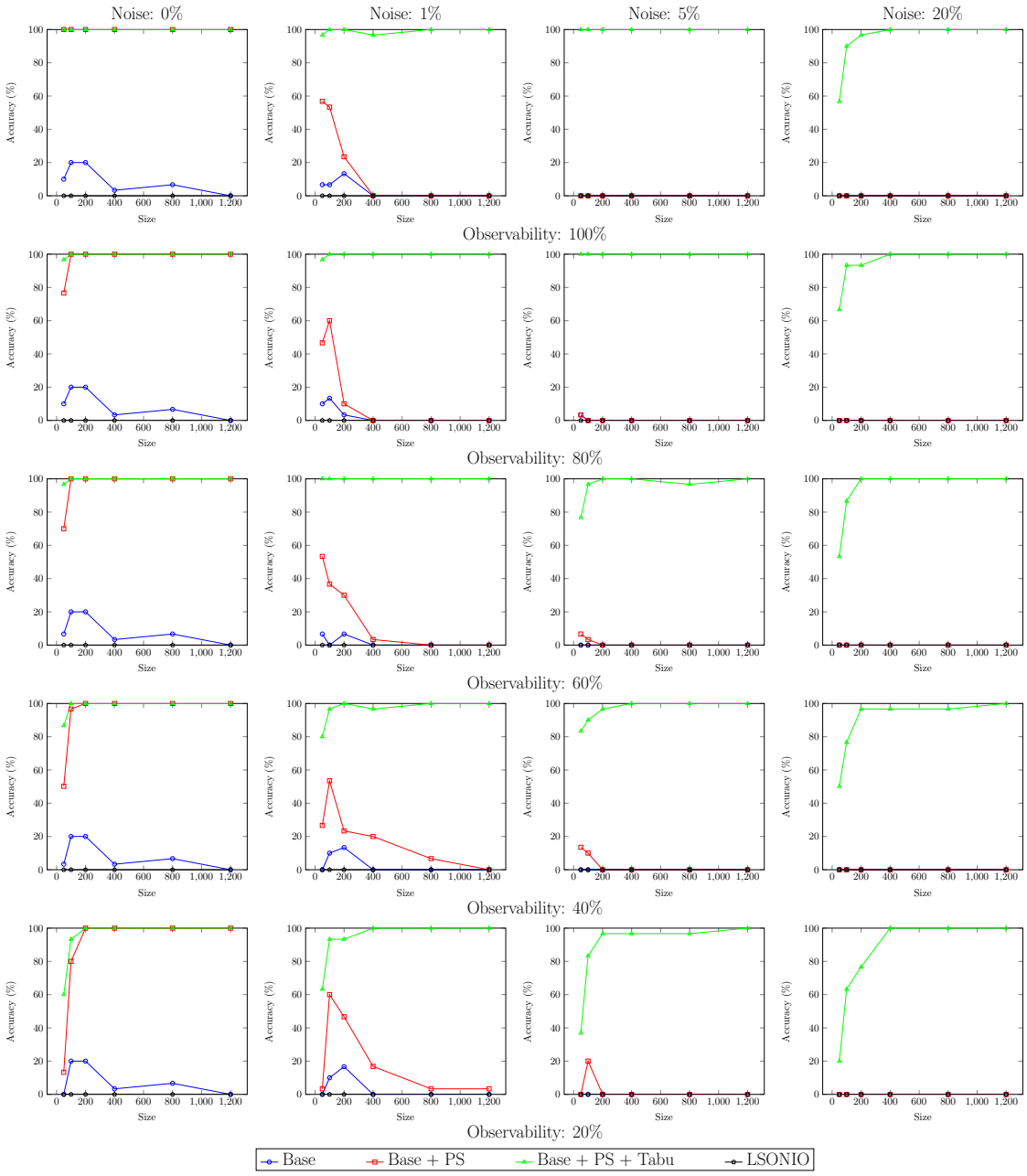


Figure B.11: Hanoi – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

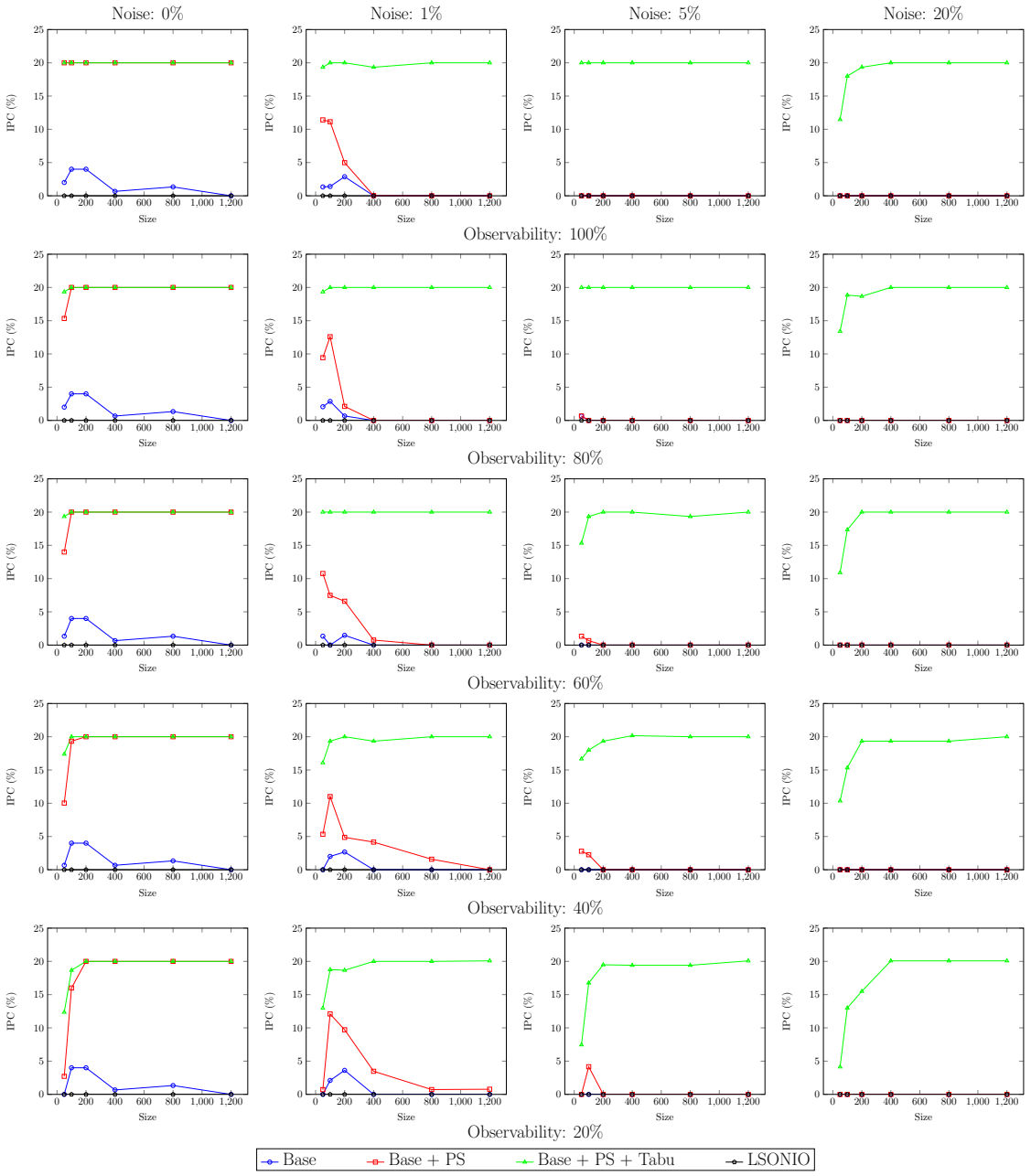


Figure B.12: Hanoi – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

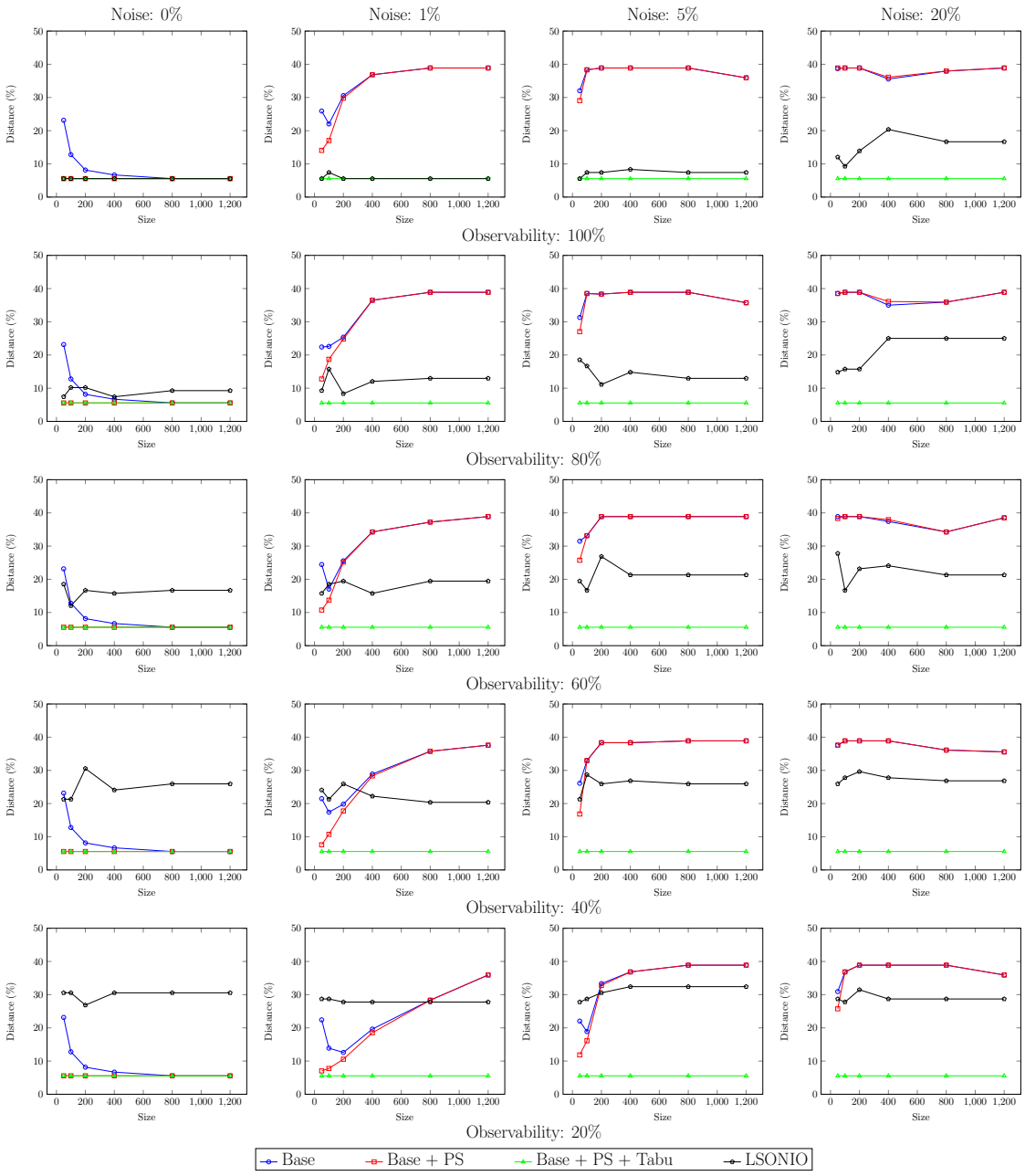


Figure B.13: N Puzzle – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

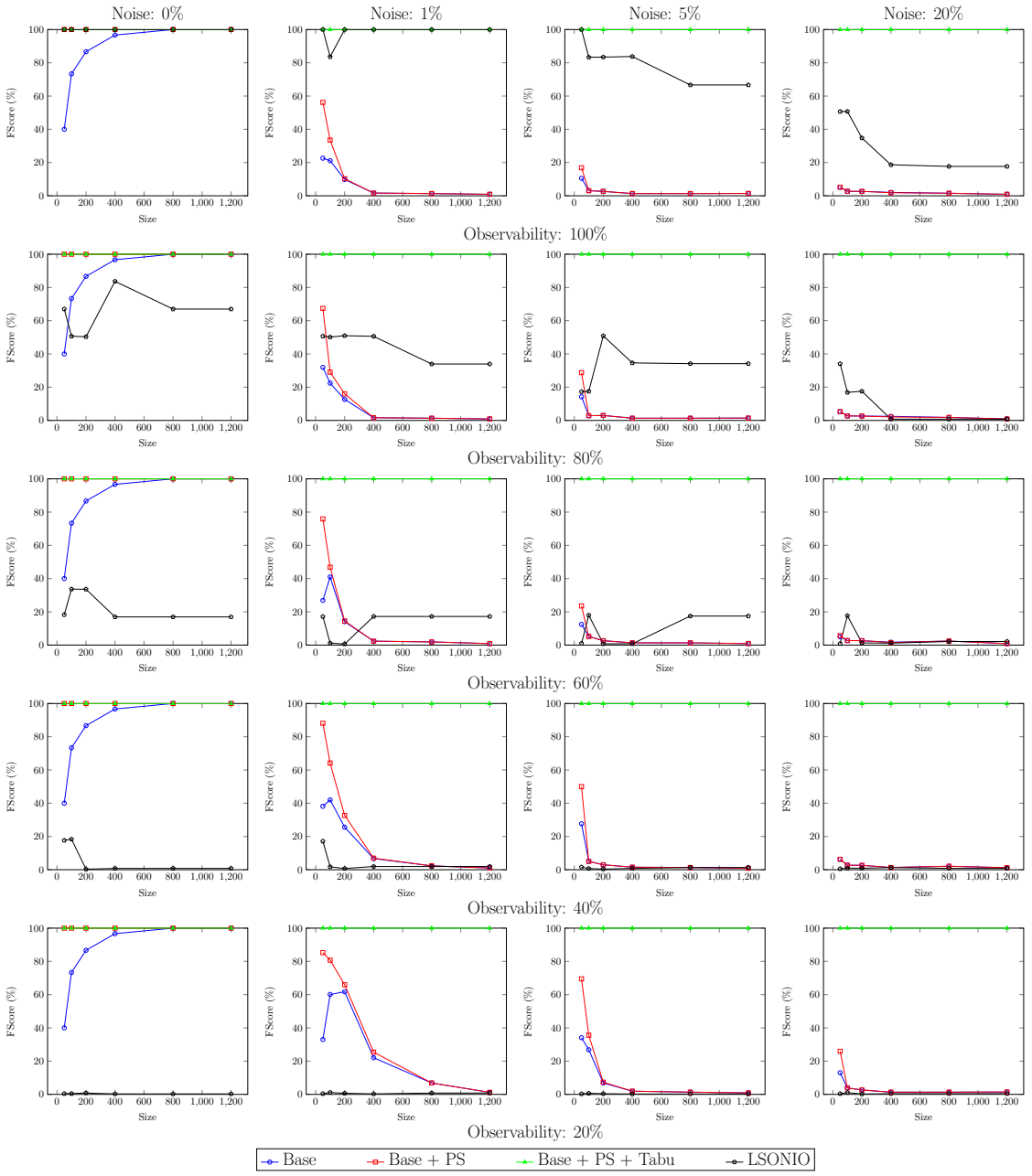


Figure B.14: N Puzzle – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

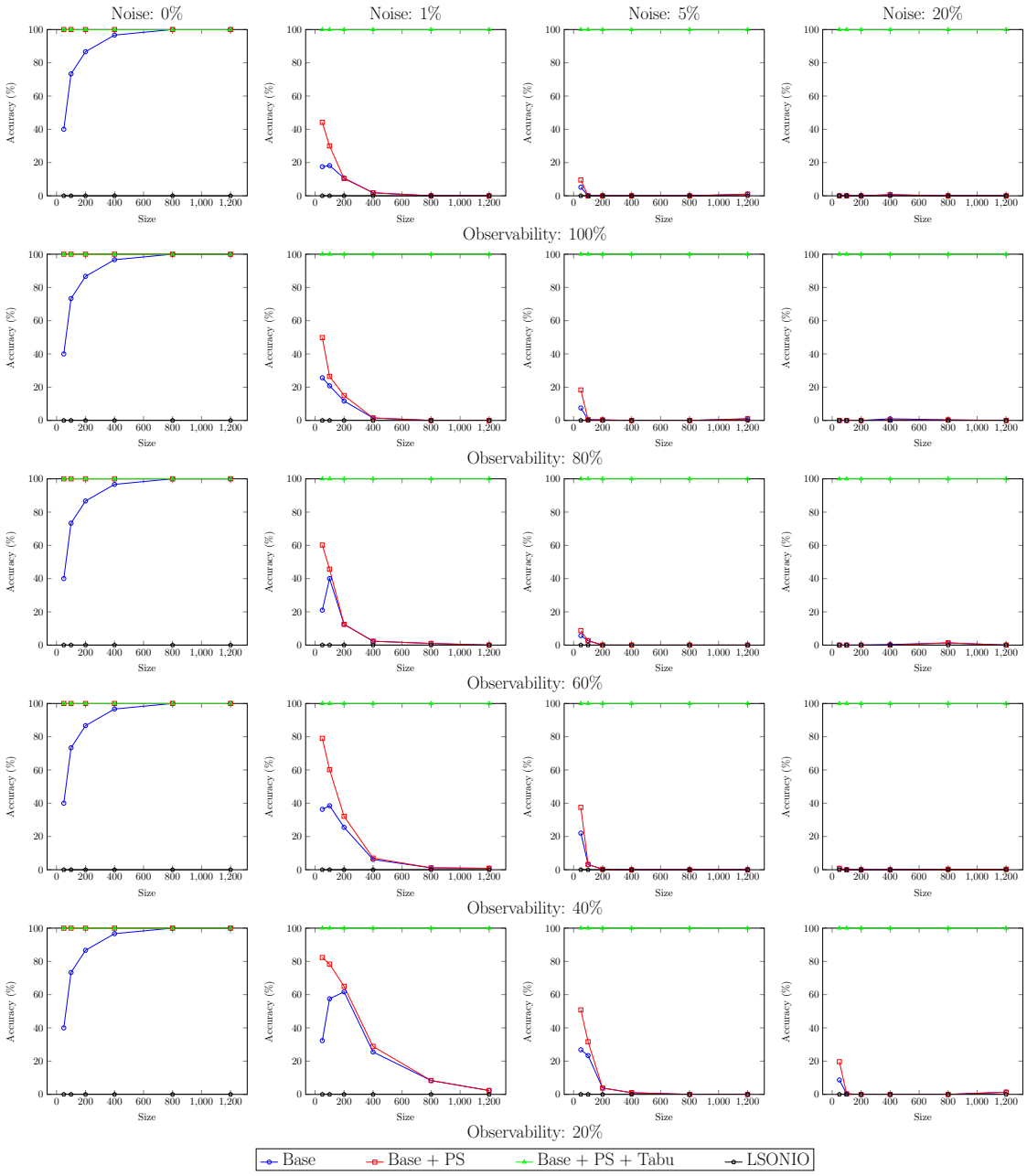


Figure B.15: N Puzzle – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

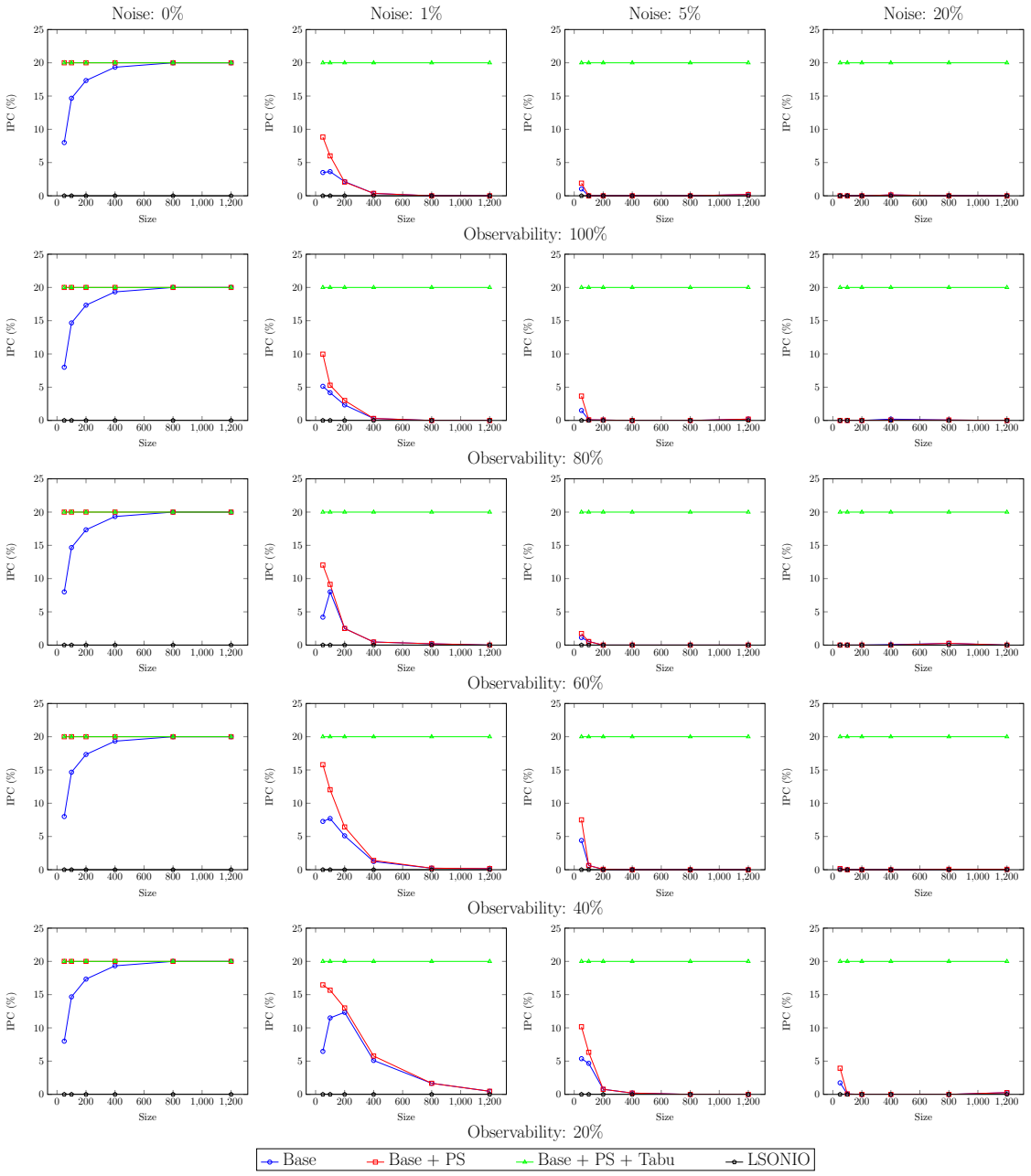


Figure B.16: N Puzzle – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

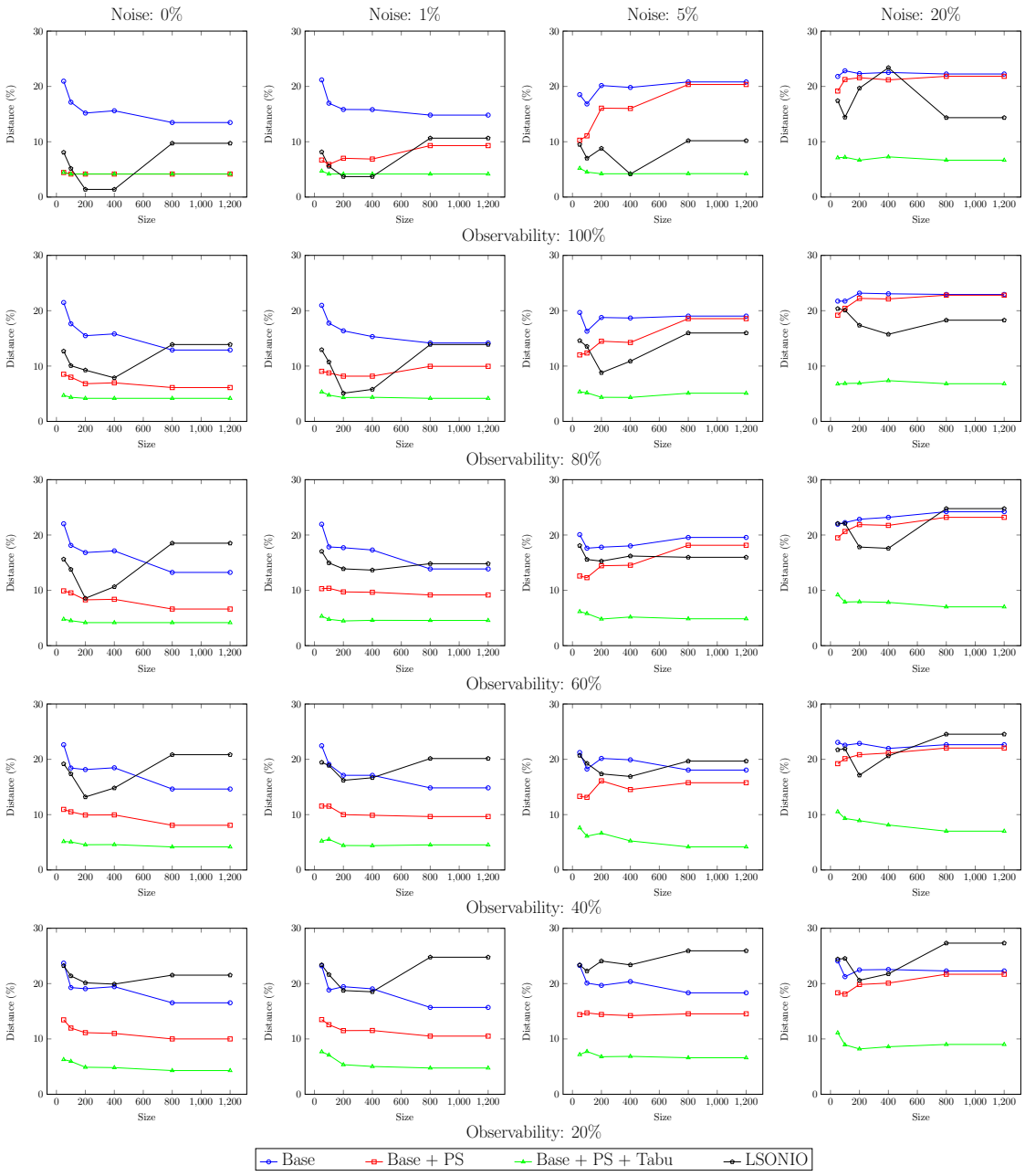


Figure B.17: Peg Solitaire – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

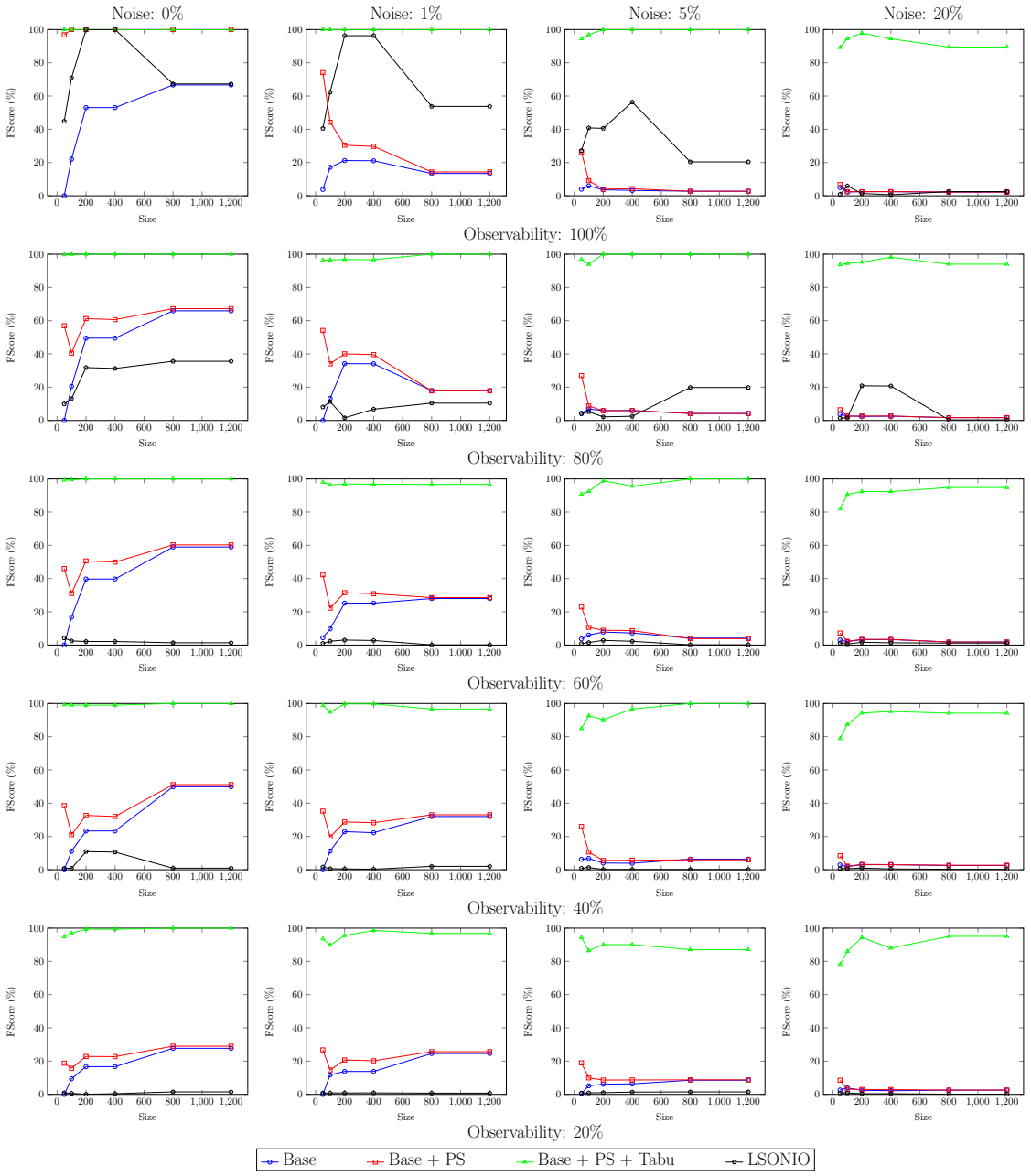


Figure B.18: Peg Solitaire – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

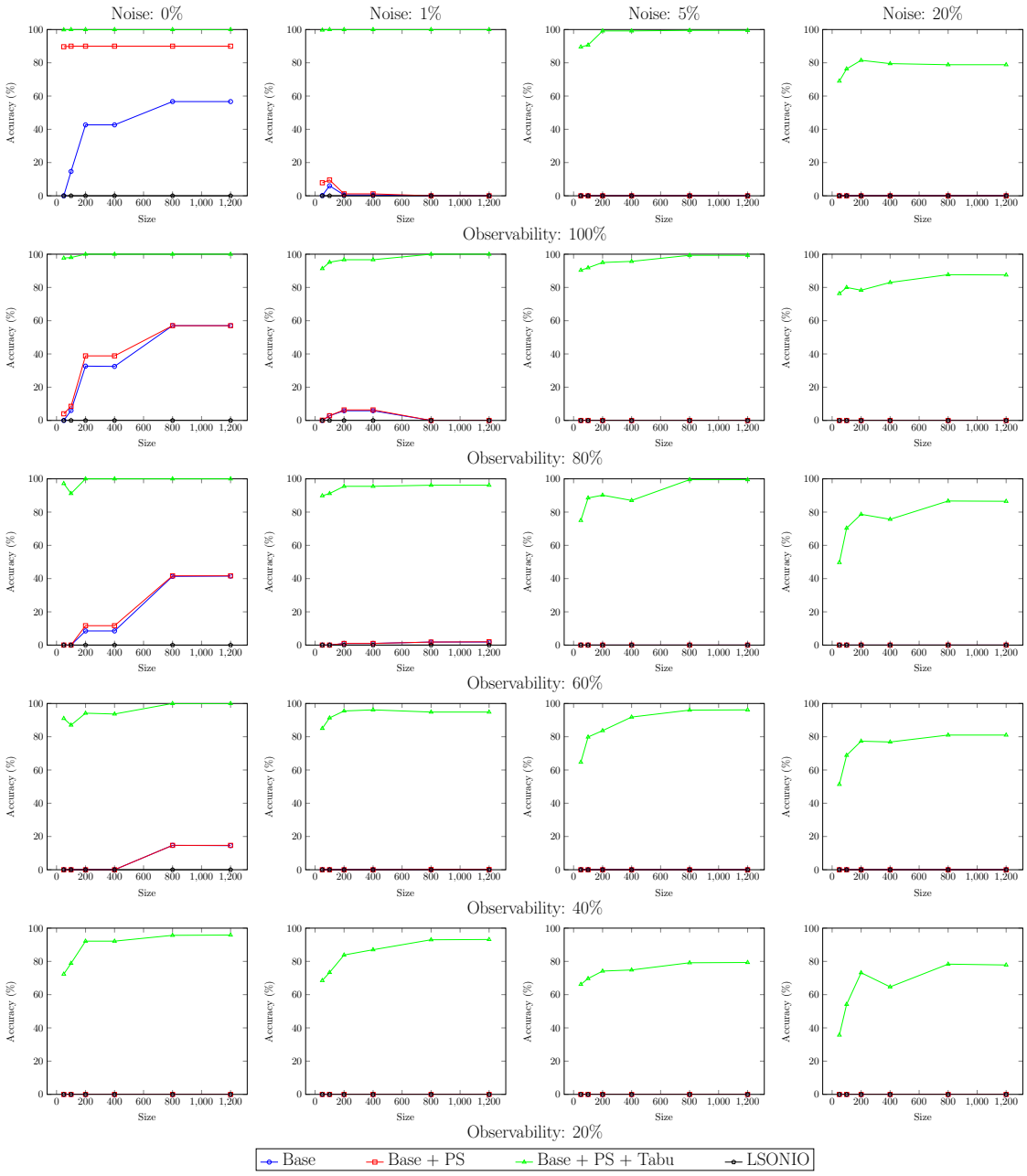


Figure B.19: Peg Solitaire – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

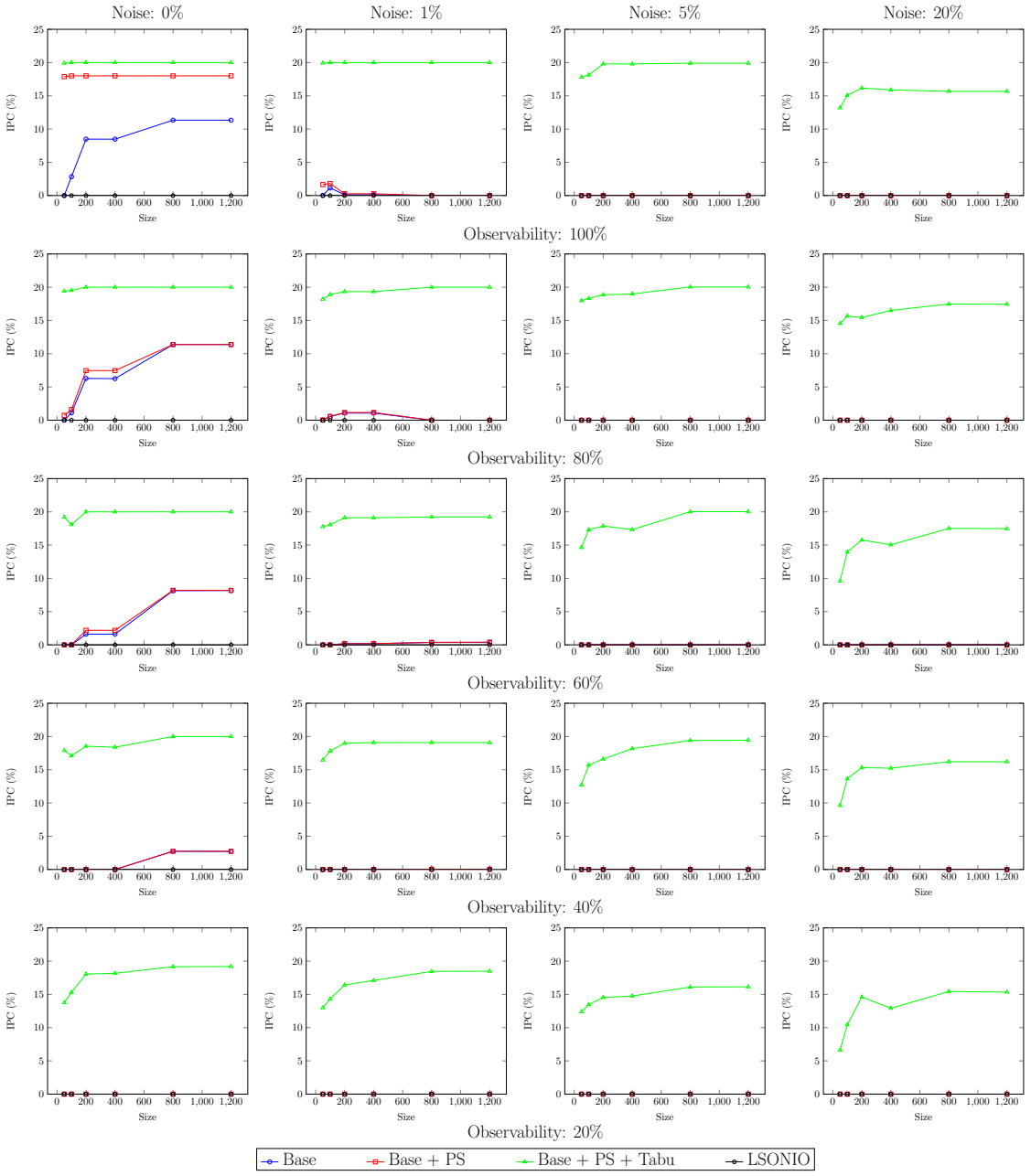


Figure B.20: Peg Solitaire – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

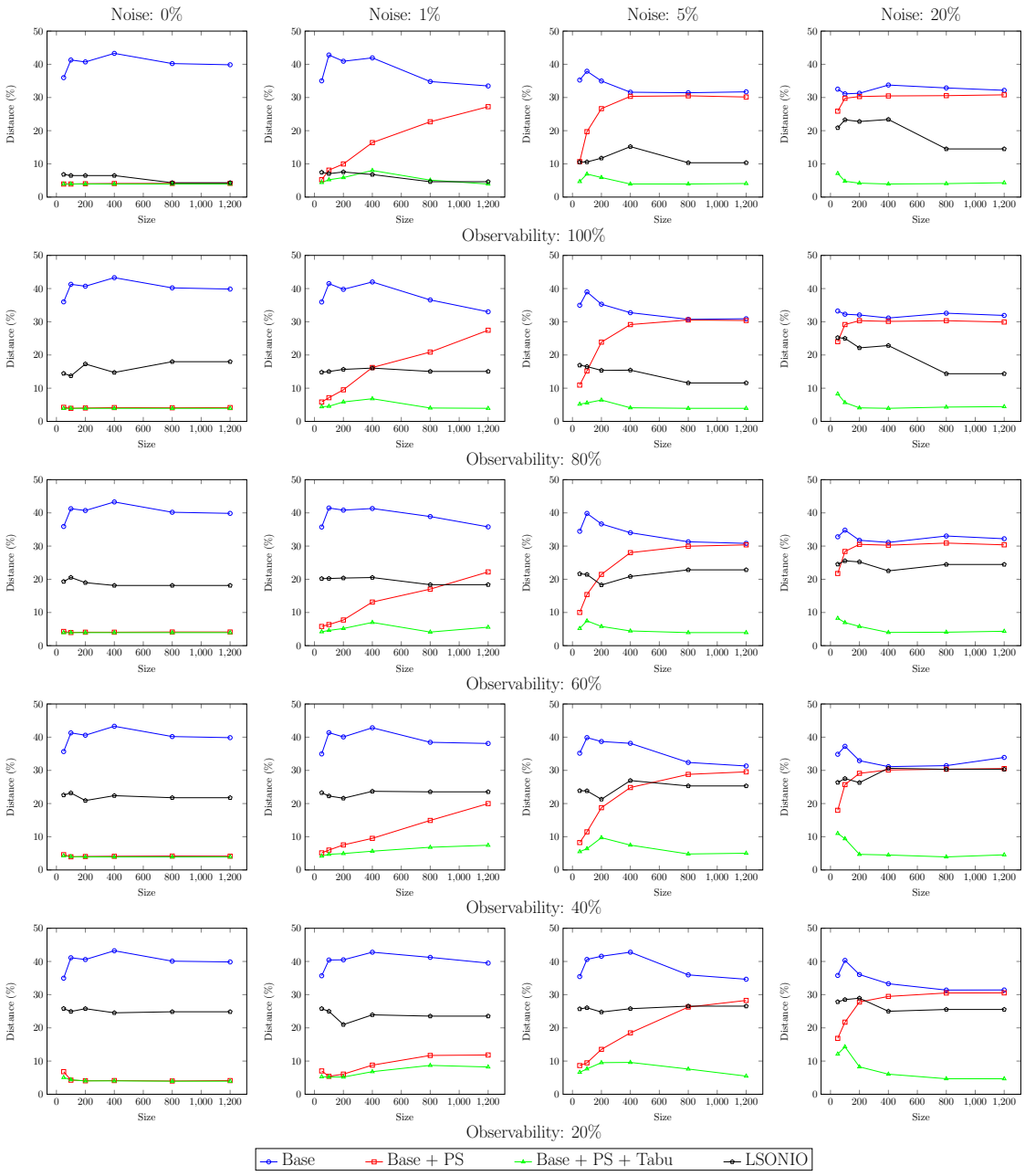


Figure B.21: Parking – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

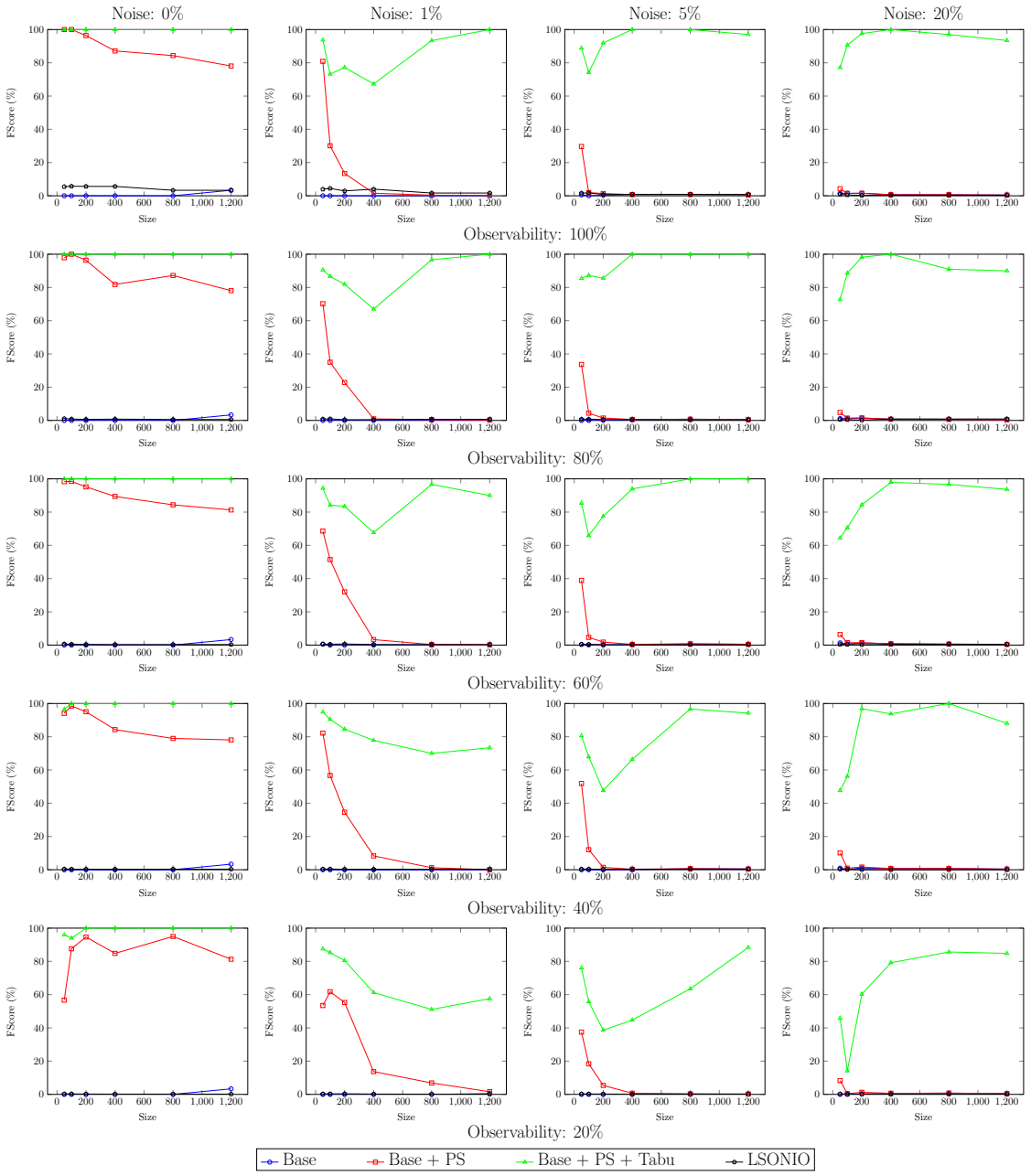


Figure B.22: Parking – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

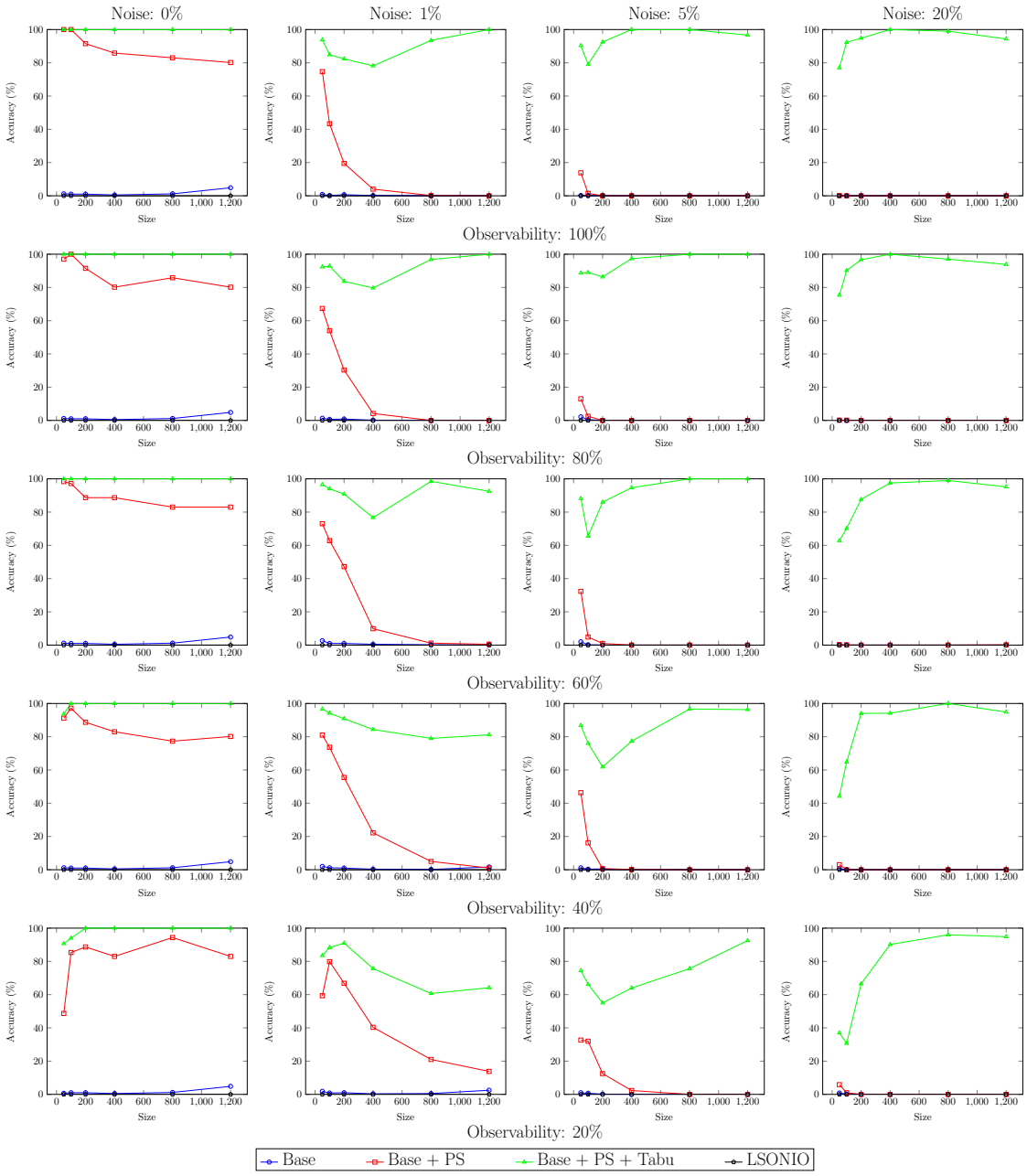


Figure B.23: Parking – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

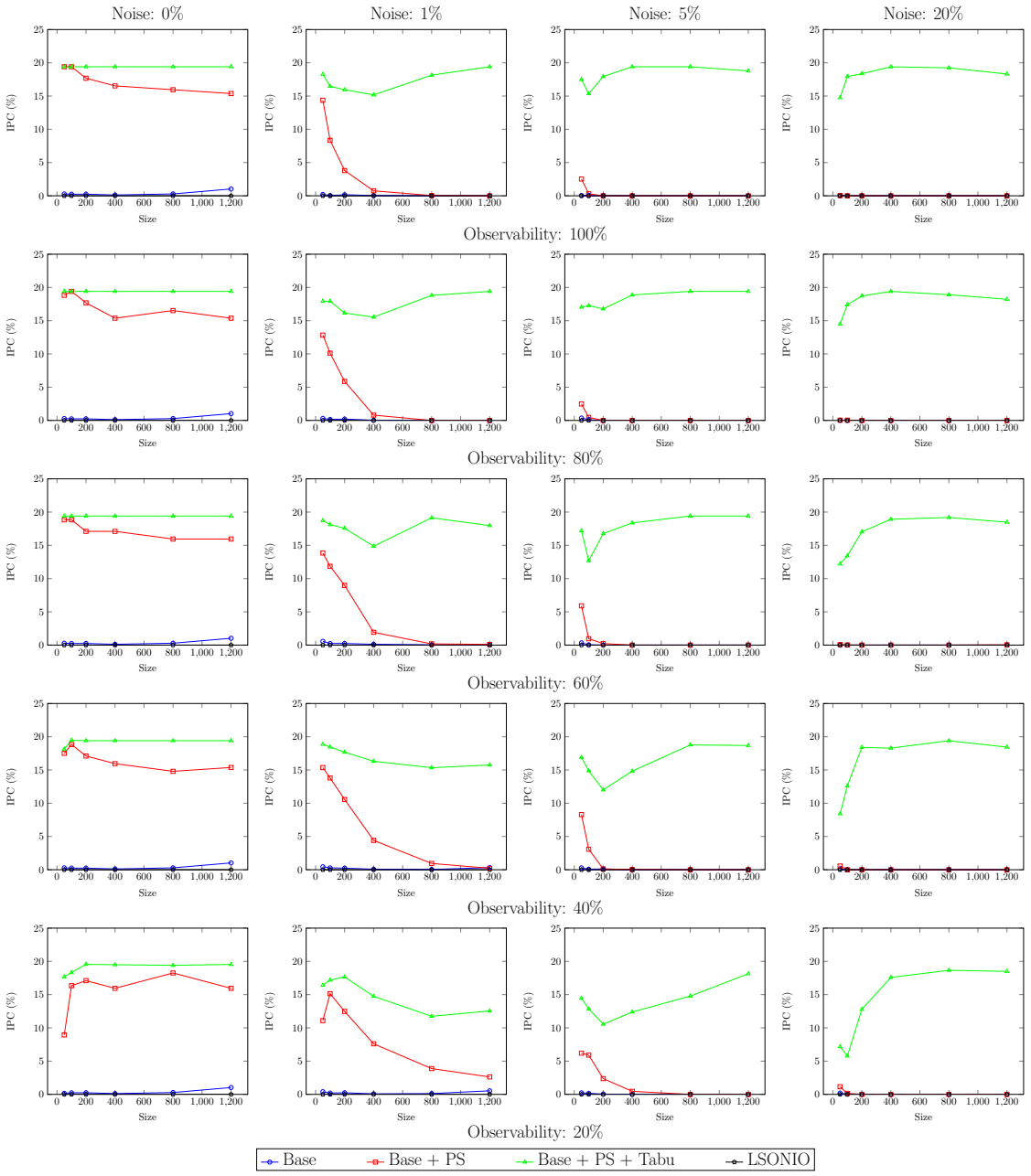


Figure B.24: Parking – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

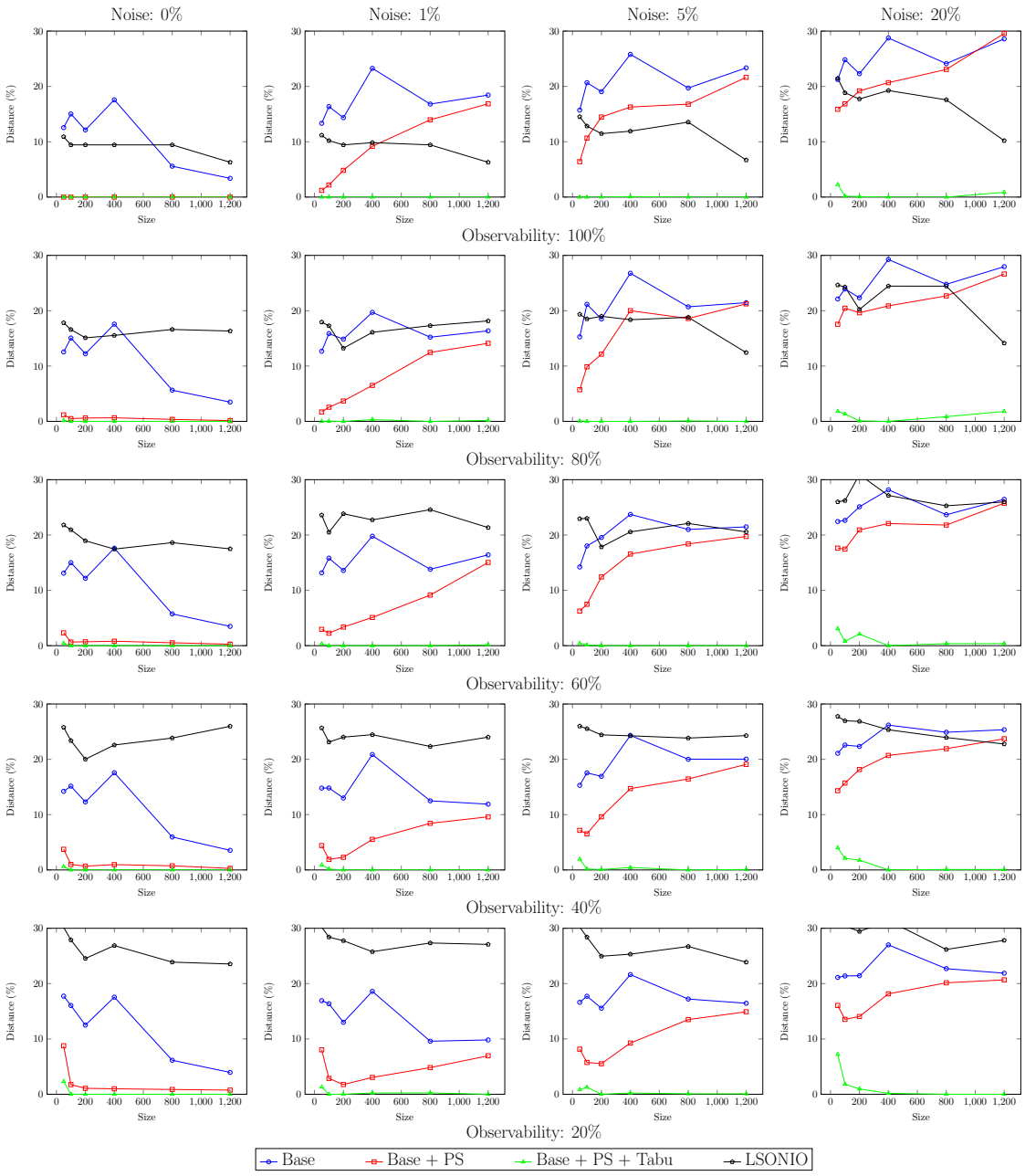


Figure B.25: Zenotravel – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

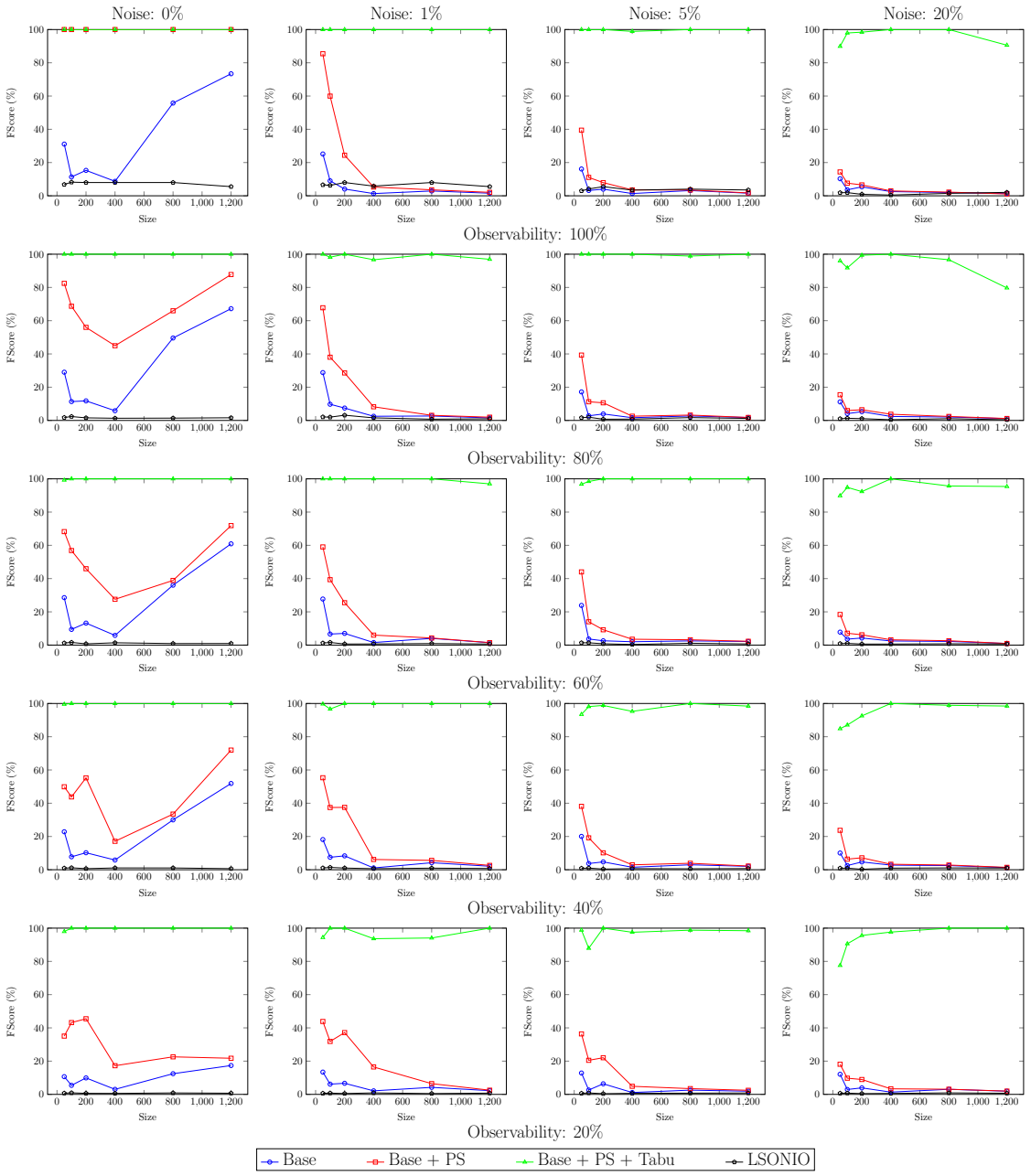


Figure B.26: Zenotravel – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

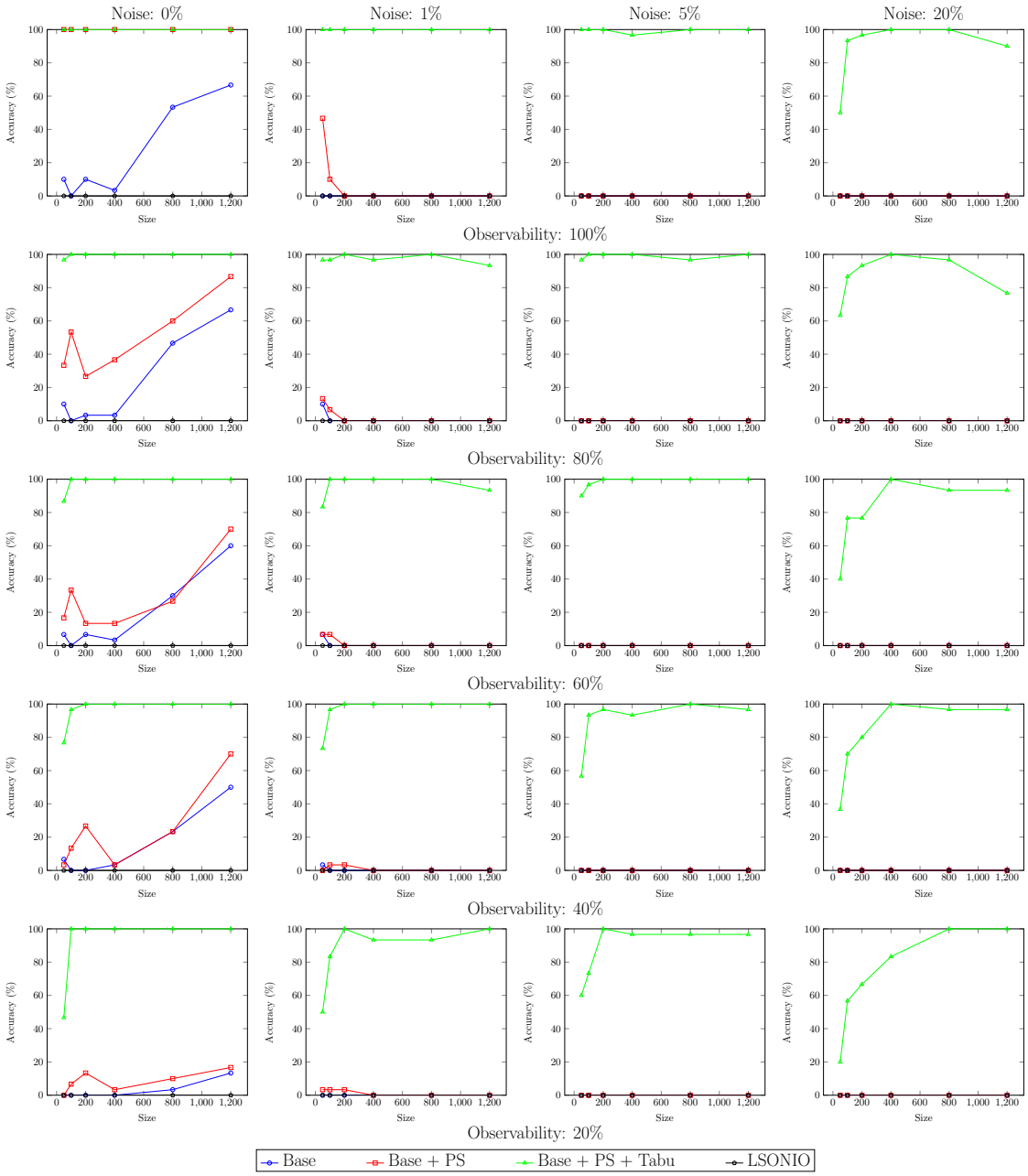


Figure B.27: Zenotravel – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

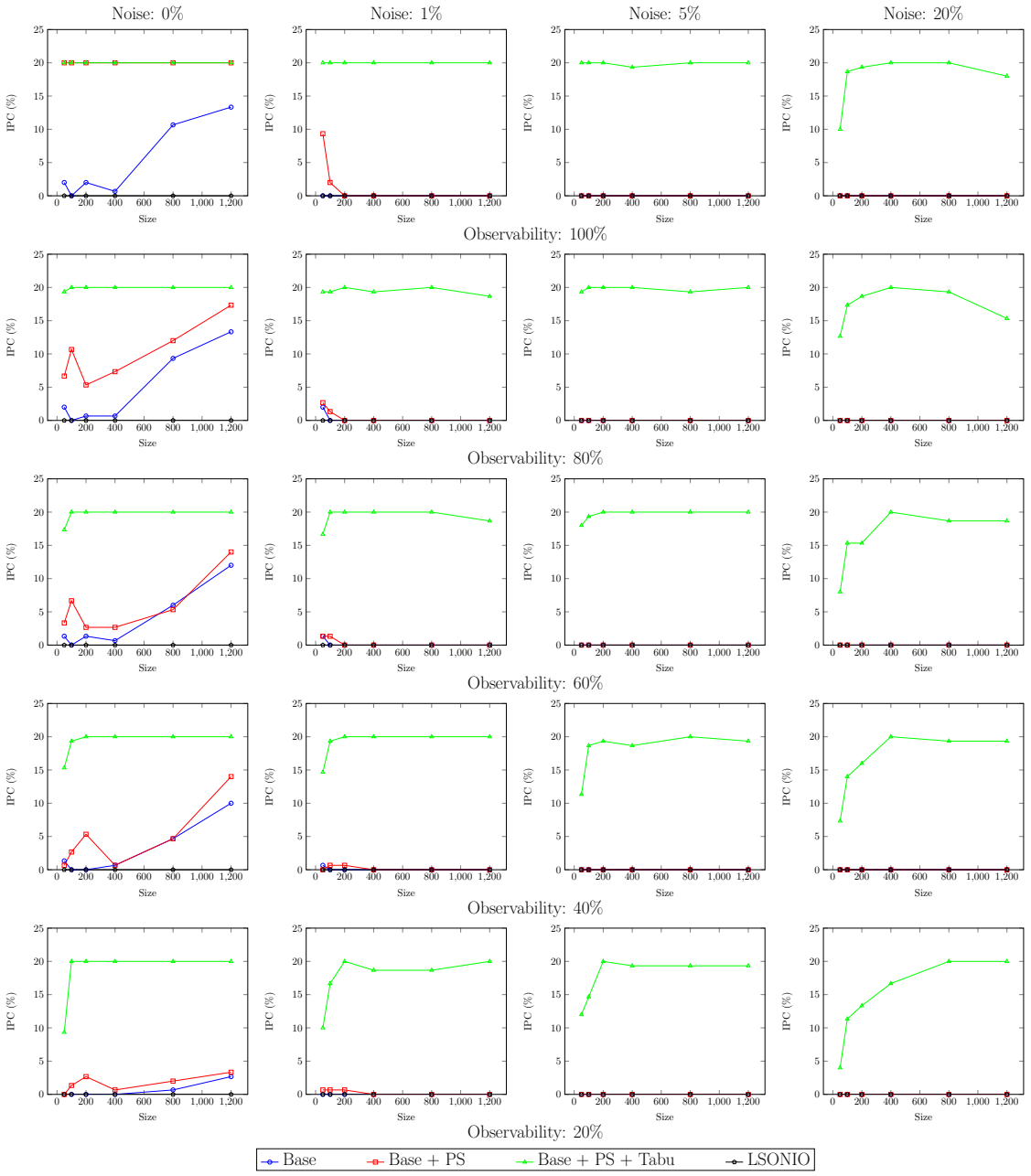


Figure B.28: Zenotravel – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

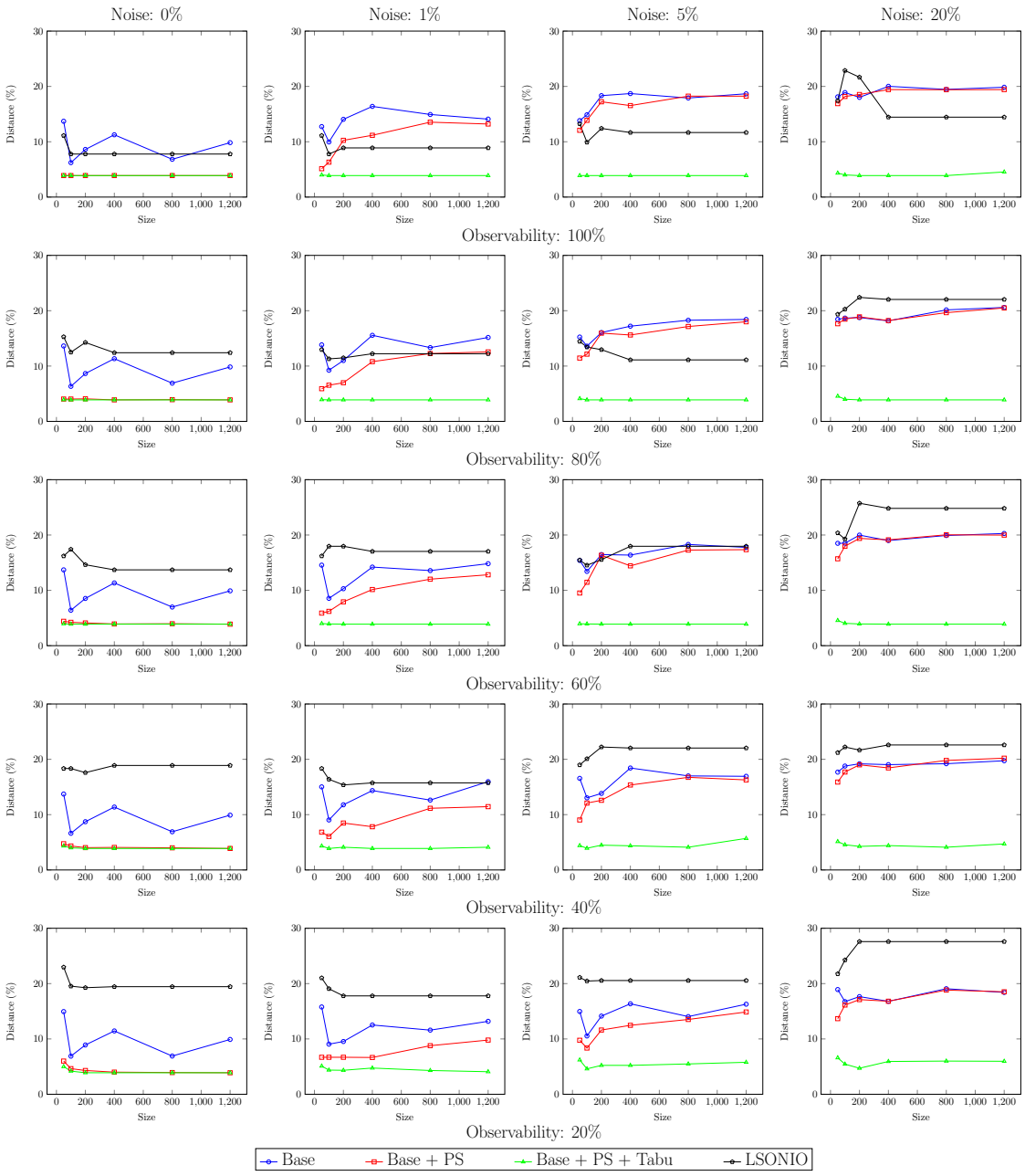


Figure B.29: Sokoban – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

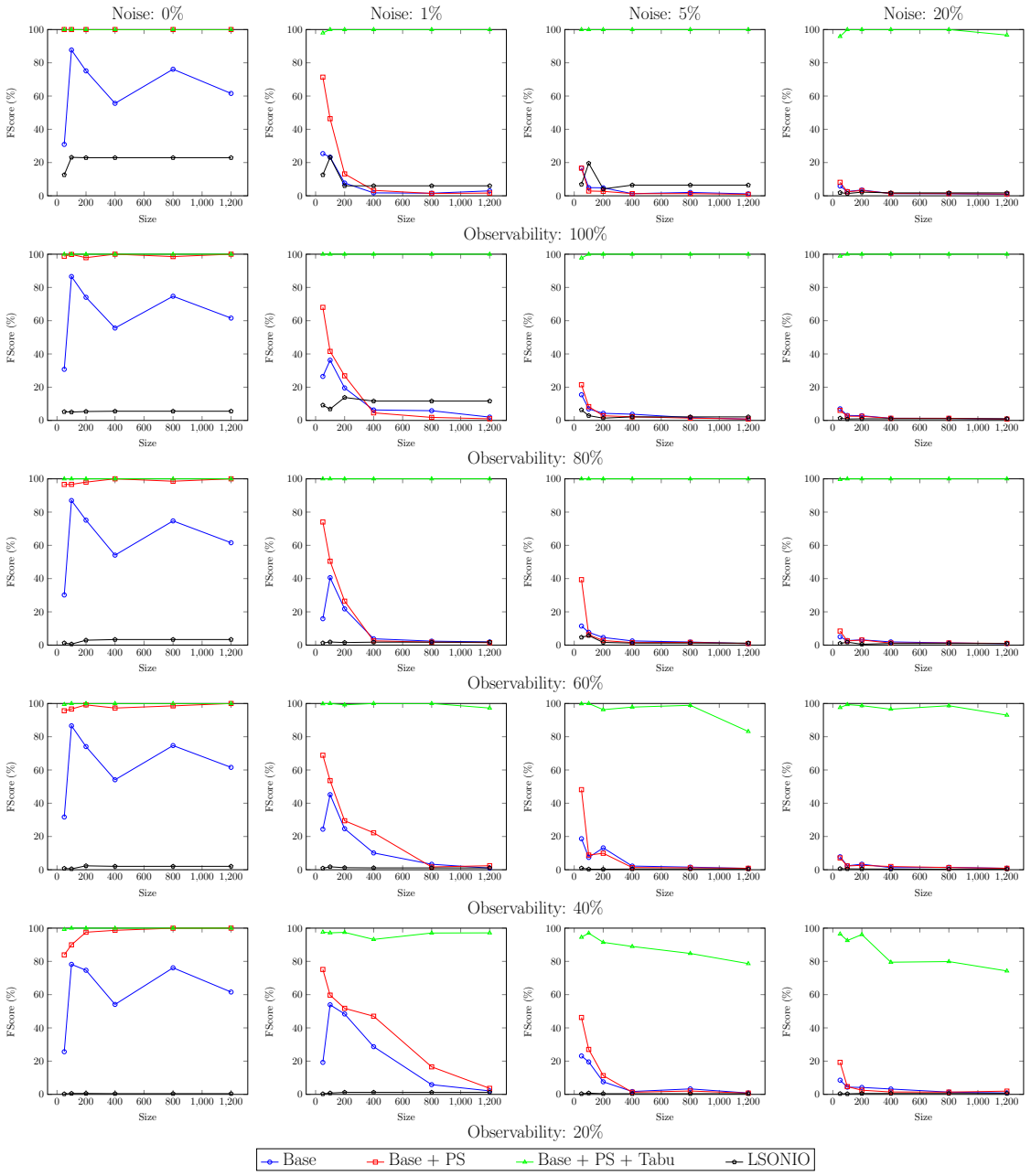


Figure B.30: Sokoban – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

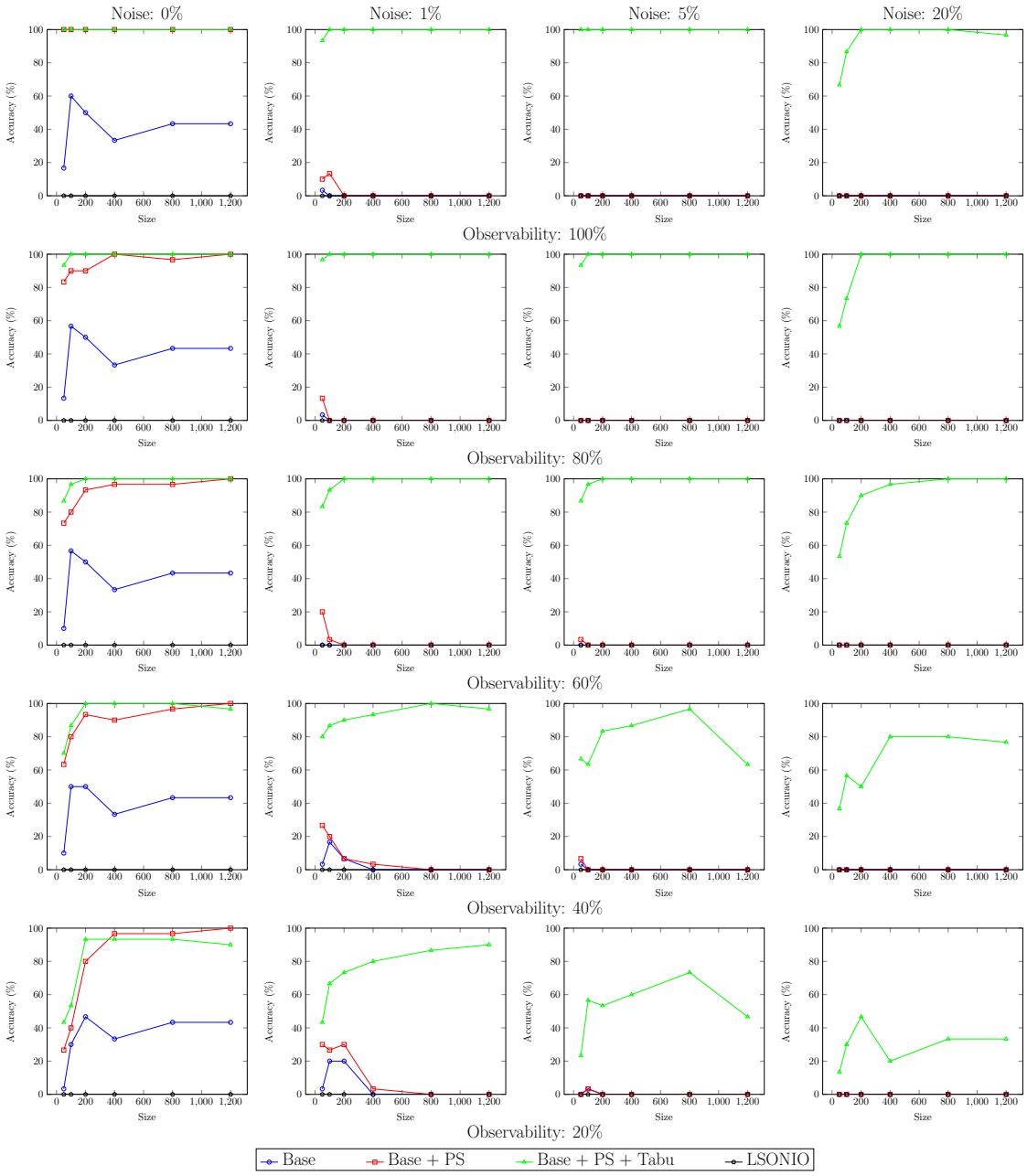


Figure B.31: Sokoban – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

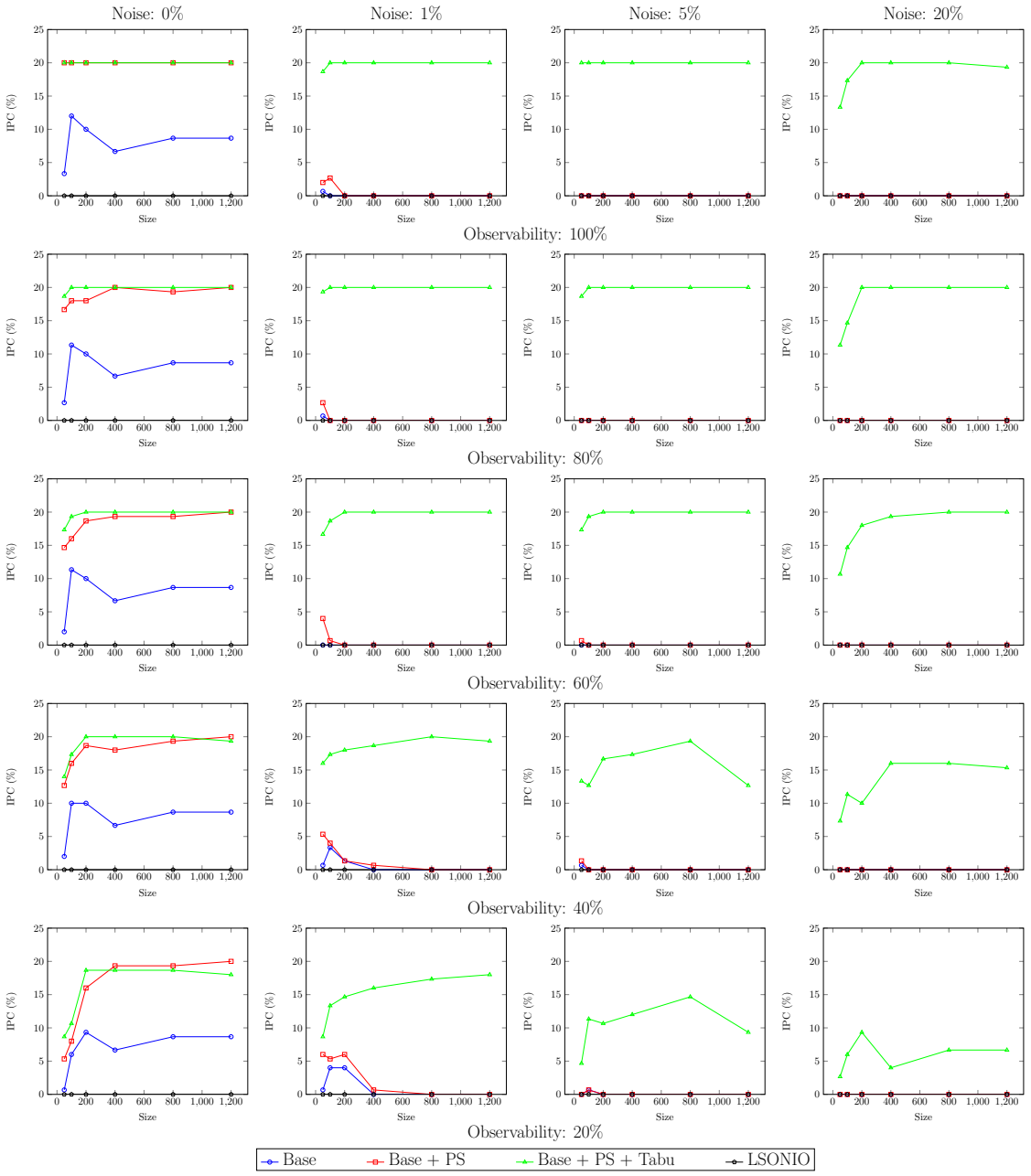


Figure B.32: Sokoban – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

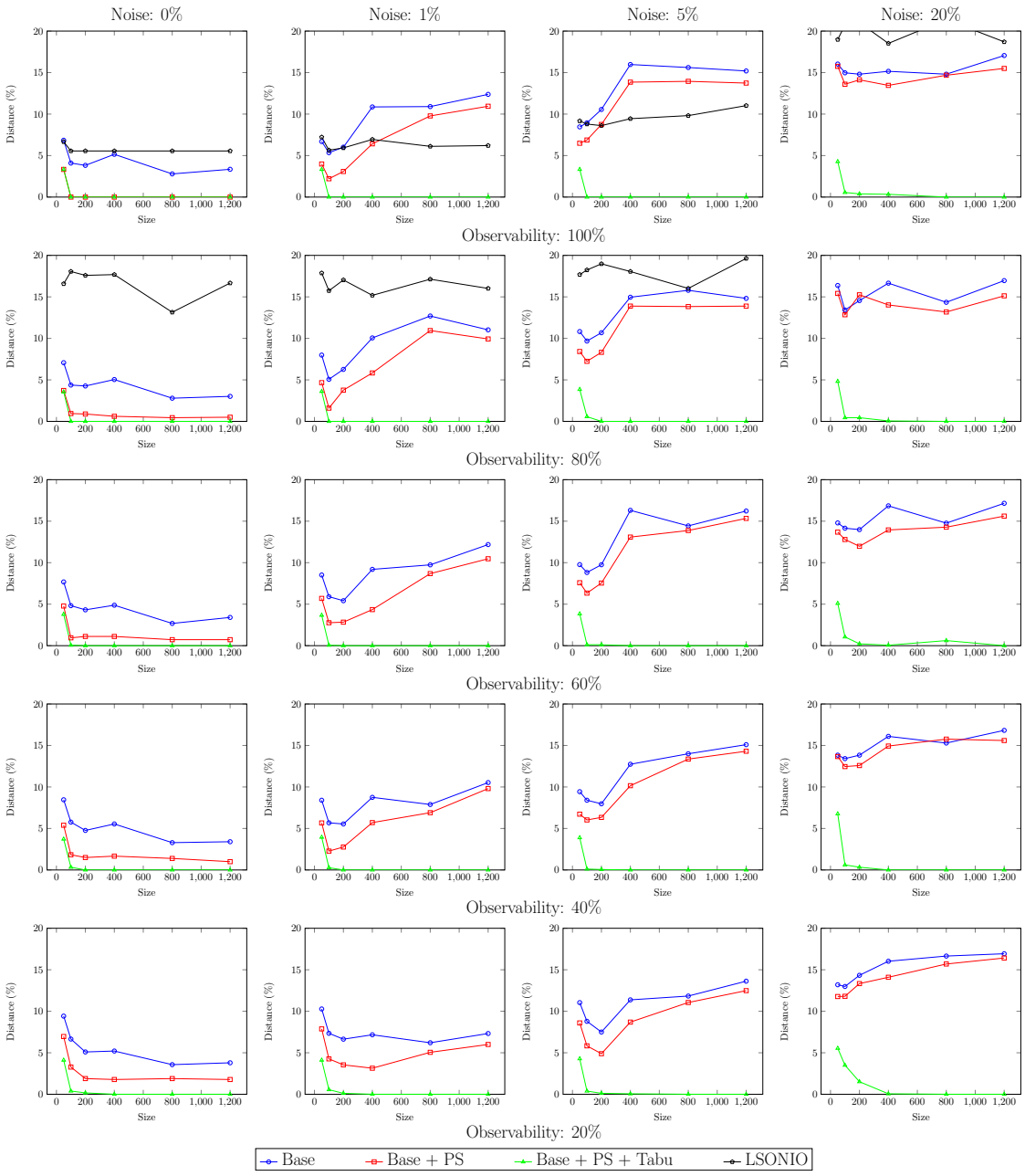


Figure B.33: Elevator – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

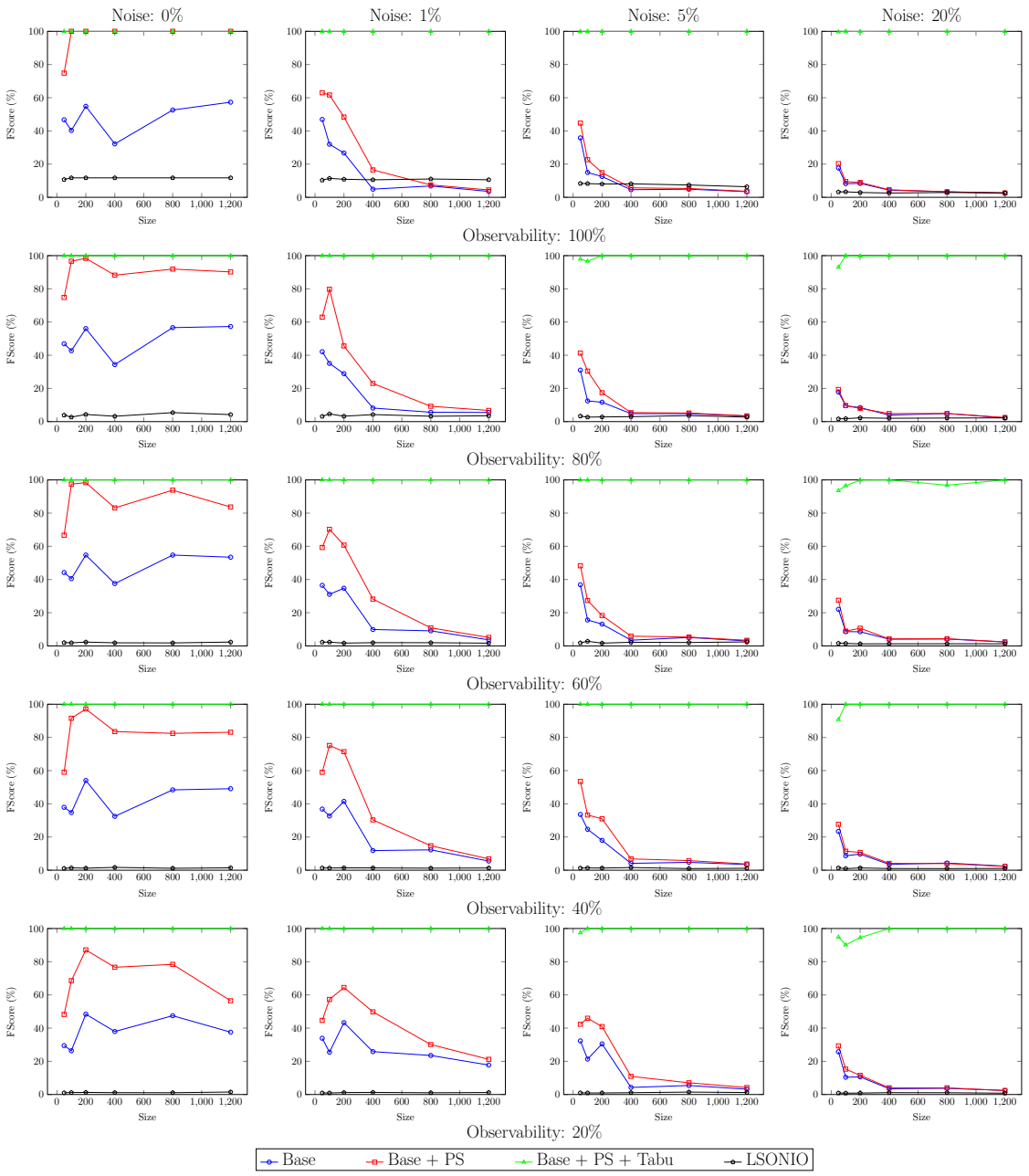


Figure B.34: Elevator – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

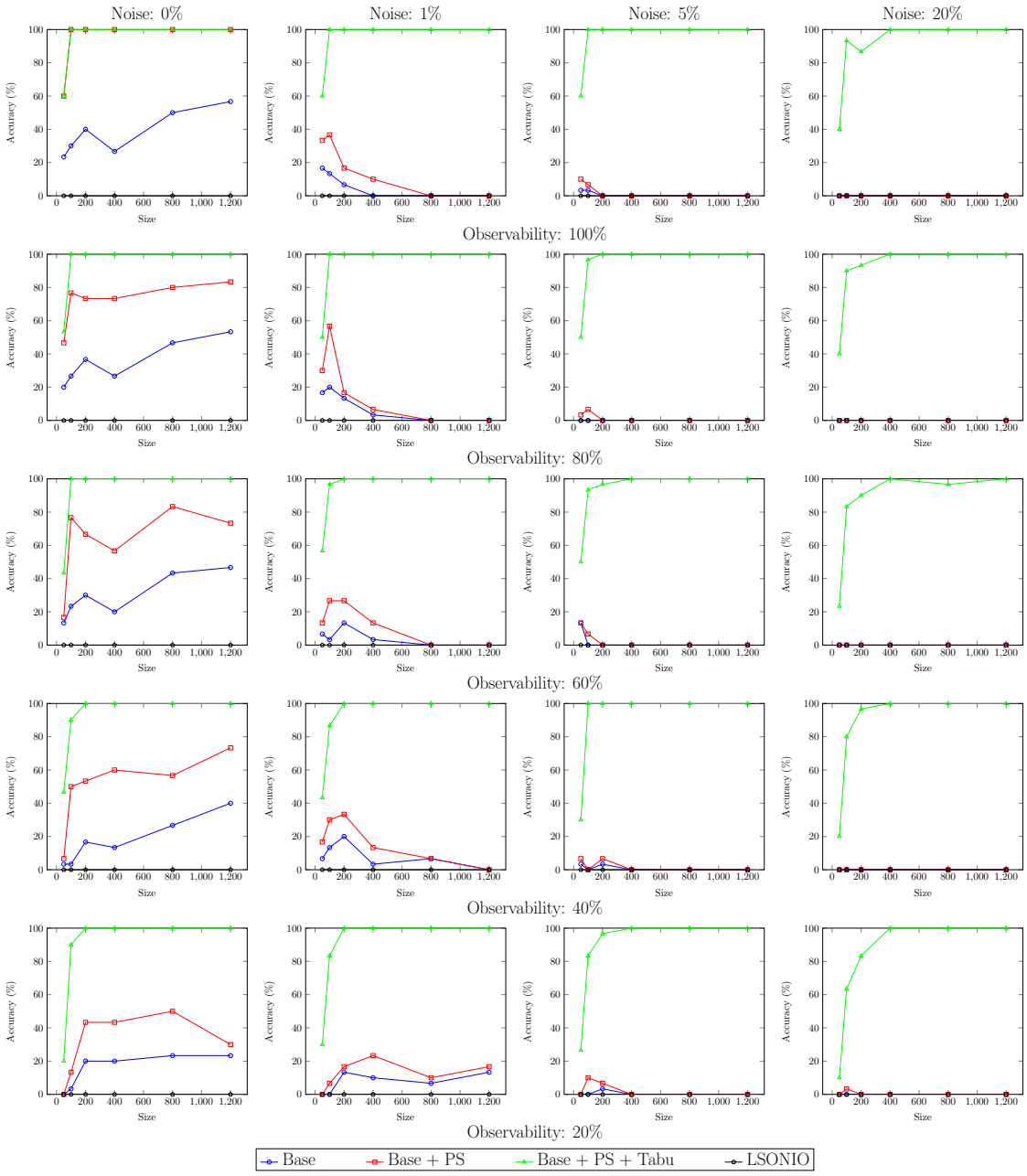


Figure B.35: Elevator – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

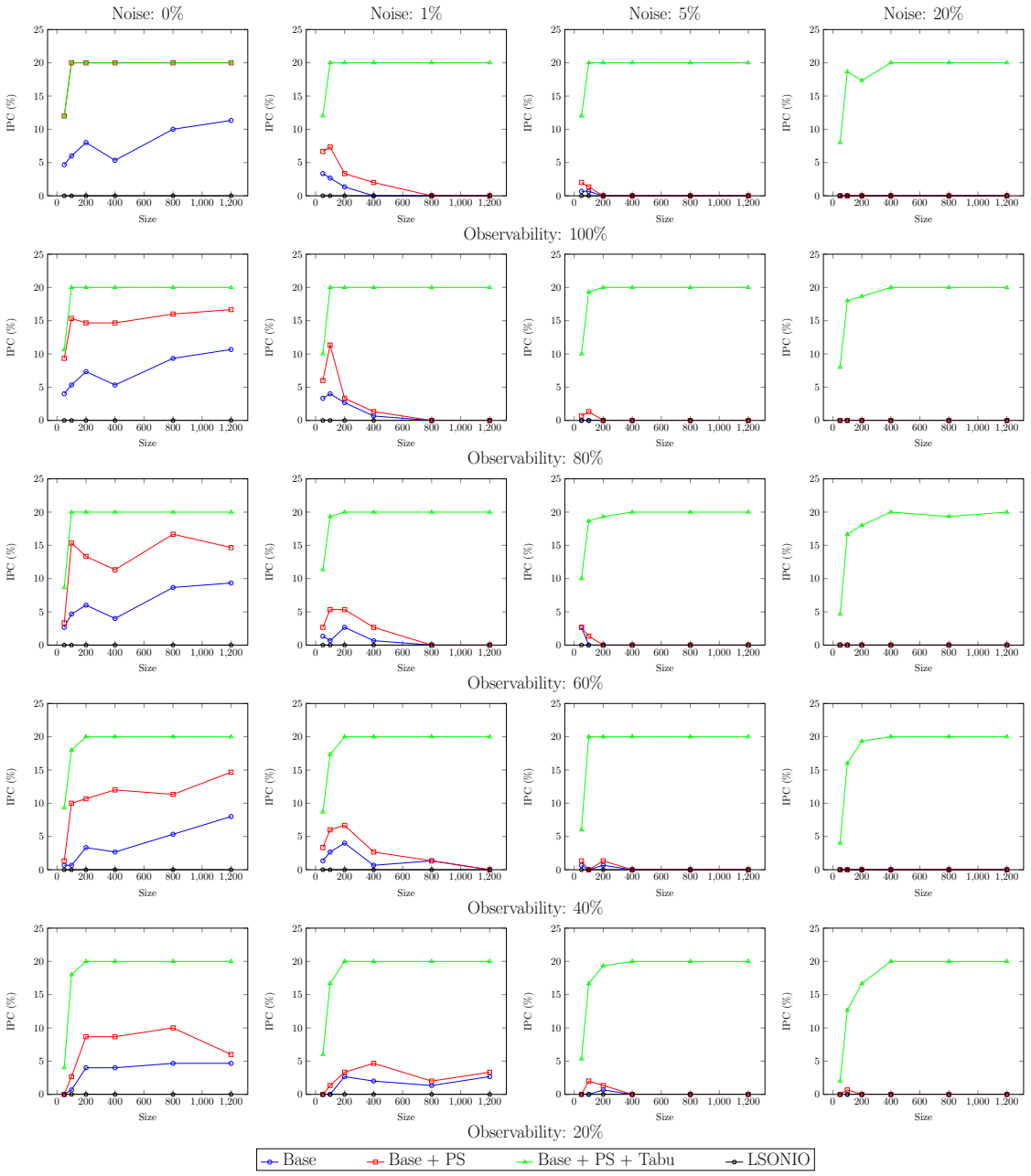


Figure B.36: Elevator – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

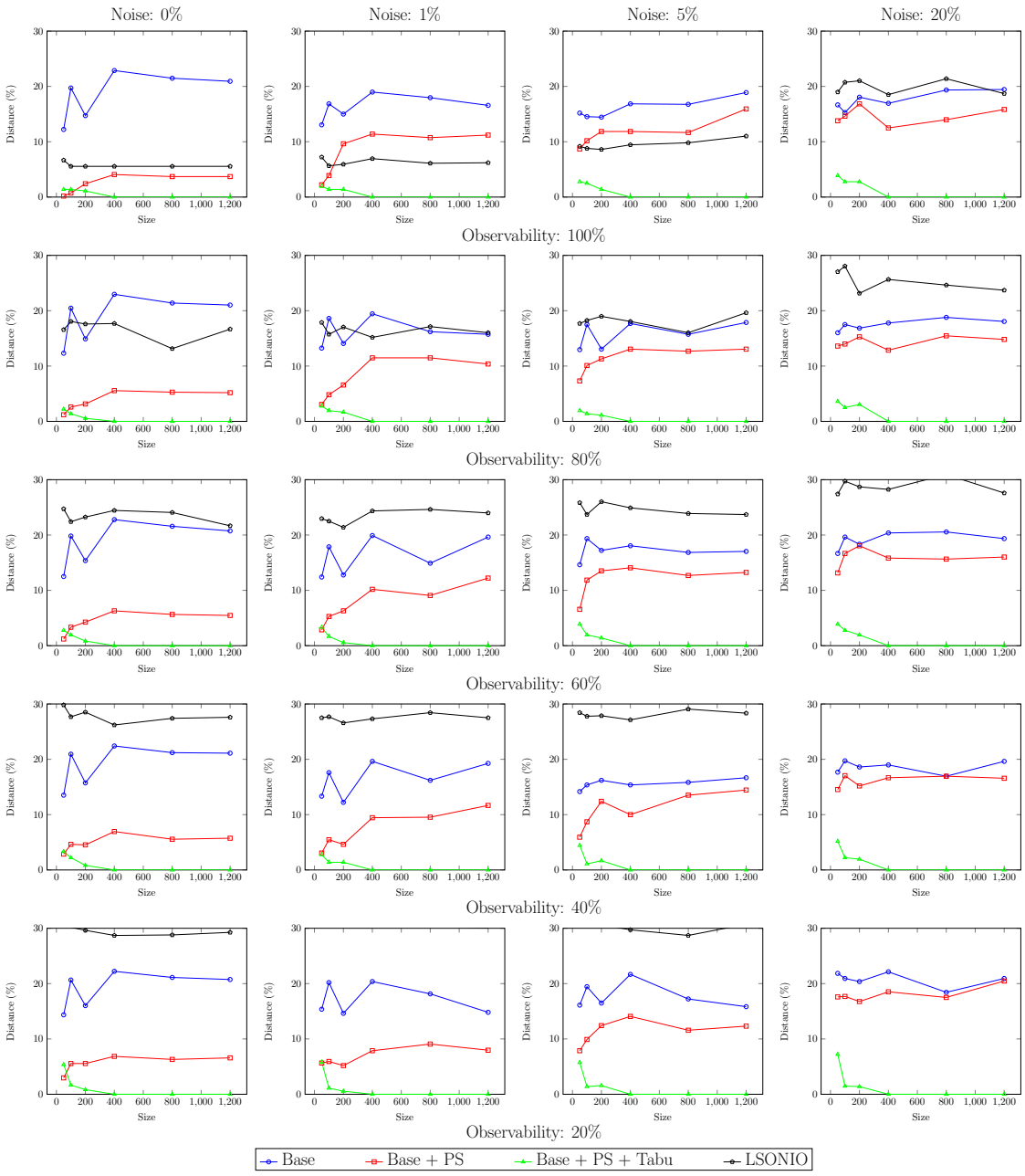


Figure B.37: Visit All – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

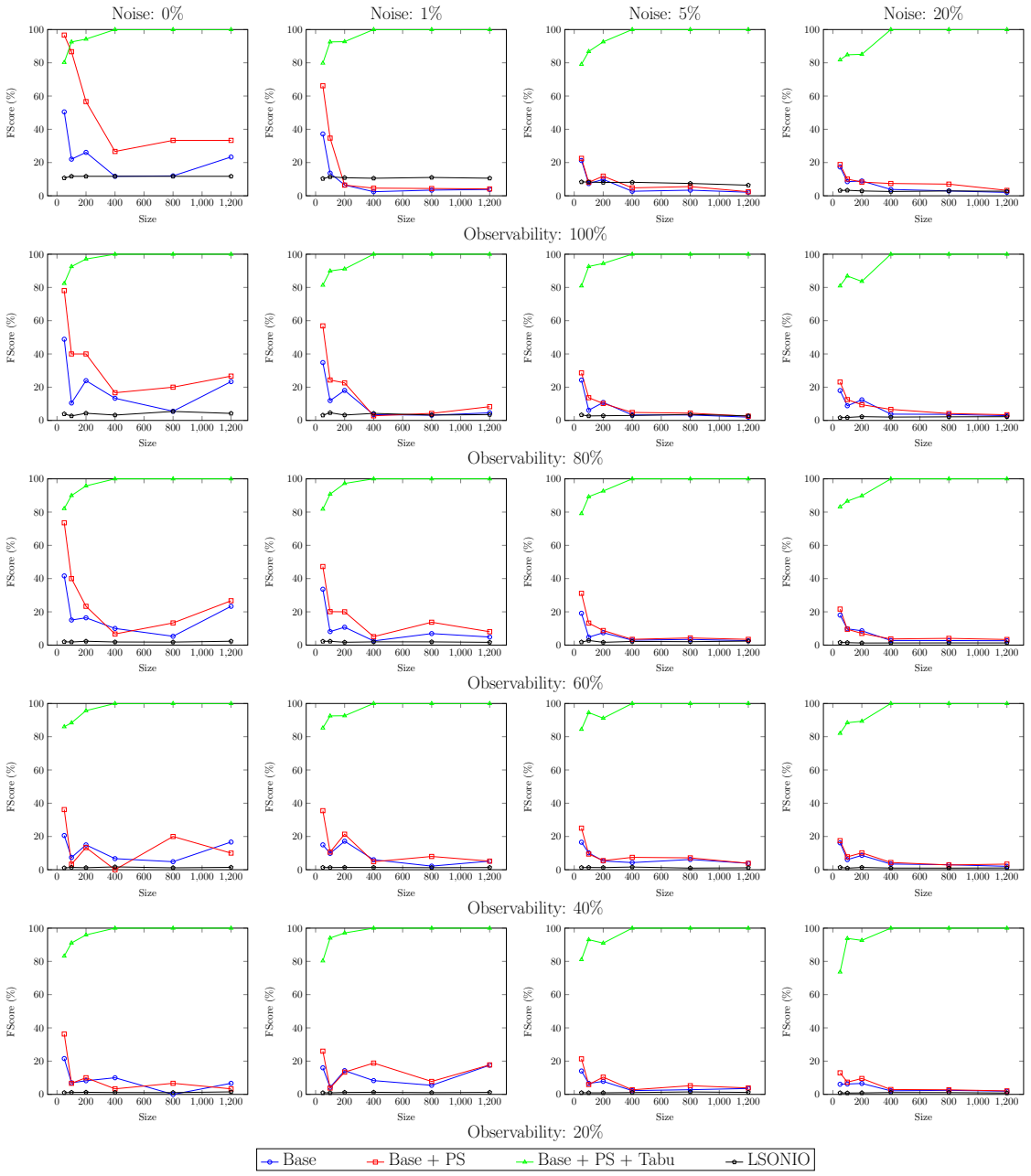


Figure B.38: Visit All – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

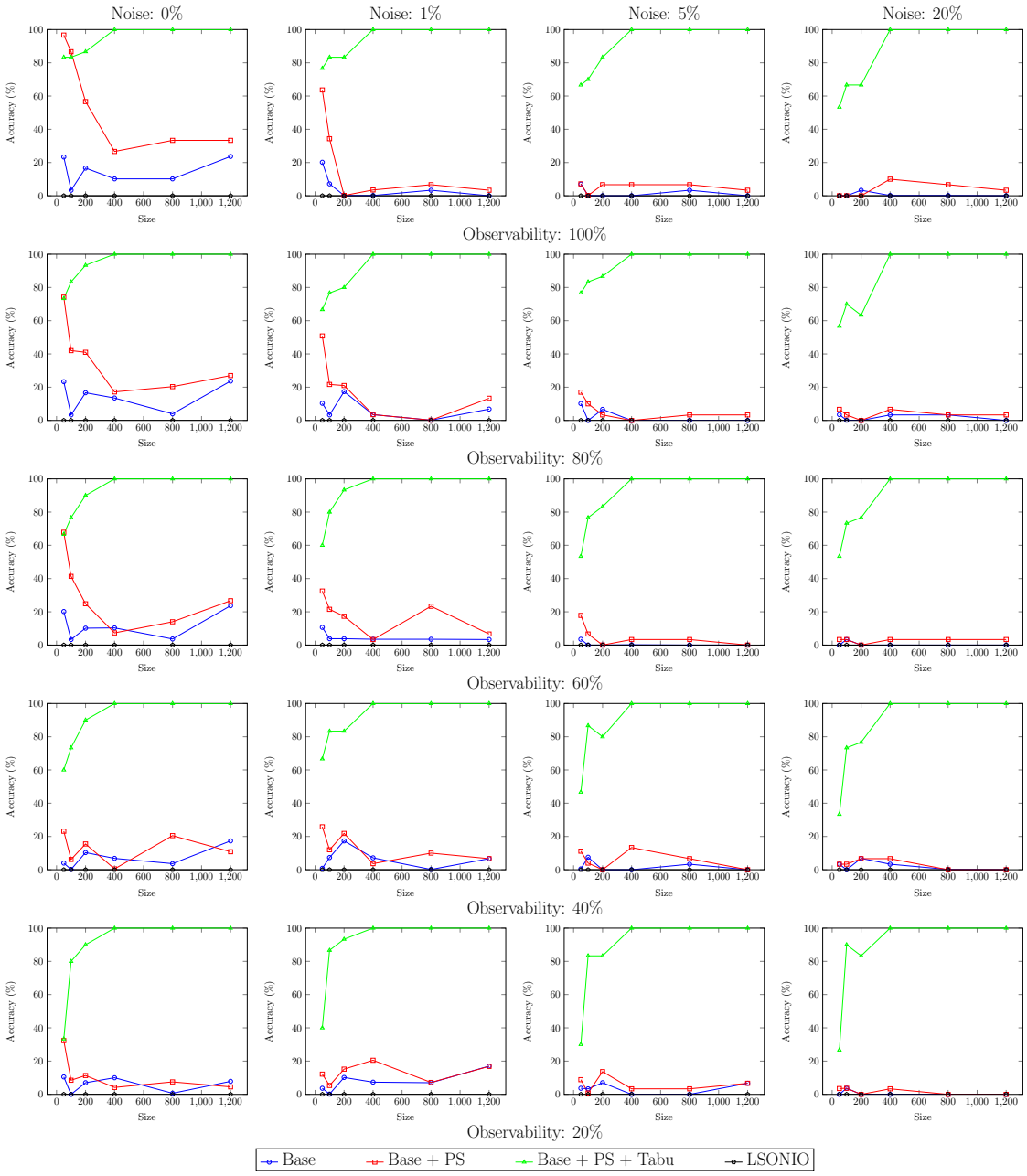


Figure B.39: Visit All – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

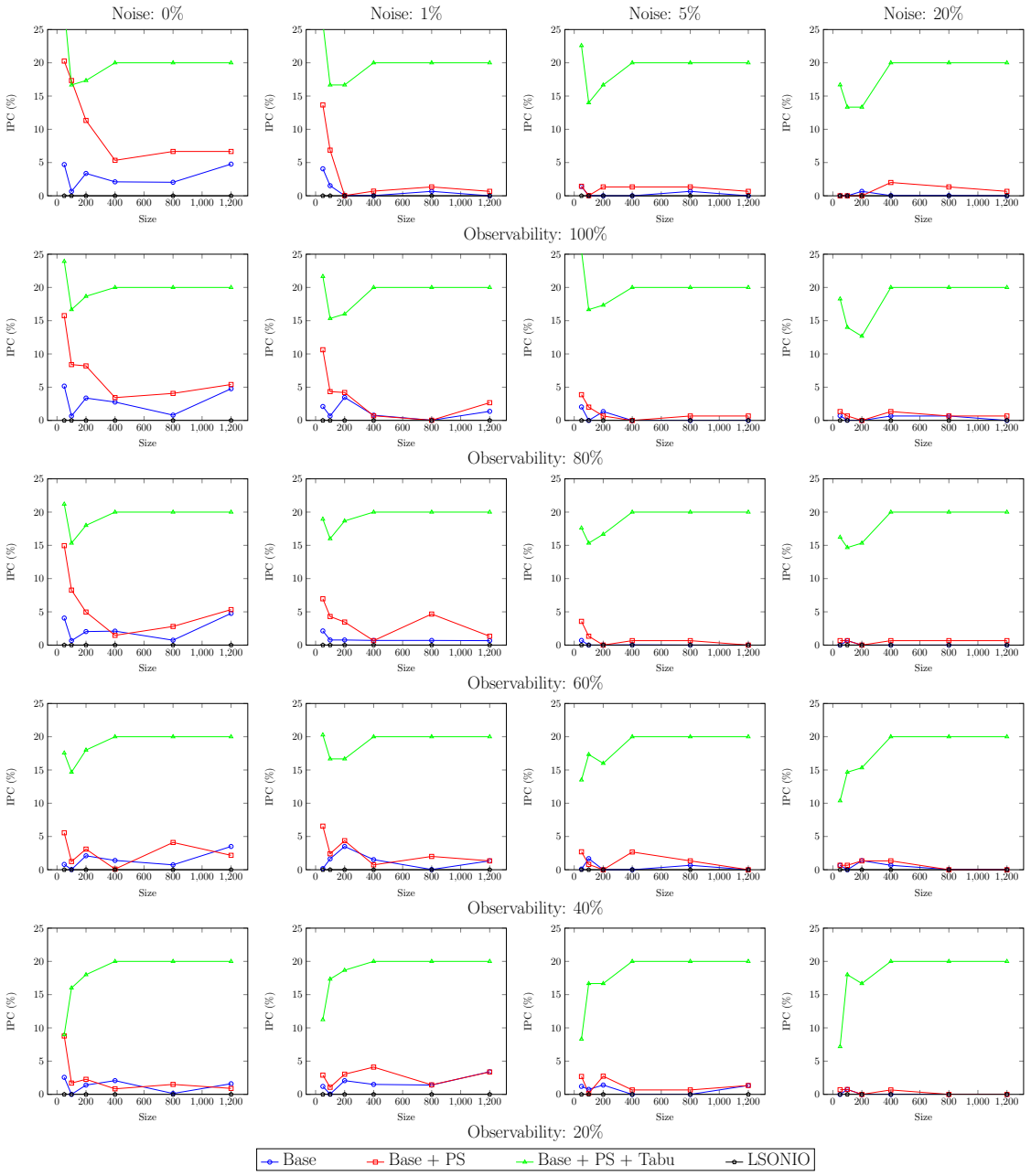


Figure B.40: Visit All – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

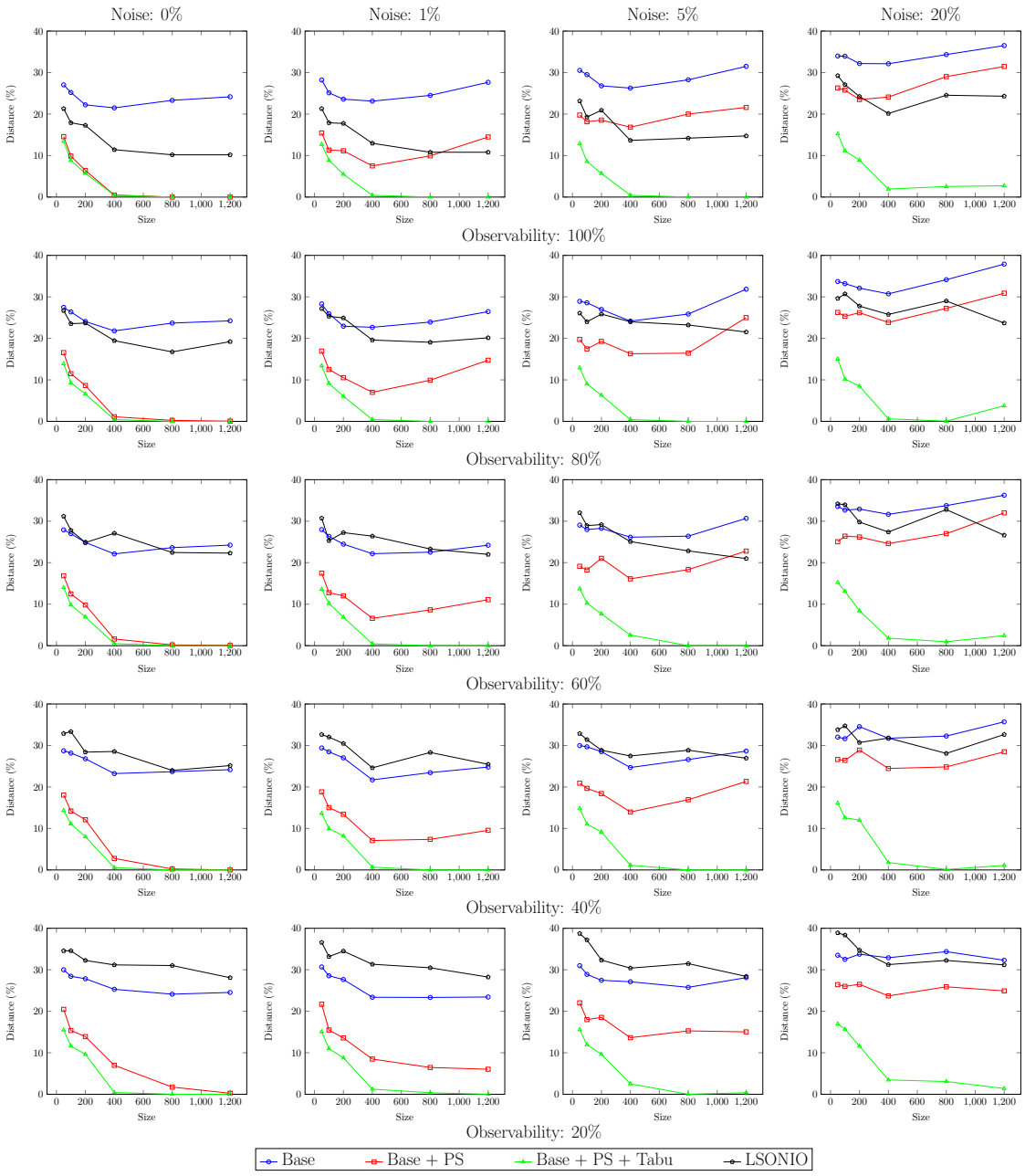


Figure B.41: Logistics – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

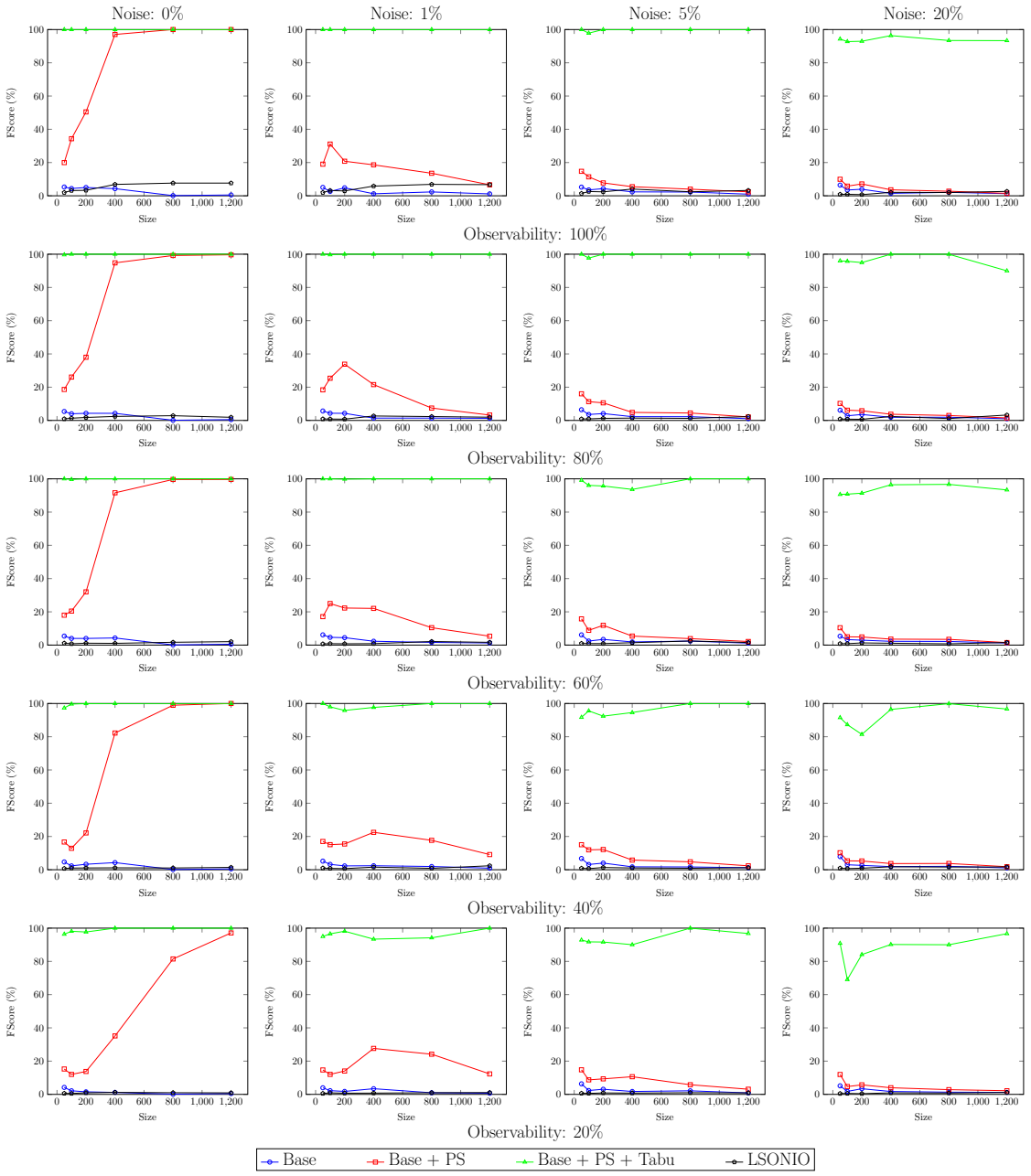


Figure B.42: Logistics – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

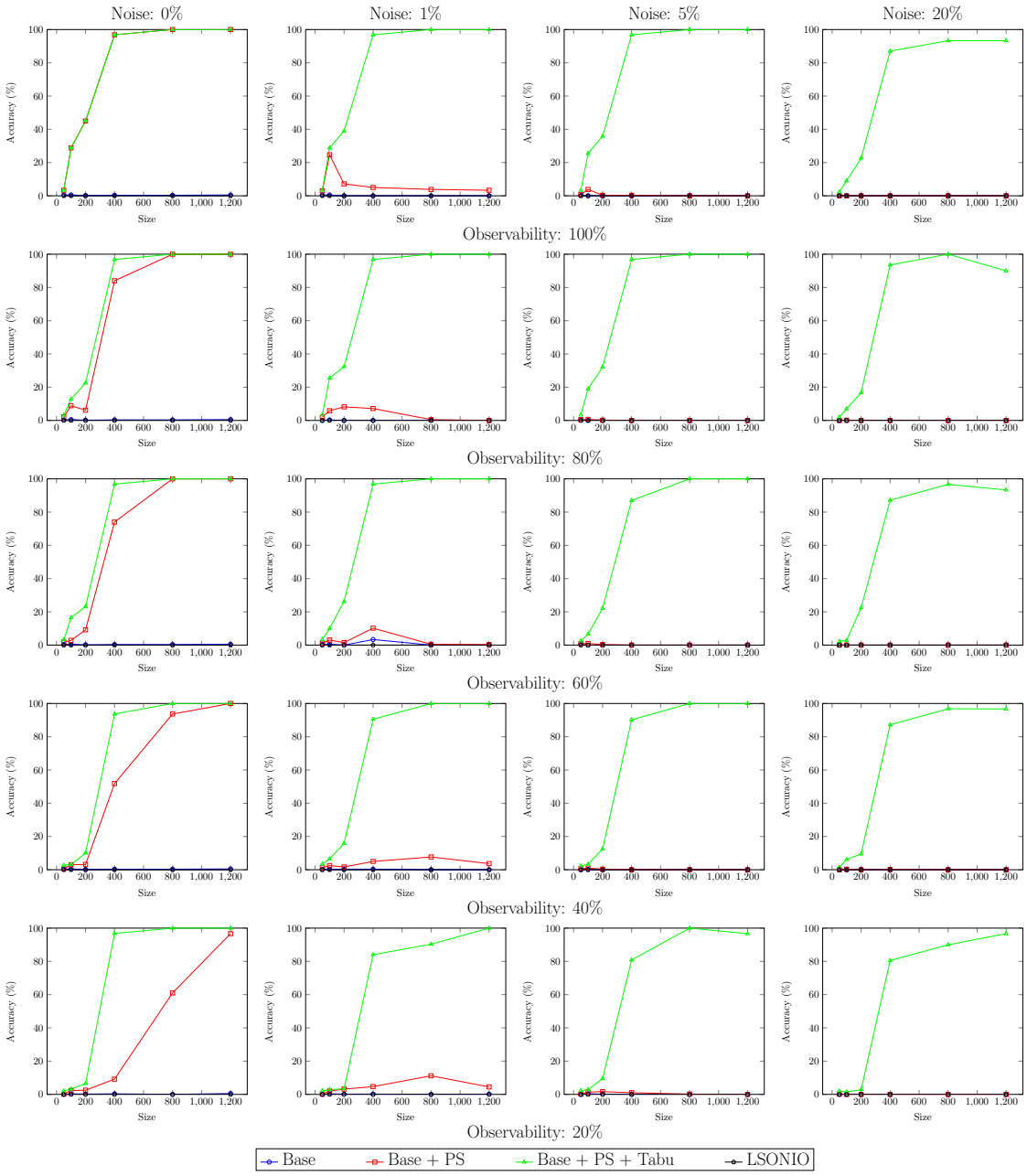


Figure B.43: Logistics – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

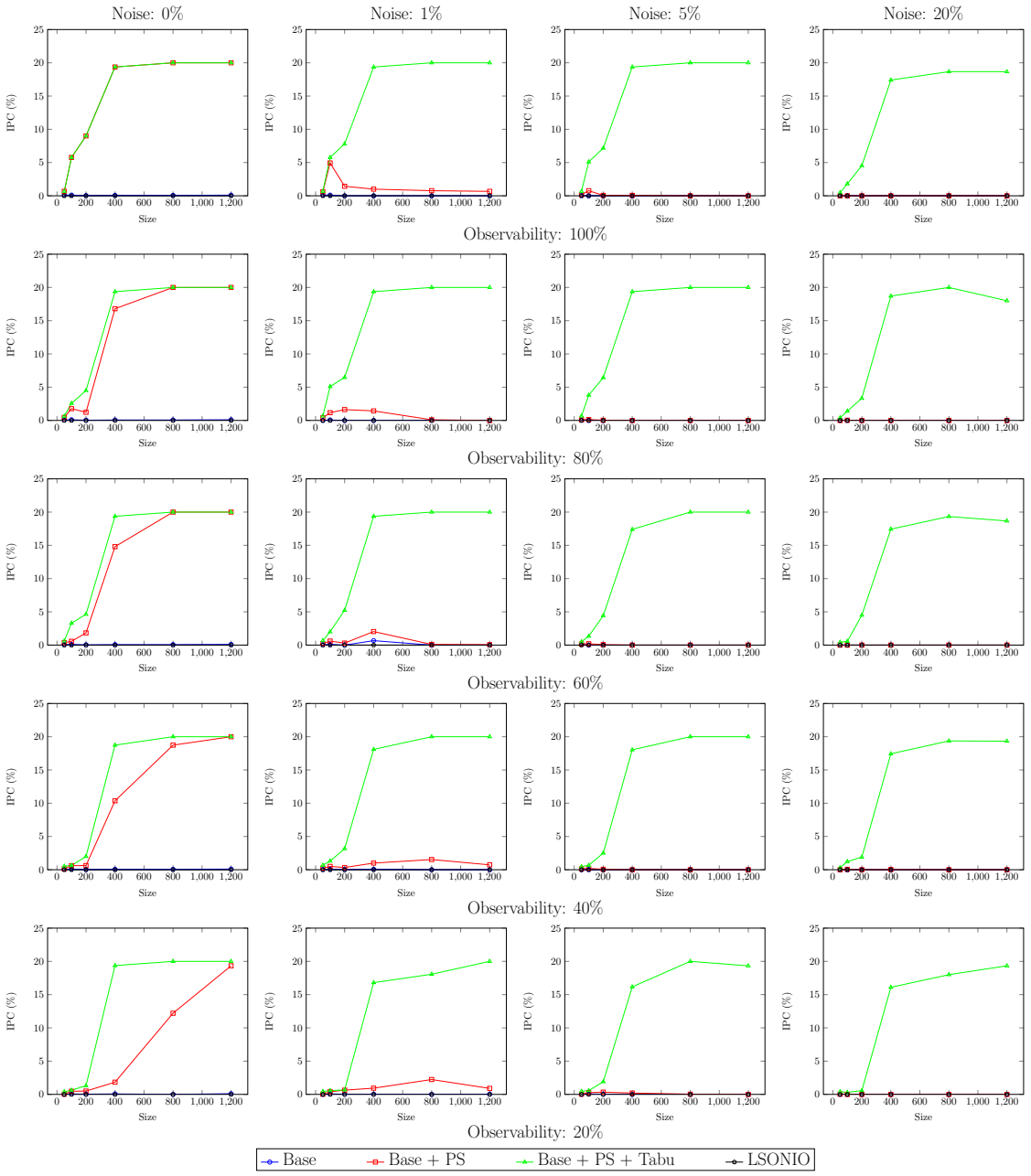


Figure B.44: Logistics – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

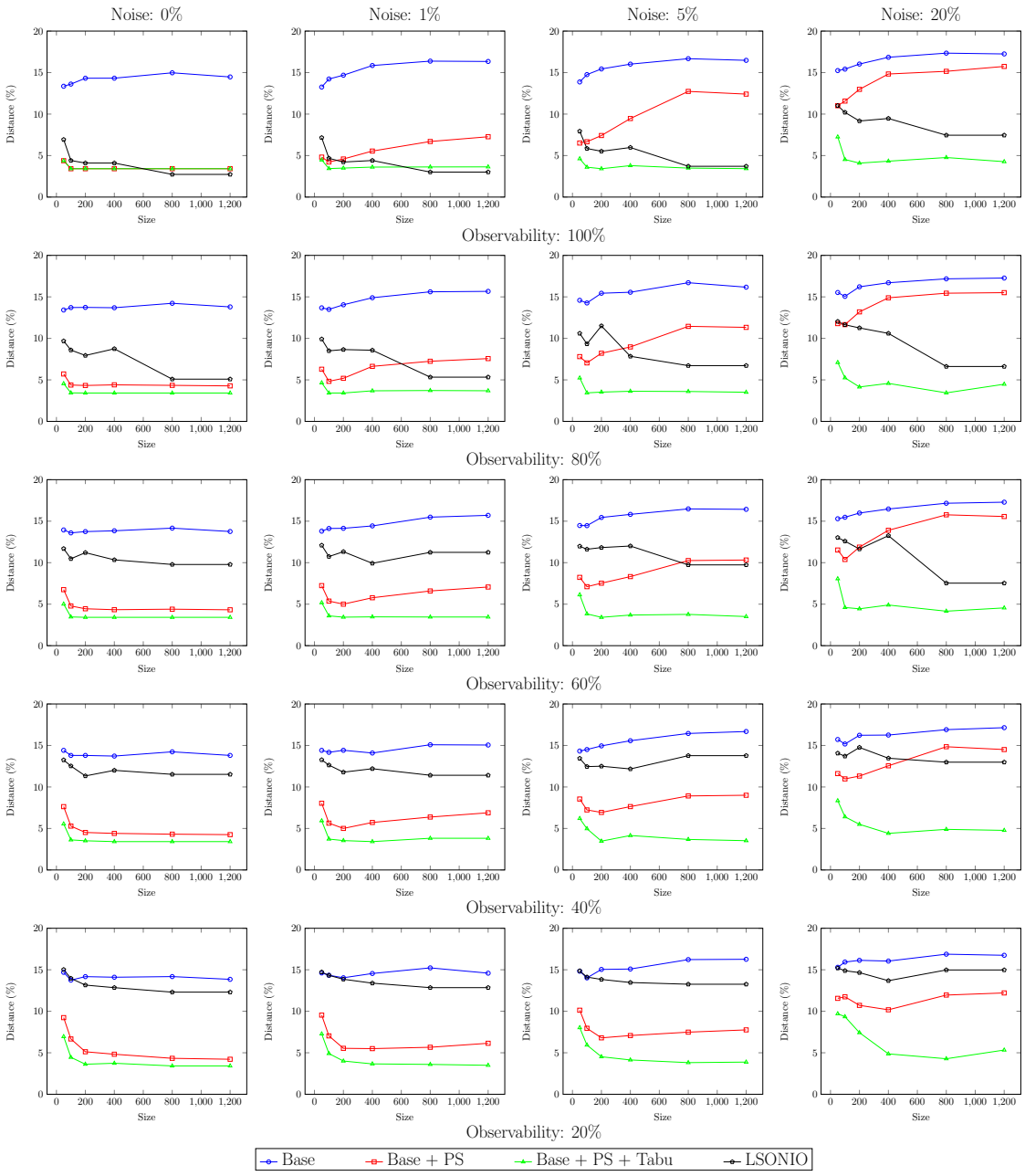


Figure B.45: Floortile – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

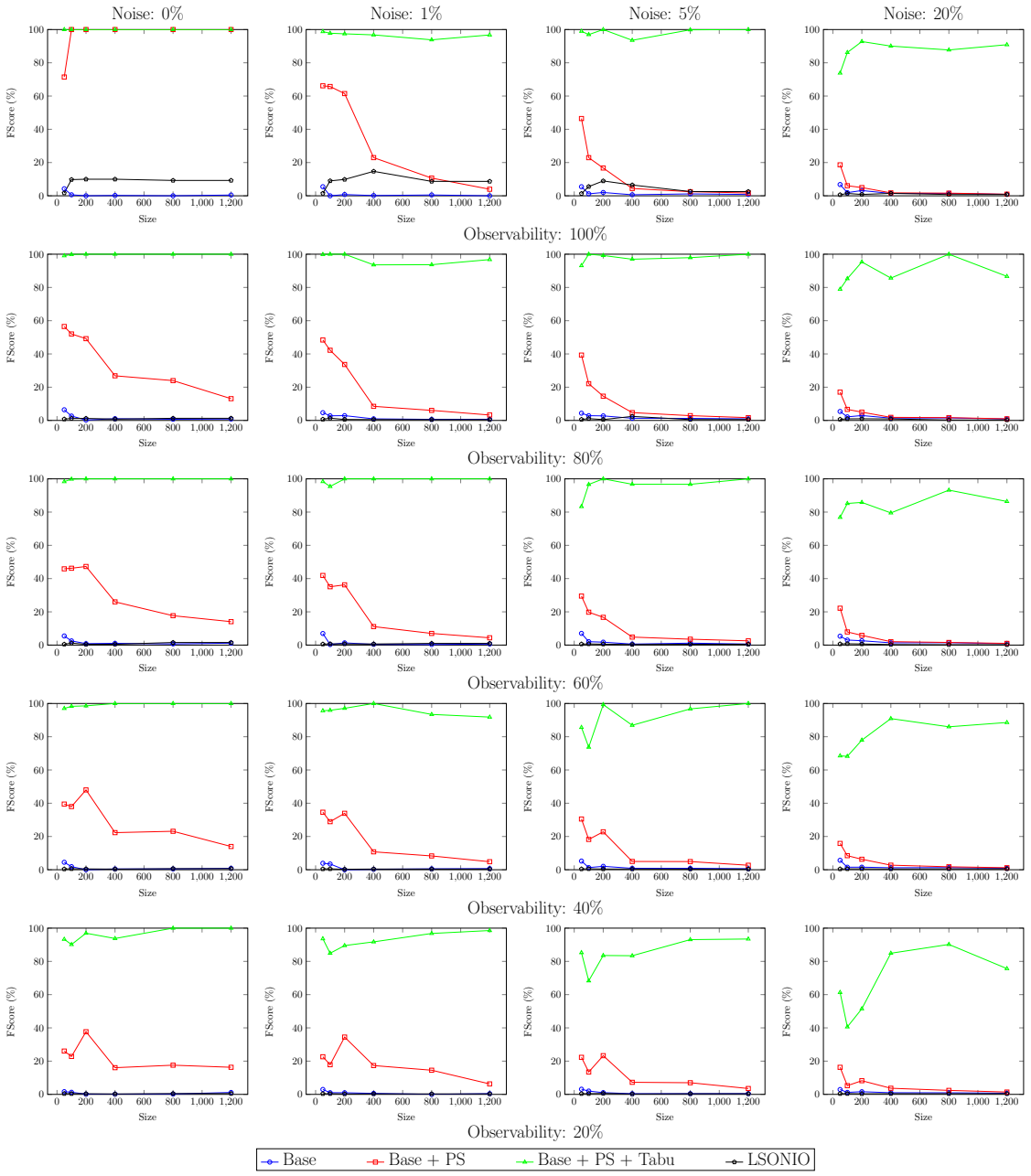


Figure B.46: Floortile – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

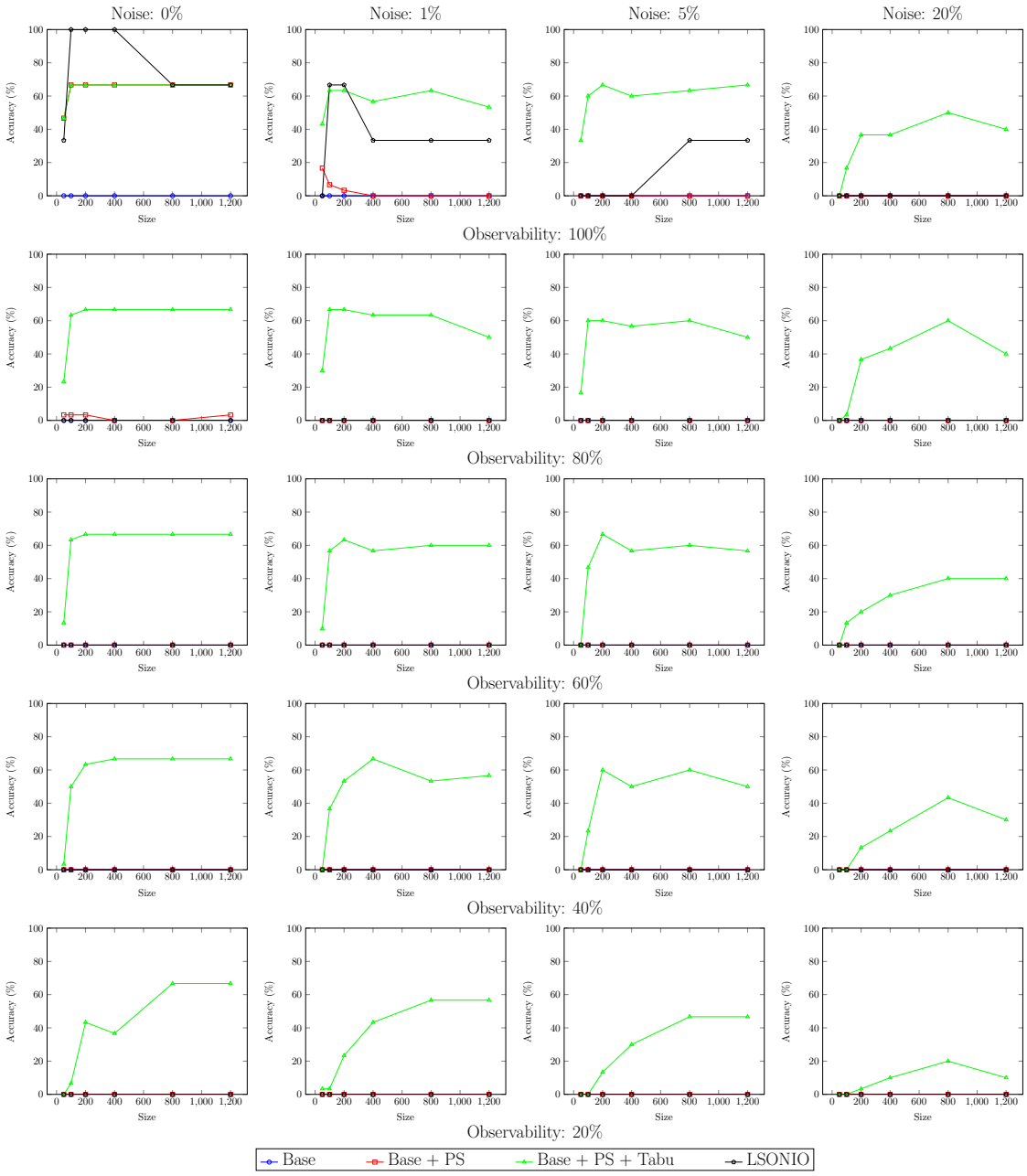


Figure B.47: Floortile – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

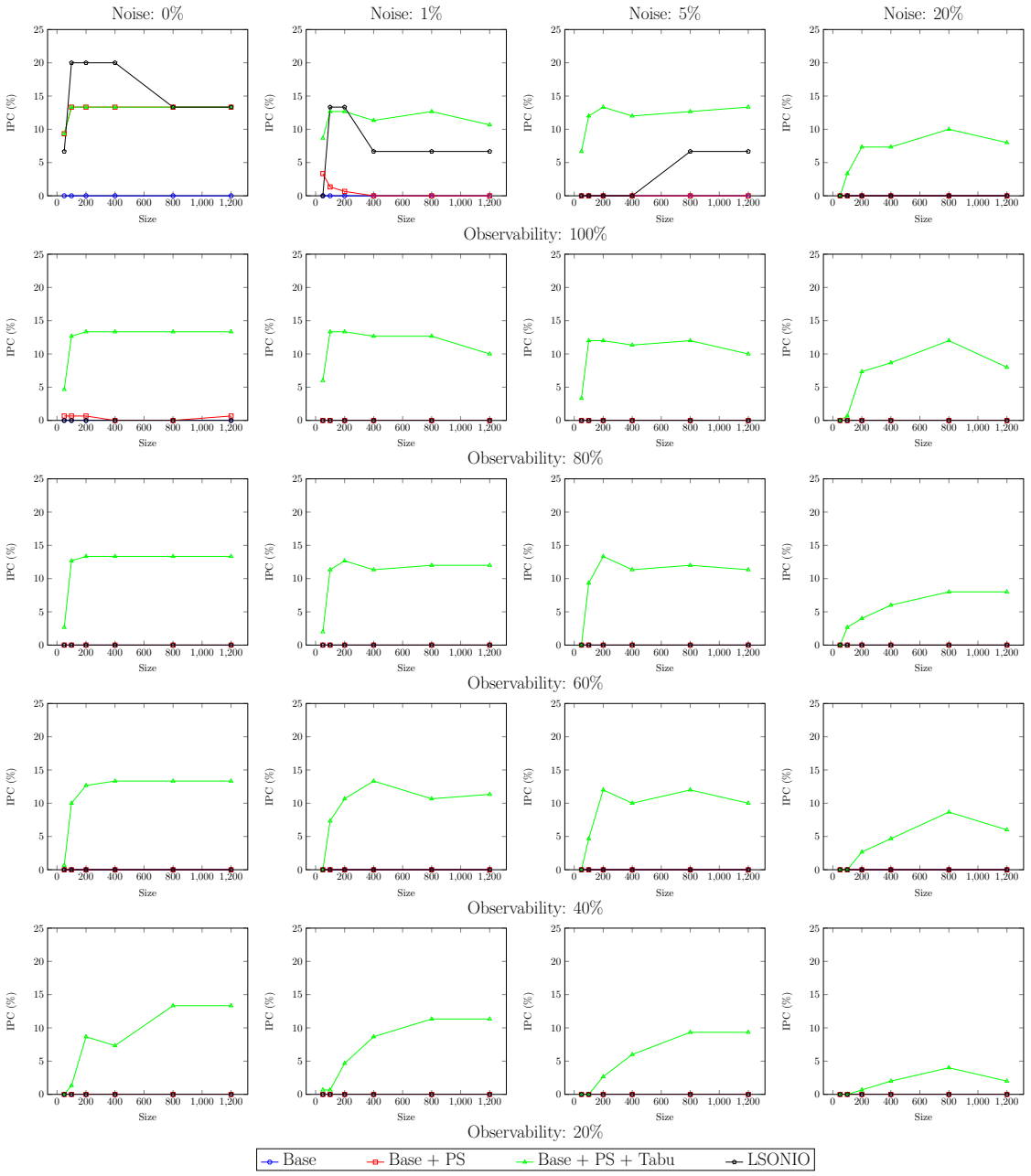


Figure B.48: Floortile – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

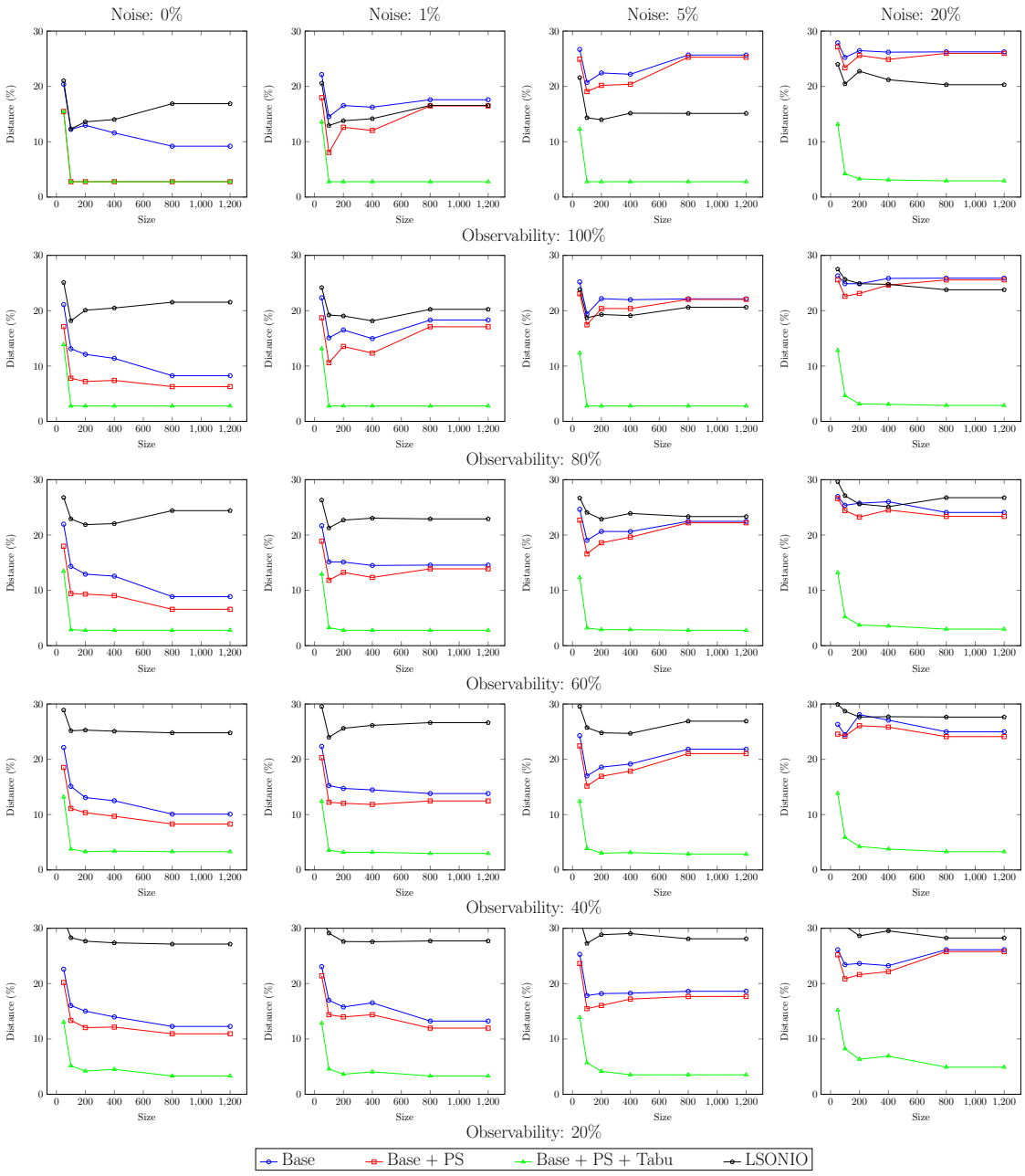


Figure B.49: Spanner – Average performances in terms of syntactical disatnce of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

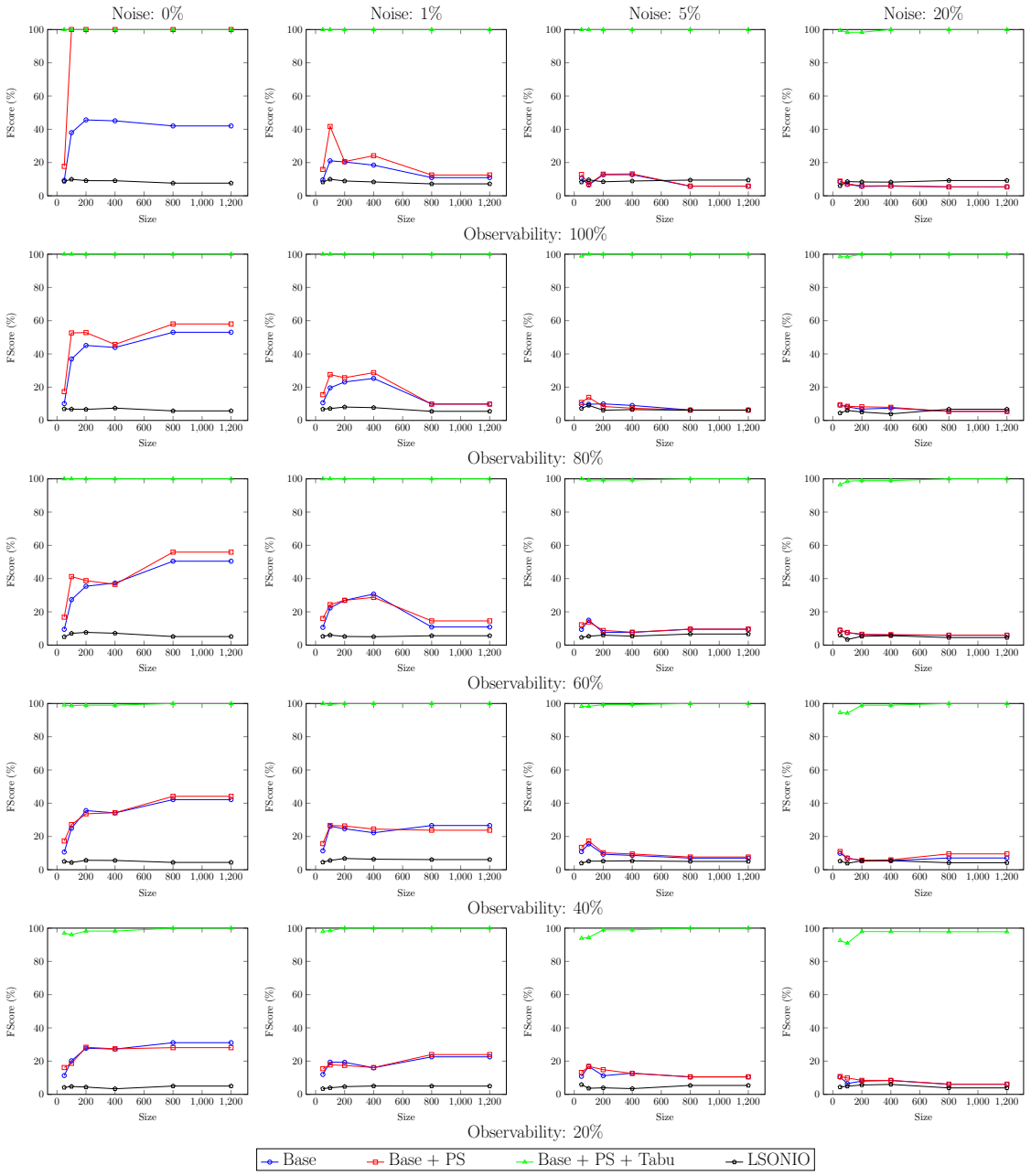


Figure B.50: Spanner – Average performances in terms of FScore of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

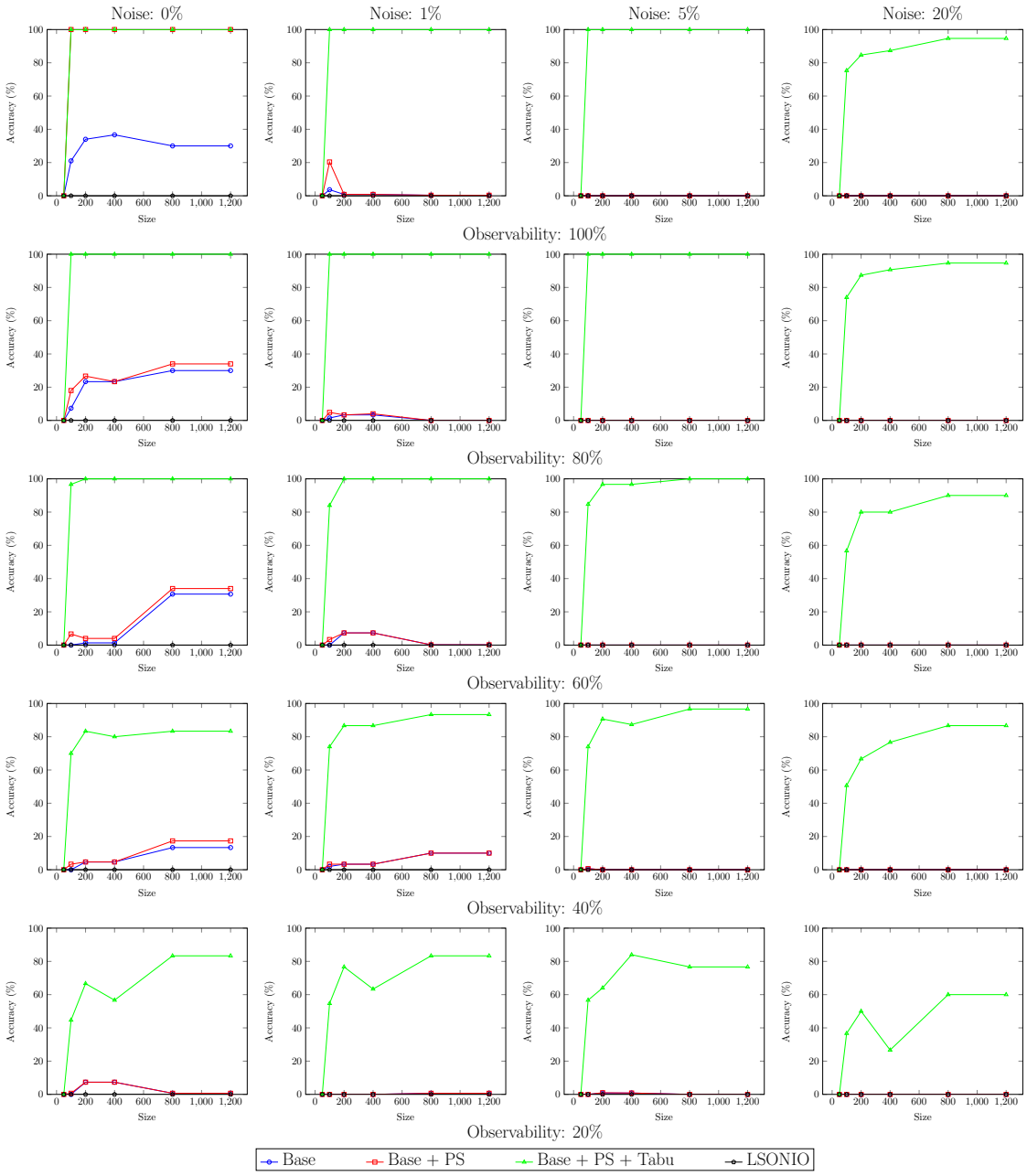


Figure B.51: Spanner – Average performances in terms of accuracy of AMLSI and LSONIO when the training dataset increases in terms of number of actions.

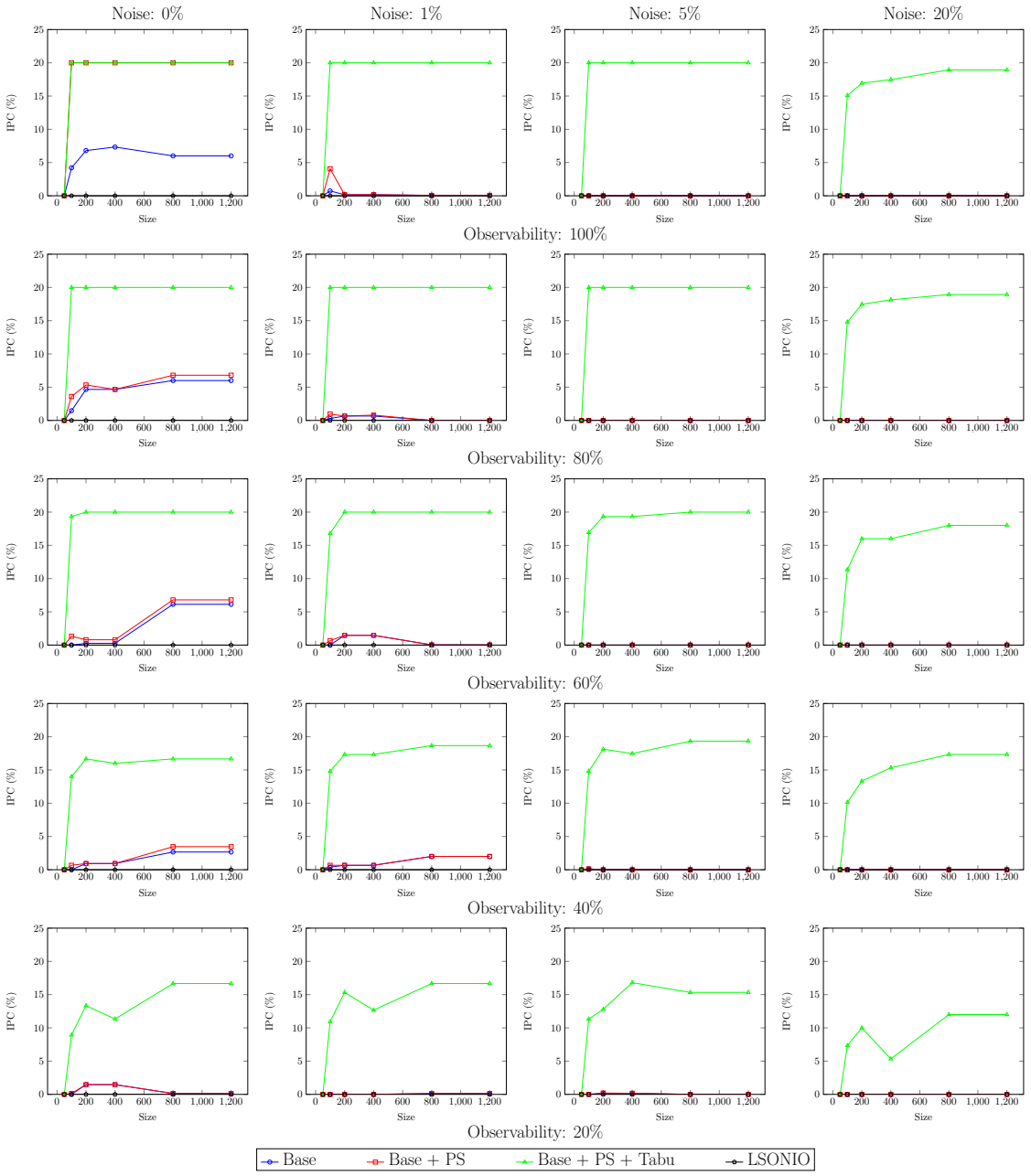
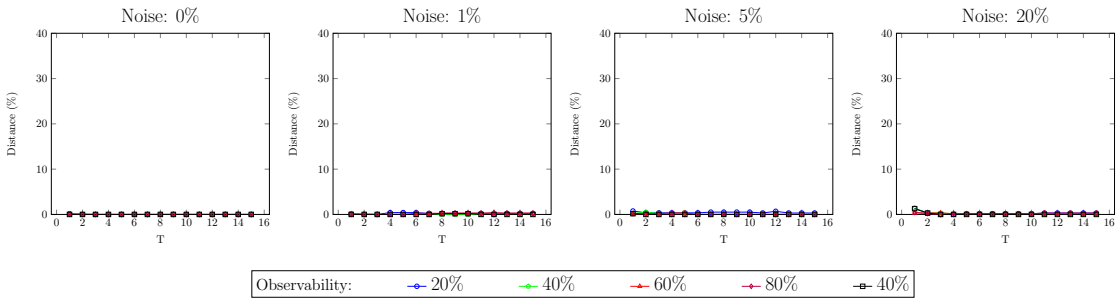
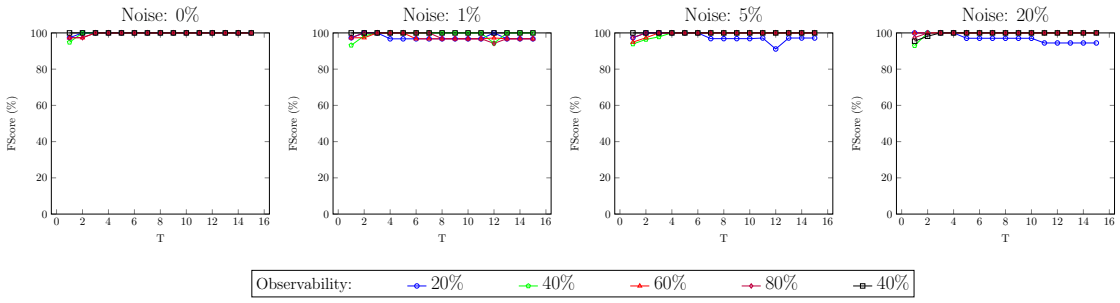


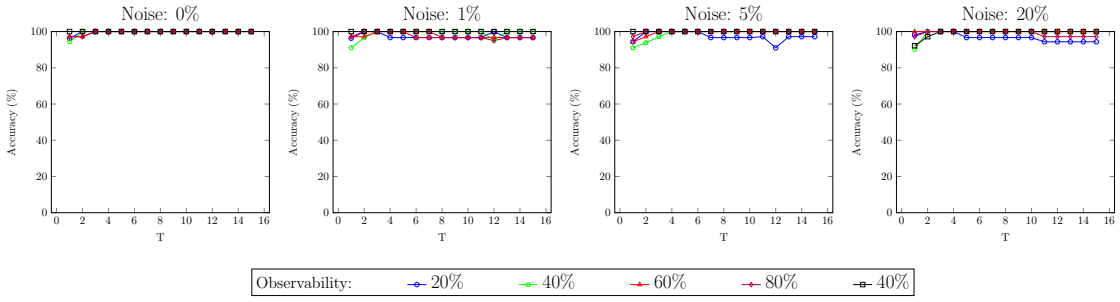
Figure B.52: Spanner – Average performances in terms of IPC score of AMLSI and LSONIO when the training dataset increases in terms of number of actions.



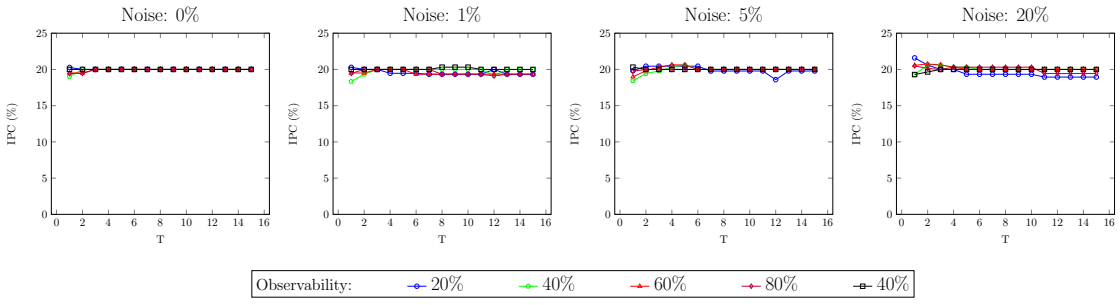
(a) Syntactical Distance



(b) FScore

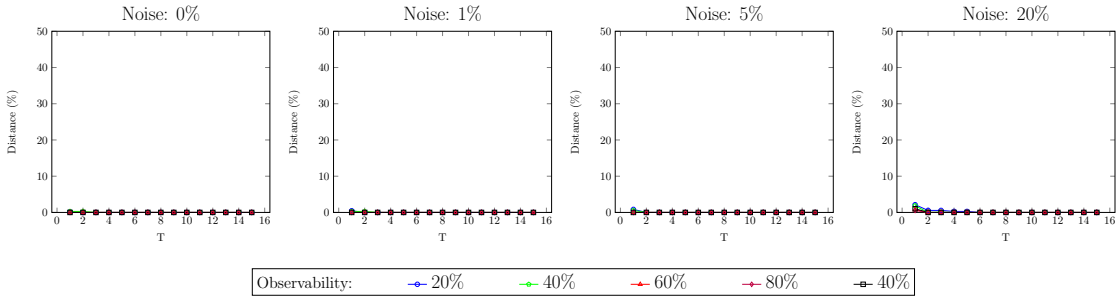


(c) Accuracy

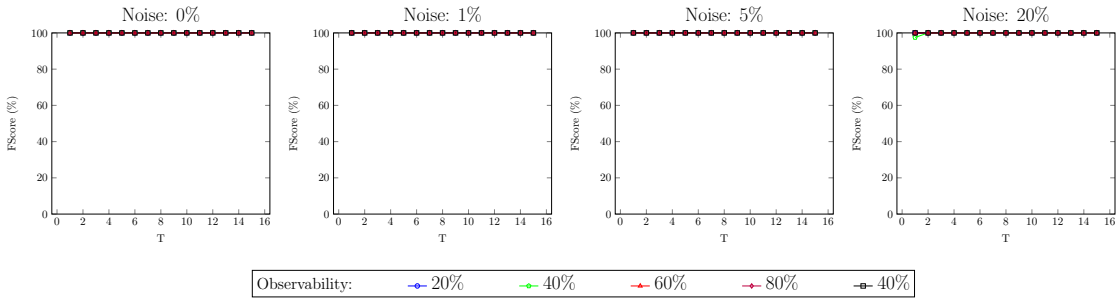


(d) IPC

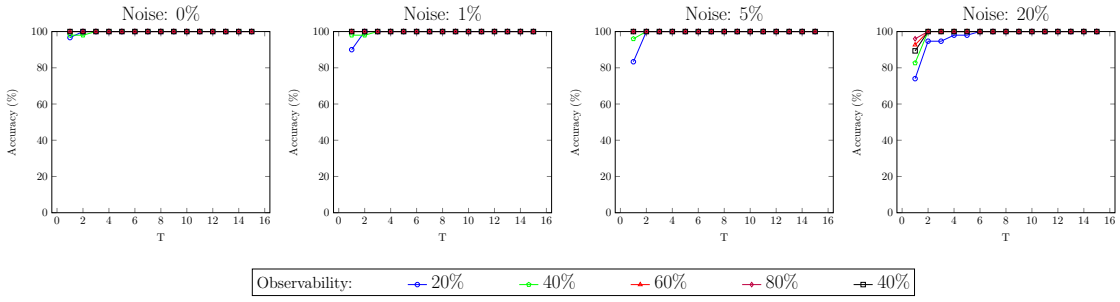
Figure B.53: Blocksworld – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



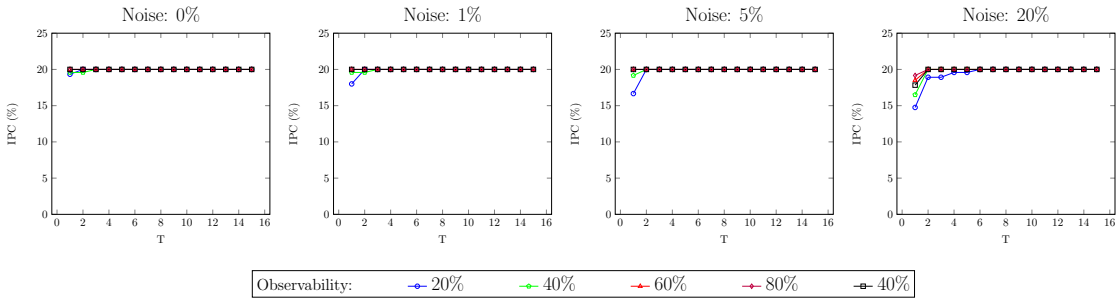
(a) Syntactical Distance



(b) FScore

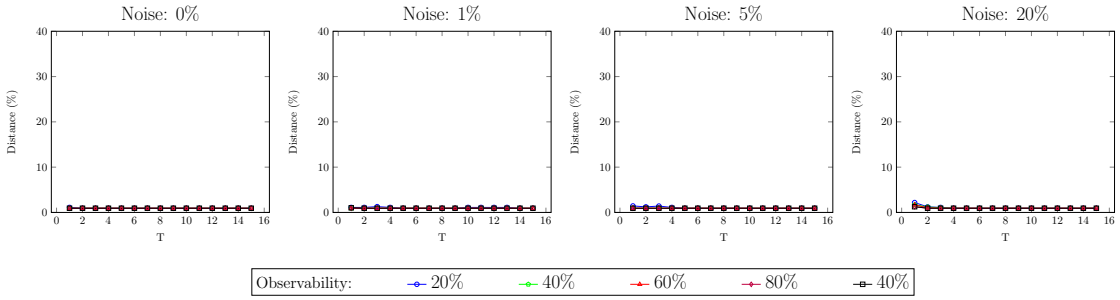


(c) Accuracy

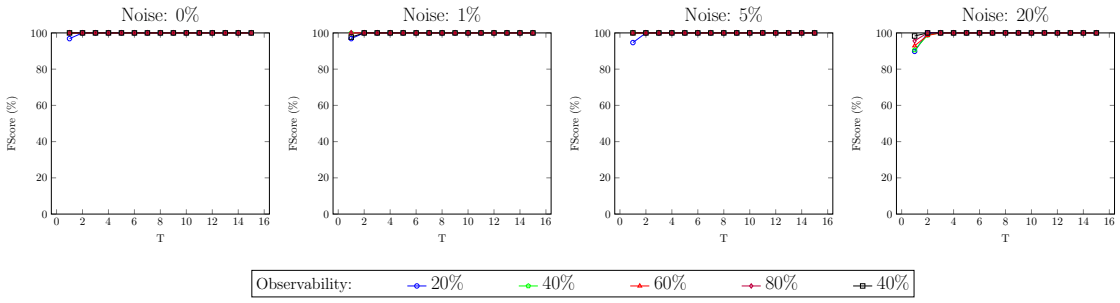


(d) IPC

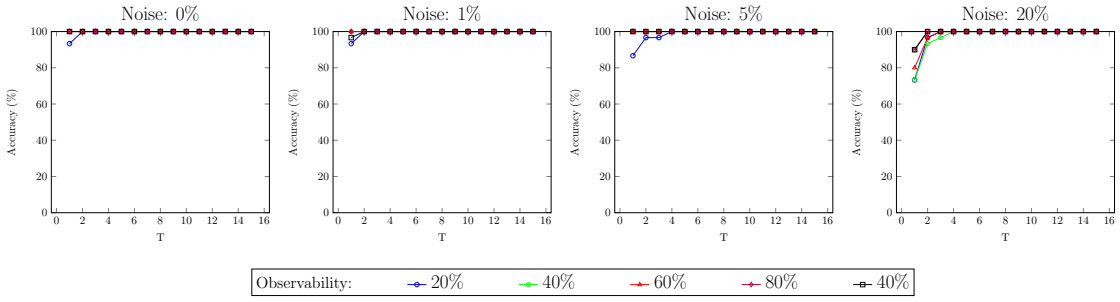
Figure B.54: Gripper – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



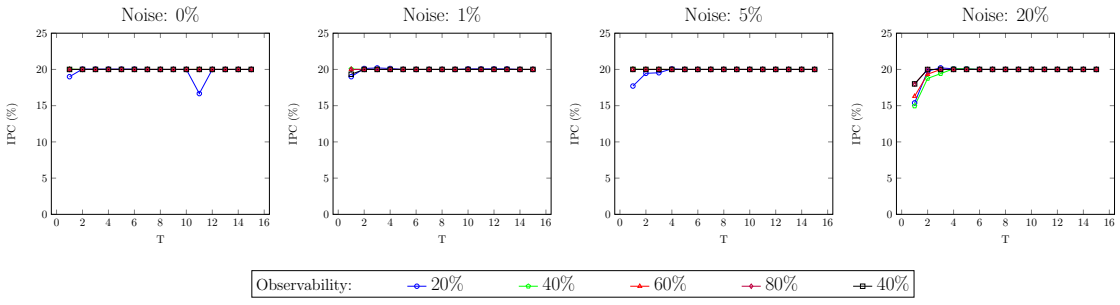
(a) Syntactical Distance



(b) FScore

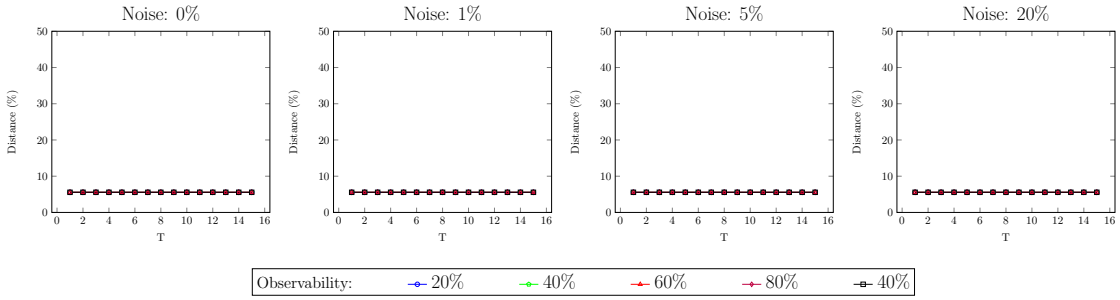


(c) Accuracy

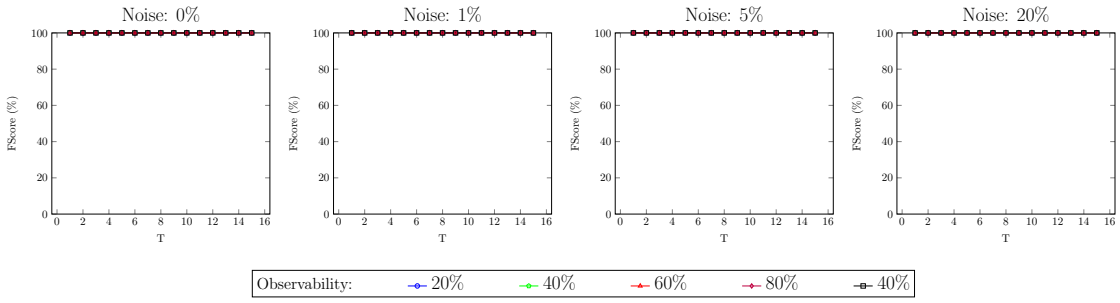


(d) IPC

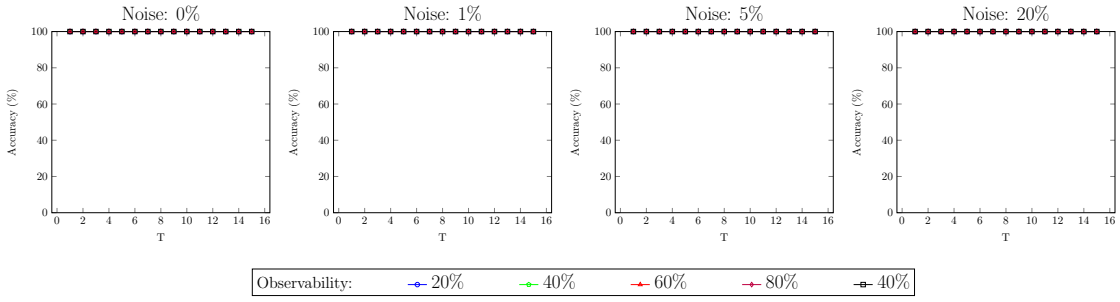
Figure B.55: Hanoi – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



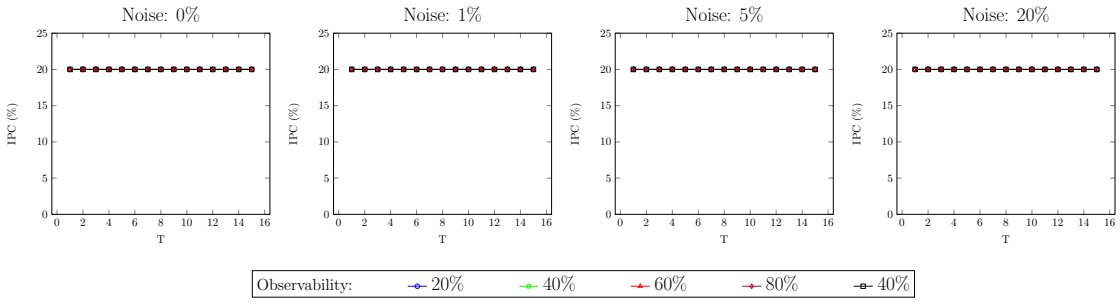
(a) Syntactical Distance



(b) FScore

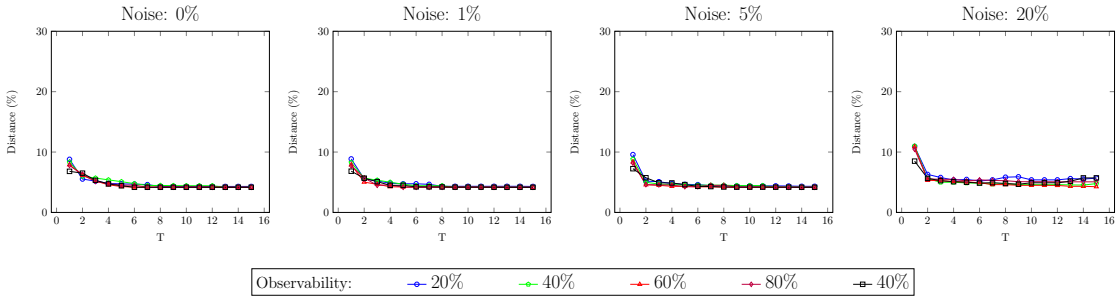


(c) Accuracy

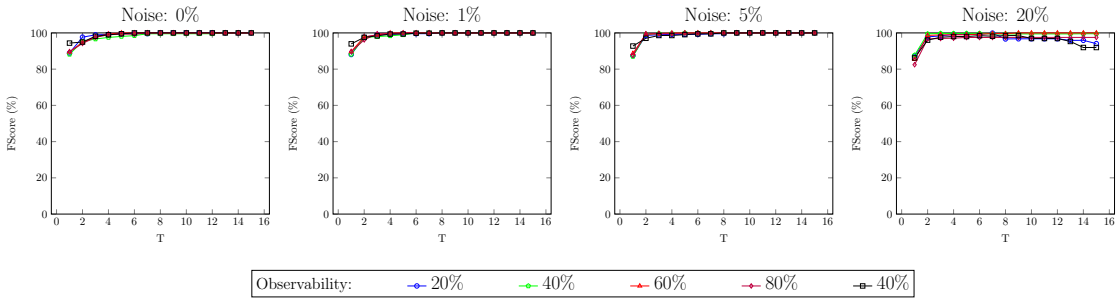


(d) IPC

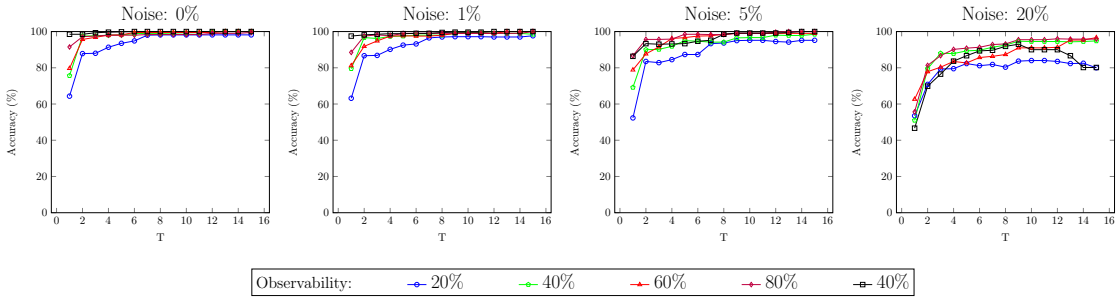
Figure B.56: N Puzzle – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



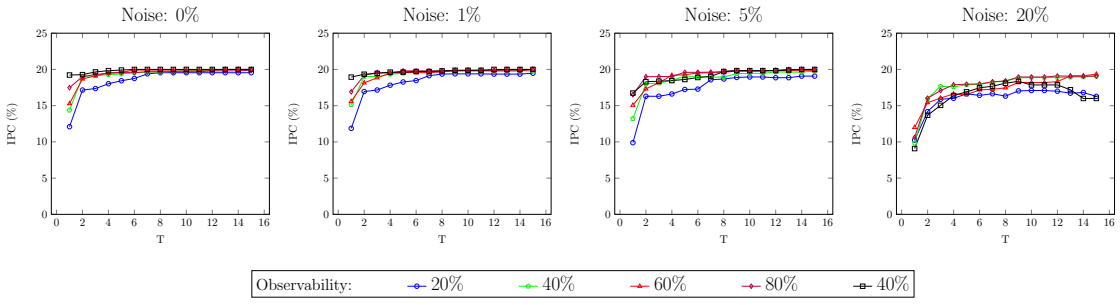
(a) Syntactical Distance



(b) FScore

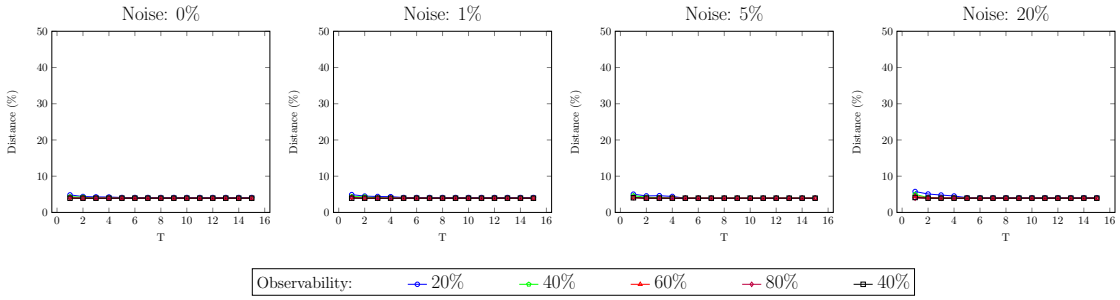


(c) Accuracy

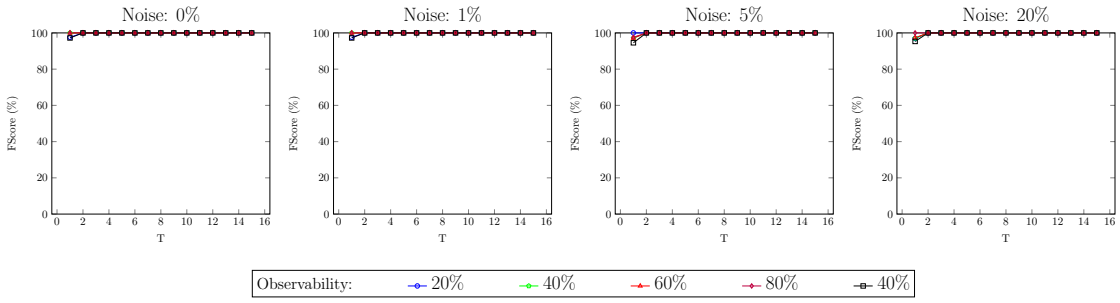


(d) IPC

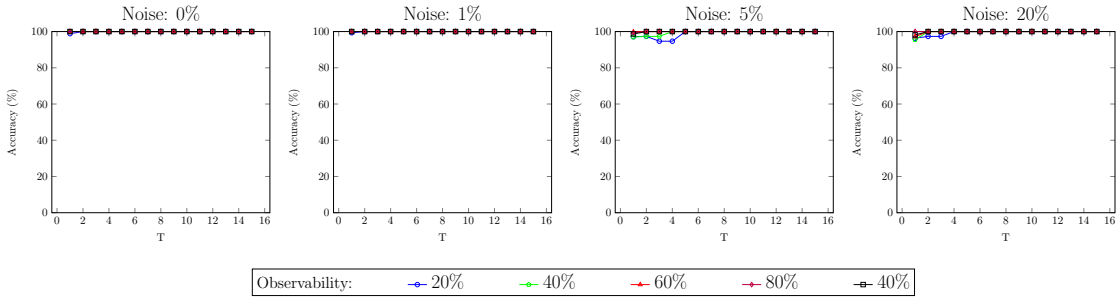
Figure B.57: Peg Solitaire – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



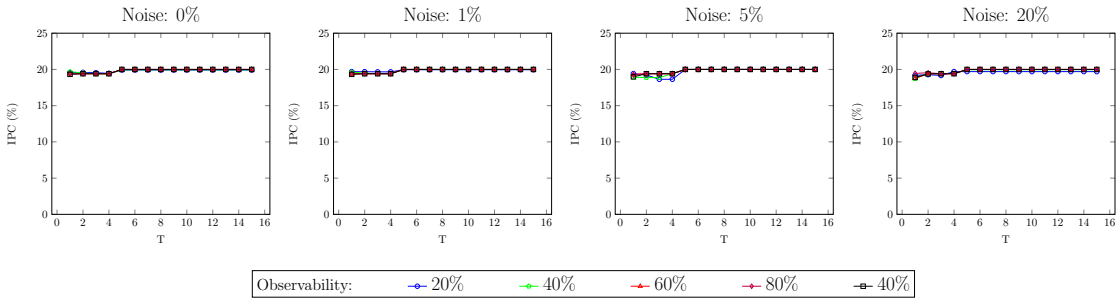
(a) Syntactical Distance



(b) FScore

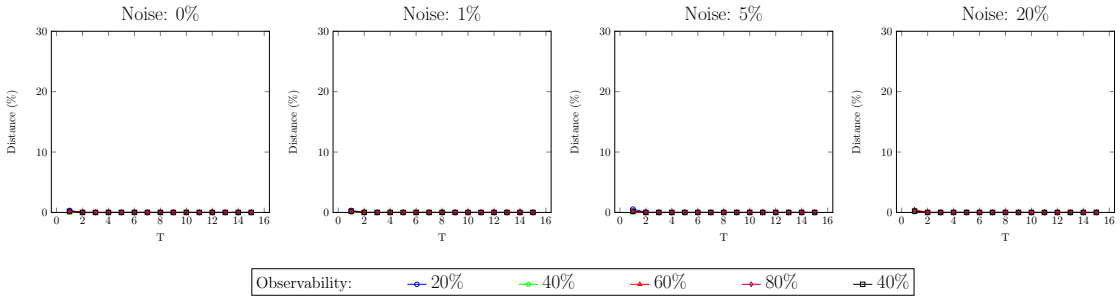


(c) Accuracy

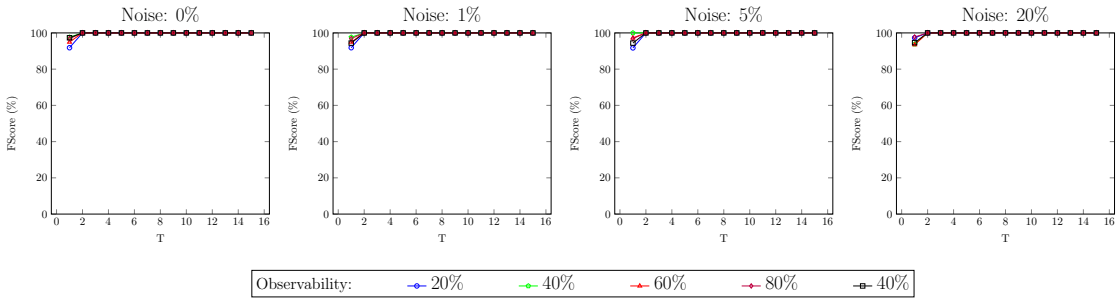


(d) IPC

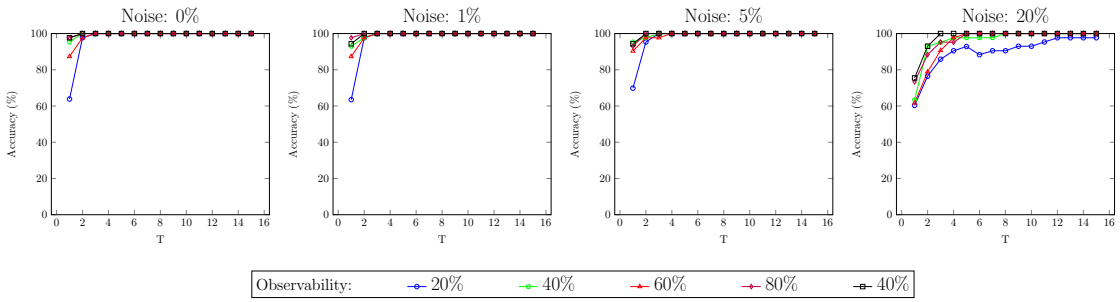
Figure B.58: Parking – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



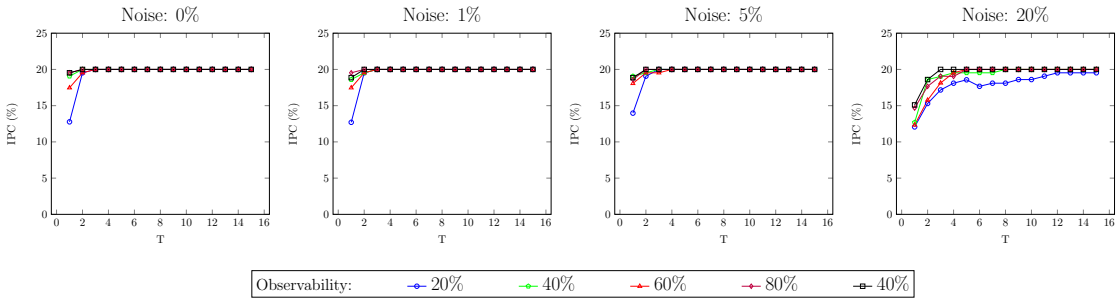
(a) Syntactical Distance



(b) FScore

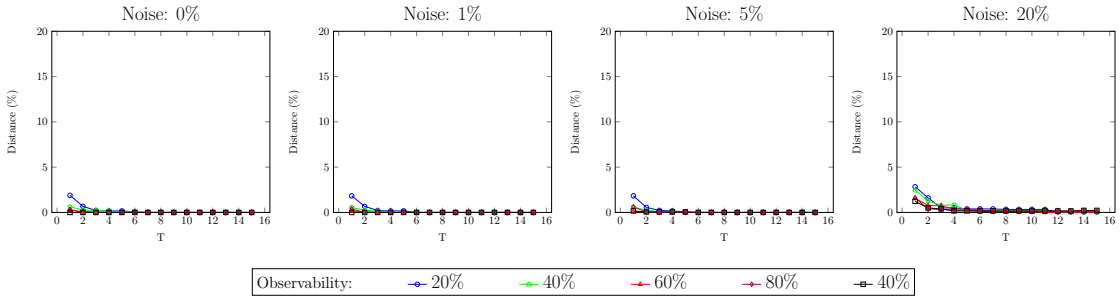


(c) Accuracy

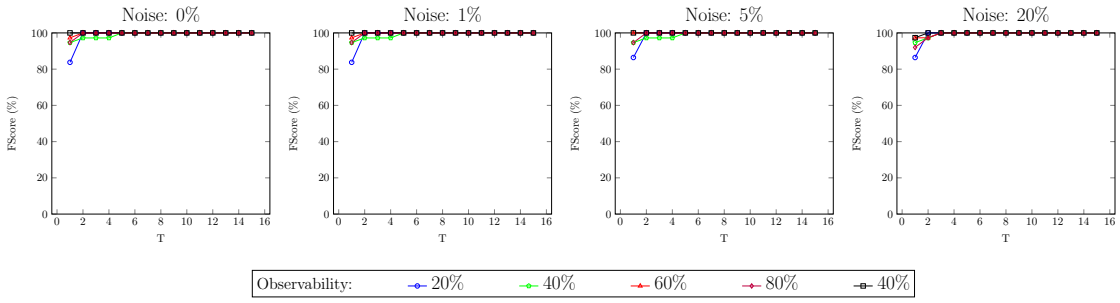


(d) IPC

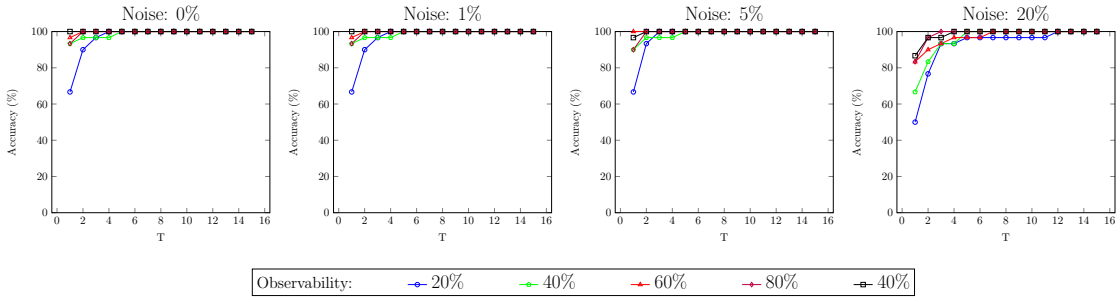
Figure B.59: Zenotravel – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



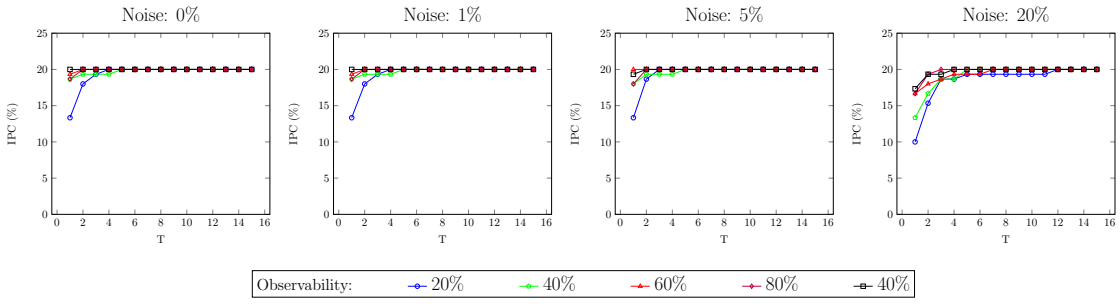
(a) Syntactical Distance



(b) FScore

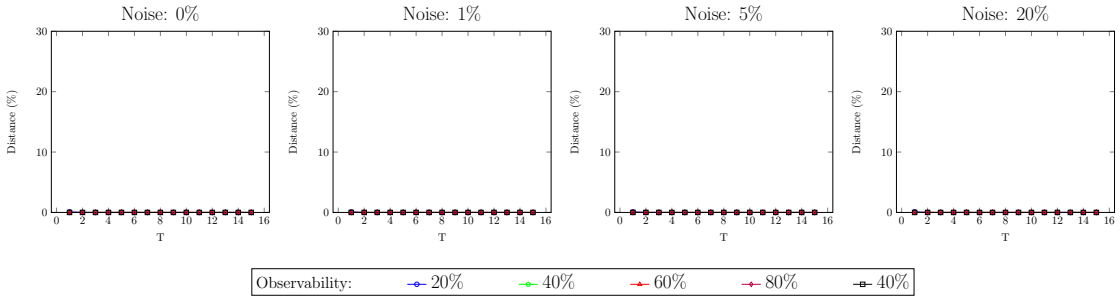


(c) Accuracy

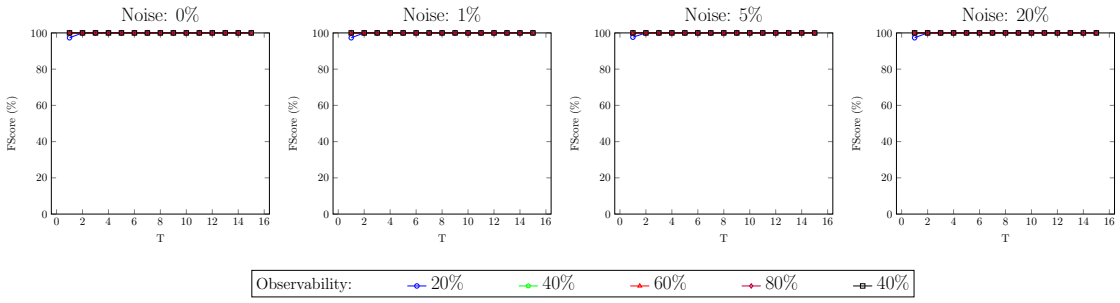


(d) IPC

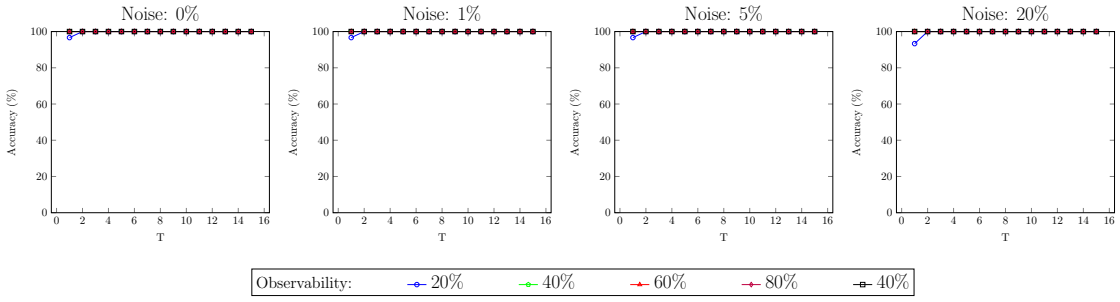
Figure B.60: Elevator – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



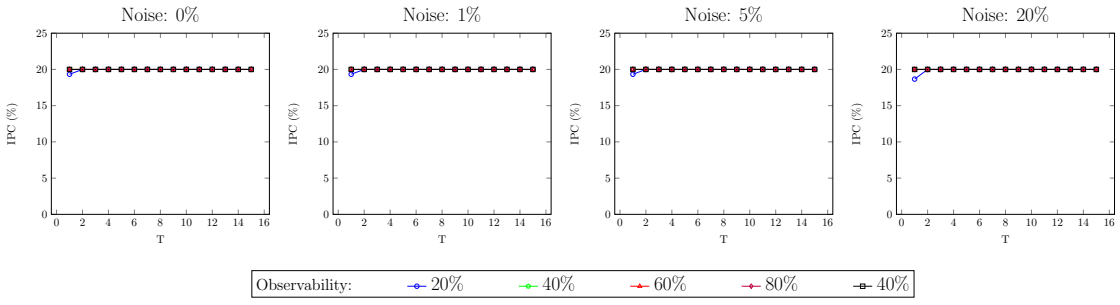
(a) Syntactical Distance



(b) FScore

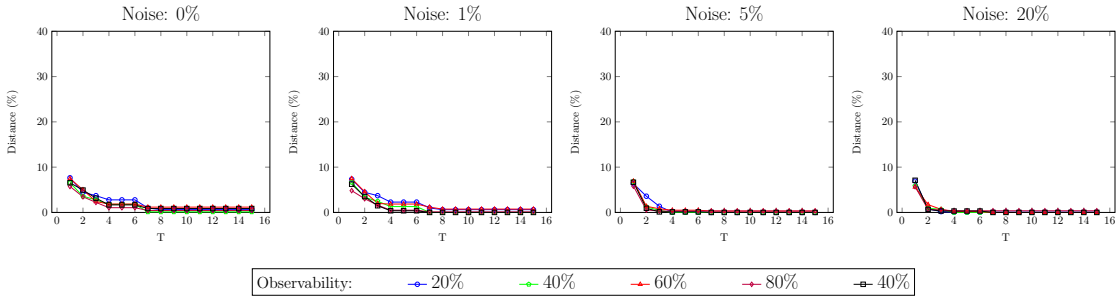


(c) Accuracy

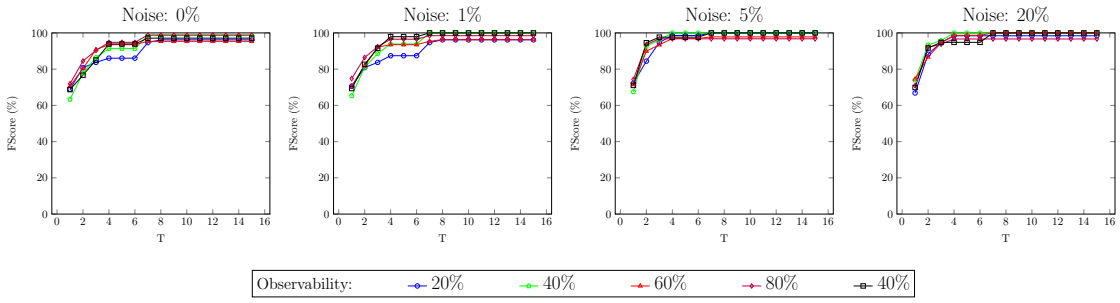


(d) IPC

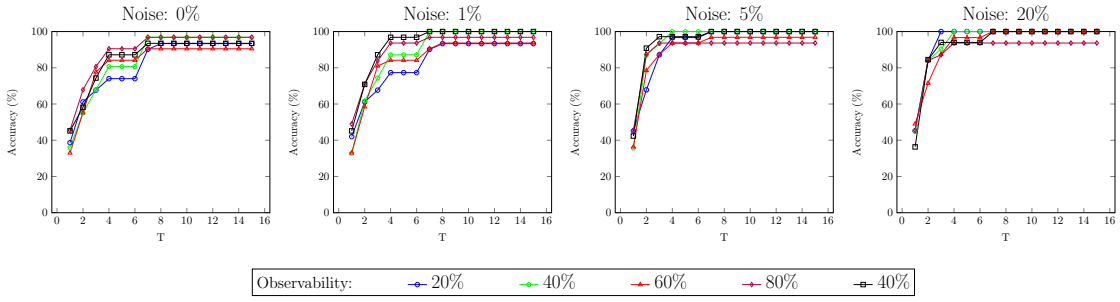
Figure B.61: Visit All – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



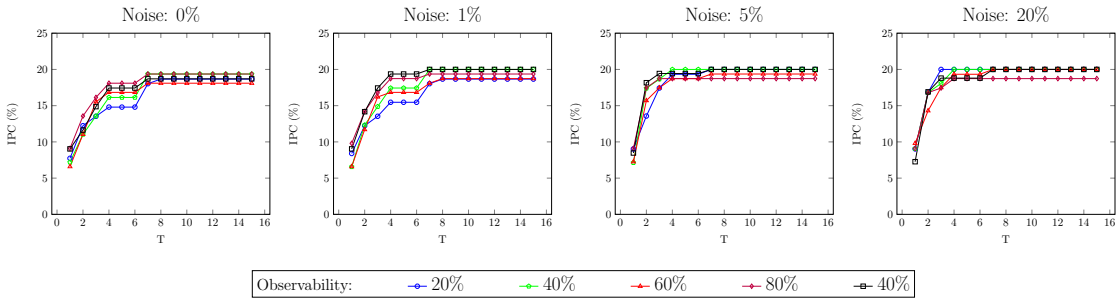
(a) Syntactical Distance



(b) FScore

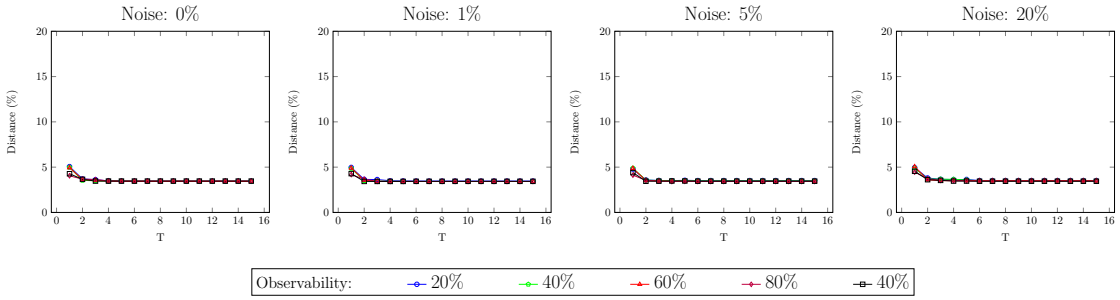


(c) Accuracy

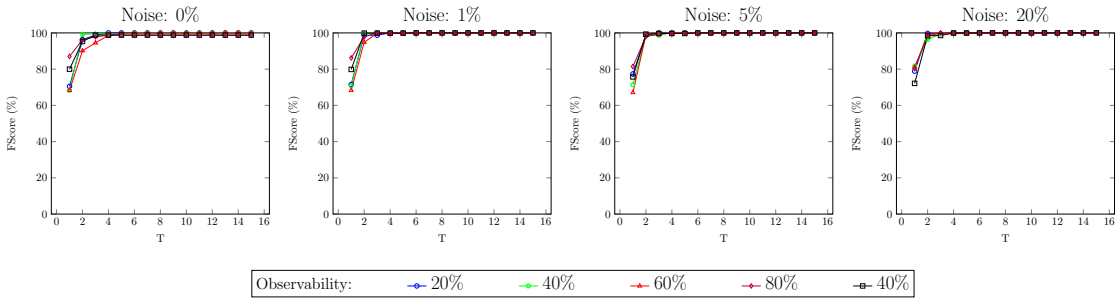


(d) IPC

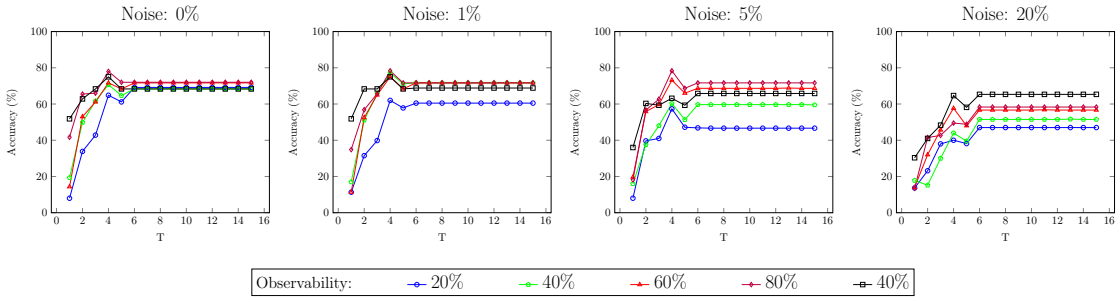
Figure B.62: Logistics – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



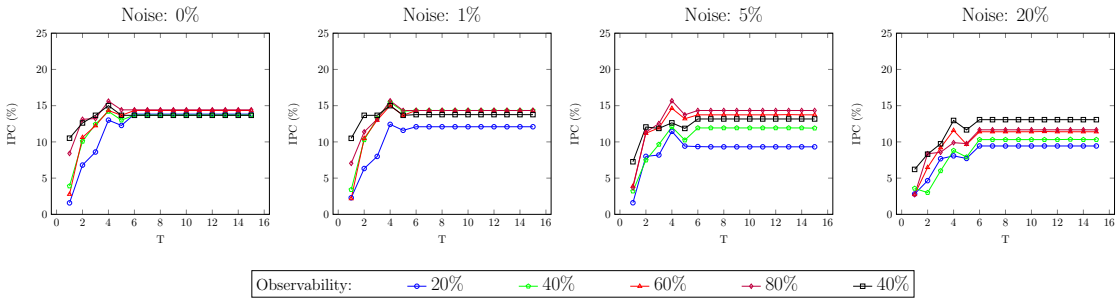
(a) Syntactical Distance



(b) FScore

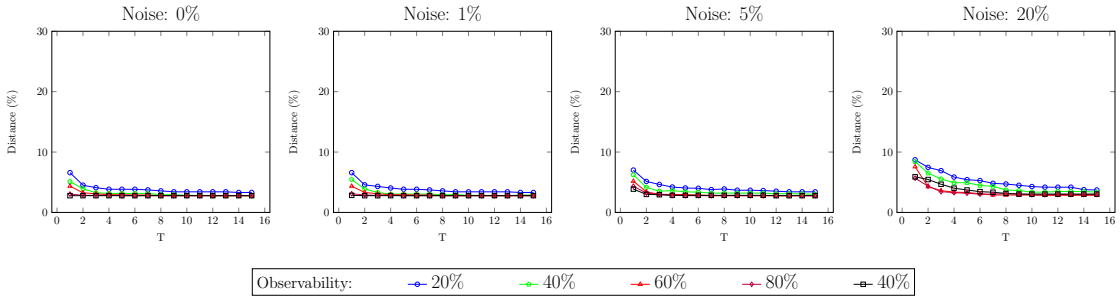


(c) Accuracy

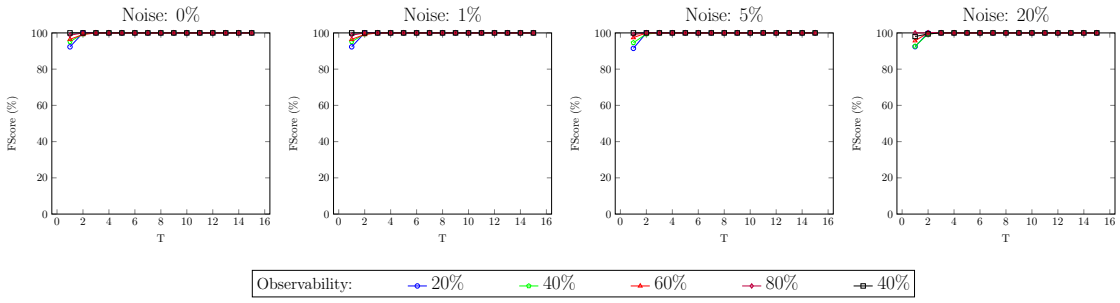


(d) IPC

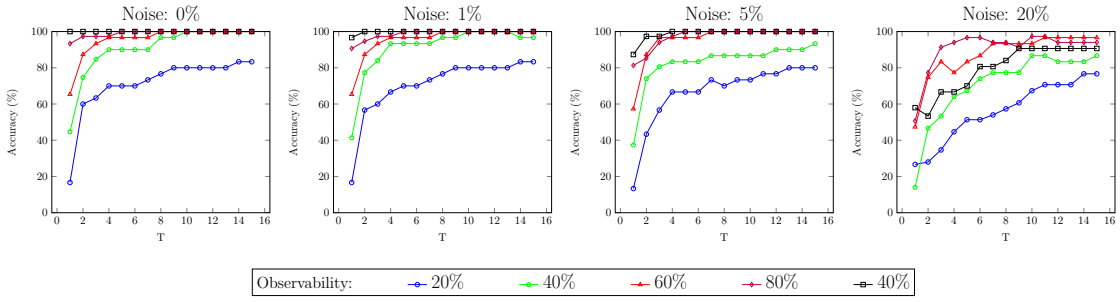
Figure B.63: Floortile – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.



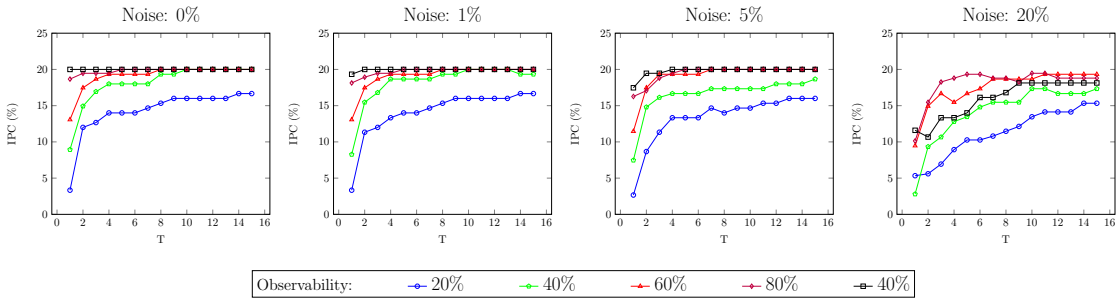
(a) Syntactical Distance



(b) FScore



(c) Accuracy



(d) IPC

Figure B.64: Spanner – Average performance of IncrAMLSI when the convergence criterion T varies between 1 and 15.

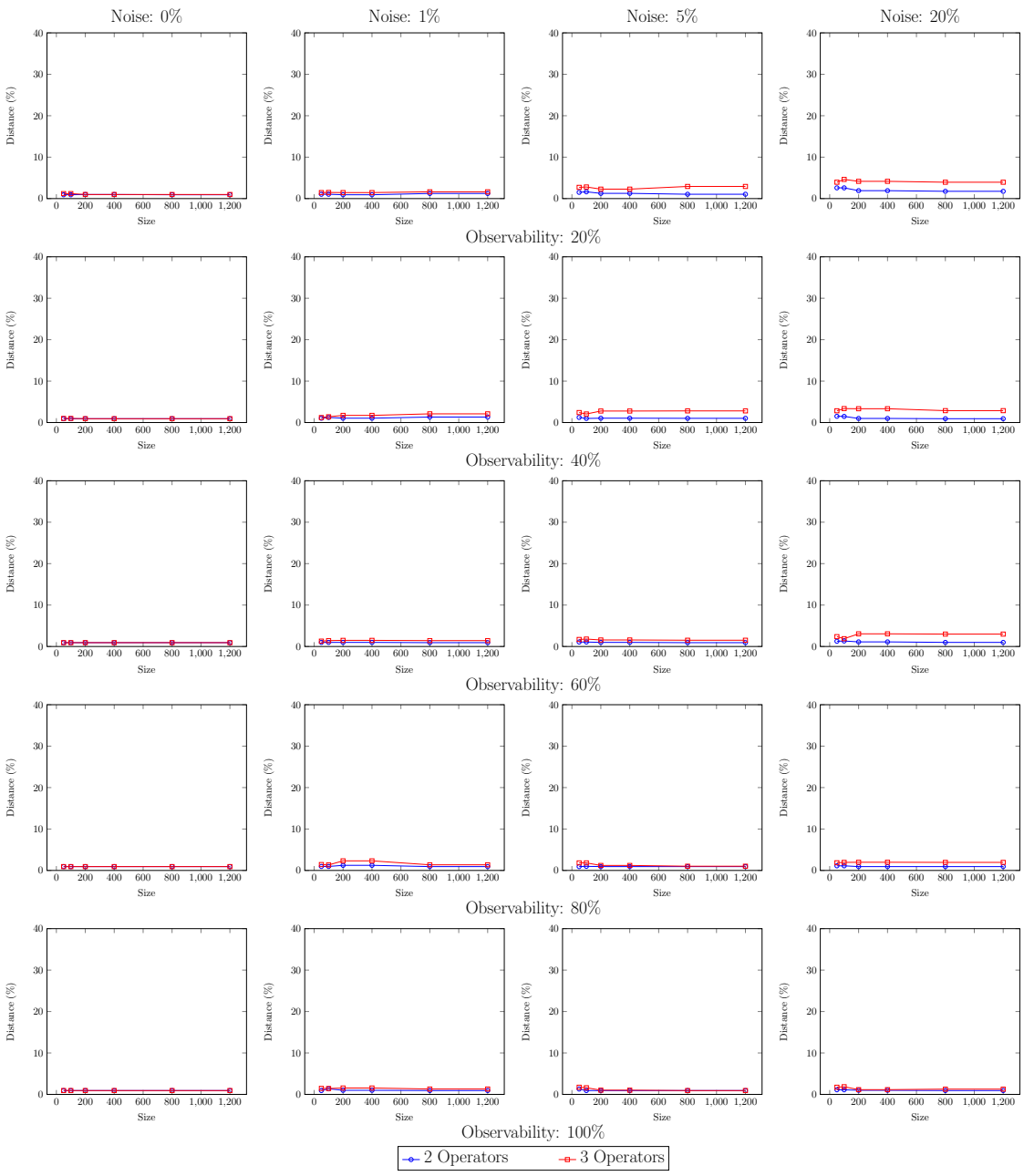


Figure B.65: Peg Solitaire – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of syntactical distance.

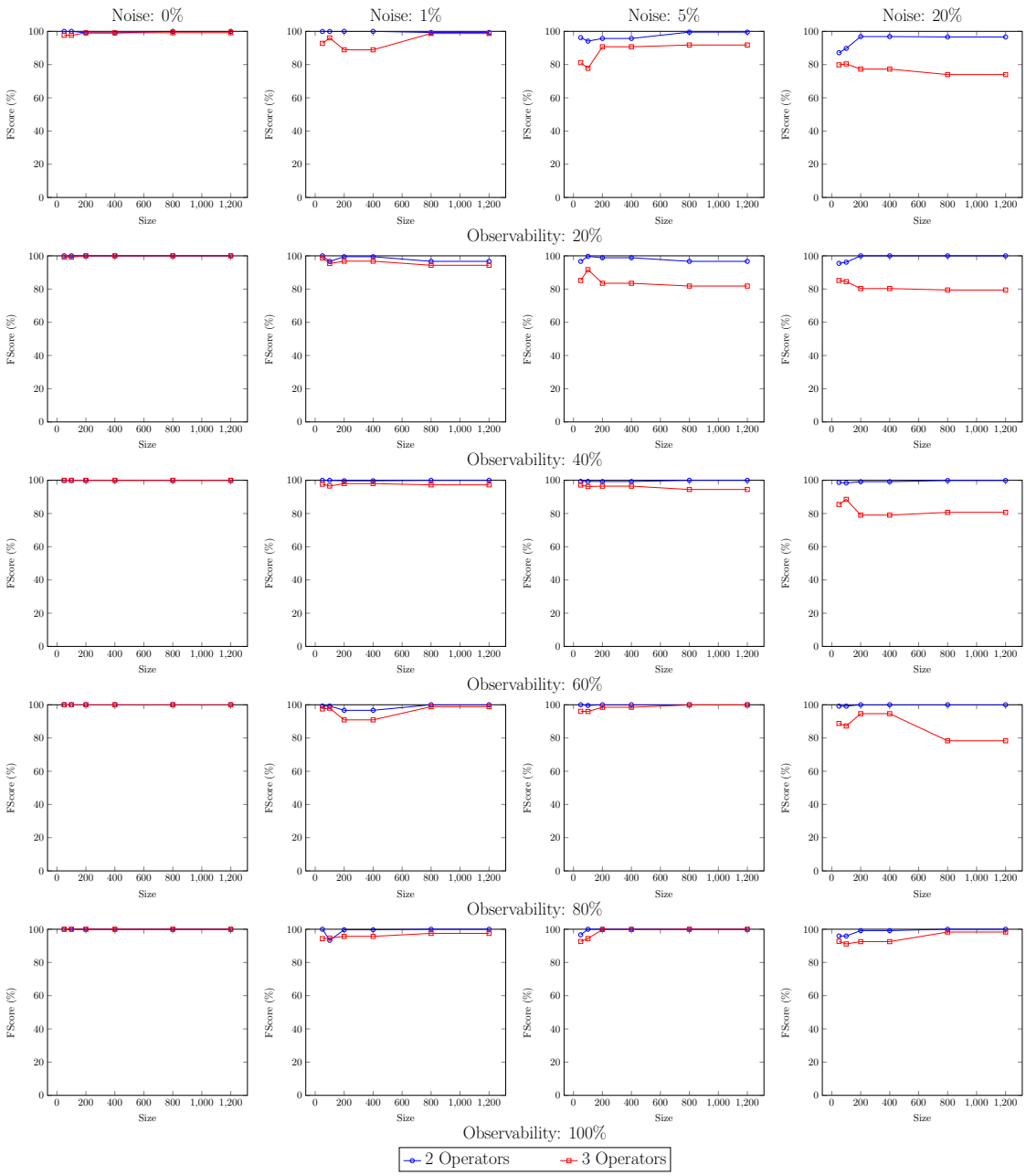


Figure B.66: Peg Solitaire – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of FScore.

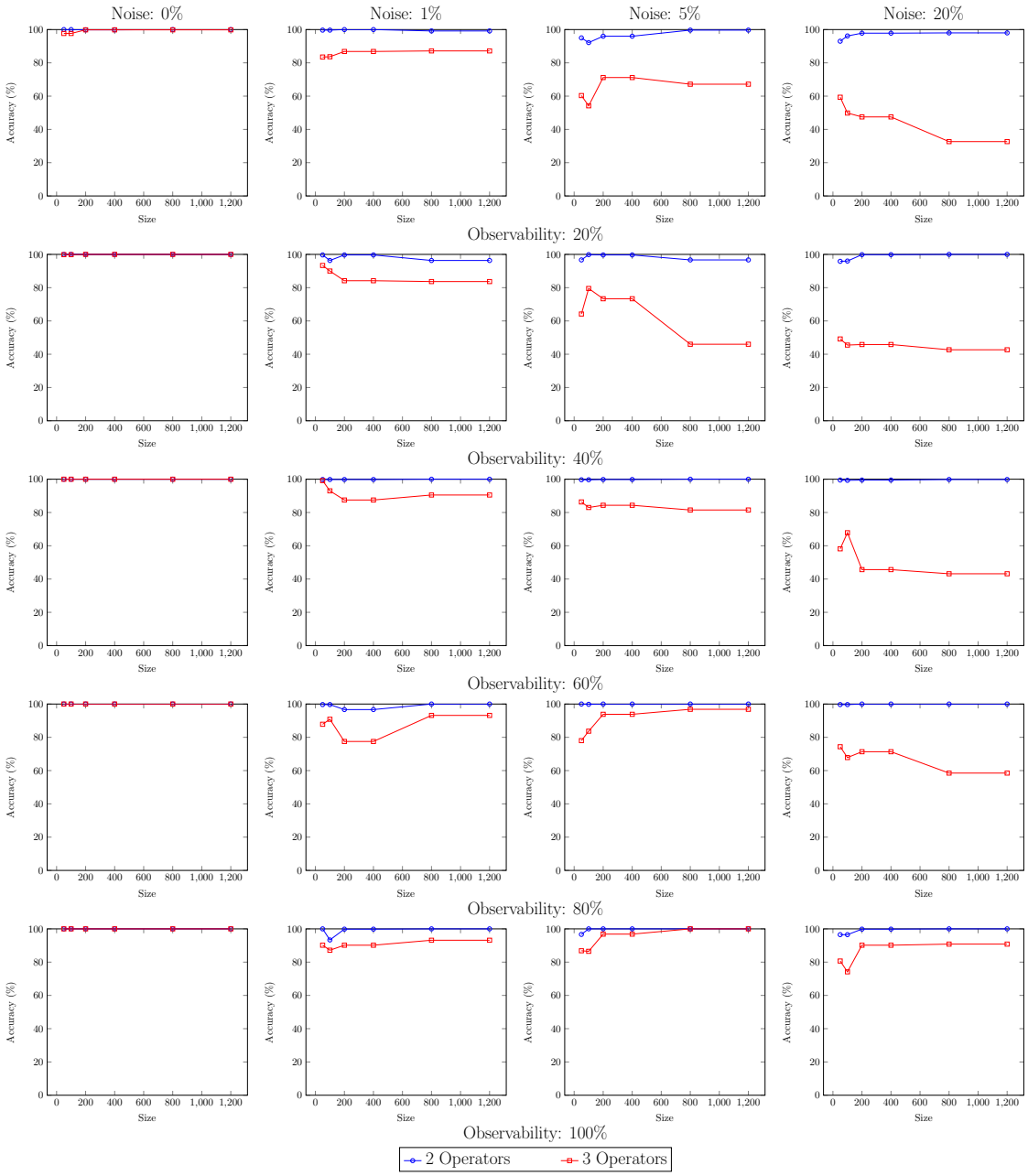


Figure B.67: Peg Solitaire – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

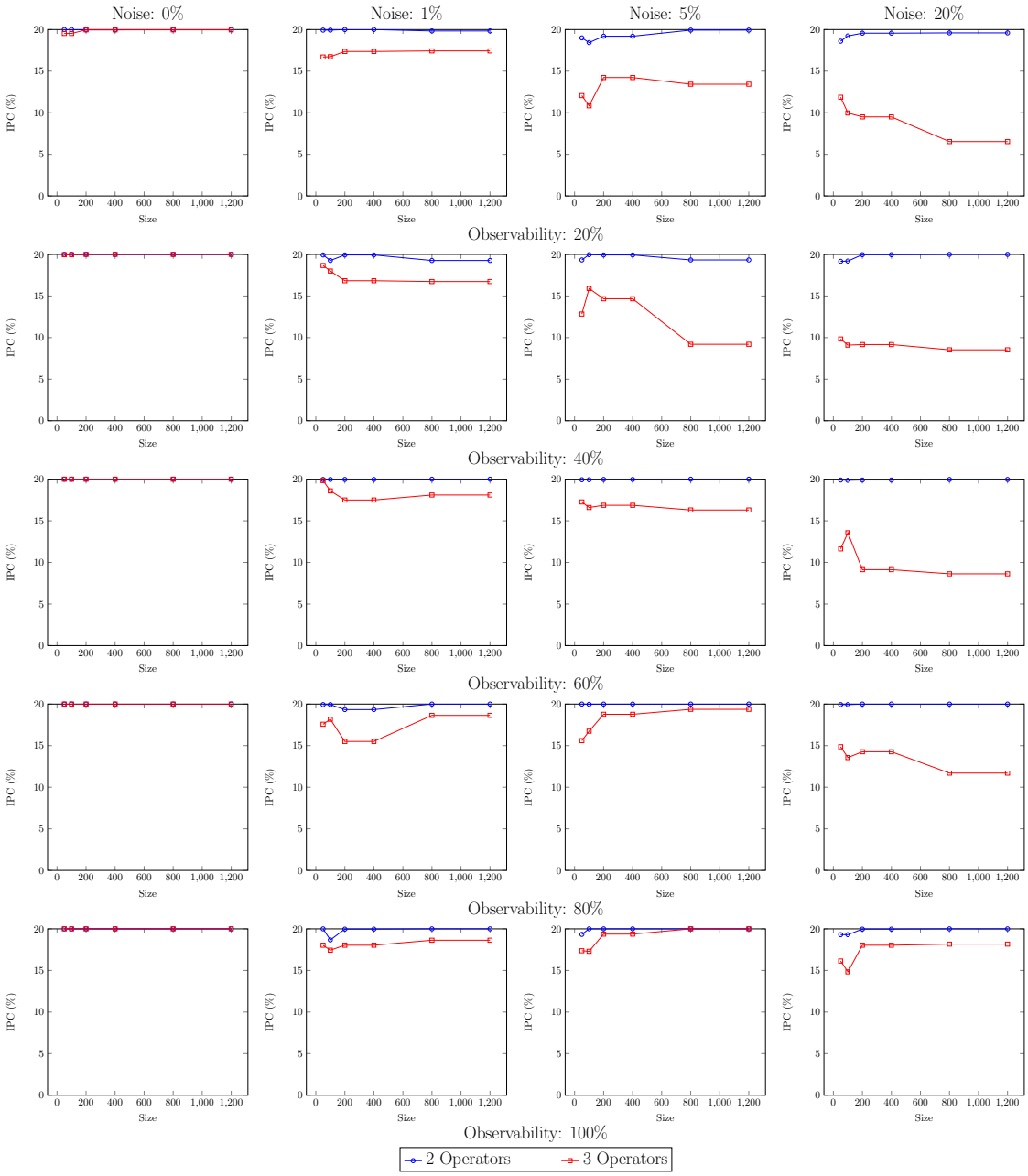


Figure B.68: Peg Solitaire – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of IPC.

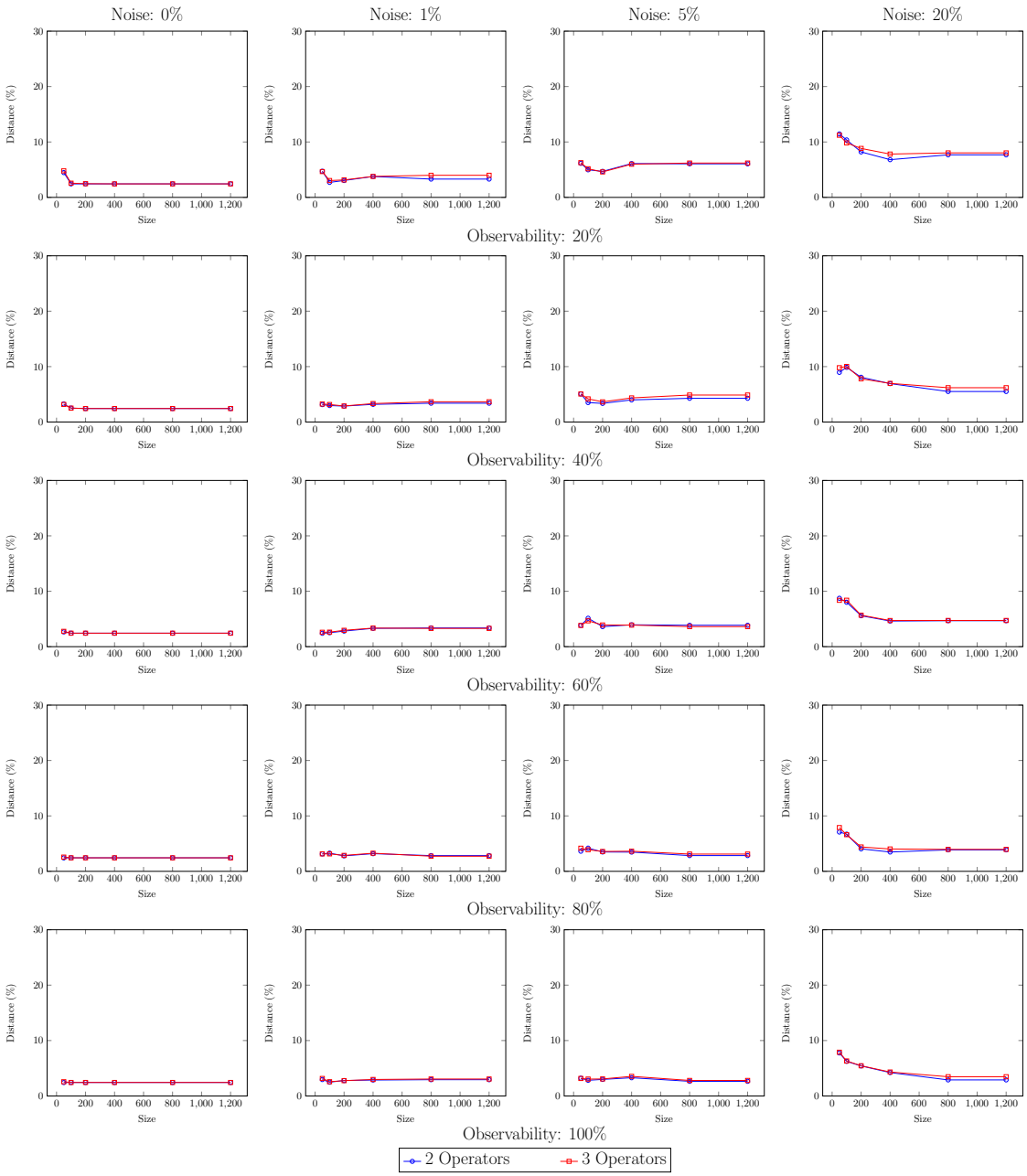


Figure B.69: Zenotravel – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of syntactical distance.

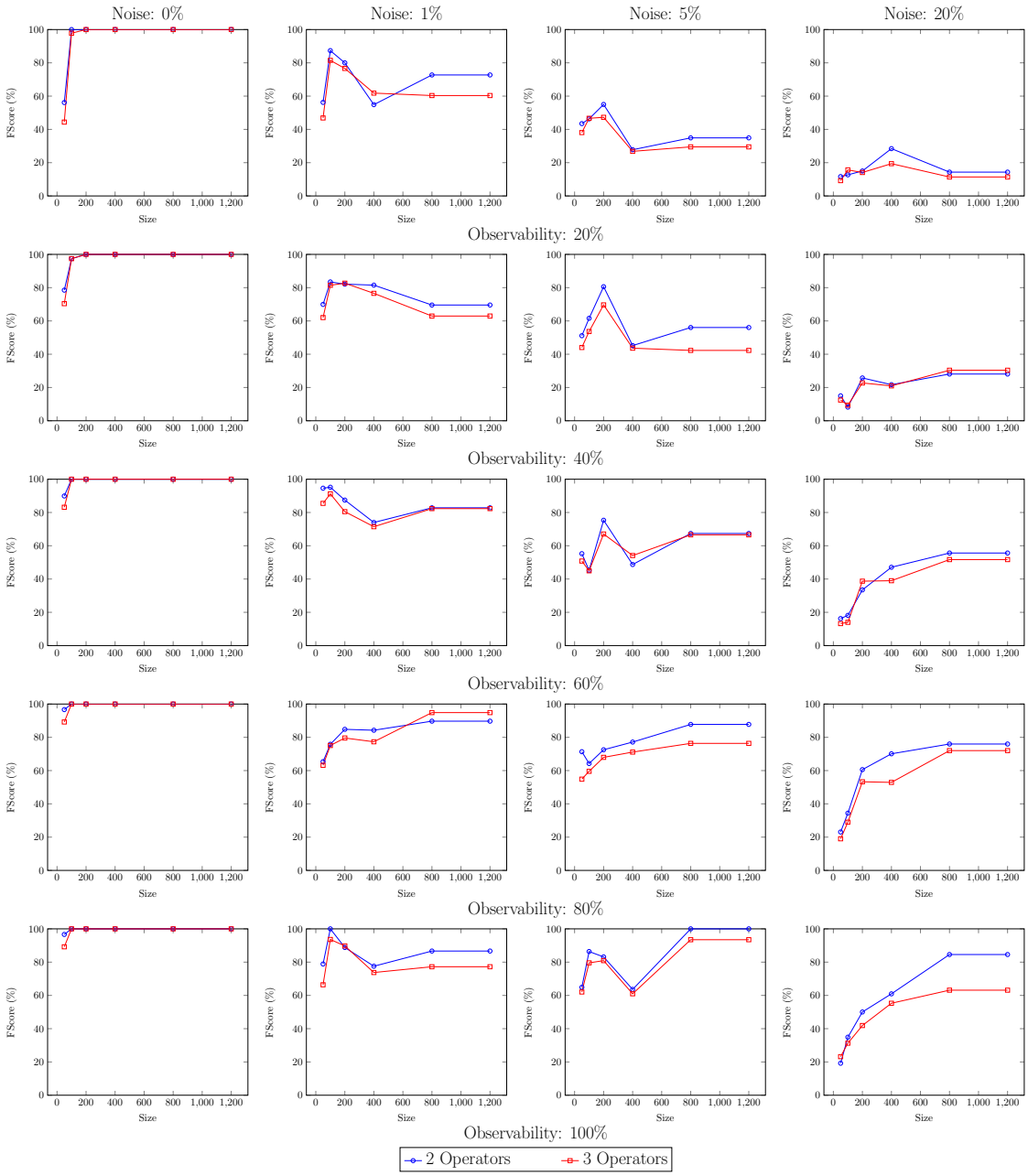


Figure B.70: Zenotravel – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of FScore.

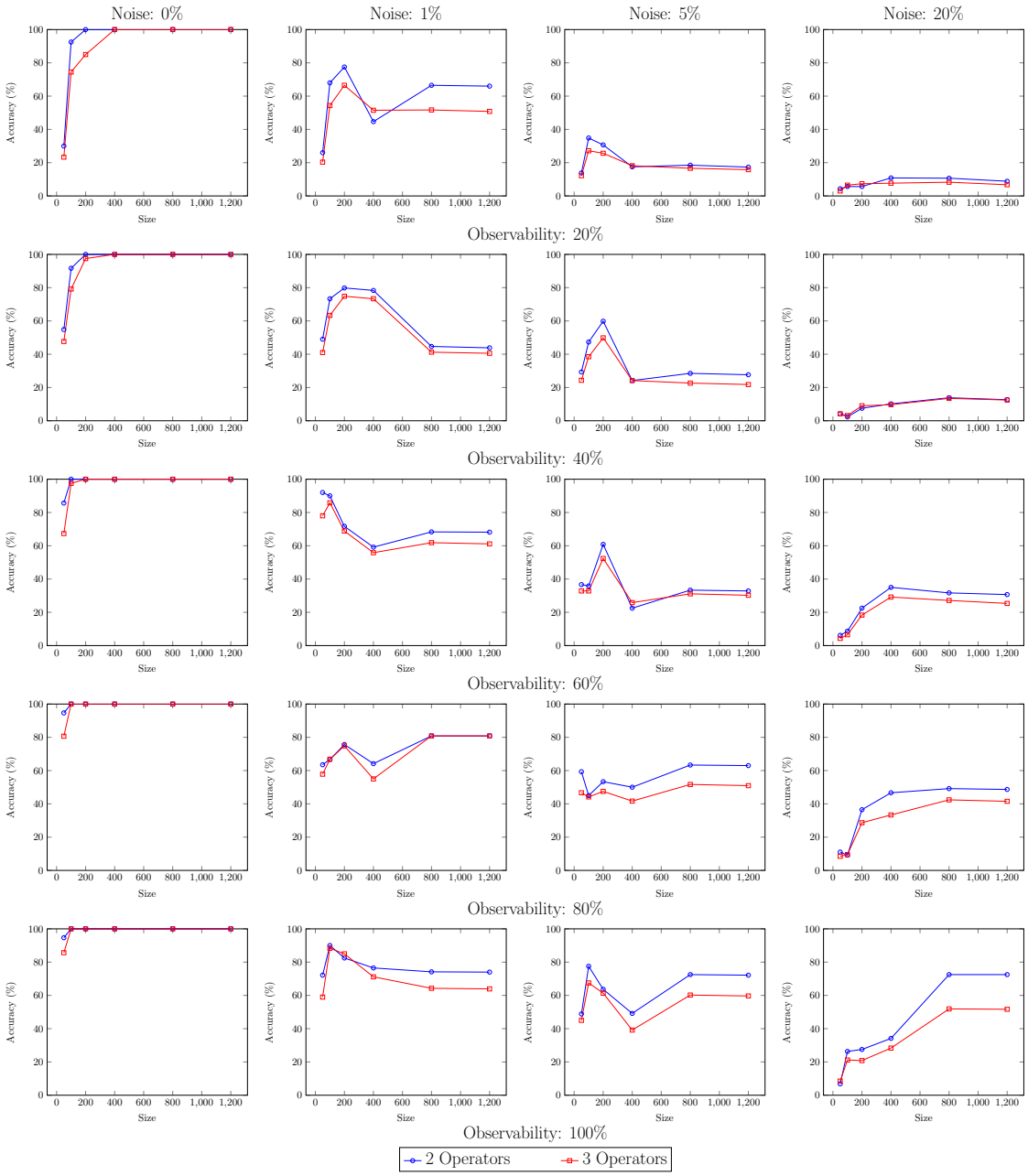


Figure B.71: Zenotravel – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

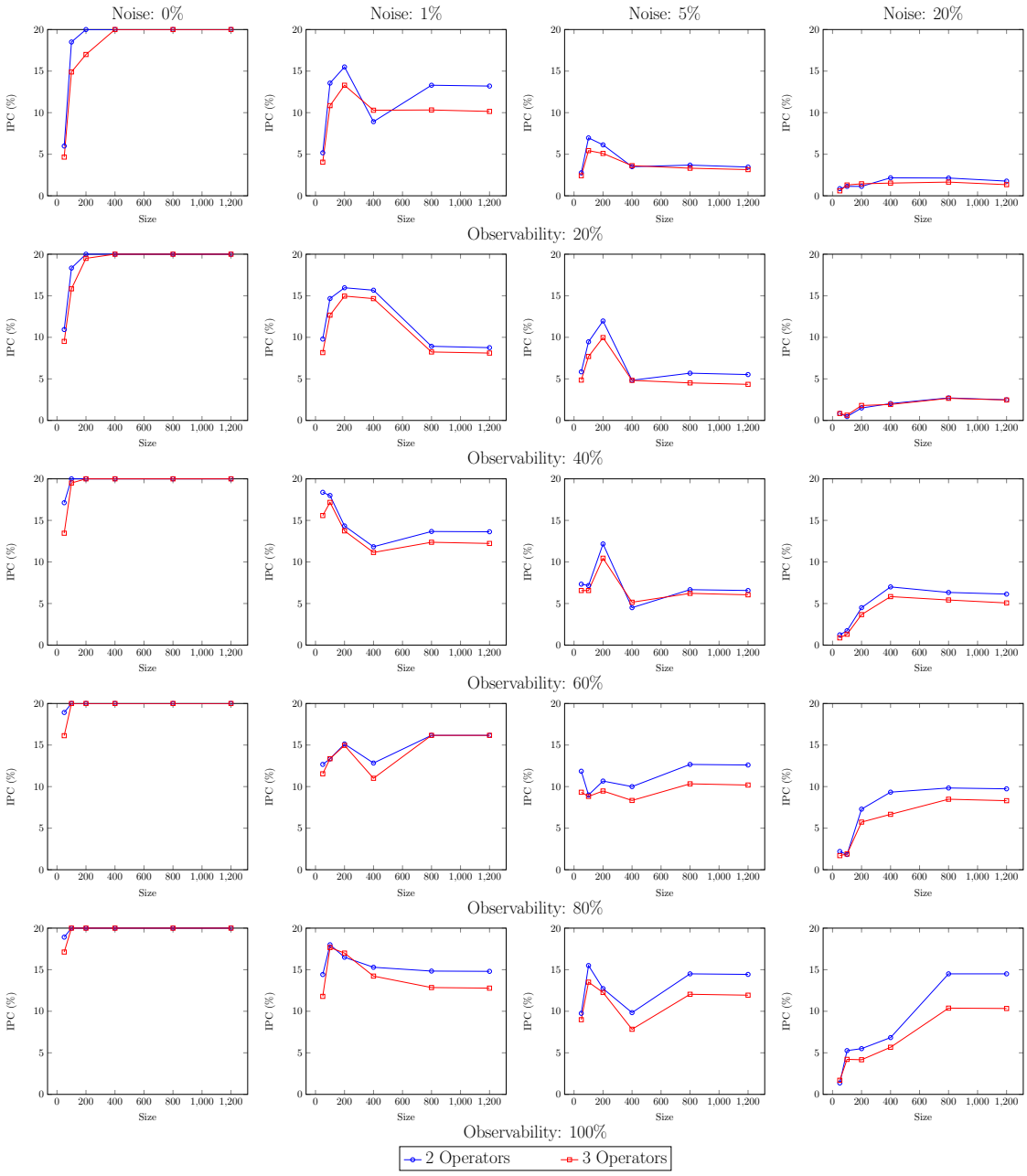


Figure B.72: Zenotravel – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of IPC.

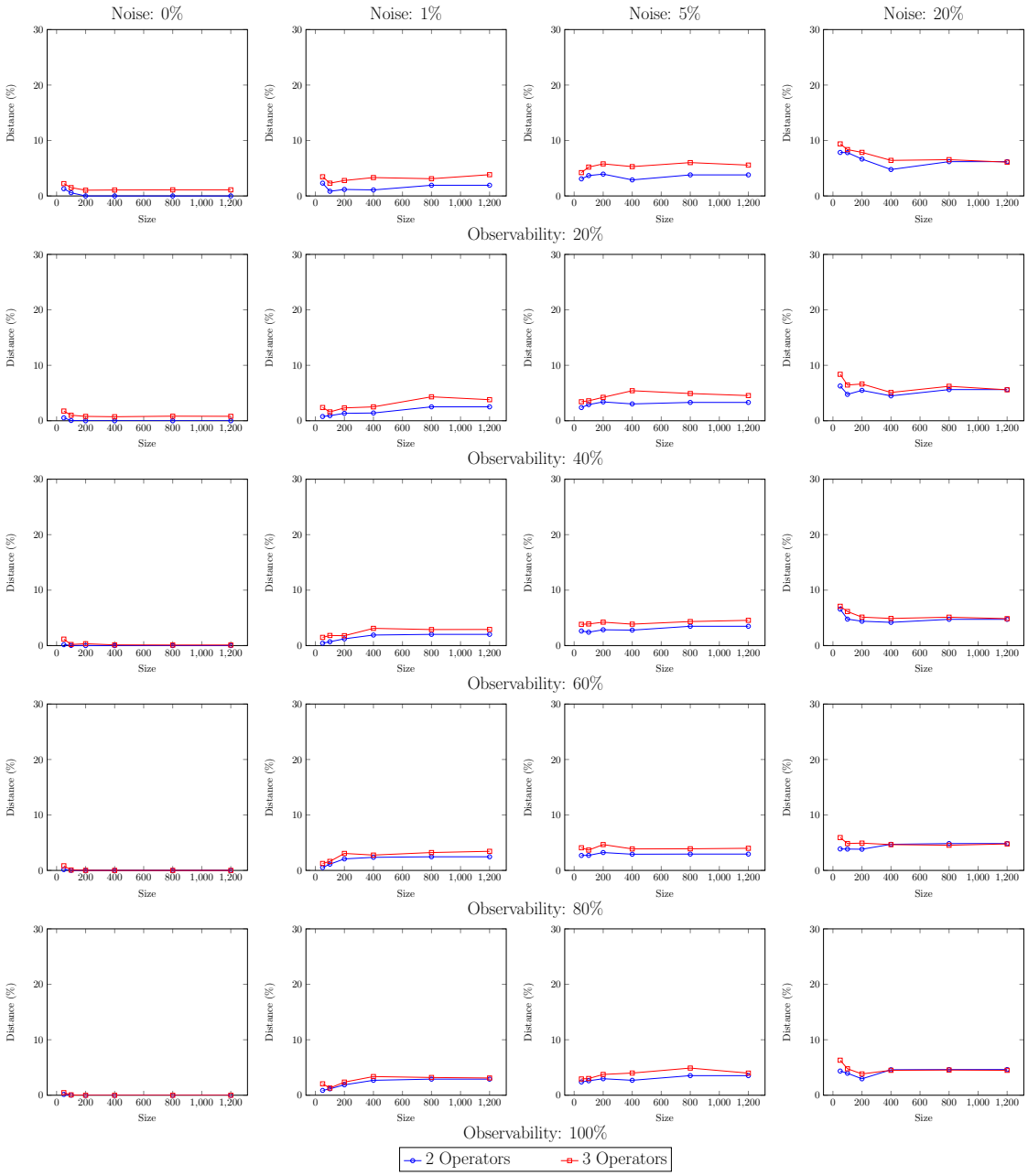


Figure B.73: Sokoban – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of syntactical distance.

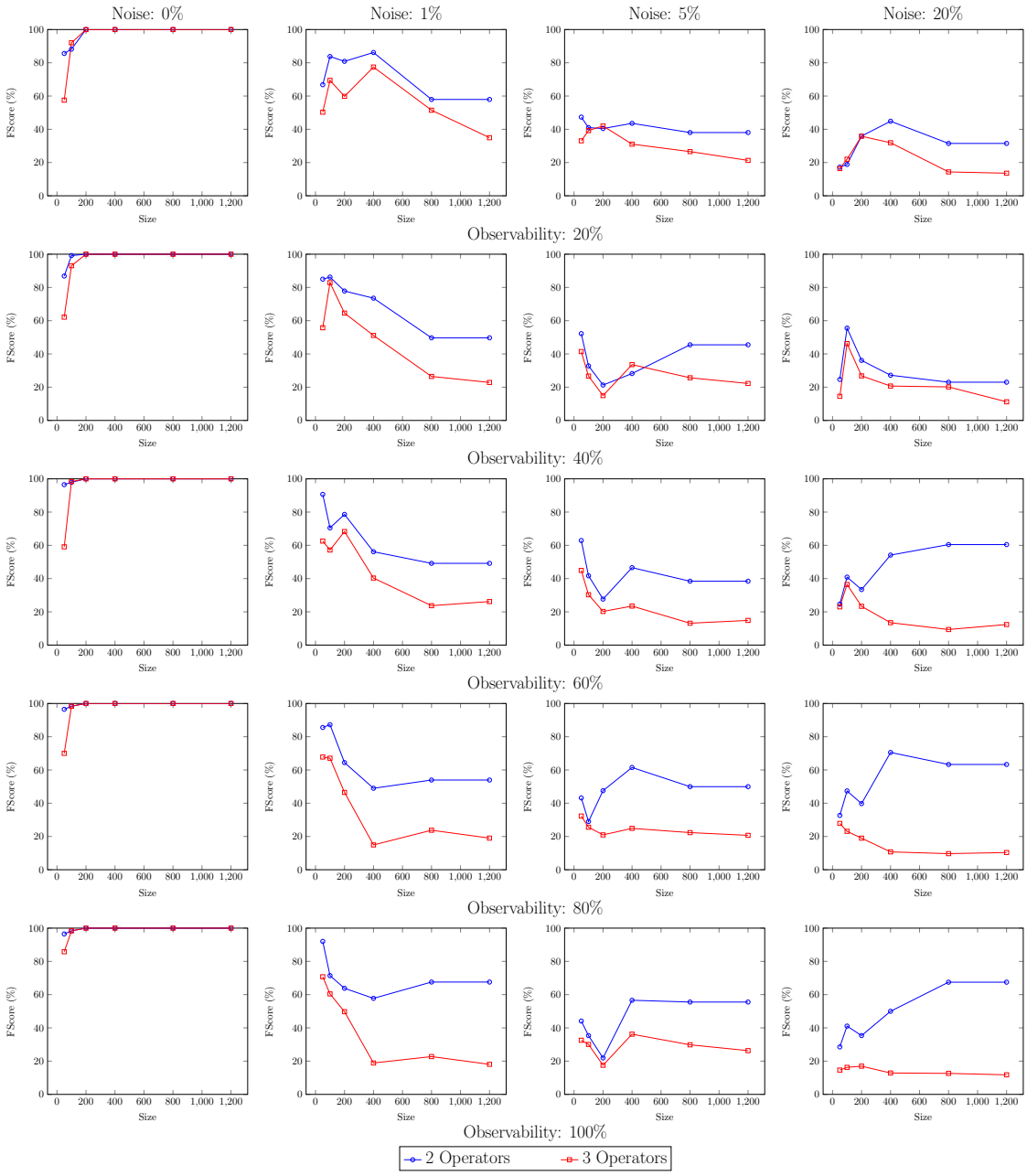


Figure B.74: Sokoban – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of FScore.

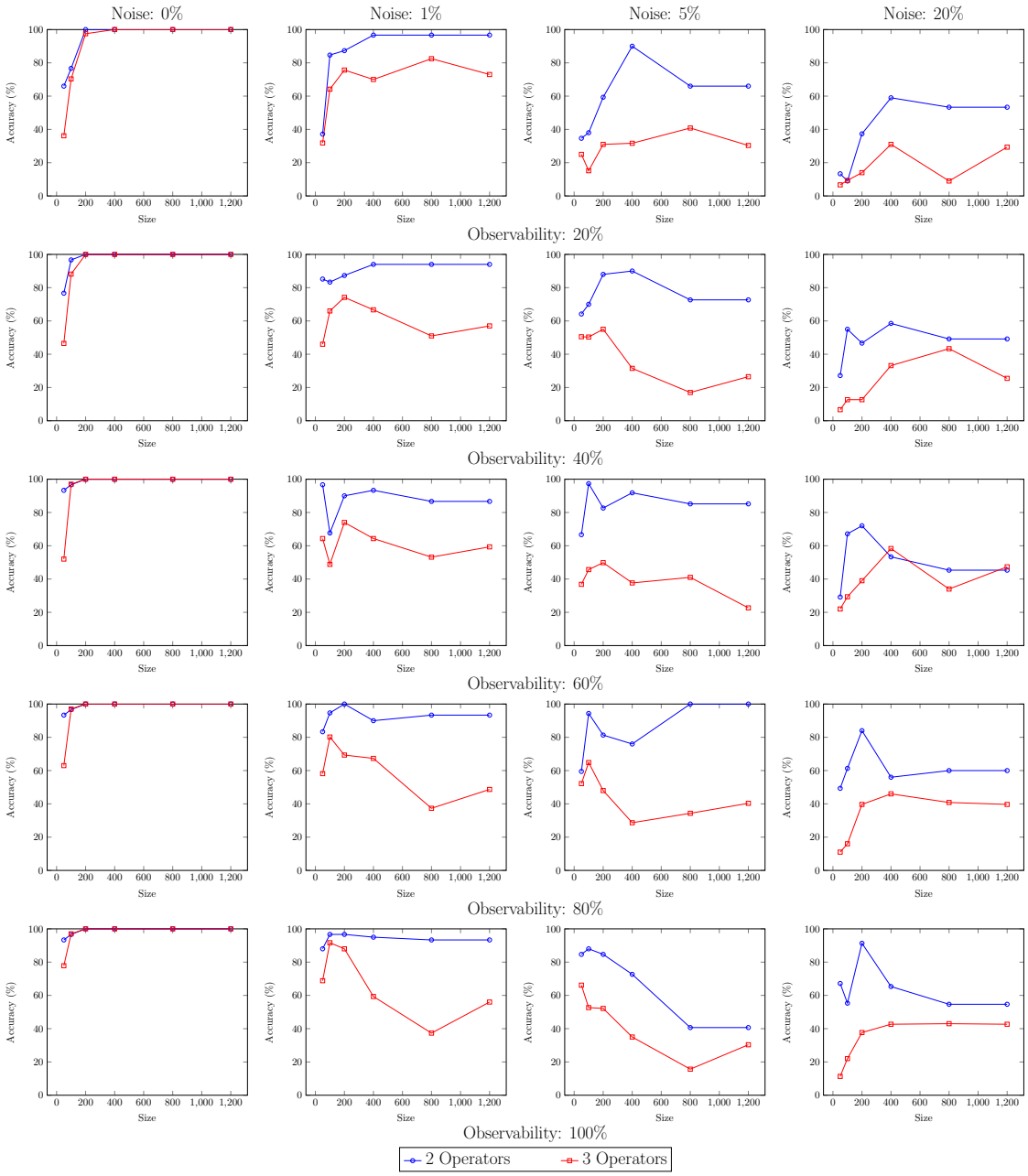


Figure B.75: Sokoban– Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

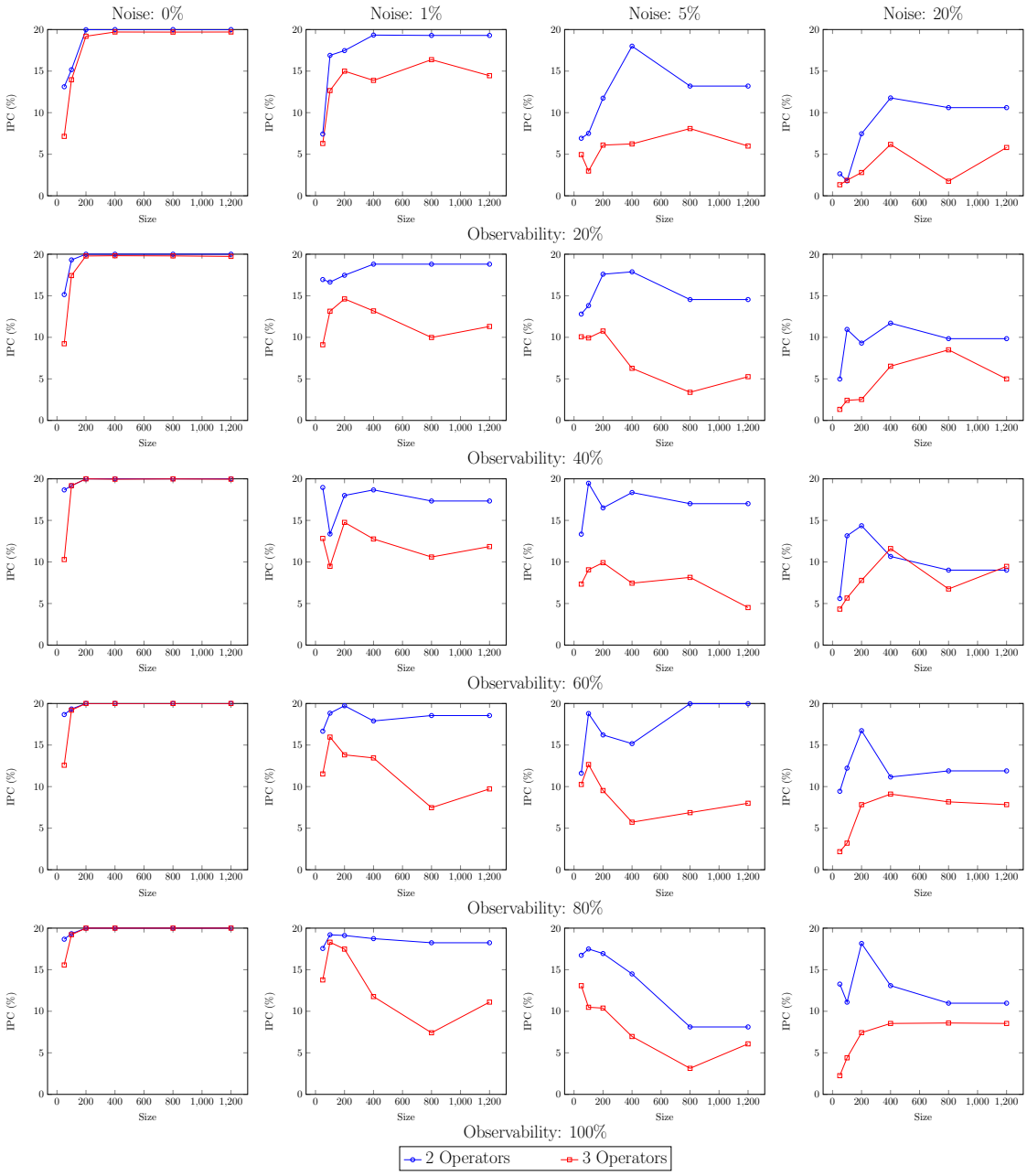


Figure B.76: Sokoban – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of IPC.

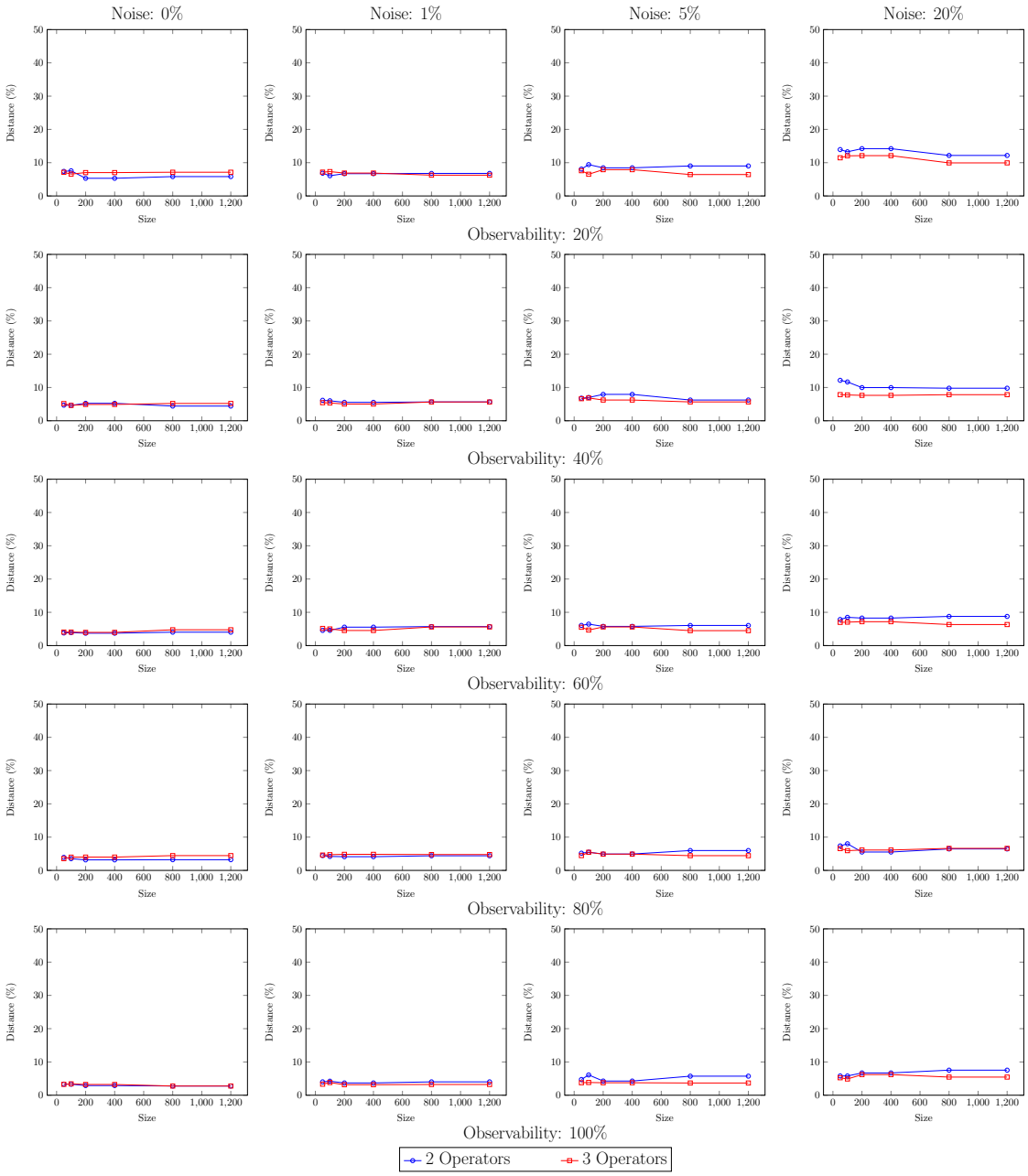


Figure B.77: Match – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of syntactical distance.

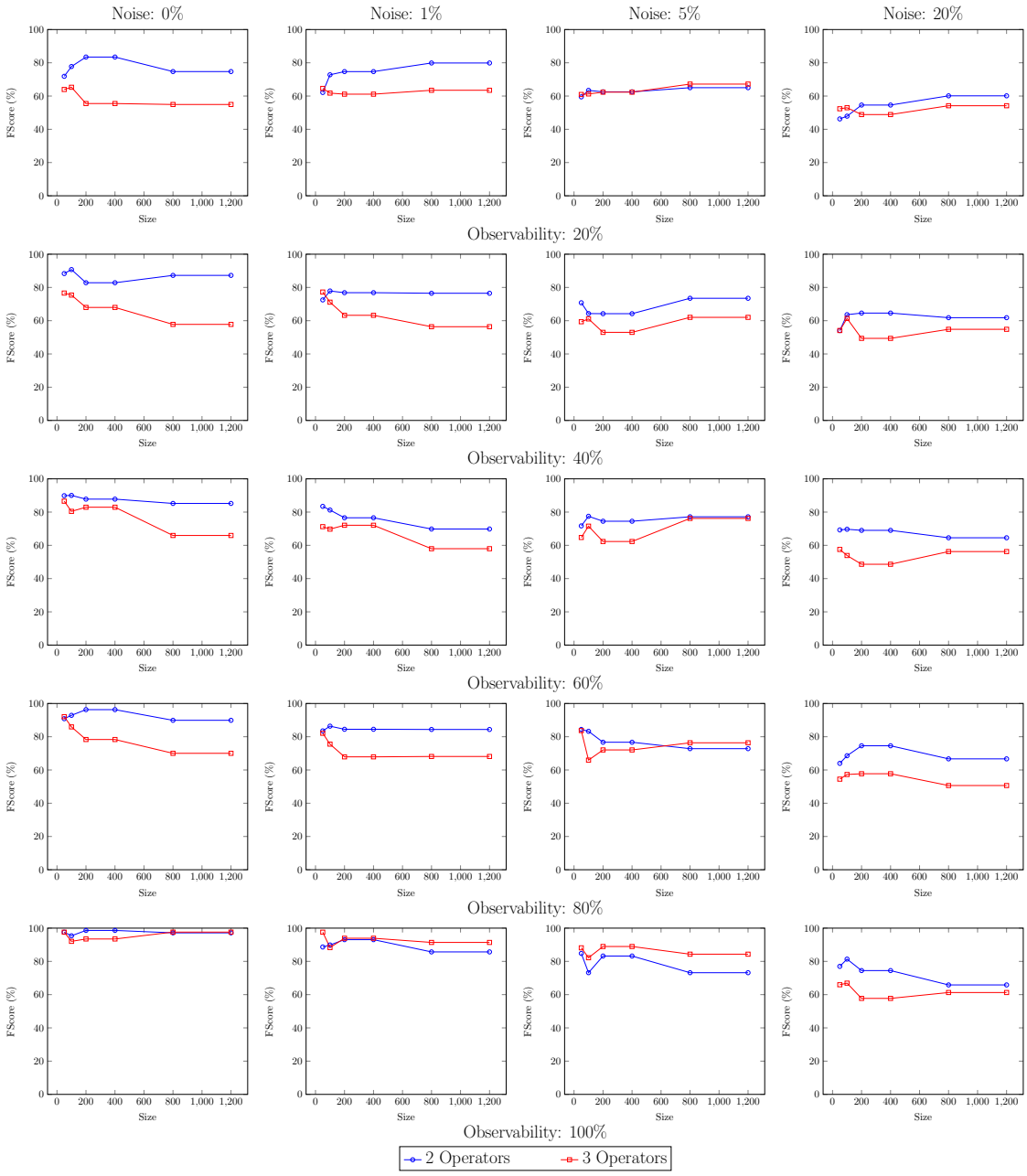


Figure B.78: Match– Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of FScore.

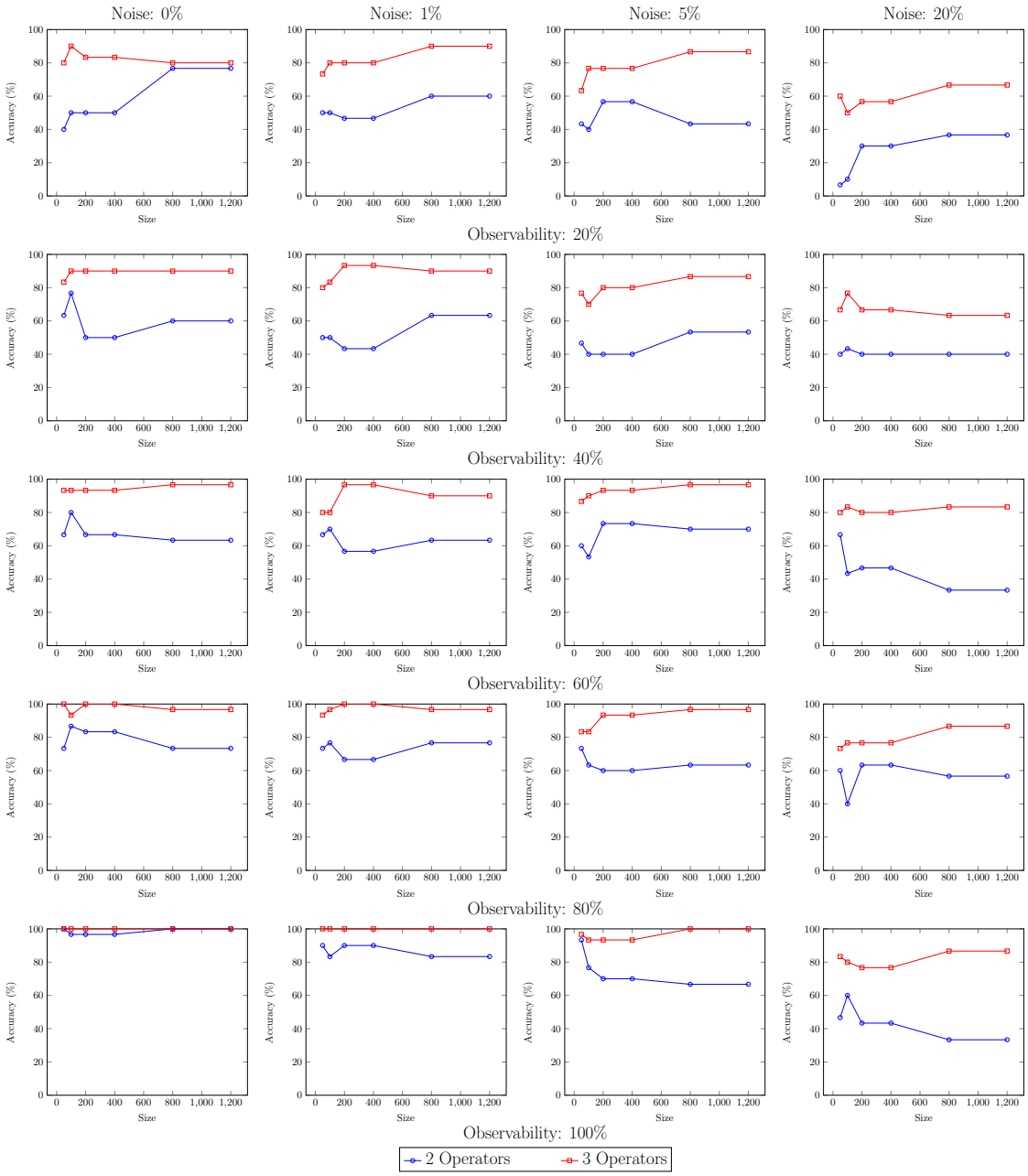


Figure B.79: Match – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

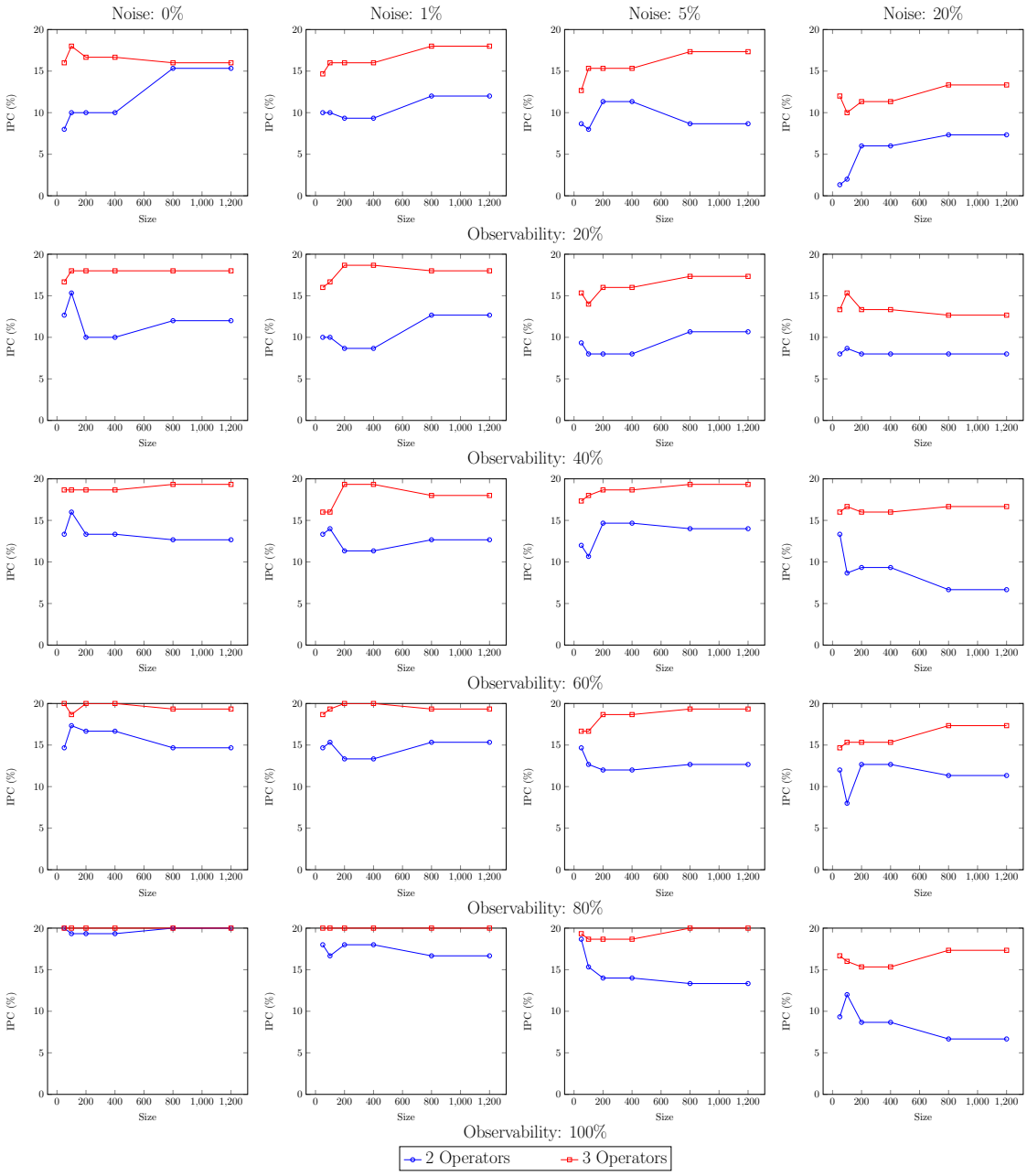


Figure B.80: Match – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of IPC.

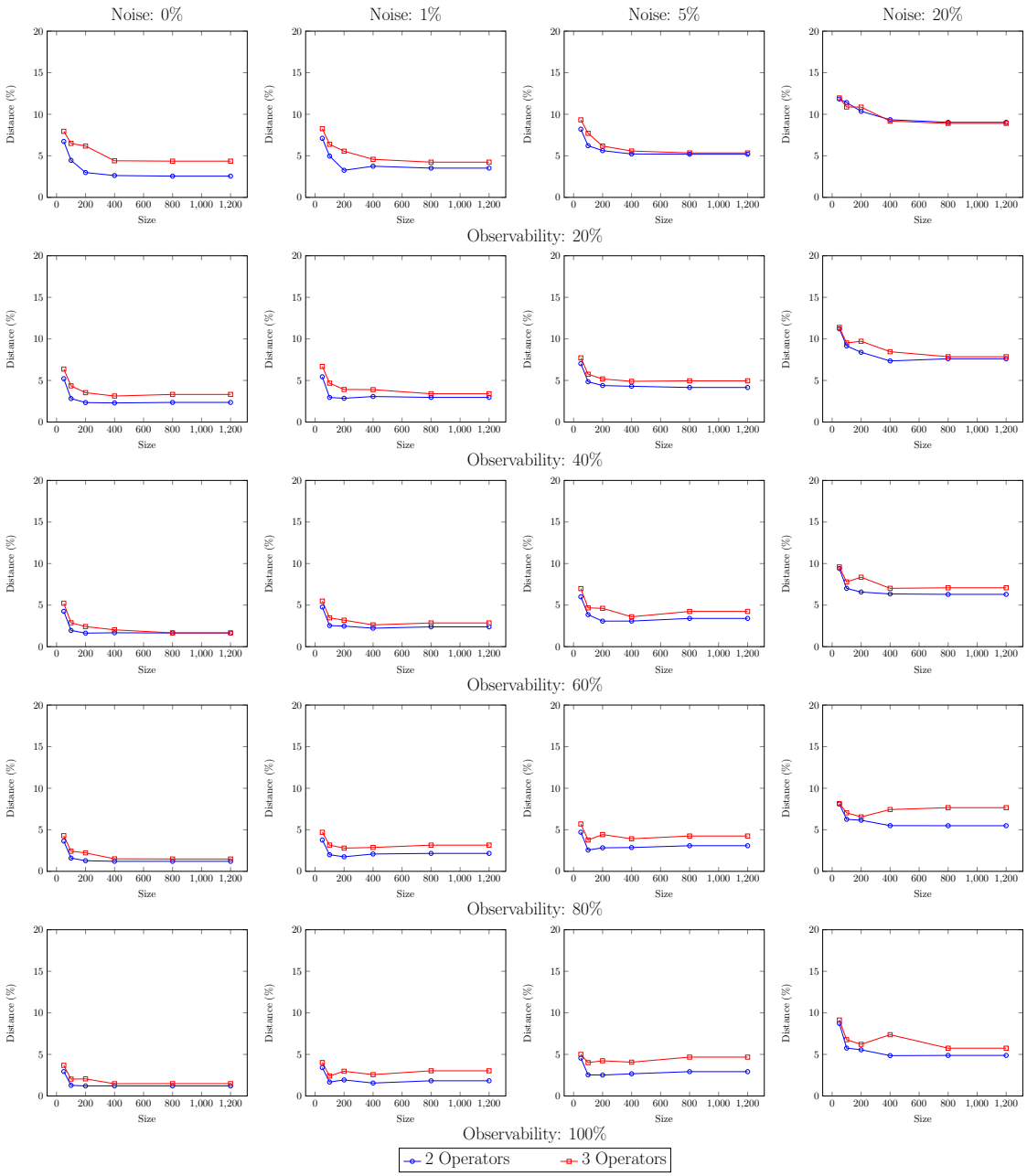


Figure B.81: Turn and Open – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of syntactical distance.

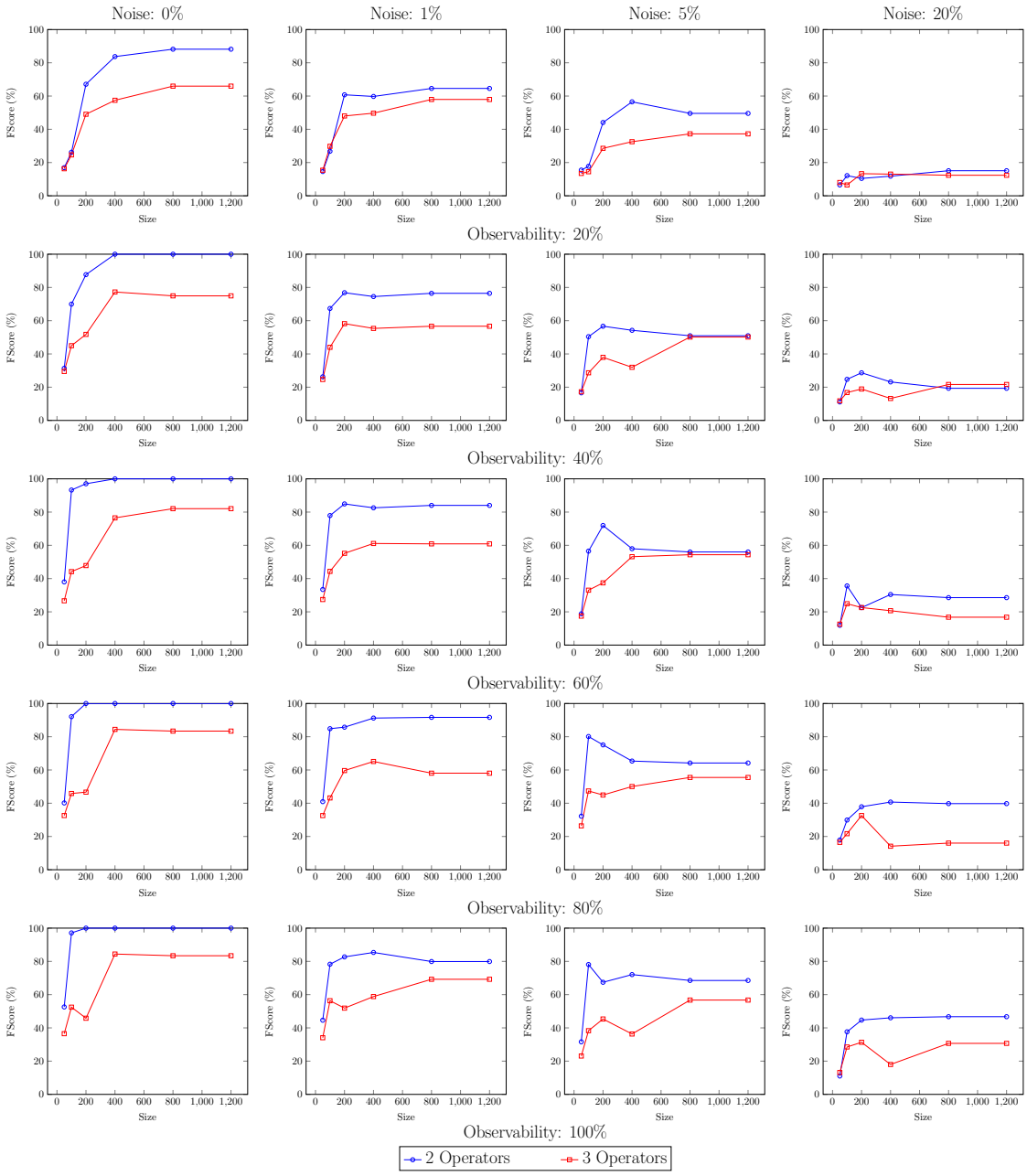


Figure B.82: Turn and Open – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of FScore.

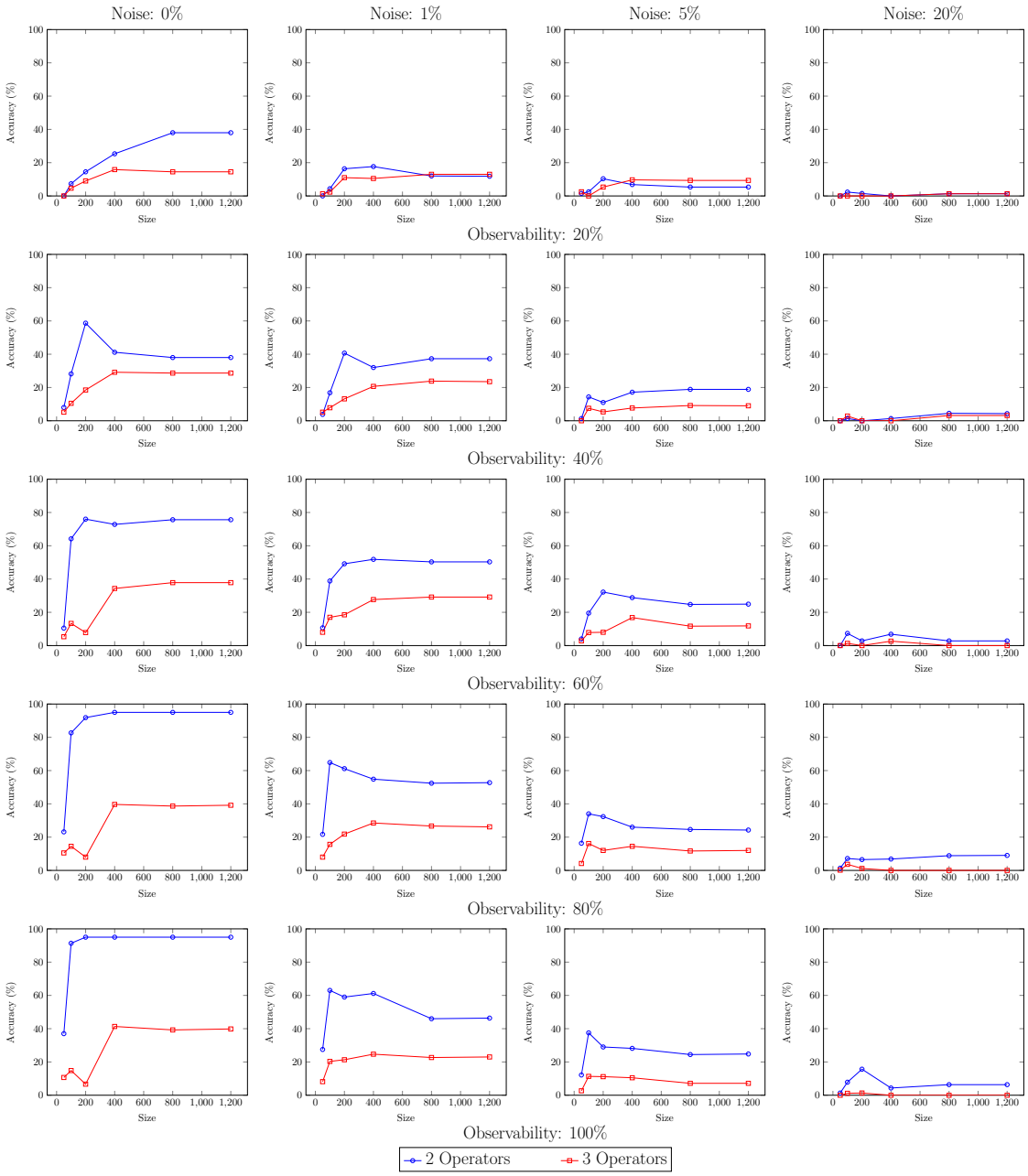


Figure B.83: Turn and Open – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

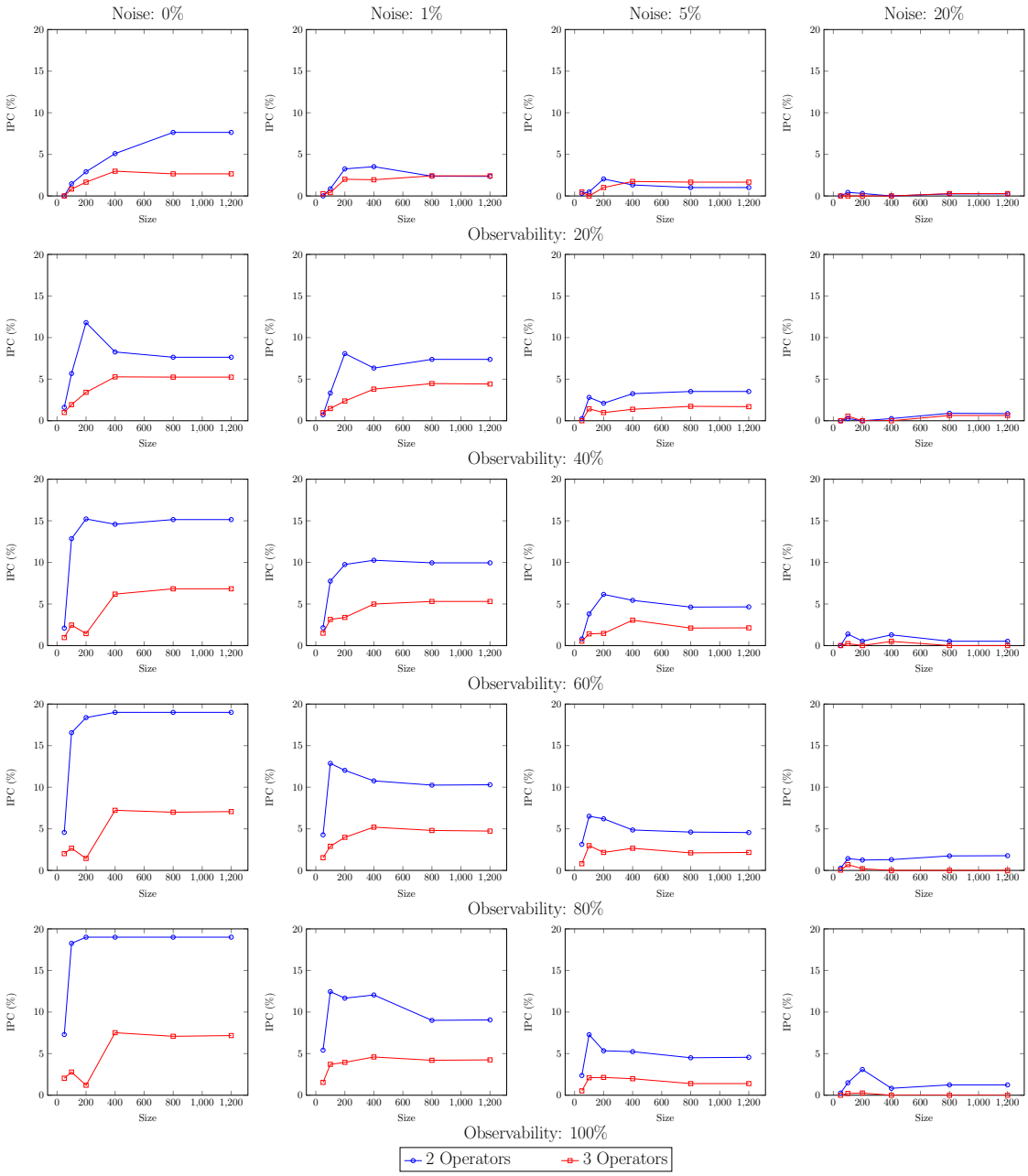


Figure B.84: Turn and Open – Average performance of TempAMLSI when the training data set size increases in number of tasks in terms of IPC.

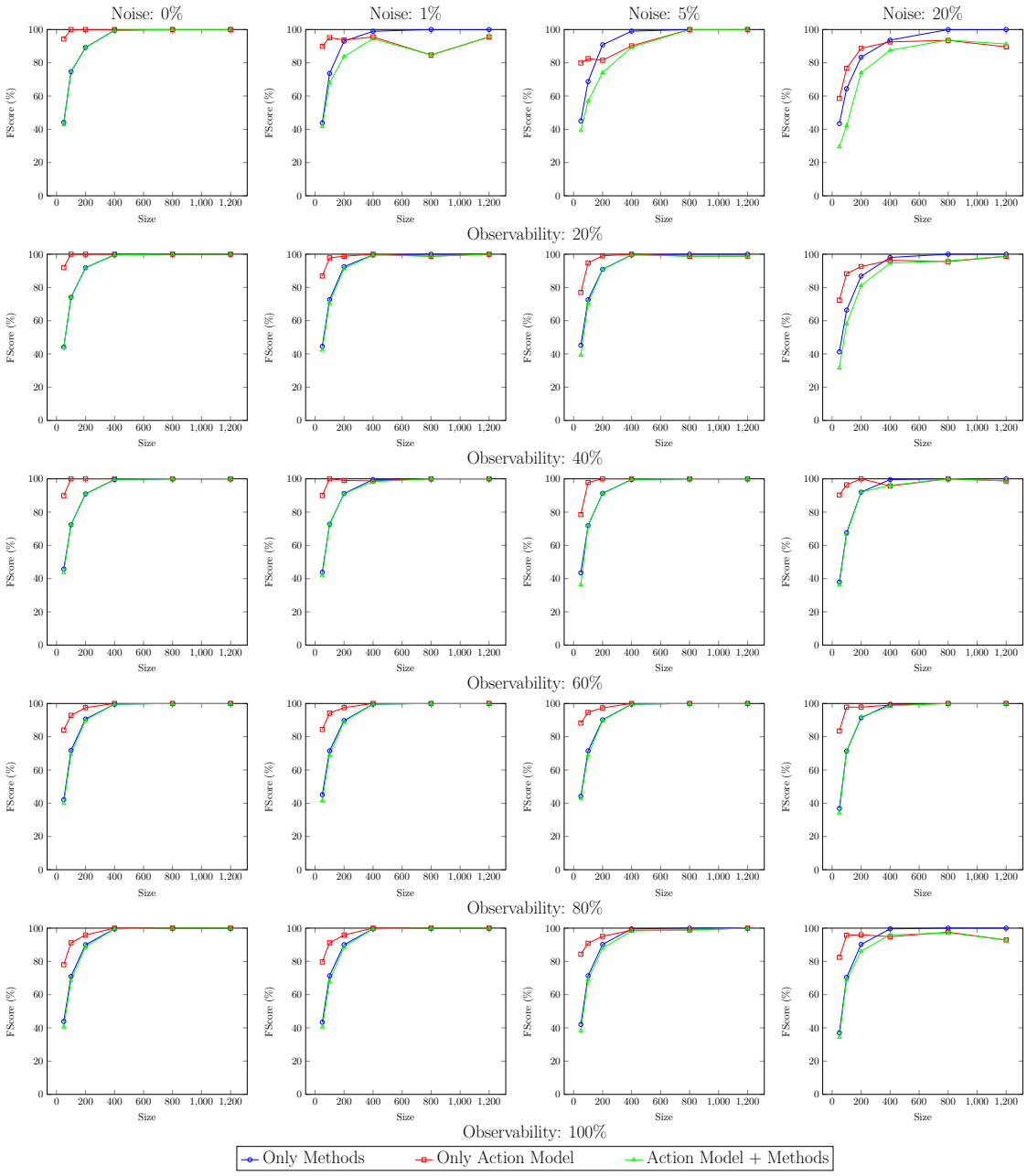


Figure B.85: Blocksworld – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of FScore.

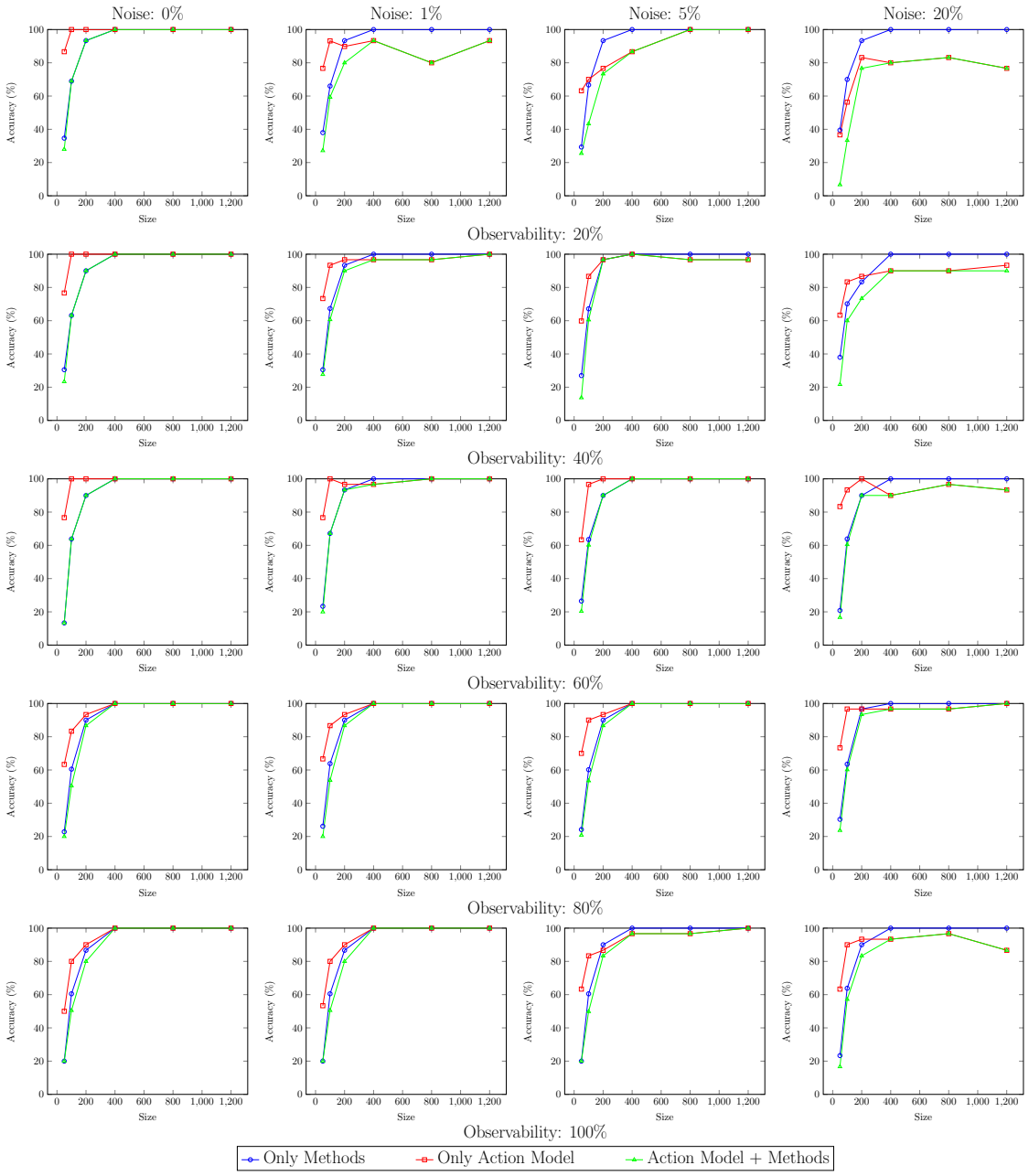


Figure B.86: Blocksworld – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

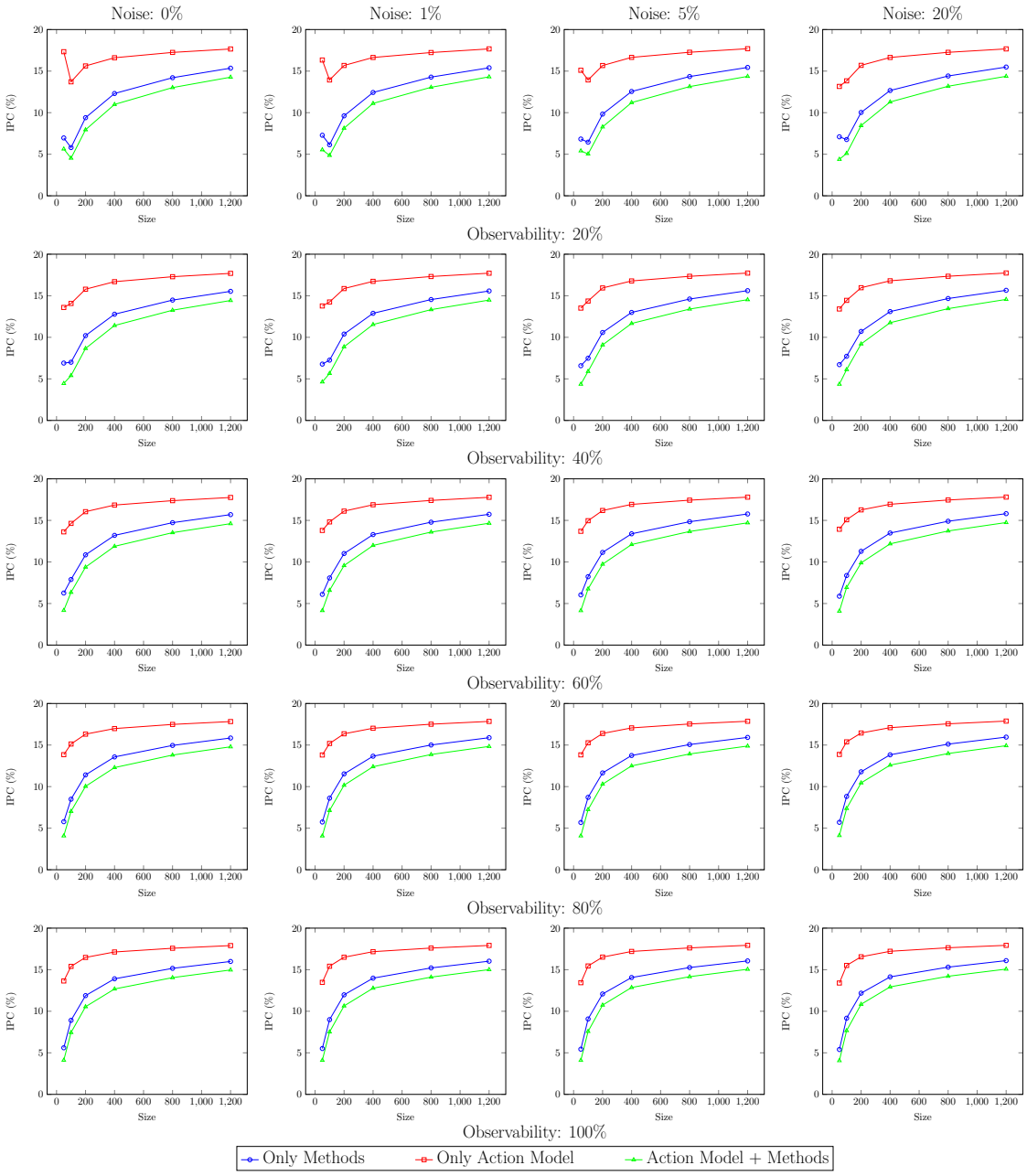


Figure B.87: Blocksworld – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of IPC.

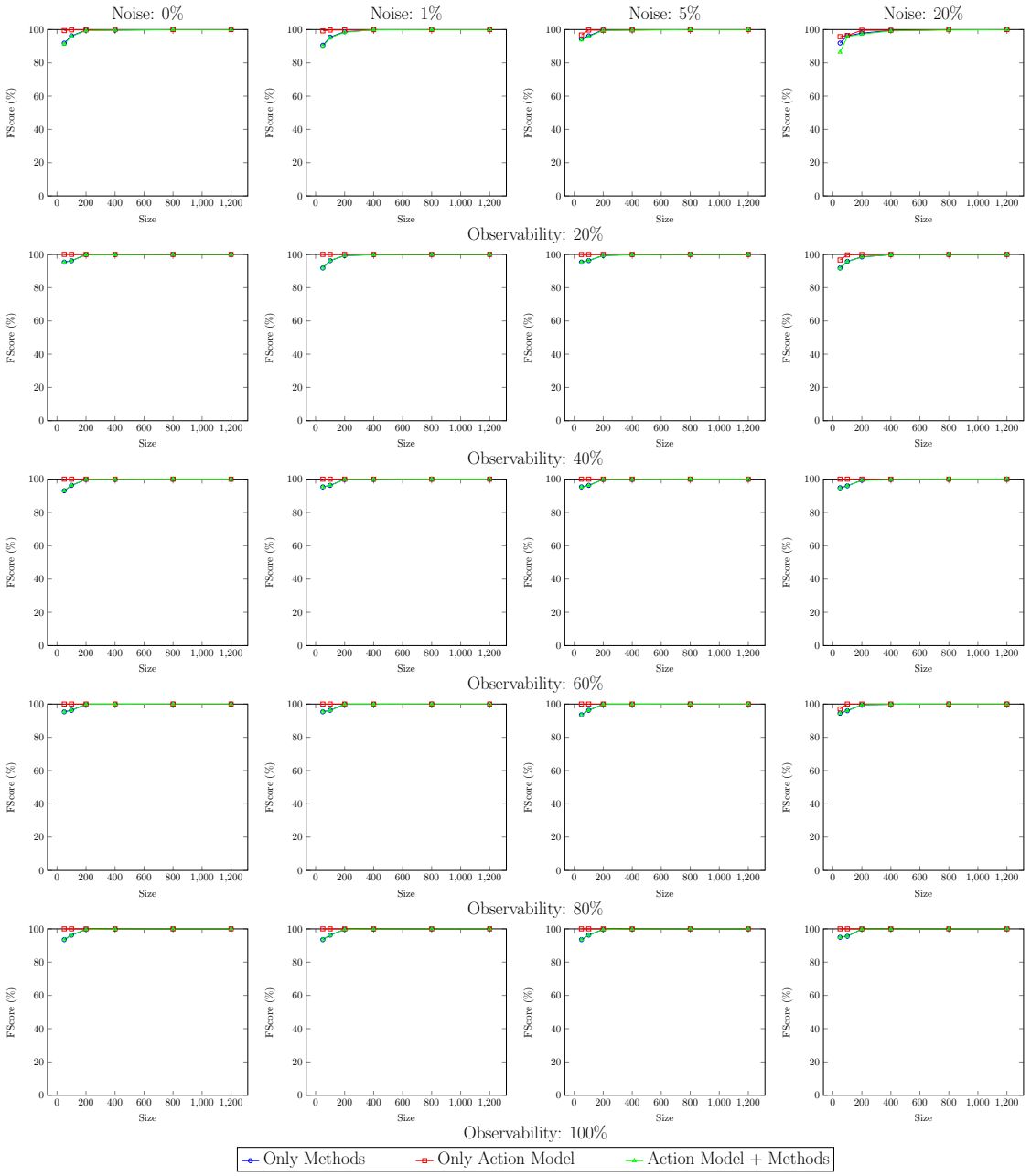


Figure B.88: Gripper – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of FScore.

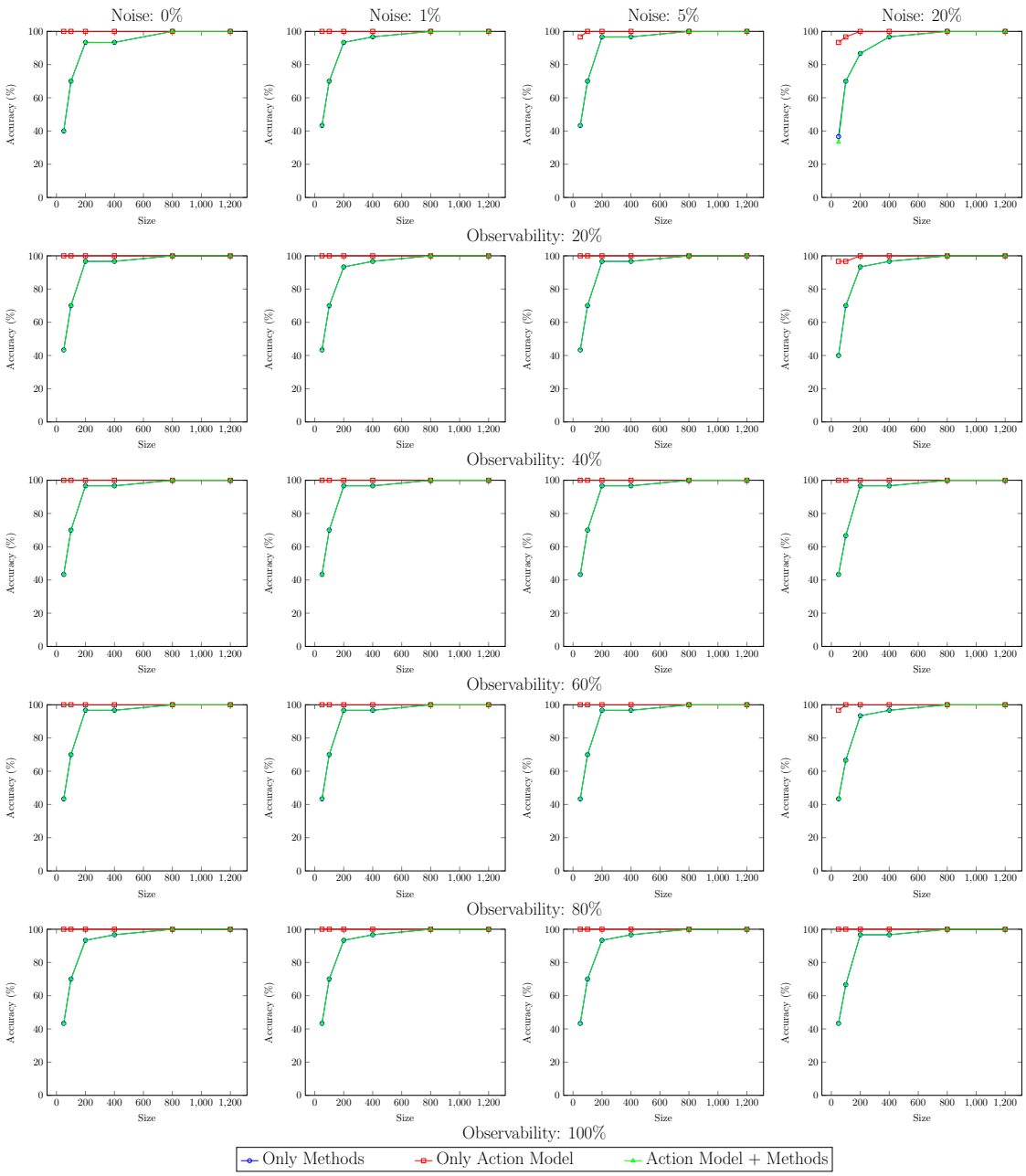


Figure B.89: Gripper – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

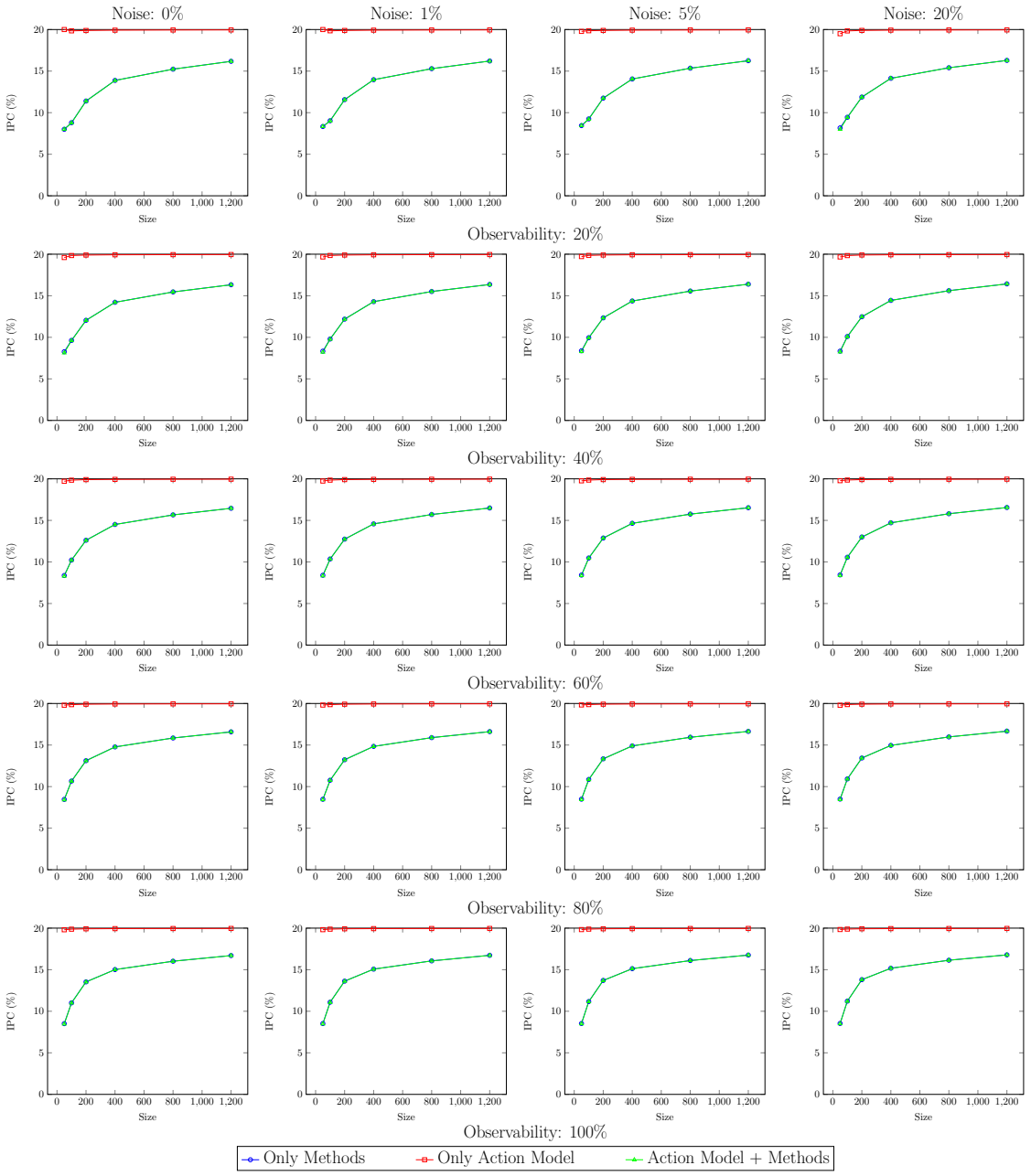


Figure B.90: Gripper – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of IPC.

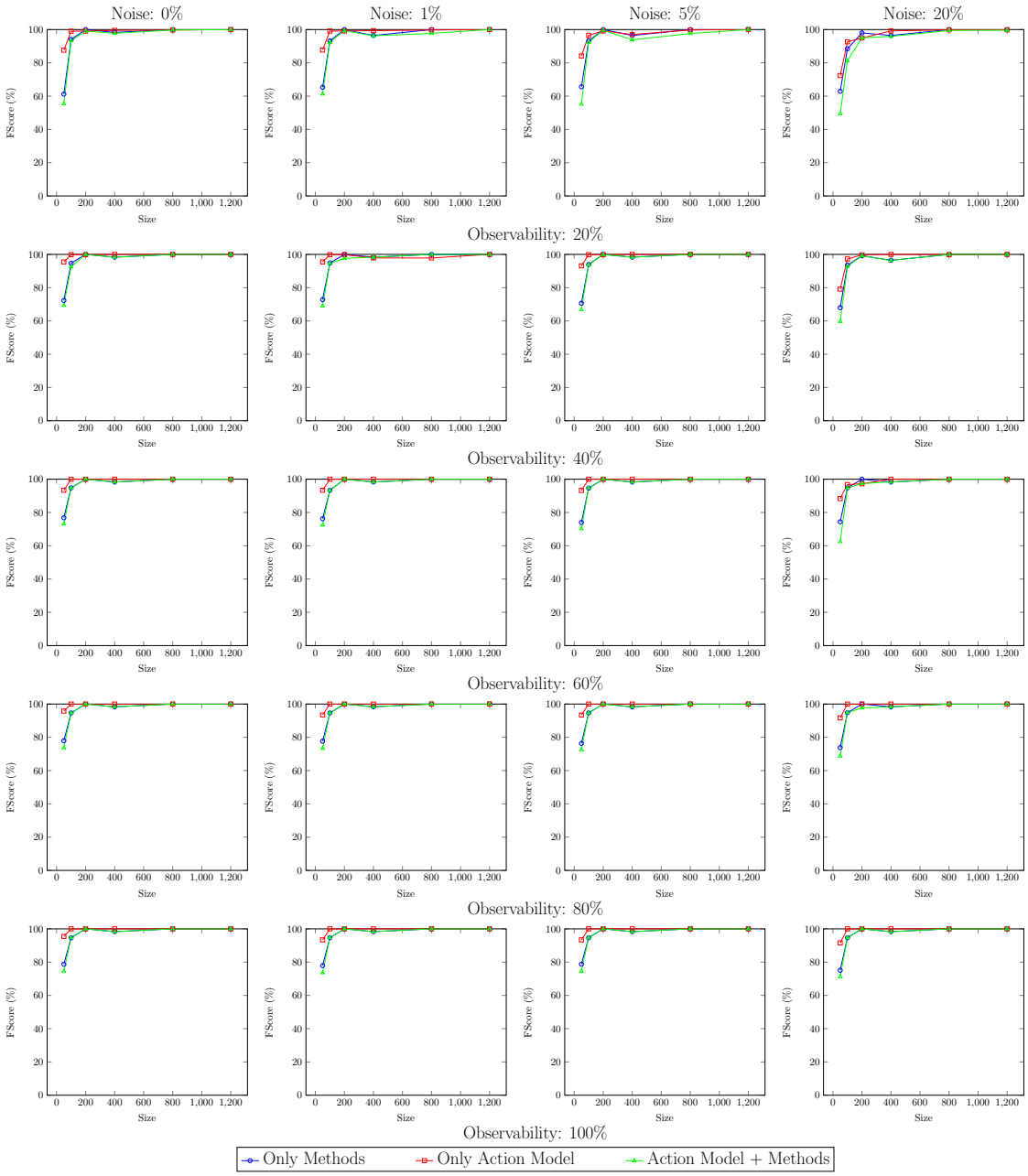


Figure B.91: Zenotravel – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of FScore.

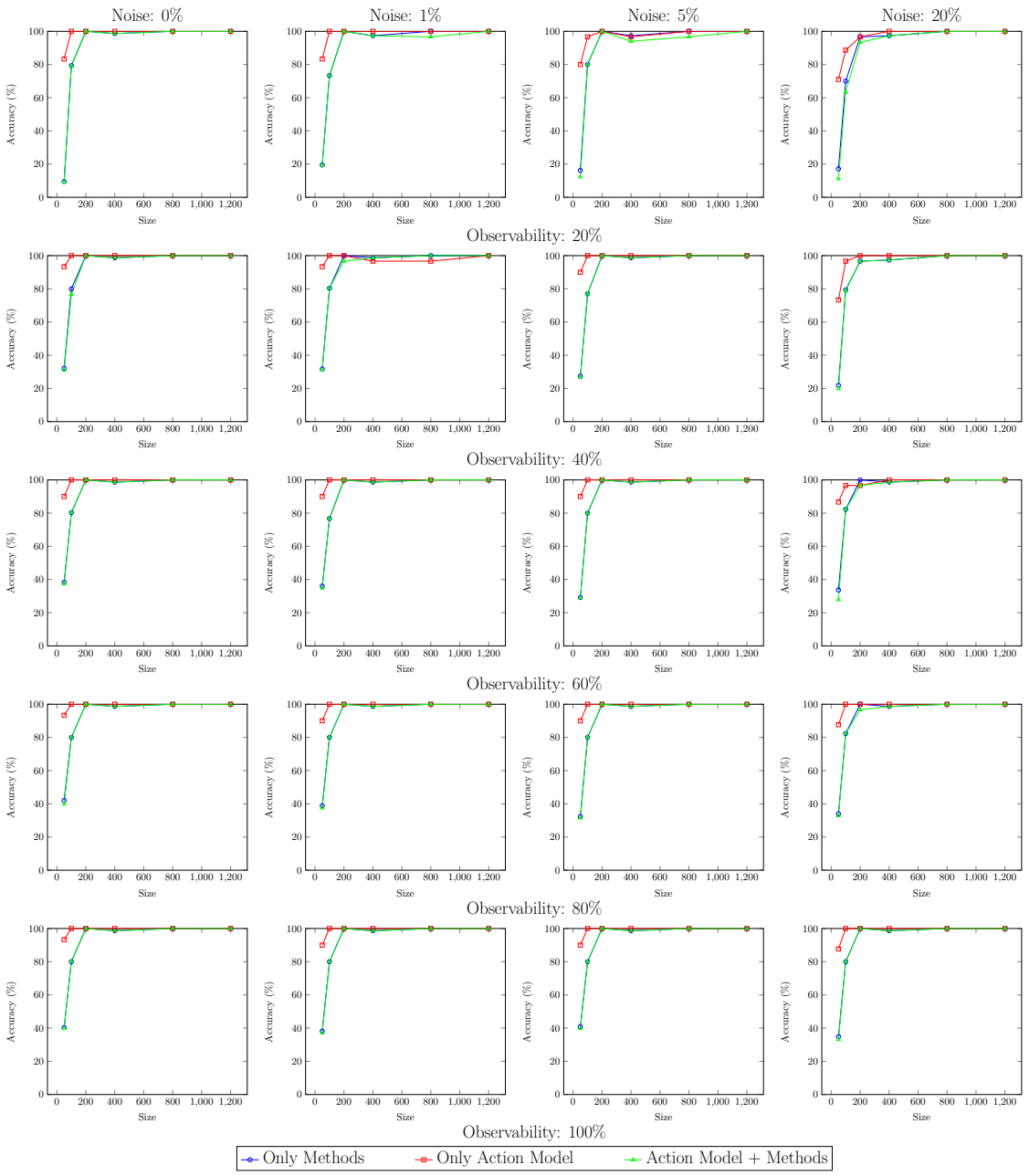


Figure B.92: Zenotravel – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

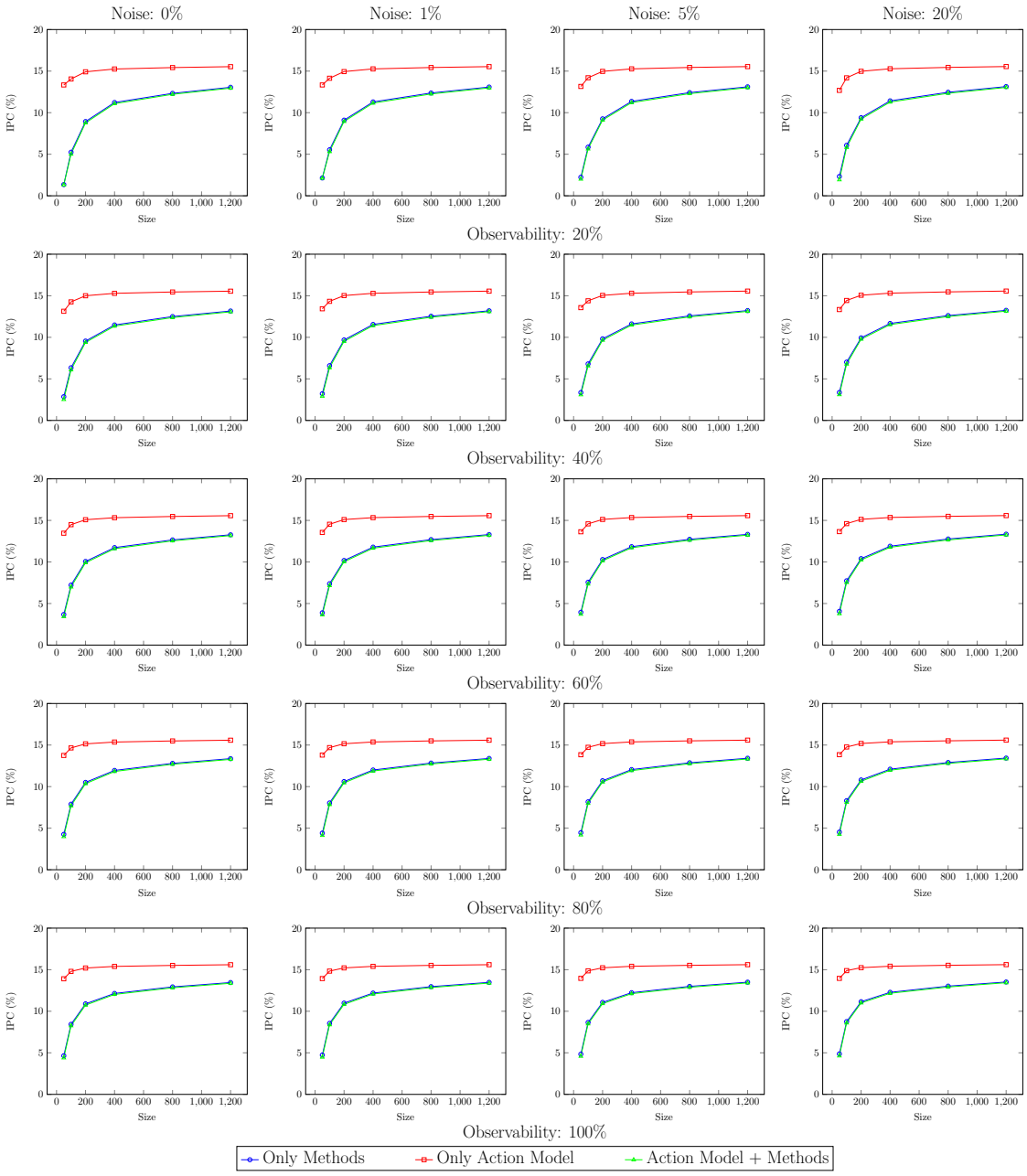


Figure B.93: Zenotravel – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of IPC.

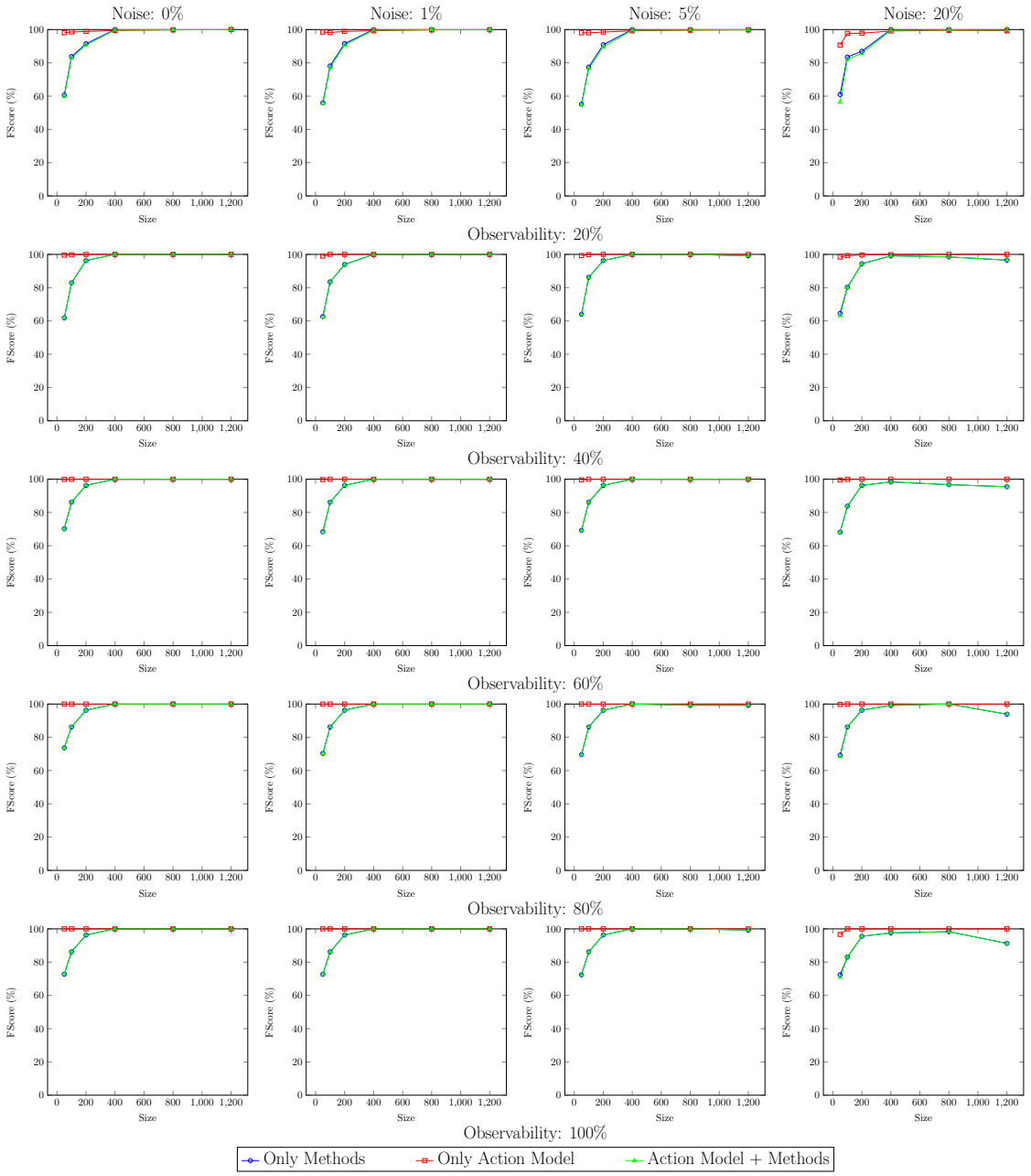


Figure B.94: Transport – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of FScore.

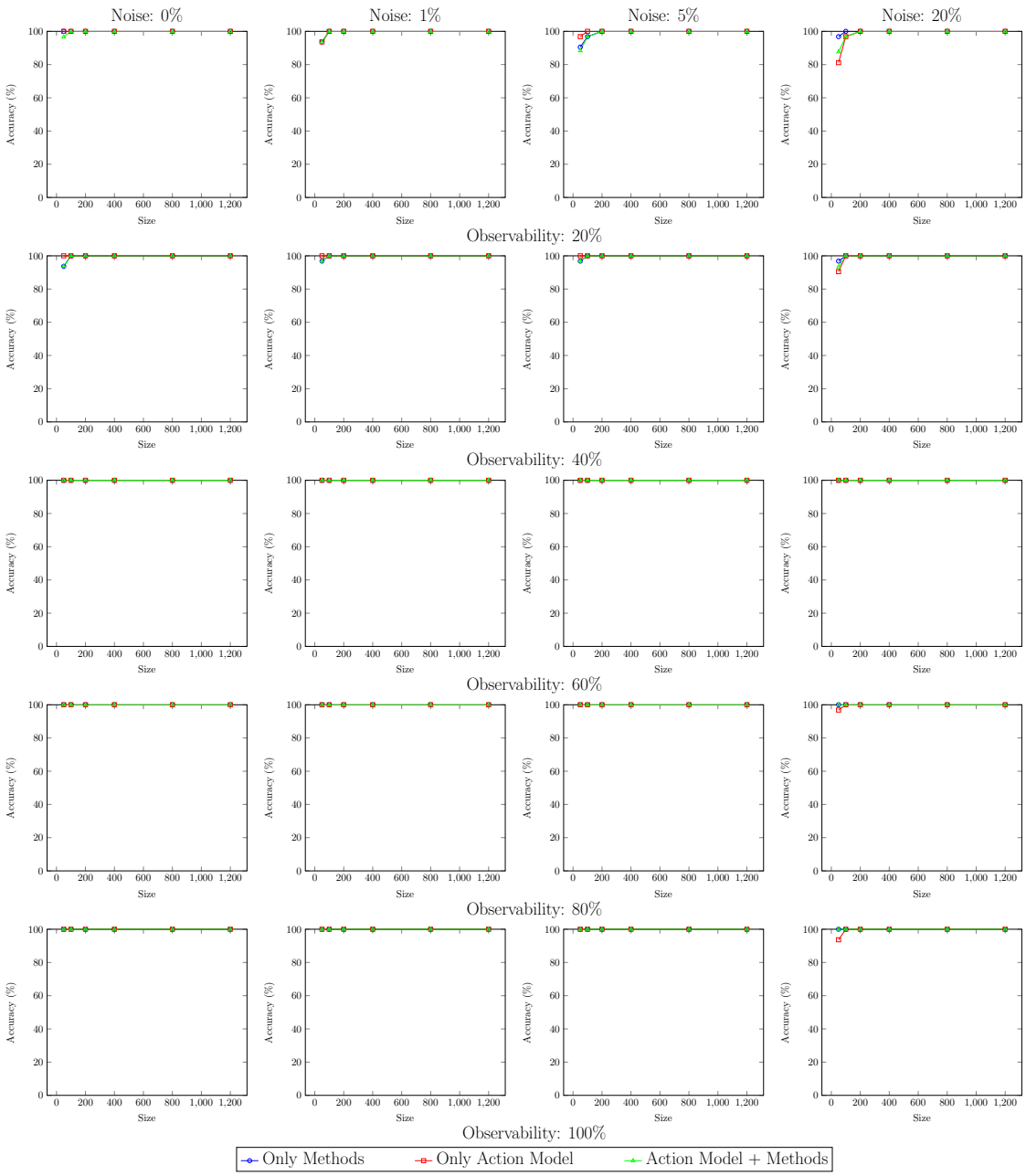


Figure B.95: Transport – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

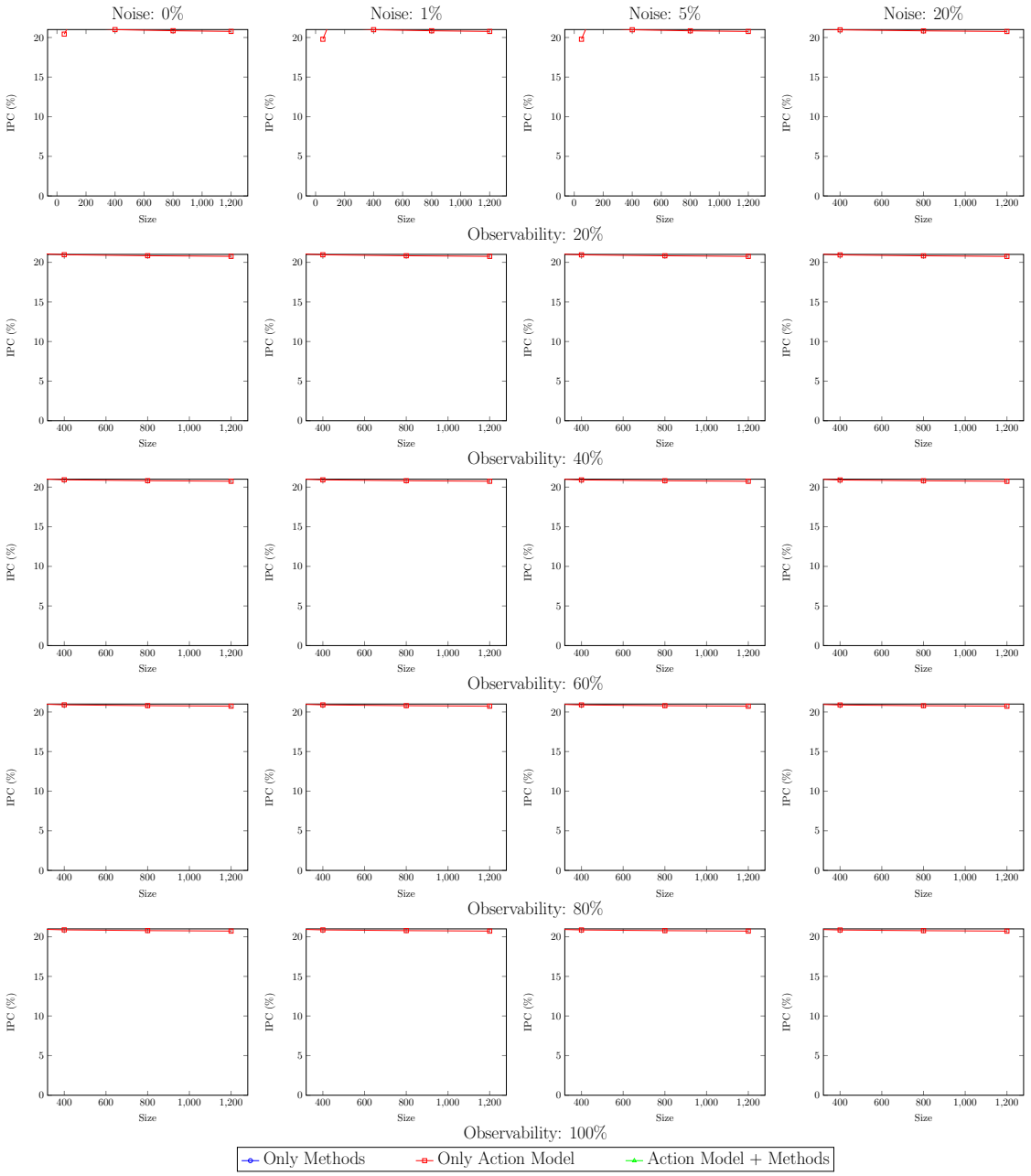


Figure B.96: Transport – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of IPC.

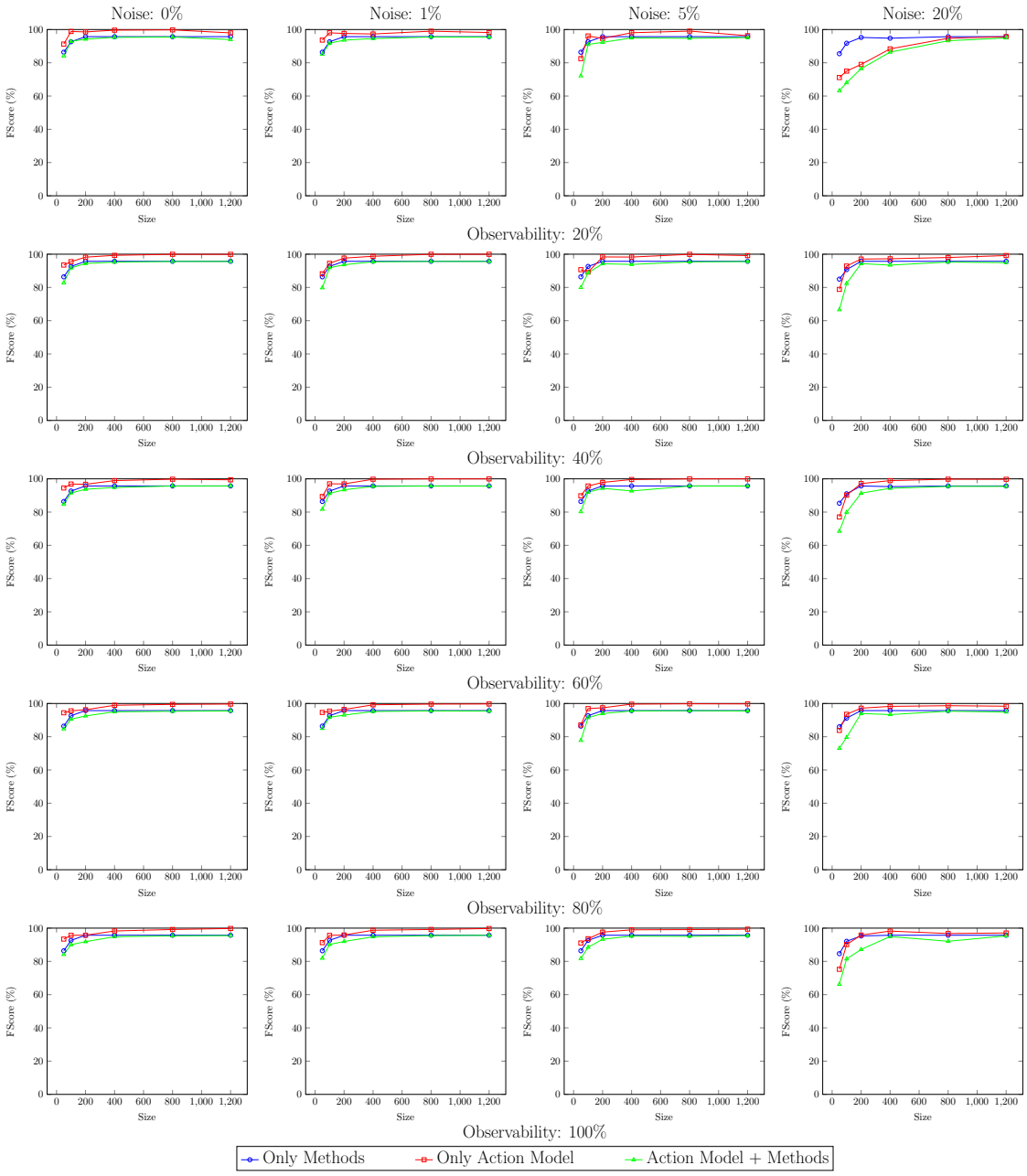


Figure B.97: Childsnack – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of FScore.

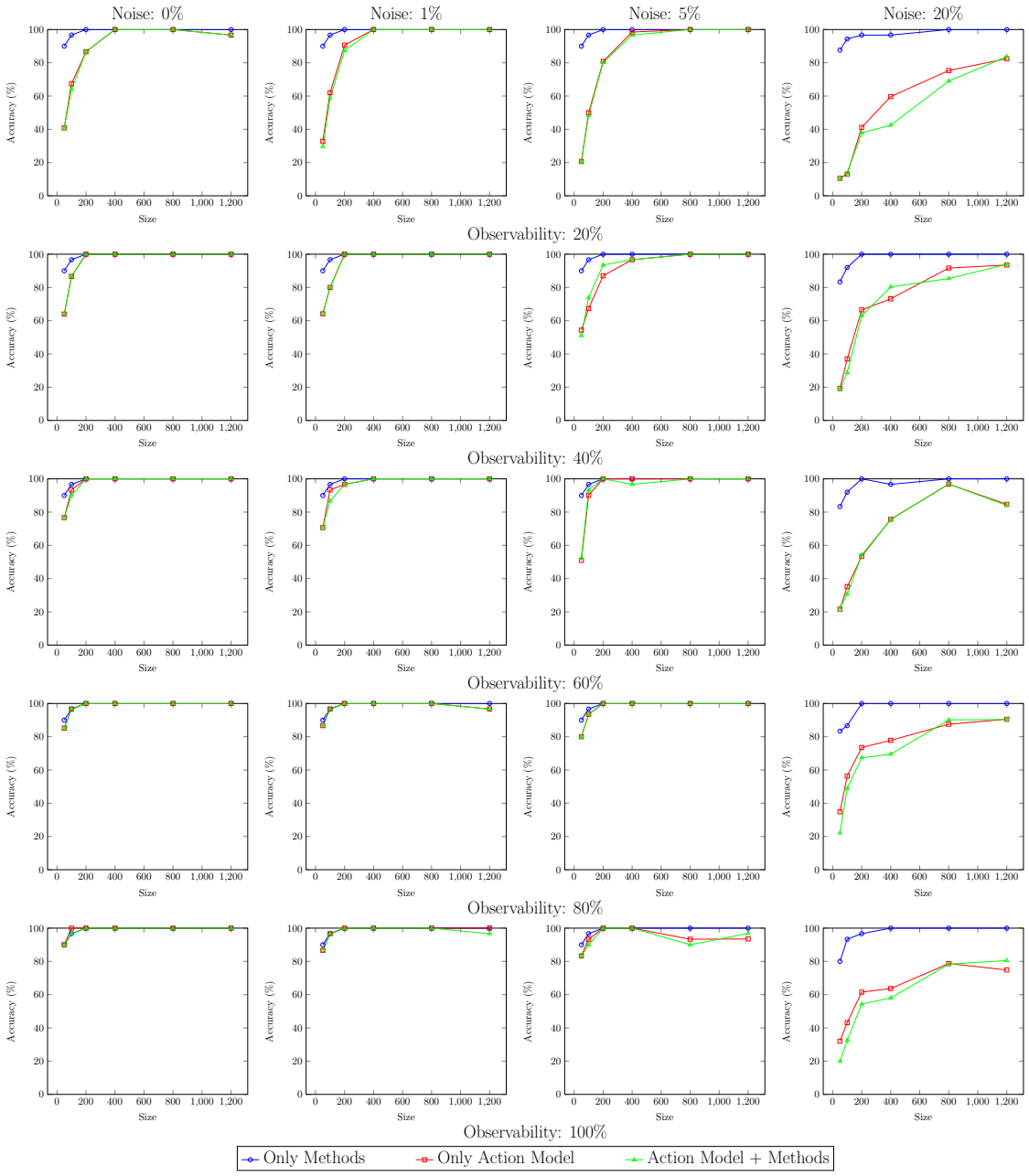


Figure B.98: Childsnack – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

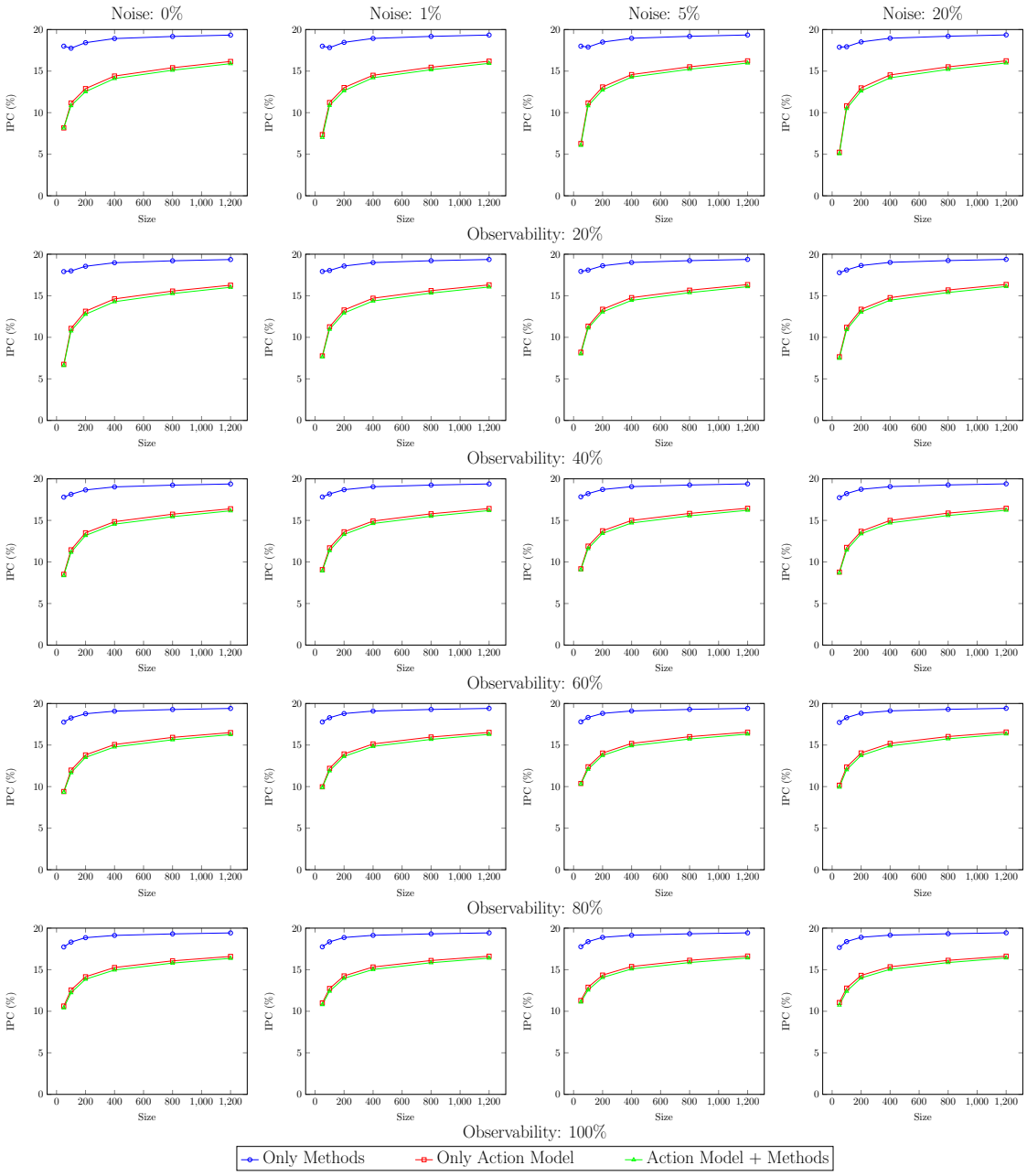


Figure B.99: Childsnack – Average performance of HierAMLSI when the training data set size increases in number of tasks in terms of IPC.

Appendix C

Benchmarks Planning Domains

Contents

C.1 Introduction

In this chapter we give a detailed description of all the action models used in this thesis. We start by describing the STRIPS action models and more particularly their declaration in the form of PDDL planning domains. For each domain, we give a description of the domain, the list of the predicates and the list of the operators with their PDDL declaration. Then, we describe the temporal action models and more precisely their declaration in the form of PDDL 2.1 planning domains. As for PDDL domains, for each domain, we give a description of the domain, the list of the predicates and the list of the operators with their PDDL 2.1 declaration. Finally, we describe the HTN task models and more precisely their declaration in the form of HDDL planning domains. For each domain, we give a description of the domain, the list of the predicates, the list of the compound tasks and the list of the primitive tasks with their HDDL declaration and the list of HDDL methods.

C.2 STRIPS Planning Domains

C.2.1 Blocksworld

Description Classical STRIPS planning domain, where stackable blocks need to be re-assembled on a table. Using its hand, an autonomous agent can stack a block onto a block, unstack a block from a block, put down a block, or pick up a block.

Predicates

- *(on ?x - block ?y - block)*: Thee block ?x is on the block ?y.
- *(ontable ?x - block)*: The block ?x is on the table.
- *(clear ?x - block)*: The block ?x is clear, i.e. there no block on ?x and it is not held .
- *(holding ?x - block)*: The block ?x is held by the autonomous agent.
- *(handempty)*: The autonomous agent does not hold any block.

Operators

- *(pick-up ?x - block)*: The autonomous agent picks a block placed on the table.

```
(:action pick-up
 :parameters (?x - block)
 :precondition (and
      (clear ?x)
      (ontable ?x)
      (handempty))
 :effect (and
      (not (ontable ?x))
      (not (clear ?x))
      (not (handempty))
      (holding ?x)))
```

- *(put-down ?x - block)*: The autonomous agent puts down a block on the table.

```
(:action put-down
 :parameters (?x - block)
 :precondition (holding ?x)
 :effect (and
      (not (holding ?x))
      (clear ?x)
      (handempty)
      (ontable ?x)))
```

- *(stack ?x - block ?y - block)*: The autonomous agent stacks a block on an other block.

```
(:action stack
 :parameters (?x ?y - block)
 :precondition (and
      (holding ?x)
      (clear ?y))
 :effect (and
      (not (holding ?x))
      (not (clear ?y))
      (clear ?x)
      (handempty)
      (on ?x ?y)))
```

- *(unstack ?x - block ?y - block)*: The autonomous agent unstacks a block from an other block.

```
(:action unstack
 :parameters (?x ?y - block)
 :precondition (and
      (on ?x ?y)
      (clear ?x)
      (handempty))
 :effect (and
      (holding ?x)
      (clear ?y)
      (not (clear ?x))
      (not (handempty))
      (not (on ?x ?y)))))
```

C.2.2 Gripper

Description In this domain, there are a robot called robby with grippers and a set of rooms containing balls. The goal is to transport balls from a given room to another.

Predicates

- *(at-robby ?r - room)*: The robot robby is in the room ?r.
- *(at ?b - ball ?r - room)*: The ball ?b is in the room ?r.
- *(free ?g - gripper)*: The Robby's gripper ?g is free.
- *(carry ?b - ball ?g - gripper)*: Robby is carrying the ball ?b with its gripper ?g.

Operators

- *(move ?from - room ?to - room)*: Robby goes to the room ?to.

```
(:action move
 :parameters  (?from ?to - room)
 :precondition (at-robby ?from)
 :effect (and
      (at-robby ?to)
      (not (at-robby ?from))))
```

- *(pick ?b - ball ?r - room ?g - gripper)*: Robby picks the ball with its gripper.

```
(:action pick
 :parameters (?b - ball ?r - room ?g - gripper)
 :precondition  (and
```

```
4       (at ?b ?r)
5       (at-robby ?r)
6       (free ?g))
7   :effect (and
8       (carry ?b ?g)
9       (not (at ?b ?r))
10      (not (free ?g))))
```

- *(drop ?b - ball ?r - room ?g - gripper)*: Robby drops the held ball.

```
1  (:action drop
2   :parameters  (?b - ball ?r - room ?g - gripper)
3   :precondition  (and
4       (carry ?b ?g)
5       (at-robby ?r))
6   :effect (and
7       (at ?b ?r)
8           (free ?g)
9           (not (carry ?b ?g)))))
```

C.2.3 Hanoi

Description The Tower of Hanoi is a puzzle consisting of several rods and a number of disks of various diameters, which can slide onto any rod. It is possible to stack a disk on a second if the latter is bigger.

Predicates

- *(on ?x - disk ?y - disk)*: The disk ?x is on the disk ?y.
- *(bigger ?x - disk ?y - disk)*: The disk ?x is bigger than disk ?y.
- *(on-location ?x - block ?l - location)*: The disk ?x is on the location ?l.
- *(clear ?x - disk)*: The disk ?x is clear.
- *(clear-location ?l - location)*: The location ?l is clear.
- *(holding ?x - disk)*: The disk ?x is held by the autonomous agent.
- *(handempty)*: The autonomous agent does not hold any block.

Operators

- *(pick-up ?x - disk ?l - location)*: The autonomous agent picks a disk placed on ?l.

```
(:action pick-up
 :parameters (?x - disk ?l - location)
 :precondition (and
      (clear ?x)
      (on-location ?x ?l)
      (handempty))
 :effect (and
      (not (on-location ?x ?l))
      (not (clear ?x))
      (not (handempty))
      (clear-location ?l)
      (holding ?x)))
```

- *(put-down ?x - disk ?l - location)*: The autonomous agent puts down a disk on ?l.

```
(:action put-down
 :parameters (?x - disk ?l - location)
 :precondition (and
      (holding ?x)
      (clear-location ?l))
 :effect (and
      (not (holding ?x))
      (clear ?x)
      (handempty)
      (not (clear-location ?l))
      (on-location ?x)))
```

- *(stack ?x - disk ?y - disk)*: The autonomous agent stacks ?x on ?y. ?y must be bigger than ?x.

```
(:action stack
 :parameters (?x ?y - disk)
 :precondition (and
      (bigger ?y ?x)
      (holding ?x)
      (clear ?y))
 :effect (and
      (not (holding ?x))
      (not (clear ?y))
      (clear ?x)
```

```
11        (handempty)
12        (on ?x ?y)))
```

- *(unstack ?x - disk ?y - disk)*: The autonomous agent unstacks ?x from ?y.

```
1  (:action unstack
2   :parameters (?x ?y - disk)
3   :precondition (and
4        (on ?x ?y)
5        (clear ?x)
6        (handempty))
7   :effect (and
8        (holding ?x)
9        (clear ?y)
10       (not (clear ?x))
11       (not (handempty))
12       (not (on ?x ?y)))))
```

C.2.4 N-Puzzle

Description The N-Puzzle (also called Mystic Square) is a sliding puzzle having $N^2 - 1$ square tiles numbered 1–$(N^2 - 1)$ on a square of $N \times N$ tiles.

Predicates

- *(at ?t - tile ?p - position)*: The tile ?t is at ?p.
- *(neighbor ?p1 - position ?p2 - position)*: ?p1 is neighbor of ?p2.
- *(empty ?p - position)*: There are no tile at ?p.

Operators

- *(move ?t - tile ?from - position ?to - position)*: The tile in position ?from is moved to the empty adjacent position ?to.

```
1  (:action move
2   :parameters (?t - tile ?from ?to - position)
3   :precondition (and
4        (neighbor ?from ?to)
5        (at ?t ?from)
6        (empty ?to))
7   :effect (and
8        (at ?t ?to)
9        (empty ?from)
10       (not (at ?t ?from))
11       (not (empty ?to))))
```

C.2.5 Peg Solitaire

Description Peg solitaire is a board game for one player involving movement of pegs on a board with holes.

Predicates

- *(in-line ?x - location ?y - location ?z - location)*: The locations ?x, ?y and ?z are in the same line.
- *(occupied ?l - location)*: The location ?l is occupied by a token.
- *(free ?l - location)*: The location ?l is not occupied by any token.
- *(move-ended)*: The move is over.
- *(last-visited ?l - location)*: The position ?l is the last location altered (by added or removed a token) during the movement.

Operators

- *(jump-new-move ?from - location ?over - location ?to - location)*: The player begins a new move. More precisely, The peg in the ?from location is moves into the ?to location. The peg placed in the ?over location is removed.

```
(:action jump-new-move
 :parameters (?from ?over ?to - location)
 :precondition (and
      (move-ended)
      (in-line ?from ?over ?to)
      (occupied ?from)
      (occupied ?over)
      (free ?to))
:effect (and
      (not (move-ended))
      (not (occupied ?from))
      (not (occupied ?over))
      (not (free ?to))
      (free ?from)
      (free ?over)
      (occupied ?to)
      (last-visited ?to)))
```

- *(jump-continue-move ?from - location ?over - location ?to - location)*: The player continues its move a remove the peg in the location ?over.

```
(:action jump-continue-move
 :parameters (?from ?over ?to - location)
 :precondition (and
      (last-visited ?from)
      (in-line ?from ?over ?to)
      (occupied ?from)
      (occupied ?over)
      (free ?to))
 :effect (and
   (not (occupied ?from))
   (not (occupied ?over))
   (not (free ?to))
   (free ?from)
   (free ?over)
   (occupied ?to)
   (not (last-visited ?from))
   (last-visited ?to)))
```

- *(end-move ?loc - location)*: The player ends its movement.

```
(:action end-move
 :parameters (?loc - location)
 :precondition (last-visited ?loc)
 :effect (and
      (move-ended)
      (not (last-visited ?loc))))
```

- *(unstack ?x - block ?y - block)*:

```
(:action unstack
 :parameters (?x - block ?y - block)
 :precondition (and
      (on ?x ?y)
      (clear ?x)
      (handempty))
 :effect (and
      (holding ?x)
      (clear ?y)
      (not (clear ?x))
      (not (handempty))
      (not (on ?x ?y)))))
```

C.2.6 Zenotravel

Description A transportation domain involves transporting people around in planes, using different modes of movement: fast and slow.

Predicates

- *(at ?p - person ?c - city)*: The person ?p is at the city ?c.
- *(at-aircraft ?a - aircraft ?c - city)*: The aircraft ?a is at the city ?c.
- *(in ?p - person ?a - aircraft)*: The person ?p is in the aircraft ?a.
- *(fuel-level ?a - aircraft ?l - flevel)*: The fuel level ?l for the aircraft ?a.
- *(next ?l1 - flevel ?l2 - flevel)*: If the flevel is ?l1 then the next level will be ?l2.

Operators

- *(board ?p - person ?a - aircraft ?c - city)*: Passenger ?p boards in ?a at the city ?c.

```
(:action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and
      (at ?p ?c)
      (at-aircraft ?a ?c))
 :effect (and
      (not (at ?p ?c))
      (in ?p ?a)))
```

- *(debark ?p - person ?a - aircraft ?c - city)*: Passenger ?p debarks from ?a at the city ?c.

```
(:action debark
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and
      (in ?p ?a)
      (at-aircraft ?a ?c))
 :effect (and
      (at ?p ?c)
      (not (in ?p ?a))))
```

- *(fly ?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel)*: ?a flies from ?c1 to ?c2. Its fuel reserves are reduced from ?l1 to ?l2.

```
(:action fly
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?l1 ?l2 - flevel)
 :precondition (and
     (at-aircraft ?a ?c1)
     (fuel-level ?a ?l1)
     (next ?l2 ?l1))
 :effect (and
     (not (at-aircraft ?a ?c1))
     (at-aircraft ?a ?c2)
     (not (fuel-level ?a ?l1))
     (fuel-level ?a ?l2)))
```

- *(zoom ?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel ?l3 - flevel)*: ?a flies from ?c1 to ?c2. Its fuel reserves are reduced from ?l1 to ?l3.

```
(:action zoom
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?l1 ?l2 ?l3 - flevel)
 :precondition (and
     (at-aircraft ?a ?c1)
     (fuel-level ?a ?l1)
     (next ?l2 ?l1)
     (next ?l3 ?l2))
 :effect (and
     (not (at-aircraft ?a ?c1))
     (at-aircraft ?a ?c2)
     (not (fuel-level ?a ?l1))
     (fuel-level ?a ?l3)))
```

- *(refuel ?a - aircraft ?c - city ?l - flevel ?l1 - flevel)*: ?a fills up with fuel in the city ?c. Its fuel reserves are increased from ?l to ?l1.

```
(:action refuel
 :parameters (?a - aircraft ?c - city ?l ?l1 - flevel)
 :precondition (and
     (fuel-level ?a ?l)
     (next ?l ?l1)
     (at-aircraft ?a ?c))
 :effect (and
     (fuel-level ?a ?l1)
     (not (fuel-level ?a ?l)))))
```

C.2.7 Parking

Description The domain involves parking cars on a street with N curb locations, and where cars can be double-parked but not triple-parked.

Predicates

- *(at-curb ?c - car)*: The car ?c is parked to a curb.
- *(at-curb-num ?c - car ?c - curb)*: The car ?c is parked to the curb ?c.
- *(behind-car ?c ?front-car - car)*: ?c is parked behind ?front-car.
- *(car-clear ?c - car)*: There are no cars in the back of ?c.
- *(curb-clear ?c - curb)*: There are no cars at ?c.

Operators

- *(move-curb-to-curb ?c - car ?csrc - curb ?cdest - curb)*: Car parked on the curb ?csrc is moved to the curb ?cdest.

```
(:action move-curb-to-curb
 :parameters (?c - car ?csrc ?cdest - curb)
 :precondition (and
        (car-clear ?c)
        (curb-clear ?cdest)
        (at-curb-num ?c ?csrc))
:effect (and
    (not (curb-clear ?cdest))
    (curb-clear ?csrc)
    (at-curb-num ?c ?cdest)
    (not (at-curb-num ?c ?csrc))))
```

- *(move-curb-to-car ?c - car ?csrc - curb ?cdest - car)*: Car parked on the curb ?csrc is moved to the back of ?cdest.

```
(:action move-curb-to-car
 :parameters (?c - car ?csrc - curb ?cdest - car)
 :precondition (and
      (car-clear ?c)
      (car-clear ?cdest)
      (at-curb-num ?c ?csrc)
      (at-curb ?cdest))
 :effect (and
      (not (car-clear ?cdest))
      (curb-clear ?csrc)
      (behind-car ?c ?cdest)
```

```
12        (not (at-curb-num ?c ?csrc))
13        (not (at-curb ?c))))
```

- *(moce-car-to-curb ?c - car ?csrc - car ?cdest - curb)*: Car parked on the back of ?csrc is moved to the curb ?cdest.

```
1  (:action move-car-to-curb
2   :parameters (?c ?csrc - car ?cdest - curb)
3   :precondition (and
4           (car-clear ?c)
5           (curb-clear ?cdest)
6           (behind-car ?c ?csrc))
7   :effect (and
8           (not (curb-clear ?cdest))
9           (car-clear ?csrc)
10          (at-curb-num ?c ?cdest)
11          (not (behind-car ?c ?csrc))
12          (at-curb ?c)))
```

- *(move-car-to-car ?c - car ?csrc - car ?cdest - car)*: Car parked on the back of ?csrc is moved to the back of ?cdest.

```
1  (:action move-car-to-car
2   :parameters (?c ?csrc ?cdest - car)
3   :precondition (and
4       (car-clear ?c)
5       (car-clear ?cdest)
6       (behind-car ?c ?csrc)
7       (at-curb ?cdest))
8   :effect (and
9       (not (car-clear ?cdest))
10      (car-clear ?csrc)
11      (behind-car ?c ?cdest)
12      (not (behind-car ?c ?csrc))))
```

C.2.8 Sokoban

Description Sokoban is a puzzle in which the player pushes boxes around in a warehouse, trying to get them to storage locations

Predicates

- *(at ?o - box ?l - location)*: the box ?o is at ?l.
- *(at-robot ?l - location)*: The robot is at ?l.

- *(adjacent ?l1 - location ?l2 - location ?d - direction)*: The robot can move from ?l1 to ?l2, and ?l2 is on the ?d of ?l1.

- *(clear ?l - location)*: There are nothing at ?l.

Operators

- *(move ?from - location ?to - location ?dir - direction)*: The player moves to the ?dir of the location ?from to the location ?to.

```
(:action move
 :parameters (?from ?to - location
              ?dir - direction)
 :precondition (and
     (clear ?to)
     (at-robot ?from)
     (adjacent ?from ?to ?dir))
 :effect (and
     (at-robot ?to)
     (not (at-robot ?from))))
```

- *(push ?rloc - location ?bloc - location ?floc - location ?dir - direction ?b - box)*: The player pushes the box ?b to the location ?floc. The initial position of the player was ?rloc and its final position is the initial position of the box: ?bloc.

```
(:action push
 :parameters  (?rloc ?bloc ?floc - location
               ?dir - direction
               ?b - box)
 :precondition (and
     (at-robot ?rloc)
     (at ?b ?bloc)
     (clear ?floc)
     (adjacent ?rloc ?bloc ?dir)
     (adjacent ?bloc ?floc ?dir))
 :effect (and
     (at-robot ?bloc)
     (at ?b ?floc)
     (clear ?bloc)
     (not (at-robot ?rloc))
     (not (at ?b ?bloc))
     (not (clear ?floc))))
```

C.2.9 Visit All

Description An agent in the middle of a square grid $n \times n$ must visit all the cells in the grid

Predicates

- *(connected ?x - place ?y - place)*: Places ?x and ?y are connected.
- *(at-robot ?x - place)*: The robot is at ?x.
- *(visited ?x - place)*: The robot has already visited ?x.

Operators

- *(move ?curpos - place ?nextpos - place)*: Agent moves from the ?curpos to the ?nextpos.

```
(:action move
 :parameters (?curpos ?nextpos - place)
 :precondition (and
     (at-robot ?curpos)
     (connected ?curpos ?nextpos))
 :effect (and
     (at-robot ?nextpos)
     (not (at-robot ?curpos))))
```

- *(visit ?pos - place)*: Agent marks its position as visited.

```
(:action visit
 :parameters (?pos - place)
 :precondition (and
     (at-robot ?pos)
     (not(visited ?pos)))
 :effect (visited ?pos))
```

C.2.10 Elevator

Description Transport a number of passengers with an elevator from their origin to their destination floors. With explicit control over the passengers that get in or out of the lift.

Predicates

- *(origin ?p - passenger ?f - floor)*: ?p waits for the elevator at the floor ?f.
- *(destin ?p - passenger ?f - floor)*: ?p wants to go to the floor ?f.
- *(ontable ?x - block)*: The block ?x is on the table.
- *(above ?f1 - floor ?f2 - floor)*: ?f1 is above ?f2.
- *(boarded ?p - passenger)*: ?p is on board of the elevator.
- *(served ?p - passenger)*: ?p has been served by the elevator.
- *(lift-at ?f - floor)*: The elevator is at floor ?f.

Operators

- *(board ?f - floor ?p - passenger)*: Passenger ?p boards the elevator at floor ?f.

```
(:action board
 :parameters (?f - floor ?p - passenger)
 :precondition (and
      (lift-at ?f)
      (origin ?p ?f)
      (not(boarded ?p))
      (not(served ?p)))
 :effect (boarded ?p))
```

- *(depart ?f - floor ?p - passenger)*: Passenger ?p exits the elevator at floor ?f.

```
(:action board
 :parameters (?f - floor ?p - passenger)
 :precondition (and
      (lift-at ?f)
      (destin ?p ?f)
      (boarded ?p)
      (not(served ?p)))
 :effect (and
      (not (boarded ?p))
      (served ?p)))
```

- *(up ?f1 - floor ?f2 - floor)*: The elevator goes up to floor ?f2 from floor ?f1.

```
(:action up
 :parameters (?f1 ?f2 - floor)
 :precondition (and
      (lift-at ?f1)
```

```
5       (above ?f1 ?f2))
6   :effect (and
7       (lift-at ?f2)
8       (not (lift-at ?f1))))
```

- *(down ?f1 - floor ?f2 - floor)*: The elevator goes down to floor ?f2 from floor ?f1.

```
1  (:action down
2   :parameters (?f1 ?f2 - floor)
3   :precondition (and
4       (lift-at ?f1)
5       (above ?f2 ?f1))
6   :effect (and
7       (lift-at ?f2)
8       (not (lift-at ?f1))))
```

C.2.11 Floortile

Description A set of robots use different colors to paint patterns in floor tiles. The robots can move around the floor tiles in four directions (up, down, left, and right). Robots paint with one color at a time but can change their spray guns to any available color. However, robots can only paint the tile that is in front (up) and behind (down) them, and once a tile has been painted, no robot can stand on it.

Predicates

- *(robot-at ?r - robot ?x - tile)*: Robot ?r is at tile ?t.
- *(robot-has ?r - robot ?c - color)*: Robot ?r has the color ?c ?t.
- *(above ?x - tile ?y - tile)*: ?y is above ?x.
- *(below ?x - tile ?y - tile)*: ?y is below ?x.
- *(rightOf ?x - tile ?y - tile)*: ?y is to the right of ?x.
- *(leftOf ?x - tile ?y - tile)*: ?y is to the left of ?x.
- *(clear ?x - tile)*: The tile ?x is clear, i.e. there no robot or paint on ?x.
- *(painted ?x - tile ?c - color)*: ?x is painted in ?c.
- *(available-color ?c - color)*: ?c is available.
- *(free-color ?r - robot)*: Robot ?x has no color.

Operators

- *(change-color ?r - robot ?c - color ?c2 - color)*: Robot ?r replaces the color ?c with the color ?c2.

```
(:action change-color
 :parameters (?r - robot
              ?c ?c2 - color)
 :precondition (and
     (robot-has ?r ?c)
     (available-color ?c2))
 :effect (and
     (not (robot-has ?r ?c))
     (robot-has ?r ?c2)))
```

- *(paint-up ?r - robot ?y - tile ?x - tile ?c - color)*: Robot ?r, place on tile ?x, paint the tile above it (?y) with color ?c.

```
(:action paint-up
 :parameters (?r - robot
              ?y ?x - tile
              ?c - color)
 :precondition (and
     (robot-has ?r ?c)
     (robot-at ?r ?x)
     (above ?y ?x)
     (clear ?y))
 :effect (and
     (not (clear ?y))
     (painted ?y ?c)))
```

- *(paint-down ?r - robot ?y - tile ?x - tile ?c - color)*: Robot ?r, place on tile ?x, paint the tile below it (?y) with color ?c.

```
(:action paint-down
 :parameters (?r - robot
              ?y ?x - tile
              ?c - color)
 :precondition (and
     (robot-has ?r ?c)
     (robot-at ?r ?x)
     (below ?y ?x)
     (clear ?y))
 :effect (and
     (not (clear ?y))
     (painted ?y ?c)))
```

- *(up ?r - robot ?x - tile ?y - tile)*: Robot ?r goes on the tile above.

```
(:action up
 :parameters (?r - robot ?x ?y - tile)
 :precondition (and
      (robot-at ?r ?x)
      (above ?y ?x)
      (clear ?y))
 :effect (and
      (robot-at ?r ?y)
      (not (robot-at ?r ?x))
      (clear ?x)
      (not (clear ?y))))
```

- *(down ?r - robot ?x - tile ?y - tile)*: Robot ?r goes on the tile below.

```
(:action down
 :parameters (?r - robot ?x ?y - tile)
 :precondition (and
      (robot-at ?r ?x)
      (below ?y ?x)
      (clear ?y))
 :effect (and
      (robot-at ?r ?y)
      (not (robot-at ?r ?x))
      (clear ?x)
      (not (clear ?y))))
```

- *(right ?r - robot ?x - tile ?y - tile)*: Robot ?r goes on the tile to its right.

```
(:action right
 :parameters (?r - robot ?x ?y - tile)
 :precondition (and
      (robot-at ?r ?x)
      (rightOf ?y ?x)
      (clear ?y))
 :effect (and
      (robot-at ?r ?y)
      (not (robot-at ?r ?x))
      (clear ?x)
      (not (clear ?y))))
```

- *(left ?r - robot ?x - tile ?y - tile)*: Robot ?r goes on the tile to its left.

```
(:action left
 :parameters (?r - robot ?x ?y - tile)
```

```
 :precondition (and
      (robot-at ?r ?x)
      (leftOf ?y ?x)
      (clear ?y))
 :effect (and
      (robot-at ?r ?y)
      (not (robot-at ?r ?x))
      (clear ?x)
      (not (clear ?y))))
```

C.2.12 Logistics

Description Transport packages within cities via trucks, and between cities via airplanes. Locations within a city are directly connected (trucks can move between any two such locations), and so are the cities.

Predicates

- *in-city ?loc - place ?c - city)*: The place ?loc is in ?c.
- *(at ?obj - physobj ?loc - place)*: The physical object ?obj is at place ?loc.
- *(in ?pkg - package ?veh - vehicle)*: The package ?pkg is in the vehicle ?veh.

Operators

- *(load-truck ?pkg - package ?t - truck ?loc - place)*: The package ?pkg is loaded in the truck ?t at the place ?loc.

```
(:action load-truck
 :parameters (?pkg - package
              ?t - truck
              ?loc - place)
 :precondition (and
      (at ?t ?loc)
      (at ?pkg ?loc))
 :effect (and
      (not (at ?pkg ?loc))
      (in ?pkg ?t)))
```

- *(unload-truck ?pkg - package ?t - truck ?loc - place)*: The package ?pkg is unloaded from the truck ?t at the place ?loc.

```
(:action unload-truck
 :parameters (?pkg - package
              ?t - truck
```

```
4                ?loc - place)
5  :precondition (and
6      (at ?t ?loc)
7      (in ?pkg ?t))
8  :effect (and
9      (not (in ?pkg ?t))
10     (at ?pkg ?loc)))
```

- *(load-airplane ?pkg - package ?a - airplane ?loc - place)*: The package ?pkg is loaded in the airplane ?a at the place ?loc.

```
1  (:action load-airplane
2   :parameters (?pkg - package
3                ?a - airplane
4                ?loc - place)
5   :precondition (and
6       (at ?a ?loc)
7       (at ?pkg ?loc))
8   :effect (and
9       (not (at ?pkg ?loc))
10      (in ?pkg ?a)))
```

- *(unload-airplane ?pkg - package ?a - airplane ?loc - place)*: The package ?pkg is unloaded from the airplane ?a at the place ?loc.

```
1  (:action unload-airplane
2   :parameters (?pkg - package
3                ?a - airplane
4                ?loc - place)
5   :precondition (and
6       (at ?a ?loc)
7       (in ?pkg ?a))
8   :effect (and
9       (not (in ?pkg ?a))
10      (at ?pkg ?loc)))
```

- *(drive-truck ?t - truck ?loc-from - place ?loc-to - place ?c - city)*: The truck moves from ?loc-from to ?loc-to in the city ?c.

```
1  (:action drive-truck
2   :parameters (?t - truck
3                ?loc-from ?loc-to - place
4                ?c - city)
5   :precondition (and
6       (at ?t ?loc-from)
7       (in-city ?loc-from ?c)
```

```
8       (in-city ?loc-to ?c))
9   :effect (and
10      (not (at ?t ?loc-from))
11      (at ?t ?loc-to)))
```

- *(fly-airplane ?a - airplane ?loc-from - airport ?loc-to - airport)*: The airplane flies from the airport ?loc-from to the airport ?loc-to.

```
1  (:action fly-airplane
2   :parameters (?a - airplane
3                ?loc-from ?loc-to - airport)
4   :precondition (at ?a ?loc-from)
5   :effect (and
6       (not (at ?a ?loc-from))
7       (at ?a ?loc-to))))
```

C.2.13 Spanner

Description A worker is in a shed, containing a number of spanners, and at a gate some distance away there are a number of nuts that must be tightened. The spanners are fragile and can be used only once to tighten a nut.

Predicates

- *(at ?m - locatable ?l - location)*: ?m is at ?l.
- *(carrying ?m - man ?s - spanner)*: The man ?m carries the spanner ?s.
- *(useable ?s - spanner)*: There are no man carrying ?s.
- *(link ?l1 - location ?l2 - location)*: There are a link between ?l1 and ?l2.
- *(tightened ?n - nut)*: The nut ?n is tightened.
- *(loose ?n - nut)*: The nut ?n is loosed.

Operators

- *(walk ?start - location ?end - location ?m - man)*: The worker ?m walks from ?start to ?end.

```
1  (:action walk
2   :parameters (?start ?end - location
3                ?m - man)
4  :precondition (and
5      (at ?m ?start)
6      (link ?start ?end))
```

```
:effect (and
    (not (at ?m ?start))
    (at ?m ?end)))
```

- *(pickup-spanner ?l - location ?s - spanner ?m - man)*: ?m picks the spanner ?s at ?l.

```
(:action pickup-spanner
 :parameters (?l - location ?s - spanner ?m - man)
 :precondition (and
    (at ?m ?l)
    (at ?s ?l))
 :effect (and
    (not (at ?s ?l))
    (carrying ?m ?s)))
```

- *(tighten-nut ?l - location ?s - spanner ?m - man ?n - nut)*: ?m tightens the nut with its spanner.

```
(:action tighten-nut
 :parameters (?l - location
              ?s - spanner
              ?m - man
              ?n - nut)
 :precondition (and
    (at ?m ?l)
        (at ?n ?l)
        (carrying ?m ?s)
        (useable ?s)
        (loose ?n))
:effect (and
    (not (loose ?n))
    (not (useable ?s))
    (tightened ?n)))
```

C.3 Temporal Planning Domains

C.3.1 Match

Description An agent needs to repair fuses. To repair a fuse the agent needs a lighted match.

Predicates

- *(handfree)*: The hand is free.
- *(unused ?m - match)*: Match ?m is unused.
- *(mended ?f - fuse)*: Fuse ?f is mended.
- *(light ?m - match)*: Match ?m is light.

Operators

- *(light-match ?m - match)*: The agent lights the match ?m.

```
(:durative-action light-match
 :parameters (?m - match)
 :duration (= ?duration 5)
 :condition (at start (unused ?m))
 :effect (and
         (at start (not (unused ?m)))
         (at start (light ?m))
         (at end (not (light ?m)))))
```

- *(mend-fuse ?f - fuse ?m - match)*: The agent repairs the fuse ?f by lighting himself with the match ?m.

```
(:durative-action mend-fuse
 :parameters (?f - fuse ?m - match)
 :duration (= ?duration 2)
 :condition (and
   (at start (handfree))
   (at start (not(mended ?f)))
   (over all (light ?m)))
 :effect (and
   (at start (not (handfree)))
   (at end (mended ?f))
   (at end (handfree))))
```

C.3.2 Turn and Open

Description In this domain, there are several robots with grippers and a set of rooms containing balls. The goal is to transport balls from a given room to another. There are doors that must be open to move from one room to another. In order to open a given door, the robot must turn the doorknob and open the door at the same time.

Predicates

- *(at-robby ?r - robot ?x - room)*: Robot ?r is at room ?x.
- *(at ?o - obj ?x - room)*: Object ?o is at room ,x.
- *(free ?r - robot ?g - gripper)*: Robot ?r's gripper ?g is free.
- *(carry ?r - robot ?o - obj ?g - gripper)*: Robot ?r is carrying ?o with its gripper ?g.
- *(connected ?x - room ?y - room ?d - door)*: Rooms ?x and ?y are connected with the door ?d.
- *(open ?d - door)*: ?d is opened.
- *(closed ?d - door)*: ?d is closes.
- *(doorknob-turned ?d - door ?g - gripper)*: The door's doorknob is turned by ?g.

Operators

- *(turn-doorknob ?r - robot ?from ?to - room ?d - door ?g - gripper)*: Robot turns the doorknob of the door.

```
(:durative-action turn-doorknob
 :parameters (?r - robot
              ?from ?to - room
              ?d - door
              ?g - gripper)
 :duration (= ?duration 3)
 :condition (and
    (over all (at-robby ?r ?from))
    (at start (free ?r ?g))
        (over all (connected ?from ?to ?d))
        (at start (closed ?d)))
 :effect (and
    (at start (not (free ?r ?g)))
    (at end (free ?r ?g))
    (at start (doorknob-turned ?d ?g))
    (at end (not (doorknob-turned ?d ?g)))))
```

- *(open-door)?r - robot ?from ?to - room ?d - door ?g - gripper*: Robot opens the door.

```
(:durative-action open-door
 :parameters (?r - robot
              ?from ?to - room
              ?d - door
              ?g - gripper)
 :duration (= ?duration 2)
 :condition (and
    (over all (at-robby ?r ?from))
    (over all (connected ?from ?to ?d))
        (over all (doorknob-turned ?d ?g))
        (at start (closed ?d)))
 :effect (and
    (at start (not (closed ?d)))
    (at end (open ?d))))
```

- *(move ?r - robot ?from ?to - room ?d - door)*: Robot goes to the room ?to. The door must be open.

```
(:durative-action move
 :parameters  (?r - robot
               ?from ?to - room
               ?d - door)
 :duration (= ?duration 1)
 :condition (and
    (at start (at-robby ?r ?from))
    (over all (connected ?from ?to ?d))
    (over all (open ?d)))
 :effect (and
    (at end (at-robby ?r ?to))
    (at start (not (at-robby ?r ?from)))))
```

- *(pick ?r - robot ?obj - obj ?room - room ?g - gripper)*: Robot picks the ball with its gripper.

```
(:durative-action pick
:parameters (?r - robot
             ?obj - obj
             ?room - room
             ?g - gripper)
:duration (= ?duration 1)
:condition (and
   (at start (at ?obj ?room))
   (at start (at-robby ?r ?room))
   (at start (free ?r ?g)))
:effect (and
```

```
    (at end (carry ?r ?obj ?g))
    (at start (not (at ?obj ?room)))
    (at start (not (free ?r ?g)))))
```

- *(drop ?r - robot ?obj - obj ?room - room ?g - gripper)*: Robot drops the ball holds by its gripper.

```
(:durative-action pick
:parameters (?r - robot
             ?obj - obj
             ?room - room
             ?g - gripper)
:duration (= ?duration 1)
:condition (and
    (at start (at ?obj ?room))
    (at start (at-robby ?r ?room))
    (at start (free ?r ?g)))
  :effect (and
    (at end (carry ?r ?obj ?g))
    (at start (not (at ?obj ?room)))
    (at start (not (free ?r ?g)))))
```

C.3.3 Peg Solitaire

Description Peg solitaire is a board game for one player involving movement of pegs on a board with holes.

Predicates

- *(in-line ?x - location ?y - location ?z - location)*: The locations ?x, ?y and ?z are in the same line.
- *(occupied ?l - location)*: The location ?l is occupied by a token.
- *(free ?l - location)*: The location ?l is not occupied by any token.

Operators

- *(jump ?from - location ?over - location ?to - location)*: The player jumps from ?from to ?to. More precisely, The peg in the ?from location is moves into the ?to location. The peg placed in the ?over location is removed.

```
(:durative-action jump
 :parameters (?from ?over ?to - location)
 :duration (= ?duration 1)
 :condition (and
```

```
    (over all (in-line ?from ?over ?to))
    (at start (occupied ?from))
    (at start (occupied ?over))
    (at start (free ?to)))
 :effect (and
     (at start (not (occupied ?from)))
     (at start (not (occupied ?over)))
     (at start (not (free ?to)))
     (at end (free ?from))
     (at end (free ?over))
     (at end (occupied ?to))))
```

C.3.4 Zenotravel

A transportation domain involves transporting people around in planes, using different modes of movement: fast and slow.

Description

Predicates

- *(at ?x - locatable ?c - city)*: The locatable ?x is at the city ?c.
- *(in ?p - person ?a - aircraft)*: The person ?p is in the aircraft ?a.
- *(fuel-level ?a - aircraft ?l - flevel)*: The fuel level ?l for the aircraft ?a.
- *(next ?l1 - flevel ?l2 - flevel)*: If the flevel is ?l1 then the next level will be ?l2.

Operators

- *(board ?p - person ?a - aircraft ?c - city)*: Passenger ?p boards in ?a at the city ?c.

```
(:durative-action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :duration (= ?duration 20)
 :condition (and
     (at start (at ?p ?c))
     (over all (at ?a ?c)))
 :effect (and
     (at start (not (at ?p ?c)))
     (at end (in ?p ?a))))
```

- *(debark ?p - person ?a - aircraft ?c - city)*: Passenger ?p debarks from ?a at the city ?c.

```
(:durative-action debark
 :parameters (?p - person ?a - aircraft ?c - city)
 :duration (= ?duration 30)
 :condition (and
    (at start (in ?p ?a))
    (over all (at ?a ?c)))
 :effect (and
    (at start (not (in ?p ?a)))
    (at end (at ?p ?c))))
```

- *(fly ?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel)*: ?a flies from ?c1 to ?c2. Its fuel reserves are reduced from ?l1 to ?l2. (slow)

```
(:durative-action fly
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?l1 ?l2 - flevel)
 :duration (= ?duration 180)
 :condition (and
    (at start (at ?a ?c1))
    (at start (fuel-level ?a ?l1))
        (at start (next ?l2 ?l1)))
 :effect (and
    (at start (not(at ?a ?c1)))
        (at end (at ?a ?c2))
    (at start (not (fuel-level ?a ?l1)))
    (at end (fuel-level ?a ?l2))))
```

- *(zoom ?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel ?l3 - flevel)*: ?a flies from ?c1 to ?c2. Its fuel reserves are reduced from ?l1 to ?l3. (quick)

```
(:durative-action zoom
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?l1 ?l2 ?l3 - flevel)
 :duration (= ?duration 100)
 :condition (and
    (at start (at ?a ?c1))
    (at start (fuel-level ?a ?l1))
    (at start (next ?l2 ?l1))
    (at start (next ?l3 ?l2)))
:effect (and
    (at start (not (at ?a ?c1)))
    (at end (at ?a ?c2))
    (at end (not (fuel-level ?a ?l1)))
```

```
15      (at end (fuel-level ?a ?l3))))
```

- *(refuel ?a - aircraft ?c - city ?l - flevel ?l1 - flevel)*: ?a fills up with fuel in the city ?c. Its fuel reserves are increased from ?l to ?l1.

```
1  (:durative-action refuel
2   :parameters (?a - aircraft
3                ?c - city
4                ?l ?l1 - flevel)
5   :duration (= ?duration 73)
6   :condition (and
7      (at start (fuel-level ?a ?l))
8      (at start (next ?l ?l1))
9      (over all (at ?a ?c)))
10  :effect (and
11     (at end (fuel-level ?a ?l1))
12     (at start (not (fuel-level ?a ?l)))))
```

C.3.5 Sokoban

Description Sokoban is a puzzle in which the player pushes stones around in a warehouse, trying to get them to storage locations

Predicates

- *(at ?t - thing ?l - location)*: The thing ?l is at ?l.
- *(adjacent ?l1 - location ?l2 - location ?d - direction)*: The robot can move from ?l1 to ?l2, and ?l2 is on the ?d of ?l1.
- *(clear ?l - location)*: There are nothing at ?l.

Operators

- *(move ?p - player ?from - location ?to - location ?dir - direction)*: The player moves to the ?dir of the location ?from to the location ?to.

```
1  (:durative-action move
2   :parameters (?p - player
3                ?from ?to - location
4                ?dir - direction)
5   :duration  (= ?duration 1)
6   :condition (and
7      (at start (at ?p ?from))
8      (at start (clear ?to))
9      (over all (adjacent ?from ?to ?dir)))
```

```
 :effect (and
    (at start (not (at ?p ?from)))
    (at start (not (clear ?to)))
    (at end (at ?p ?to))
    (at end (clear ?from))))
```

- *(push ?p - player ?s - stone ?ppos - location ?from - location ?to - location ?dir - direction)*: The player pushes the stone ?s to the location ?to. The initial position of the player was ?ppos and its final position is the initial position of the stone: ?from.

```
(:durative-action push
 :parameters (?p - player
              ?s - stone
              ?ppos ?from ?to - location
              ?dir - direction)
 :duration (= ?duration 1)
 :condition (and
    (at start (at ?p ?ppos))
    (at start (at ?s ?from))
    (at start (clear ?to))
    (over all (adjacent ?ppos ?from ?dir))
    (over all (adjacent ?from ?to ?dir)))
:effect (and
    (at start (not (at ?p ?ppos)))
    (at start (not (at ?s ?from)))
    (at start (not (clear ?to)))
    (at end (at ?p ?from))
    (at end (at ?s ?to))
    (at end (clear ?ppos))))
```

C.4 HTN Planning Domains

C.4.1 Blocksworld

Description HTN version of the classical STRIPS planning domain Blocksworld, where stackable blocks need to be re-assembled on a table. Using its hand, an autonomous agent can stack a block onto a block, unstack a block from a block, put down a block, or pick up a block.

Predicates

- *(on ?x - block ?y - block)*: Thee block ?x is on the block ?y.
- *(ontable ?x - block)*: The block ?x is on the table.

- *(clear ?x - block)*: The block ?x is clear, i.e. there no block on ?x and it is not held .
- *(holding ?x - block)*: The block ?x is held by the autonomous agent.
- *(handempty)*: The autonomous agent does not hold any block.

Compound Tasks

- *(do-put-on ?x - block ?y - block)*: Put ?x on ?y.
- *(do-on-table)*: Put ?x on the table.
- *(do-move ?x - block ?y - block)*: Move ?x on ?y.
- *(do-clear ?x - block)*: Clear block ?x.

Primitive Tasks

- *(pick-up ?x - block)*: The autonomous agent picks a block placed on the table.

```
(:action pick-up
 :parameters (?x - block)
 :precondition (and
      (clear ?x)
      (ontable ?x)
      (handempty))
 :effect (and
      (not (ontable ?x))
      (not (clear ?x))
      (not (handempty))
      (holding ?x)))
```

- *(put-down ?x - block)*: The autonomous agent puts down a block on the table.

```
(:action put-down
 :parameters (?x - block)
 :precondition (holding ?x)
 :effect (and
      (not (holding ?x))
      (clear ?x)
      (handempty)
      (ontable ?x)))
```

- *(stack ?x - block ?y - block)*: The autonomous agent stacks a block on an other block.

```
(:action stack
 :parameters (?x ?y - block)
 :precondition (and
      (holding ?x)
      (clear ?y))
 :effect (and
      (not (holding ?x))
      (not (clear ?y))
      (clear ?x)
      (handempty)
      (on ?x ?y)))
```

- *(unstack ?x - block ?y - block)*: The autonomous agent unstacks a block from an other block.

```
(:action unstack
 :parameters (?x ?y - block)
 :precondition (and
      (on ?x ?y)
      (clear ?x)
      (handempty))
 :effect (and
      (holding ?x)
      (clear ?y)
      (not (clear ?x))
      (not (handempty))
      (not (on ?x ?y)))))
```

Methods

- *(m0-do-put-on ?x - block ?y - block)*: ?x is already on ?y.

```
(:method m0-do-put-on
 :parameters (?x ?y - block)
 :task (do-put-on ?x ?y)
 :precondition (and
    (on ?x ?y)
    (handempty))
 :ordered-subtasks (and))
```

- *(m1-do-put-on ?x - block ?y - block)*: To put ?x on ?y, the agent clears ?x and ?y and moves ?x on ?y.

```
(:method m1-do-put-on
 :parameters (?x  ?y - block)
```

```
 :task (do-put-on ?x ?y)
 :precondition (handempty)
 :ordered-subtasks (and
     (do-clear ?x)
     (do-clear ?y)
     (do-move ?x ?y)))
```

- *(m2-do-on-table ?x - block ?y - block)*: If ?x in on ?y, the agent unstacks ?x from ?y and puts down it on the table.

```
(:method m2-do-on-table
 :parameters (?x ?y - block)
 :task (do-on-table ?x)
 :precondition (and
     (clear ?x)
     (handempty)
     (not (ontable ?x)))
 :ordered-subtasks(and
     (unstack ?x ?y)
     (put-down ?x)))
```

- *(m3-do-on-table ?x - block)*: ?x is already on the table.

```
(:method m3-do-on-table
 :parameters (?x - block)
 :task (do-on-table ?x)
 :precondition (and
     (clear ?x)
     (ontable ?x)
     (handempty))
 :ordered-subtasks (and))
```

- *(m4-do-move ?x - block ?y - block)*: If ?x is on the table, the agents picks up ?x and stacks it on ?y;

```
(:method m4-do-move
 :parameters (?x ?y - block)
 :task (do-move ?x ?y)
 :precondition (and
     (clear ?x)
     (clear ?y)
     (handempty)
     (ontable ?x))
 :ordered-subtasks(and
     (pick-up ?x)
     (stack ?x ?y)))
```

- *(m5-do-move ?x - block ?y - block)*: If ?x is on ?z, the agent unstacks ?x from ?z and stacks it on ?y.

```
(:method m5-do-move
 :parameters (?x ?y ?z - block)
 :task (do-move ?x ?y)
 :precondition (and
    (clear ?x)
    (clear ?y)
    (on ?x ?z)
    (handempty)
    (not (ontable ?x)))
 :ordered-subtasks(and
    (unstack ?x ?z)
    (stack ?x ?y)))
```

- *(m6-do-clear ?x - block)*: ?x is already clear.

```
(:method m6-do-clear
  :parameters (?x - block)
  :task (do-clear ?x)
  :precondition (and
     (clear ?x)
     (handempty))
  :ordered-subtasks(and))
```

- *(m7-do-clear ?x - block ?y - block)*: If ?y is on ?x, the agents clear ?y and puts it on the table.

```
(:method m7-do-clear
 :parameters ( ?x ?y - block )
 :task (do-clear ?x)
 :precondition (and
    (not (clear ?x))
    (on ?y ?x)
    (handempty))
 :ordered-subtasks(and
    (do-clear ?y)
    (unstack ?y ?x)
    (put-down ?y)))
```

C.4.2 Gripper

Description In this domain, there are a robot called robby with grippers and a set of rooms containing balls. The goal is to transport balls from a given room to another.

Predicates

- *(at-robby ?r - room)*: The robot robby is in the room ?r.
- *(at ?b - ball ?r - room)*: The ball ?b is in the room ?r.
- *(free ?g - gripper)*: The Robby's gripper ?g is free.
- *(carry ?b - ball ?g - gripper)*: Robby is carrying the ball ?b with its gripper ?g.

Compound Tasks

- *(move-two-balls ?b1 - ball ?b2 - ball ?r - room)*: Move balls ?b1 ?b2 in ?r.
- *(move-one-ball ?b - ball ?r - room)*: Move ball ?b in ?r.
- *(goto ?r - room)*: Go to room ?r.

Primitive Tasks

- *(move ?from - room ?to - room)*: Robby goes to the room ?to.

```
(:action move
 :parameters  (?from ?to - room)
 :precondition (at-robby ?from)
 :effect (and
      (at-robby ?to)
      (not (at-robby ?from))))
```

- *(pick ?b - ball ?r - room ?g - gripper)*: Robby picks the ball with its gripper.

```
(:action pick
 :parameters (?b - ball ?r - room ?g - gripper)
 :precondition  (and
      (at ?b ?r)
      (at-robby ?r)
      (free ?g))
 :effect (and
      (carry ?b ?g)
      (not (at ?b ?r))
      (not (free ?g))))
```

- *(drop ?b - ball ?r - room ?g - gripper)*: Robby drops the held ball.

```
(:action drop
 :parameters  (?b - ball ?r - room ?g - gripper)
 :precondition  (and
      (carry ?b ?g)
```

```
    (at-robby ?r))
 :effect (and
    (at ?b ?r)
        (free ?g)
        (not (carry ?b ?g)))))
```

Methods

- *(move-two-balls-0 ?b1 - ball ?b2 - ball ?r - room ?rb - room ?g1 - gripper ?g2 - gripper))*: To move ?b1 and ?b2 in the room ?rb. Robby goes to ?ra, picks the balls, goes to ?rb and drops the balls.

```
(:method move-two-balls-0
 :parameters (?b1 ?b2 - ball
              ?r ?rb - room
              ?g1 ?g2 - gripper)
 :task (move-two-balls ?b1 ?b2 ?r)
 :precondition (and
   (at ?b1 ?rb)
   (at ?b2 ?rb))
 :ordered-subtasks (and
   (goto ?rb)
   (pick ?b1 ?rb ?g1)
   (pick ?b2 ?rb ?g2)
   (move ?rb ?r)
   (drop ?b1 ?r ?g1)
   (drop ?b2 ?r ?g2)))
```

- *(move-two-balls-0 ?b1 - ball ?b2 - ball ?r - room ?rb - room ?g1 - gripper ?g2 - gripper))*: To move ?b in the room ?rb. Rooby goes to ?ra, picks the ball, goes to ?rb and drops the ball.

```
(:method move-one-balls-0
 :parameters (?b - ball
              ?r ?rb - room
              ?g - gripper)
 :task (move-one-ball ?b ?r)
 :precondition (and
   (at ?b ?rb))
 :ordered-subtasks (and
   (goto ?rb)
   (pick ?b ?rb ?g)
   (move ?rb ?r)
   (drop ?b ?r ?g)))
```

- *(goto-0 ?r - room)*: Robby is already at ?r.

```
(:method goto-0
 :parameters (?r - room)
 :task (goto ?r)
 :precondition (and
   (at-robby ?r))
 :ordered-subtasks ())
```

- *(goto-1 ?from - room ?to - room)*: Robby goes to the room ?to.

```
(:method goto-1
 :parameters (?from ?to - room)
 :task (goto ?to)
 :precondition (at-robby ?from)
 :ordered-subtasks (move ?from ?to))
```

C.4.3 Zenotravel

A transportation domain involves transporting people around in planes.

Description

Predicates

- *(at ?t - thing ?c - city)*: The thing ?t is at the city ?c.
- *(in ?p - person ?a - aircraft)*: The person ?p is in the aircraft ?a.
- *(fuel-level ?a - aircraft ?l - flevel)*: The fuel level ?l for the aircraft ?a.
- *(next ?l1 - flevel ?l2 - flevel)*: If the flevel is ?l1 then the next level will be ?l2.
- *(max ?l - flevel)*: The max level of fuel is ?l.
- *(min ?l - flevel)*: The min level of fuel is ?l.
- *(not-min ?l - flevel)*: The min level of fuel is not ?l.

Compound Tasks

- *(transport-person ?p - person ?c - city)*: Transport ?p at ?c.
- *(transport-aircraft ?a - aircraft ?c - city)*: Transport ?a at ?c.

Primitive Tasks

- *(board ?p - person ?a - aircraft ?c - city)*: Passenger ?p boards in ?a at the city ?c.

```
(:action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and
     (at ?p ?c)
     (at-aircraft ?a ?c))
 :effect (and
     (not (at ?p ?c))
     (in ?p ?a)))
```

- *(debark ?p - person ?a - aircraft ?c - city)*: Passenger ?p debarks from ?a at the city ?c.

```
(:action debark
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and
     (in ?p ?a)
     (at-aircraft ?a ?c))
 :effect (and
     (at ?p ?c)
     (not (in ?p ?a))))
```

- *(fly ?a - aircraft ?c1 - city ?c2 - city ?l1 - flevel ?l2 - flevel)*: ?a flies from ?c1 to ?c2. Its fuel reserves are reduced from ?l1 to ?l2.

```
(:action fly
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?l1 ?l2 - flevel)
 :precondition (and
     (at-aircraft ?a ?c1)
     (fuel-level ?a ?l1)
     (next ?l2 ?l1))
 :effect (and
     (not (at-aircraft ?a ?c1))
     (at-aircraft ?a ?c2)
     (not (fuel-level ?a ?l1))
     (fuel-level ?a ?l2)))
```

- *(refuel ?a - aircraft ?l - flevel ?l1 - flevel)*: ?a fills up with fuel in the city ?c. Its fuel reserves are increased from ?l to ?l1.

```
(:action refuel
 :parameters (?a - aircraft ?l ?l1 - flevel)
 :precondition (and
      (fuel-level ?a ?l)
      (next ?l ?l1))
 :effect (and
      (fuel-level ?a ?l1)
      (not (fuel-level ?a ?l)))))
```

Methods

- *(transport-person-0 ?a - aircraft ?c1 - city ?c2 - city ?p - person)*: ?a transports ?p from ?c1 to ?c2. First of all, ?a goes to ?c1, then ?p boards in ?a, then ?p goes to ?c2 and finally ?p debarks from ?a.

```
(:method transport-person-0
 :parameters (?a - aircraft
              ?c1 ?c2 - city
              ?p - person)
 :task (transport-person ?p ?c2)
 :precondition (and
         (at ?p ?c1)
    (not(at ?p ?c2)))
 :ordered-subtasks (and
    (transport-aircraft ?a ?c1)
         (board ?p ?a ?c1)
         (transport-aircraft ?a ?c2)
         (debark ?p ?a ?c2)))
```

- *(transport-person-1 ?c - city ?p - person)*: ?p is already in its destination.

```
(:method transport-person-1
:parameters (?c - city ?p - person)
:task (transport-person ?p ?c)
:precondition (at ?p ?c)
:ordered-subtasks )
```

- *(transport-aircraft-0 ?a - aircraft ?c - city)*: ?a is already in its destination.

```
(:method transport-aircraft-0
 :parameters (?a - aircraft ?c - city)
 :task (transport-aircraft ?a ?c)
 :precondition (at ?a ?c)
 :ordered-subtasks )
```

- *(transport-aircraft-1 ?a - aircraft ?c - city ?other - city ?l1 - flevel ?l2 - flevel)*: ?a flies from ?other to ?c.

```
(:method transport-aircraft-1
 :parameters (?a - aircraft
              ?c ?other - city
              ?l1 ?l2 - flevel)
 :task (transport-aircraft ?a ?c)
 :precondition (and
    (not (at ?a ?c))
         (at ?a ?other)
    (fuel-level ?a ?l1)
         (next ?l2 ?l1)
    (not-min ?l1))
 :ordered-subtasks (fly ?a ?other ?c ?l1 ?l2))
```

- *(transport-aircraft-2 ?a - aircraft ?c - city ?other - city ?l1 - flevel ?lmax - flevel)*: ?a flies from ?other to ?c. Before its departure, ?a fills his fuel tank.

```
(:method transport-aircraft-2
 :parameters (?a - aircraft
              ?c ?other - city
              ?l1 ?lmax - flevel)
 :task (transport-aircraft ?a ?c)
 :precondition (and
    (not (at ?a ?c))
    (at ?a ?other)
    (fuel-level ?a ?l1)
    (min ?l1)
    (max ?lmax))
 :ordered-subtasks (and
     (refuel ?a ?l1 ?lmax)
     (transport-aircraft ?a ?c)))
```

C.4.4 Childsnack

Description This domain plan how to make and serve sandwiches for a group of children in which some are allergic to gluten.

Predicates

- *(at-kitchen-bread ?b - bread-portion)*: The bread portion ?b is at the kitchen.
- *(at-kitchen-content ?c - content-portion)*: The content portion ?c is at the kitchen.

- *(at-kitchen-sandwich ?s - sandwich)*: The sandwich ?s is at the kitchen.
- *(no-gluten-content ?c - content-portion)*: The content portion ?c is gluten free.
- *(no-gluten-bread ?b - bread-portion)*: The bread portion ?b is gluten free.
- *(ontray ?s - sandwich ?t - tray)*: The sandwich ?s is on ?t.
- *(no-gluten-sandwich ?s - sandwich)*: The sandwich ?s is gluten free.
- *(allergic-gluten ?c - child)*: The child is allergic to gluten.
- *(not-allergic-gluten ?c - child)*: The child is not allergic to gluten.
- *(served ?c - child)*: The child has been served.
- *(waiting ?c - child ?p - place)*: The child wait for its sandwich at ?p.
- *(at ?t - tray ?p - place)*: ?t is at ?p.
- *(not-exist ?s - sandwich)*: The sandwich has not yet been prepared.

Compound Tasks

- *(serve ?c - child)*: Make a sandwich for the child ?c.

Primitive Tasks

- *(make-sandwich-no-gluten ?s - sandwich ?b - bread-portion ?c content-portion)*: Make a sandwich with gluten free bread and content.

```
(:action make-sandwich-no-gluten
 :parameters (?s - sandwich
              ?b - bread-portion
              ?c - content-portion)
 :precondition (and
    (at-kitchen-bread ?b)
         (at-kitchen-content ?c)
         (no-gluten-bread ?b)
         (no-gluten-content ?c)
         (not-exist ?s))
 :effect (and
    (not (at-kitchen-bread ?b))
    (not (at-kitchen-content ?c))
    (at-kitchen-sandwich ?s)
    (no-gluten-sandwich ?s)
    (not (not-exist ?s))))
```

- *(make-sandwich ?s - sandwich ?b - bread-portion ?c content-portion)*: Make a sandwich with bread and content containing gluten.

```
(:action make-sandwich
 :parameters (?s - sandwich
              ?b - bread-portion
              ?c - content-portion)
 :precondition (and
    (at-kitchen-bread ?b)
        (at-kitchen-content ?c)
        (not-exist ?s))
 :effect (and
    (not (at-kitchen-bread ?b))
    (not (at-kitchen-content ?c))
    (at-kitchen-sandwich ?s)
    (not (not-exist ?s))))
```

- *(put-on-tray ?s - sandwich ?t - tray ?kitchen - place)*: Put a sandwich on the tray of the kitchen.

```
(:action put-on-tray
 :parameters (?s - sandwich
              ?t - tray
              ?kitchen - place)
 :precondition (and
    (at-kitchen-sandwich ?s)
    (at ?t ?kitchen))
 :effect (and
    (not (at-kitchen-sandwich ?s))
    (ontray ?s ?t)))
```

- *(serve-sandwich-no-gluten ?s - sandwich ?c - child ?t - tray ?p - place)*: Serve a gluten free sandwich to a child.

```
(:action serve-sandwich-no-gluten
 :parameters (?s - sandwich
              ?c - child
              ?t - tray
              ?p - place)
 :precondition (and
    (allergic-gluten ?c)
    (ontray ?s ?t)
    (waiting ?c ?p)
    (no-gluten-sandwich ?s)
    (at ?t ?p))
 :effect (and
```

```
    (not (ontray ?s ?t))
    (served ?c)
    (not(waiting ?c ?p))))
```

- *(serve-sandwich ?s - sandwich ?c - child ?t - tray ?p - place)*: Serve a sandwich containing gluten to a child.

```
(:action serve-sandwich
:parameters (?s - sandwich
             ?c - child
             ?t - tray
             ?p - place)
 :precondition (and
    (not-allergic-gluten ?c)
    (ontray ?s ?t)
    (waiting ?c ?p)
    (at ?t ?p))
 :effect (and
    (not (ontray ?s ?t))
    (served ?c)
    (not(waiting ?c ?p))))
```

- *(move-tray ?t - tray ?p1 - place ?p2 - place)*: Move the tray from ?p1 to ?p2.

```
(:action move-tray
 :parameters (?t - tray
              ?p1 ?p2 - place)
 :precondition (and
    (at ?t ?p1))
 :effect (and
    (not (at ?t ?p1))
    (at ?t ?p2)))
```

Methods

- *(serve-0 ?c - child ?s - sandwich ?p2 - place ?t - tray ?b - bread-portion ?cont - content-portion ?kitchen - place)*: Prepare and serve a gluten free sandwich to an allergic child.

```
(:method serve-0
 :parameters (?c - child
              ?s - sandwich
              ?p2 - place
              ?t - tray
              ?b - bread-portion
```

```
                ?cont - content-portion
                ?kitchen - place)
 :task (serve ?c)
 :precondition (and
    (allergic-gluten ?c)
        (not-exist ?s)
        (waiting ?c ?p2)
        (no-gluten-bread ?b)
        (no-gluten-content ?cont))
 :ordered-subtasks (and
    (make-sandwich-no-gluten ?s ?b ?cont)
    (put-on-tray ?s ?t ?kitchen)
        (move-tray ?t ?kitchen ?p2)
        (serve-sandwich-no-gluten ?s ?c ?t ?p2)
        (move-tray ?t ?p2 ?kitchen)))
```

- *(serve-1 ?c - child ?s - sandwich ?p2 - place ?t - tray ?b - bread-portion ?cont - content-portion ?kitchen - place)*: Prepare and serve a sandwich to a non-allergic child.

```
(:method serve-1
 :parameters (?c - child
              ?s - sandwich
              ?p2 - place
              ?t - tray
              ?b - bread-portion
              ?cont - content-portion
              ?kitchen - place)
 :task (serve ?c)
 :precondition (and
    (not-allergic-gluten ?c)
        (not-exist ?s)
        (waiting ?c ?p2)
        (not (no-gluten-bread ?b))
        (not(no-gluten-content ?cont)))
 :ordered-subtasks (and
    (make-sandwich ?s ?b ?cont)
    (put-on-tray ?s ?t ?kitchen)
        (move-tray ?t ?kitchen ?p2)
        (serve-sandwich ?s ?c ?t ?p2)
        (move-tray ?t ?p2 ?kitchen)))
```

C.4.5 Transport

Description A transport domain where each vehicle can transport some packages depending on its capacity and moving has a cost depending on the length of the road

Predicates

- *(road ?l1 - location ?l2 - location)*: There are a road between ?l1 and ?l2.
- *(at ?x - locatable ?l - location)*: ?x is at ?l.
- *(in ?p - package ?v - vehicle)*: ?p is in ?v.
- *(capacity ?v - vehicle ?n - capacity-number)*: The capacity of ?v is ?n.
- *(capacity-predecessor ?n0 - capacity-number ?n1 - capacity-number)*: ?n1 = ?n0 - 1.

Compound Tasks

- *(deliver ?p - package ?l - location)*: Deliver ?p at ?l.
- *(get-to ?v - vehicle ?l - location)*: Go to location ?l with ?v.
- *(load ?v - vehicle ?l - location ?p - package)*: Load ?p in ?v at ?l.
- *(unload ?v - vehicle ?l - location ?p - package)*: Unload ?p from ?v at ?l.

Primitive Tasks

- *(drive ?v - vehicle ?l1 - location ?l2 - location)*: ?v goes to ?l2 from ?l1. A road connecting ?l1 to ?l2 is required.

```
(:action drive
 :parameters (?v - vehicle
              ?l1 ?l2 - location)
 :precondition (and
        (at ?v ?l1)
        (road ?l1 ?l2))
 :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)))
```

- *(pick-up ?v - vehicle ?l - location ?p - package ?s1 - capacity-number ?s2 - capacity-number)*: Pick ?p in ?v.

```
(:action pick-up
 :parameters (?v - vehicle
              ?l - location
              ?p - package
              ?s1 ?s2 - capacity-number)
:precondition (and
    (at ?v ?l)
    (at ?p ?l)
    (capacity-predecessor ?s1 ?s2)
    (capacity ?v ?s2))
 :effect (and
    (not (at ?p ?l))
    (in ?p ?v)
    (capacity ?v ?s1)
    (not (capacity ?v ?s2))))
```

- *(drop ?v - vehicle ?l - location ?p - package ?s1 - capacity-number ?s2 - capacity-number)*: Drop ?p from ?v.

```
(:action drop
 :parameters (?v - vehicle
              ?l - location
              ?p - package
              ?s1 ?s2 - capacity-number)
 :precondition (and
    (at ?v ?l)
    (in ?p ?v)
    (capacity-predecessor ?s1 ?s2)
    (capacity ?v ?s1))
:effect (and
    (not (in ?p ?v))
    (at ?p ?l)
    (capacity ?v ?s2)
    (not (capacity ?v ?s1))))
```

Methods

- *(deliver-0 ?l1 - location ?l2 - location ?p - package ?v - vehicle)*: Deliver ?p to a given location. ?v goes to the location of ?p, picks ?p, goes to the final location and drops ?p.

```
(:method load-0
 :parameters (?l1 ?l2 - location
              ?p - package
              ?v - vehicle)
```

```
 :task (deliver ?p ?l2)
 :precondition
 :ordered-subtasks (and
        (get-to ?v ?l1)
        (load ?v ?l1 ?p)
        (get-to ?v ?l2)
        (unload ?v ?l2 ?p)))
```

- *(load-0 ?l - location ?p - package ?s1 - capacity-number ?s2 - capacity-number ?v - vehicle)*: Load ?p in ?v.

```
(:method load-0
 :parameters (?l - location
              ?p - package
              ?s1 ?s2 - capacity-number
              ?v - vehicle)
 :task (load ?v ?l ?p)
 :precondition
 :ordered-subtasks (pick-up ?v ?l ?p ?s1 ?s2))
```

- *(unload-0 ?l - location ?p - package ?s1 - capacity-number ?s2 - capacity-number ?v - vehicle)*: Unload ?p in ?v.

```
(:method unload-0
 :parameters (?l - location
              ?p - package
              ?s1 ?s2 - capacity-number
              ?v - vehicle)
 :task (load ?v ?l ?p)
 :precondition
 :ordered-subtasks (pick-up ?v ?l ?p ?s1 ?s2))
```

- *(get-to-0 ?l - location ?v - vehicle)*: ?v is already at ?l.

```
(:method get-to-0
 :parameters (?l - location ?v - vehicle)
 :task (get-to ?v ?l)
 :precondition (at ?v ?l)
 :ordered-subtasks )
```

- *(get-to-1 ?l1 - location ?l2 - location ?v - vehicle)*: ?v goes to a location ?l1 connected to ?l2, and moves from ?l1 to ?l2.

```
(:method get-to-1
 :parameters (?l1 ?l2 - location
              ?v - vehicle)
```

```
4  :task (get-to ?v ?l2)
5  :precondition
6  :ordered-subtasks (and
7       (get-to ?v ?l1)
8       (drive ?v ?l1 ?2)))
```

Appendix D

Résumé

Contents

D.1 Introduction

Le domaine de l'intelligence artificielle (IA) vise à développer des agents autonomes capables de percevoir, d'apprendre et d'agir sans aucune intervention humaine pour accomplir des tâches complexes. Comme l'indique (Russell and Norvig, 2021), un agent autonome est un système intelligent qui peut décider de ce qu'il faut faire pour réaliser une tâche. Un agent autonome perçoit son environnement, planifie les meilleures actions possibles et les exécute. Pour planifier les meilleures actions, l'agent autonome doit donc être capable de prendre des décisions. Il existe plusieurs approches permettant aux agents autonomes de prendre des décisions. L'une d'entre elles est la *planification automatique* (Fikes and Nilsson, 1971; Ghallab et al., 2004).

D.1.1 Planification automatique

L'objectif de la planification est la résolution de *problèmes de planification*. La description d'un problème de planification se fait de manière déclarative. La déclaration d'un problème de planification est composée de : (1) l'état initial de l'environnement, (2) un but et (3) l'ensemble des actions qui peuvent

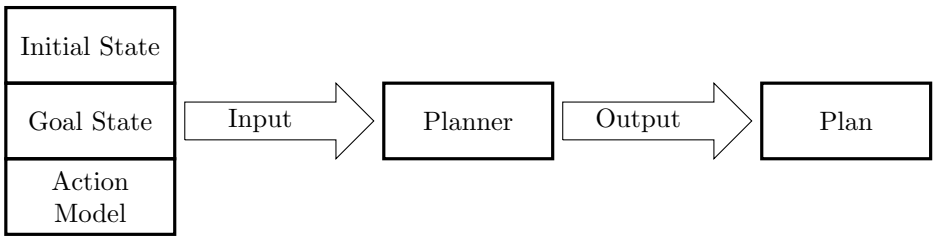


Figure D.1: Résolution d'un problème de planification

être exécutées par l'agent autonome pour atteindre le but spécifié. Pour résoudre le problème de planification, l'agent autonome recherche une séquence d'actions, appelée plan, permettant d'atteindre l'état but à partir de l'état initial. La résolution des problèmes de planification est basée sur des modèles descriptifs des actions. Ils décrivent quel état ou ensemble d'états possibles peut résulter de l'exécution d'une action. Plus précisément, les *modèles d'actions* sont généralement utilisés pour décrire les actions. Les modèles d'actions définissent les actions en termes de *pré-conditions* et d'*effets*. Les pré-conditions expriment les propriétés de l'environnement qui doivent être satisfaites pour exécuter l'action et les effets expriment les conséquences de l'exécution de l'action sur l'environnement. En pratique, les modèles d'actions sont encodés manuellement à l'aide d'un langage déclaratif tel que le PDDL (Planning Domain Description Language) (Ghallab et al., 1998). Enfin, la Figure D.1 donne l'architecture traditionnelle pour résoudre un problème de planification : un solveur, appelé *planificateur*, prend en entrée l'état initial, l'état but et le modèle d'actions et retourne le plan solution.

La manière classique de représenter des modèles d'actions est la modélisation STRIPS (STanford Research Institute Problem Solver) (Fikes and Nilsson, 1971). Les pré-conditions, les effets et les états initiaux et buts sont représentés sous la forme d'un ensemble de propositions logiques décrivant l'environnement. La modélisation STRIPS se repose sur plusieurs hypothèses restrictives :

- L'environnement est déterministe et entièrement observable. L'agent autonome connaît à tout moment l'état dans lequel il se trouve et peut donc prédire l'état dans lequel il se trouvera après l'exécution d'une action.
- Le but est spécifié en utilisant un ensemble de propriétés, c'est-à-dire des propositions logiques, que l'état final atteint par l'agent autonome doit satisfaire.
- L'environnement est statique. Seules les actions exécutées par l'agent autonome modifient l'état de l'environnent. L'agent autonome est seul et l'environnent n'a pas de dynamique interne.
- L'exécution des actions est atomique. Les actions n'ont pas de durée, les effets sont appliqués instantanément et la concurrence entre les actions n'est pas prise en compte.

- Les pré-conditions et les effets sont des ensembles de propositions logiques.

Plusieurs extensions de la modélisation STRIPS ont été proposées pour relâcher certaines hypothèses. Par exemple, les problèmes de planification temporelle (Fox and Long, 2003) permettent de modéliser des actions qui peuvent être exécutées simultanément. Cela peut être utile pour des robots à bras multiples qui peuvent manipuler plusieurs objets simultanément. Un autre exemple est celui des problèmes de planification HTN (Erol et al., 1994) qui déclarent des tâches plutôt que des actions. Il existe deux types de tâches. Les tâches primitives, similaires aux actions classiques, et les tâches complexes. Les tâches complexes sont des tâches qui, lors de la résolution du problème de planification, sont décomposées en sous-tâches (primitives et/ou complexes). Un autre exemple est celui des problèmes de planification ADL (Pednault, 1994) qui permettent de modéliser des pré-conditions et des effets plus complexes avec des quantificateurs logiques, des effets conditionnels (les effets s'appliquent lorsque certaines conditions sont satisfaites), etc. Un dernier exemple est celui des problèmes de planification multi-agents (Brenner, 2003) qui permettent de représenter des problèmes de planification où plusieurs agents autonomes interagissent pour atteindre un but. Cela peut être utile si nous avons plusieurs robots qui interagissent afin d'atteindre un objectif.

Plus nous relâchons des hypothèses de la modélisation STRIPS, plus les modèles d'actions sont difficiles à concevoir "à la main". D'un autre coté, plusieurs outils basés sur la planification automatique ont été développés, nous pouvons citer par exemple l'aérospatial (Fisher et al., 2000; Backes et al., 2004; Bresina et al., 2005), les véhicules autonomes (Urmson and Whittaker, 2008), la logistique (Cross and Walker, 1994), la robotique (Dvorak et al., 2014; Lallement et al., 2018; Liang et al., 2022), l'industrie (Hoffmann et al., 2009), la cybersécurité (Edelkamp et al., 2009). Les hypothèses de la modélisation STRIPS étant trop restrictives pour ces applications, nous avons donc besoins relâcher certaines d'entre elles ce qui rend la conception de ces modèles d'actions plus difficile. Dans cette thèse, nous nous intéressons aux outils permettant de faciliter l'acquisition de ces modèles d'actions.

D.1.2 Revue de littérature

Des outils d'ingénierie des connaissances facilitant l'écriture de modèles d'actions ont été développés. Ces outils permettent de vérifier les erreurs de syntaxe, de visualiser le modèle, etc. Cependant, ces outils nécessitent une grande expertise en planification automatique et des connaissances en génie logiciel. Des approches d'apprentissage automatique ont également été proposées pour générer automatiquement des modèles d'actions (Arora et al., 2018a; Celorrio et al., 2012; Jilani et al., 2014). Généralement, les approches d'apprentissage automatique sont utilisées pour des applications déjà existantes : soit pour automatiser des processus, comme un processus industriel par

exemple, soit pour bénéficier de la flexibilité de la planification automatique, pour faciliter la maintenance de ces applications par exemple. Ces approches prennent en entrée un jeu de données d'apprentissage et apprennent un modèle d'actions. Un jeu de données d'apprentissage est une collection de données permettant d'apprendre un modèle. Dans le contexte de la planification automatique, les jeux de données d'apprentissage sont généralement des exemples d'exécutions d'agents autonomes. Ces approches sont prometteuses et certaines d'entre elles réduisent l'effort humain (Zhuo et al., 2010a) nécessaire à l'acquisition des modèles d'actions. Cependant, ces approches ne sont pas assez efficaces pour être utilisées dans des applications réelles.

Tout d'abord, la plupart de ces approches d'apprentissage apprennent des modèles d'actions STRIPS, et comme nous l'avons vu précédemment, les modèles d'actions STRIPS sont basés sur des hypothèses trop restrictives pour être utilisées dans des applications réelles. Ensuite, l'acquisition des jeux de données d'apprentissage est un processus difficile et coûteux. En effet, il faut générer les exemples, les exécuter, stocker les résultats de l'exécution. De plus, certaines approches nécessitent également d'analyser ces exemples pour en extraire une représentation symbolique. Il existe principalement deux types de jeux de données : (1) les traces de plans et (2) les marches aléatoires. La plupart des approches d'apprentissage utilisent des traces d'exécution ayant résolues un problème comme données d'apprentissage ((Yang et al., 2007; Aineto et al., 2019; Segura-Muros et al., 2018; Kucera and Barták, 2018)) et quelques approches utilisent des traces d'exécution générées aléatoirement ((Rodrigues et al., 2010a; Mourão et al., 2012)).

Les traces de plans sont des traces d'exécution d'une séquence d'actions ayant accompli une tâche donnée. Par exemple, pour un véhicule autonome, une trace de plan sera la séquence d'actions permettant au véhicule de se rendre de Grenoble à Lyon. Le principal problème avec ce type de traces est qu'elles sont biaisées par la tâche à réaliser, et si nous avons trop peu de données, il y a un risque de sur-apprentissage. Par exemple, pour notre véhicule autonome, si notre jeu de données d'apprentissage ne contient que des trajets où les deux villes sont reliées par une autoroute, il est possible que le modèle d'actions appris ne puisse pas être utilisé pour des trajets reliant deux villes qui ne sont pas reliées par une autoroute. Pour limiter le risque de sur-apprentissage, nous avons donc besoin d'un grand nombre de traces d'exécution acquises à partir d'une grande variété de tâches. Cependant, comme nous l'avons mentionné précédemment, l'acquisition de ces traces est difficile et coûteuse. Les marches aléatoires sont des séquences d'actions générées aléatoirement. L'avantage de ce type de trace est qu'elle n'est pas biaisée par une tâche. En générant aléatoirement des séquences, nous pouvons couvrir un grand nombre d'exemples d'exécution tout en limitant le risque de sur-apprentissage. De plus, les marches aléatoires permettent d'acquérir des séquences d'actions infaisables, c'est-à-dire des séquences d'actions où une ou plusieurs actions ne sont pas réalisables. De plus, généralement en plus des séquences d'actions, les traces d'exécution contiennent des observations : c'est-à-dire les différents états

de l'environnement observés lors de l'exécution des séquences d'actions. Ces états sont des ensembles de propositions logiques décrivant l'environnement. En pratique, obtenir des observations complètes et sans bruit est une tâche difficile et généralement les observations seront partielles et bruitées. Un état observé partiel est un état dans lequel les valeurs de certaines propositions sont inconnues. De même, un état observé bruité est un état où les valeurs de certaines propositions sont erronées. Cependant, la majorité des approches d'apprentissage ne sont capables de traiter que des observations complètes ou partielles et non bruitées (Wang, 1995; Yang et al., 2007; Aineto et al., 2019). De plus, même si certaines approches ((Mourão et al., 2012; Segura-Muros et al., 2018; Rodrigues et al., 2010a)) sont capables d'apprendre à partir d'observations partielles et bruyantes, peu d'approches sont capables de gérer des niveaux très élevés comme on peut en rencontrer dans les applications du monde réel. Enfin, les approches d'apprentissage ne sont généralement pas capables d'apprendre des modèles d'actions utilisables par les planificateurs. Une étape de relecture par un expert en planification est généralement nécessaire.

Dans cette thèse, nous soutenons que, pour être efficace, une approche d'apprentissage doit aborder le triple problématique suivante :

1. **Sortie:** Comme nous l'avons mentionné plus haut, la planification STRIPS est trop restrictive pour les applications du monde réel. Nous devons donc proposer des approches apprenant des modèles d'actions moins restrictifs.

2. **Entrée:** L'acquisition de l'ensemble de données d'apprentissage doit être peu coûteuse et nécessiter peu d'efforts humains.

3. **Performance:** Le modèle d'actions appris doit être *correct*. Un modèle d'actions est *correct* s'il peut être utilisé par les planificateurs pour résoudre des problèmes de planification sans nécessiter une étape de relecture. De plus, même avec des entrées ne nécessitant pas beaucoup d'efforts humains, l'acquisition de l'ensemble de données d'apprentissage est difficile et coûteuse. L'approche d'apprentissage doit nécessiter peu de données pour apprendre le modèle d'actions. Enfin, l'approche d'apprentissage doit être robuste aux observations partielles et bruitées, c'est-à-dire que l'approche d'apprentissage doit apprendre des modèles d'actions corrects même avec un niveau élevé de partialité et de bruit dans les observations.

Dans cette thèse, nous nous attaquons au défi de recherche suivant : *Comment apprendre automatiquement des modèles d'actions à partir d'observations partielles et bruitées ?*

D.2 Contribution

Cette thèse contribue au domaine de la planification automatique, et plus particulièrement au domaine de l'acquisition de modèles d'actions.

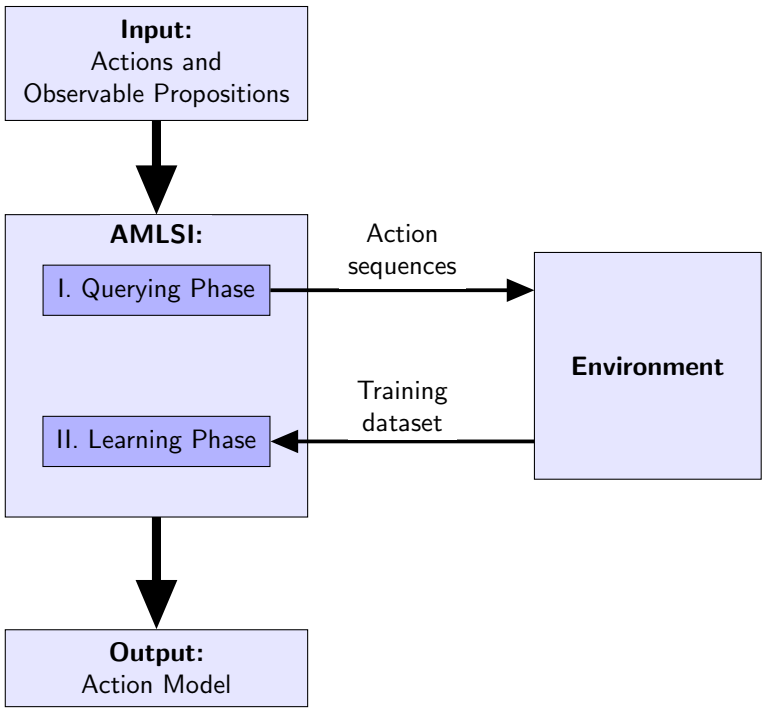


Figure D.2: Aperçu de l'approche AMLSI

Dans cette thèse, nous proposons AMLSI (Action Model Learning with State machine Interaction), une approche d'apprentissage pour l'acquisition de modèles d'actions. La figure D.2 donne un aperçu de cette approche. L'idée centrale de l'approche AMLSI est d'interagir avec l'environnement dans lequel l'agent autonome devra résoudre des problèmes de planification pour apprendre le modèle d'actions : AMSLI teste différentes actions, observe l'évolution de l'environnement lorsque ces actions sont exécutées et apprend le modèle d'actions à partir de ses observations. Cette approche est divisée en deux phases :

Phase de requêtes AMLSI teste des séquences d'actions faisables et infaisables dans l'environnement et perçoit des observations. Ces observations peuvent être partielles et/ou bruitées. L'approche AMLSI teste à la fois les séquences d'actions faisables et infaisables car les séquences d'actions infaisables permettent de minimiser le nombre d'actions (faisables) à exécuter dans l'environnement et donc de minimiser le coût d'acquisition des données d'apprentissage. De plus, les séquences d'actions seront générées de manière aléatoire afin d'éviter le problème de sur-apprentissage. Cette phase prend en entrée l'ensemble des actions que l'agent autonome peut exécuter et l'ensemble des propositions observables décrivant l'environnement. Cette phase de requêtes permet à AMLSI de construire un jeu de données d'apprentissage qui sera ensuite utilisé comme entrée de la phase d'apprentissage.

Phase d'apprentissage C'est durant cette phase qu'AMLSI apprend le modèle d'actions. Pour apprendre le modèle d'actions, AMLSI va s'appuyer sur l'induction grammatical régulière. Comme mentionné dans le chapitre 4, les problèmes de planification sont liés à des machines à états et ces machines à états sont équivalentes à des grammaires régulières. De plus, comme mentionné dans le chapitre 3, l'induction grammatical régulière est un problème bien défini (Gold, 1967; De La Higuera, 2010), et de nombreux algorithmes ont été proposés pour le résoudre comme, par exemple l'algorithme RPNI (Oncina and Garcia, 1992).

- **Apprentissage** STRIPS **:** Dans un premier temps, nous montrons dans le chapitre 4 que notre approche est capable d'apprendre des modèles d'actions STRIPS corrects. L'idée centrale de notre approche est d'apprendre la machine à états reliée au problème de planification en utilisant des algorithmes d'induction grammatical régulière puis d'induire le modèle d'actions à partir de cette machine à états. De plus, comme les problèmes de planification sont déclarés à l'aide d'un domaine de planification, l'approche AMLSI représente le modèle d'actions STRIPS sous la forme d'un domaine de planification PDDL. Nous montrons expérimentalement que l'approche AMLSI apprend des modèles d'actions précis même avec un niveau élevé de partialité et/ou de bruit dans les observations. De plus, comme l'approche AMLSI utilise à la fois des séquences d'actions faisables et infaisables, elle nécessite peu de données d'apprentissage pour apprendre des modèles d'actions précis. Enfin, nous montrons que l'approche AMLSI est plus performante que les approches de l'état de l'art.

- **Apprentissage incrémental :** Ensuite, nous présentons dans le chapitre 5 IncrAMLSI, une extension incrémentale de l'approche AMLSI. Proposer une approche incrémentale permet de répondre à deux besoins. Tout d'abord, l'acquisition de données est un processus long, les données d'apprentissage deviennent disponibles progressivement, et sont difficiles et coûteuses à obtenir. Il est donc important de pouvoir mettre à jour les modèles d'actions appris en fonction des nouvelles données entrantes sans recommencer le processus d'apprentissage depuis le début. Aussi, il est important de savoir quand arrêter l'apprentissage, c'est-à-dire avoir un critère d'arrêts, pour savoir quand arrêter le processus d'acquisition des données et donc minimiser le coût de ce processus.

- **Apprentissage temporelle :** Ensuite, nous présentons dans le chapitre 6 TempAMLSI, une approche apprenant des modèles d'actions temporels. Plusieurs planificateurs (Fox and Long, 2002a; Halsey et al., 2004; Celorrio et al., 2015; Furelos Blanco et al., 2018) résolvent des problèmes de planification temporels en traduisant les problèmes temporels vers en problèmes STRIPS. TempAMLSI réutilise cette idée pour l'apprentissage de modèle : TempAMLSI apprend un modèle STRIPS en utilisant

l'approche AMLSI puis traduit le modèle STRIPS en un modèle temporel. Nous montrons dans ce chapitre que TempAMLSI apprend des modèles contenant différentes formes de concurrences tel que les enveloppes (Coles et al., 2009).

- **Apprentissage HTN :** Enfin, nous présentons dans le chapitre 7 HierAMLSI, une approche apprenant des modèles de tâches HTN. Comme pour TempAMLSI, HierAMLSI est basé sur l'approche AMLSI et utilise l'apprentissage STRIPS pour apprendre des modèles moins restrictifs. Nous montrons dans ce chapitre que HierAMLSI est capable d'apprendre le modèle de tâches primitives, le modèle de tâches complexes et les deux à la fois.

D.3 Perspectives

Nous allons maintenant voir les différentes perspectives possibles pour nos travaux.

D.3.1 Extension de l'approche AMLSI

Une première perspective consiste à étendre l'approche AMLSI.

La principale limite de l'approche AMLSI est l'expressivité des modèles d'actions appris. Bien que l'approche AMLSI apprenne des modèles moins restrictifs que les modèles STRIPS, les pré-conditions et les effets des actions sont STRIPS, c'est-à-dire que les pré-conditions et les effets sont des ensembles de propositions logiques. Cependant, il est possible d'avoir des pré-conditions et des effets plus complexes, comme la modélisation ADL par exemple. Au cours de cette thèse, nous avons testé une extension d'AMLSI pour les modèles d'actions ADL. Cependant, cette extension n'a pas été capable d'apprendre des modèles d'actions corrects lorsque les observations étaient bruitées et/ou partielles. De plus, l'approche AMLSI n'apprend pas les aspects numériques. En effet, certaines modélisations permettent de prendre en compte des caractéristiques numériques tel que les effets probabilistes (Younes and Littman, 2004; Sanner, 2010), les fonctions numériques (Fox and Long, 2003, 2002b, 2006). Une première perspective pour notre approche serait de l'étendre pour répondre à ces limitations.

Comme mentionné dans le chapitre 2), les problèmes de planification moins restrictifs que les problèmes STRIPS sont plus complexes à résoudre. Des techniques basées sur l'apprentissage automatique ont été proposées pour faciliter la résolution de ces problèmes, comme les macro-actions : (Dawson and Siklóssy, 1977; Korf, 1985; Botea et al., 2005; Castellanos-Paez et al., 2018), la planification généralisée (Minton, 2012; Borrajo and Veloso, 1997; de la Rosa et al., 2007, 2008) et l'apprentissage d'heuristique (De La Rosa et al., 2009; Yoon et al., 2006). L'objectif de ces méthodes est d'apprendre des macro-actions ou

des heuristiques facilitant la résolution de problèmes de planification à partir de traces d'exécution. Par exemple, les macro-actions sont des actions composées de plusieurs actions. Par exemple, pour Blocksworld, nous pourrions avoir la macro action *pick-up-stack* composée des actions *pick-up* et *stack*. Pour apprendre ces macros, les approches prennent généralement en entrée un ensemble de plans de solution et renvoient les macros qui ont souvent été observées. Une perspective possible serait de réutiliser l'idée centrale d'AMLSI pour apprendre ces macros et heuristiques : apprendre la machine d'état liée au problème de planification en utilisant des algorithmes d'induction de grammaire et induire à partir de cette machine d'état ces macros et heuristiques. Plus généralement, nous pourrions étendre l'approche AMLSI afin de pouvoir à la fois apprendre des modèles d'actions et apprendre à résoudre efficacement les problèmes de planification en utilisant les modèles d'actions appris.

D.3.2 Applications

Comme nous l'avons vu précédemment, le principal intérêt des approches apprenant des modèles d'actions est de faciliter l'acquisition de ces modèles afin de les utiliser dans des applications réelles tel que l'aérospatial (Fisher et al., 2000; Backes et al., 2004; Bresina et al., 2005), les véhicules autonomes (Urmson and Whittaker, 2008), la logistique (Cross and Walker, 1994), la robotique (Dvorak et al., 2014; Lallement et al., 2018; Liang et al., 2022), l'industrie (Hoffmann et al., 2009), la cybersécurité (Edelkamp et al., 2009). Une perspective directe pour notre travail serait donc d'utiliser l'approche AMLSI pour faciliter l'acquisition de ces modèles d'actions.

Une deuxième application serait de profiter de l'aspect interactif de l'approche AMLSI pour faciliter le développement d'outils utilisant des interactions avec l'utilisateur comme par exemple la programmation de robots, et plus précisément, la programmation par démonstration de robots (Billard et al., 2008). La programmation par démonstration est une technique de programmation qui permet à l'utilisateur d'enseigner de nouvelles compétences à un robot en montrant des démonstrations de ces différentes compétences. AMLSI peut être adapté à ce contexte pour apprendre des modèles d'actions. Comme nous l'avons vu précédemment, la première étape de l'approche AMLSI est une phase de requêtes. Dans le contexte de la programmation par démonstration, cette phase de requêtes pourrait être l'interaction entre le robot et l'utilisateur : le robot pourrait demander à l'utilisateur quelles sont les actions ou les tâches faisables et infaisables. Dans ce contexte, les démonstrations seraient l'exécution des séquences d'actions générées.

Enfin, dans cette thèse, nous avons tiré profit du fait que les problèmes de planification sont équivalents à des grammaires pour apprendre des modèles d'actions. De plus, nous avons vu dans le chapitre 3 que l'induction grammaticale a plusieurs applications (Adriaans and van Zaanen, 2004; Dupont et al., 2008; Boström, 1996; Boström, 1998; Cruz-Alcázar and Vidal, 1998; Bex et al., 2006; Cruz-Alcázar and Vidal, 2008; Stein et al., 2006; Bréhélin et al.,

2001; Raffelt and Steffen, 2006; Berg et al., 2006). Une dernière application pourrait donc consister à utiliser AMLSI dans ces contextes applicatifs où une représentation sous forme de modèles d'actions peut présenter des avantages. Par exemple, dans le domaine de la modélisation comportementale des systèmes, des algorithmes d'induction grammatical sont utilisés pour induire le comportement d'un système, d'un logiciel, d'un processus industriel, etc. Le comportement de ces systèmes est représenté sous la forme de grammaires et peut ensuite être analysé pour les automatiser (Dupont et al., 2008), détecter des intrusions (Su and Wassermann, 2006; Godefroid et al., 2008) etc. Dans ce contexte, AMLSI pourrait être utilisé pour apprendre des modèles d'actions représentant le comportement de ces systèmes. L'avantage serait d'avoir une représentation plus compacte et plus lisible de ces systèmes. De plus, les modèles d'actions appris pourraient être utilisés directement pour automatiser ces systèmes, détecter les intrusions, etc.

Bibliography

Adriaans, P. W. and van Zaanen, M. (2004). Computational grammar induction for linguists. *Grammars*, 7:57–68.

Aineto, D., Celorrio, S. J., and Onaindia, E. (2019). Learning action models with minimal observability. *Artificial Intelligence*, 275:104–137.

Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350.

Angluin, D. (1981). A note on the number of queries needed to identify regular languages. *Information and control*, 51(1):76–87.

Angluin, D. (1988). Queries and concept learning. *Machine learning*, 2(4):319–342.

Arora, A., Fiorino, H., Pellier, D., Métivier, M., and Pesty, S. (2018a). A review of learning planning action models. *Knowledge Engineering Review*, 33.

Arora, A., Fiorino, H., Pellier, D., and Pesty, S. (2017). Action model acquisition using sequential pattern mining. In *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 286–292. Springer.

Arora, A., Fiorino, H., Pellier, D., and Pesty, S. (2018b). Action model acquisition using LSTM. *CoRR*, abs/1810.01992.

Asai, M. (2019). Unsupervised grounding of plannable first-order logic representation from images. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 583–591.

Backes, P., Norris, J., Powell, M., and Vona, M. (2004). Multi-mission activity planning for mars lander and rover missions. In *Proc. of the IEEE Aerospace Conference*, volume 2, pages 877–886 Vol.2.

Balac, N., Gaines, D. M., and Fisher, D. (2000a). Learning action models for navigation in noisy environments. In *Proc. of the Workshop on Machine Learning of Spatial Knowledge*.

Balac, N., Gaines, D. M., and Fisher, D. (2000b). Using regression trees to learn action models. In *Proc. of the IEEE International Conference on Systems, Man & Cybernetics*, pages 3378–3383.

Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., Morris, P., Ong, J., Remolina, E., Smith, T., et al. (2012). Europa: A platform for ai planning, scheduling, constraint programming, and optimization. *International Competition on Knowledge Engineering for Planning and Scheduling*.

Barzdinš, J. (1974). Two theorems on the limiting synthesis of functions. *Theory of algorithms and programs*, 1(210):82–88.

Berg, T., Jonsson, B., and Raffelt, H. (2006). Regular inference for state machines with parameters. In *Fundamental Approaches to Software Engineering*, pages 107–121.

Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. (2006). Inference of concise dtds from XML data. In *International Conference on Very Large Data Bases*, pages 115–126.

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2008). Robot programming by demonstration. In *Springer Handbook of Robotics*, pages 1371–1394.

Bonet, B. and Geffner, H. (2020). Learning first-order symbolic representations for planning from the structure of the state space. In *Proc. of the European Conference on Artificial Intelligence*, pages 2322–2329.

Bonet, B., Palacios, H., and Geffner, H. (2009). Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. of the International Conference on Automated Planning and Scheduling*.

Borchers, B. and Furman, J. (1998). A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306.

Borrajo, D. and Veloso, M. M. (1997). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, 11(1-5):371–405.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proc. of the workshop on Computational Learning Theory*, pages 144–152. ACM.

Boström, H. (1996). Theory-guided induction of logic programs by inference of regular languages. In *Proc. of the International Conference on Machine Learning*.

Boström, H. (1998). Predicate invention and learning from positive examples only. In *European Conference on Machine Learning*, pages 226–237.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-ff: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621.

Bréhélin, L., Gascuel, O., and Caraux, G. (2001). Hidden markov models with patterns to learn boolean vector sequences and application to the built-in self-test for integrated circuits. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):997–1008.

Brenner, M. (2003). A multiagent planning language. In *Proc. of ICAPS Workshop on PDDL*, volume 3, page 6.

Bresina, J. L., Jónsson, A. K., Morris, P. H., and Rajan, K. (2005). Activity planning for the mars exploration rovers. In *Proc. of the International Conference on Automated Planning and Scheduling*, volume 2005, pages 40–49.

Carrasco, R. C. and Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In *Proc. of the International Colloquium on Grammatical Inference*, pages 139–152. Springer.

Carreno, Y., Pairet, E., Petillot, Y., and Petrick, R. P. A. (2020). Task allocation strategy for heterogeneous robot teams in offshore missions. In *Proc. of the International Conference on Autonomous Agents and MultiAgent Systems*, page 222–230.

Case, J. and Smith, C. (1983). Comparison of identification criteria for machine inductive inference. *Theoretical Computer Science*, 25:193–220.

Castellanos-Paez, S., Pellier, D., Fiorino, H., and Pesty, S. (2018). Mining useful macro-actions in planning. *CoRR*, abs/1810.09145.

Celorrio, S. J., de la Rosa, T., Fernández, S., Fernández, F., and Borrajo, D. (2012). A review of machine learning for automated planning. *Knowledge Engineering Review*, 27(4):433–467.

Celorrio, S. J., Fernández, F., and Borrajo, D. (2008). The PELA architecture: Integrating planning and learning to improve execution. In *Proc. of the AAAI Conference on Artificial Intelligence*, pages 1294–1299.

Celorrio, S. J., Jonsson, A., and Palacios, H. (2015). Temporal planning with required concurrency using classical planning. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 129–137.

Chvátal, V. (1979). A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235.

Coles, A., Fox, M., Halsey, K., Long, D., and Smith, A. (2009). Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1–44.

Cresswell, S. and Gregory, P. (2011). Generalised domain model acquisition from action traces. In *Proc. of the International Conference on Automated Planning and Scheduling*.

Cresswell, S., McCluskey, T. L., and West, M. M. (2013). Acquiring planning domain models using *LOCM*. *Knowledge Engineering Review*, 28(2):195–213.

Croonenborghs, T., Ramon, J., Blockeel, H., and Bruynooghe, M. (2007). Online learning and exploiting relational models in reinforcement learning. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 726–731.

Cross, S. E. and Walker, E. (1994). Dart: applying knowledge-based planning and scheduling to crisis action planning.

Cruz-Alcázar, P. P. and Vidal, E. (1998). Learning regular grammars to model musical style: Comparing different coding schemes. In *Grammatical Inference*, pages 211–222.

Cruz-Alcázar, P. P. and Vidal, E. (2008). Two grammatical inference applications in music processing. *Applied Artificial Intelligence*, 22(1&2):53–76.

Cushing, W., Kambhampati, S., Mausam, and Weld, D. S. (2007). When is temporal planning really temporal? In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 1852–1859.

Dawson, C. and Siklóssy, L. (1977). The role of preprocessing in problem solving systems. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 465–471.

De La Higuera, C. (2010). *Grammatical Inference: learning automata and grammars*. Cambridge University Press.

de la Rosa, T., Celorrio, S. J., and Borrajo, D. (2008). Learning relational decision trees for guiding heuristic planning. In *Pro. of the International Conference on Automated Planning and Scheduling*, pages 60–67.

De La Rosa, T., Jiménez, S., Garcıa-Durán, R., Fernández, F., Garcıa-Olaya, A., and Borrajo, D. (2009). Three relational learning approaches for lookahead heuristic planning. In *Proc. of the Workshop on Planning and Learning*, pages 37–44.

de la Rosa, T., Olaya, A. G., and Borrajo, D. (2007). Using cases utility for heuristic planning improvement. In *Proc. of the International Conference on Case-Based Reasoning*, pages 137–148.

Dupont, P. (1996). Incremental regular inference. In *Proc. of the International Colloquium on Grammatical Inference*, pages 222–237. Springer.

Dupont, P., Lambeau, B., Damas, C., and van Lamsweerde, A. (2008). The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*.

Dupont, P., Miclet, L., and Vidal, E. (1994). What is the search space of the regular inference? In *Proc. of the International Conference on Grammatical Inference*, pages 25–37.

Dvorak, F., Bit-Monnot, A., Ingrand, F., and Ghallab, M. (2014). A flexible ANML actor and planner in robotics. In *Proc. of the Planning and Robotics Workshop*.

Edelkamp, S., Elfers, C., Horstmann, M., Schröder, M.-S., Sohr, K., and Wagner, T. (2009). Early warning and intrusion detection based on combined ai methods. In *Proc. of the International Conference on Automated Planning and Scheduling*.

Erol, K., Hendler, J. A., and Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *Proc. of the National Conference on Artificial Intelligence*, pages 1123–1128.

Esposito, F., Semeraro, G., Fanizzi, N., and Ferilli, S. (2000). Multistrategy theory revision: Induction and abduction in INTHELEX. *Machine Learning*, 38(1-2):133–156.

Eyraud, R., De La Higuera, C., and Janodet, J.-C. (2007). Lars: A learning algorithm for rewriting systems. *Machine Learning*, 66(1):7–31.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208. Publisher: Elsevier.

Fisher, F., Knight, R., Engelhardt, B., Chien, S., and Alejandre, N. (2000). A planning approach to monitor and control for deep space communications. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)*, volume 2, pages 311–320 vol.2. ISSN: 1095-323X.

Fox, M. and Long, D. (2002a). Fast temporal planning in a graphplan framework. In *Proc. of the Workshop on Planning for Temporal Domains*, pages 9–17.

Fox, M. and Long, D. (2002b). PDDL+: Modeling continuous time dependent effects. In *Proc. of the International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34.

Fox, M. and Long, D. (2003). PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

Fox, M. and Long, D. (2006). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297.

Fu, K. and Booth, T. L. (1975). Grammatical inference: Introduction and survey - part II. *Transactions on Systems, Man, and Cybernetics*, 5(4):409–423.

Furelos Blanco, D., Jonsson, A., Palacios Verdes, H. L., and Jiménez, S. (2018). Forward-search temporal planning with simultaneous events. In *Proc. of the Constraint Satisfaction Techniques for Planning and Scheduling Workshop*.

Gabel, M. and Su, Z. (2010). Online inference and enforcement of temporal properties. In *Proc. of International Conference on Software Engineering*, pages 15–24.

Gaglione, J., Neider, D., Roy, R., Topcu, U., and Xu, Z. (2021). Learning linear temporal properties from noisy data: A maxsat-based approach. In *Proc. of the International Symposium on Automated Technology for Verification and Analysis*, pages 74–90.

Garcia, P., Segarra, E., Vidal, E., and Galiano, I. (1990). On the use of the morphic generator grammatical inference (mggi) methodology in automatic speech recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 4(04):667–685.

Garcia, P. and Vidal, E. (1990). Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925.

Garcia-Martinez, R. and Borrajo, D. (2000). An integrated approach of learning, planning, and execution. *Journal of Intelligent and Robotic Systems*, 29(1):47–78.

Garland, A. and Lesh, N. (2003). Learning hierarchical task models by demonstration. *Mitsubishi Electric Research Laboratory (MERL), USA–(January 2002)*.

Garrido, A. and Jiménez, S. (2020). Learning temporal action models via constraint programming. In *Proc. of the European Conference on Artificial Intelligence*, pages 2362–2369.

Genesereth, M. R. and Nilsson, N. J. (1988). *Logical foundations of artificial intelligence*. Morgan Kaufmann.

Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D., Sun, Y., and Weld, D. (1998). Pddl - the planning domain definition language. *Technical Report*.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory and Practice*. Elsevier.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Machine Learning Proceedings*, pages 87–95. Elsevier.

Glover, F. W. and Laguna, M. (1997). *Tabu Search*. Kluwer.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 206–215.

Gold, E. M. (1967). Language identification in the limit. *Information and control*, 10(5):447–474.

Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320.

Goldberg, D. E. (1989). *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley.

González-Ferrer, A., Fernández-Olivares, J., Castillo, L., et al. (2009). Jabbah: a java application framework for the translation between business process models and htn. *Proc. of the International Competition on Knowledge Engineering for Planning and Scheduling*.

Grand, M., Fiorino, H., and Pellier, D. (2020a). AMLSI: A novel accurate action model learning algorithm. In *Proc. of the workshop on Knowledge Engineering for Planning and Scheduling*.

Grand, M., Fiorino, H., and Pellier, D. (2020b). Retro-engineering state machines into PDDL domains. In *Proc. of the International Conference on Tools with Artificial Intelligence*, pages 1186–1193.

Grand, M., Fiorino, H., and Pellier, D. (2021). Incramlsi: Incremental learning of accurate planning domains from partial and noisy observations. In *Proc. of the International Conference on Tools with Artificial Intelligence*, pages 121–128.

Grand, M., Fiorino, H., and Pellier, D. (2022a). An accurate HDDL domain learning algorithm from partial and noisy observations. In *Proc. of the HPlan workshop*.

Grand, M., Pellier, D., and Fiorino, H. (2022b). Tempamlsi: Temporal action model learning based on STRIPS translation. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 597–605.

Gregory, P. and Cresswell, S. (2015). Domain model acquisition in the presence of static relations in the LOP system. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 97–105.

Gregory, P. and Lindsay, A. (2016). Domain model acquisition in domains with action costs. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 149–157.

Guttman, O. et al. (2006). *Probabilistic automata and distributions over sequences*. PhD thesis, The Australian National University.

Halsey, K., Long, D., and Fox, M. (2004). Crikey-a temporal planner looking at the integration of scheduling and planning. In *Proc. of the Integrating Planning into Scheduling Workshop*, pages 46–52.

Helmert, M. (2006). The Fast Downward planning system. *Artificial Intelligence*, 26:191–246.

Hoffmann, J., Weber, I., and Kraft, F. (2009). Planning@sap: An application in business process management. In *Proc. of the Scheduling and Planning Applications woRKshop*.

Hogg, C., Kuter, U., and Munoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Hogg, C., Kuter, U., and Munoz-Avila, H. (2010). Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Proc. of the AAAI Conference on Artificial Intelligence*, volume 24.

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). Htn-maker: Learning htns with minimal additional knowledge engineering required. In *Proc. of the AAAI Conference on Artificial Intelligence*, pages 950–956.

Höller, D. (2021). Translating totally ordered htn planning problems to classical planning problems using regular approximation of context-free languages. In *Proc. of the International Conference on Automated Planning and Scheduling*, volume 31, pages 159–167.

Höller, D., Behnke, G., Bercher, P., and Biundo, S. (2014). Language classification of hierarchical planning problems. In *Proc. of the European Conference on Artificial Intelligence*, pages 447–452.

Höller, D., Behnke, G., Bercher, P., and Biundo, S. (2016). Assessing the expressivity of planning formalisms through the comparison to formal languages. In *In Proc. of the International Conference on Planning and Scheduling*, page 158–165.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., and Alford, R. (2019). Hierarchical planning in the IPC. *CoRR*, abs/1909.04405.

Howey, R. and Long, D. (2003). Val's progress: The automatic validation tool for pddl2. 1 used in the international planning competition. In *Proc. of the Workshop on the IPC*, pages 28–37.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., and Alford, R. (2020). HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proc. of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891.

Ilghami, O., Muñoz-Avila, H., Nau, D. S., and Aha, D. W. (2005). Learning approximate preconditions for methods in hierarchical plans. In *Proc. of the International Conference on Machine Learning*, pages 337–344.

Ilghami, O., Nau, D. S., and Muñoz-Avila, H. (2006). Learning to do HTN planning. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 390–393.

Ilghami, O., Nau, D. S., Muñoz-Avila, H., and Aha, D. W. (2002). Camel: Learning method preconditions for HTN planning. In *Proc. of the International Conference on Artificial Intelligence Planning Systems*, pages 131–142.

Jilani, R., Crampton, A., Kitchin, D. E., and Vallati, M. (2014). Automated knowledge engineering tools in planning: state-of-the-art and future challenges. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling*.

Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Proc. of the Symposium on the Complexity of Computer Computations*, pages 85–103.

Kautz, H. A. and Selman, B. (1996). Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. of the National Conference on Artificial Intelligence*, pages 1194–1201.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77.

Koshiba, T., Mäkinen, E., and Takada, Y. (2000). Inferring pure context-free languages from positive data. *Acta Cybernetica*, 14(3):469–477.

Kucera, J. and Barták, R. (2018). LOUGA: learning planning operators using genetic algorithms. In *Proc. of the Pacific Rim Knowledge Acquisition Workshop*, pages 124–138.

Kudo, M. and Shimbo, M. (1988). Efficient regular grammatical inference techniques by the use of partial similarities and their logical relationships. *Pattern recognition*, 21(4):401–409.

Lallement, R., Silva, L. d., and Alami, R. (2018). HATP: Hierarchical agent-based task planner. In *Proc. of the International Conference on Autonomous Agents and MultiAgent Systems*, pages 1823–1825.

Lamanna, L., Saetti, A., Serafini, L., Gerevini, A., and Traverso, P. (2021a). Online learning of action models for PDDL planning. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling*.

Lamanna, L., Saetti, A., Serafini, L., Gerevini, A., and Traverso, P. (2021b). Online learning of action models for PDDL planning. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling*.

Lang, K. (1998). Evidence driven state merging with search. *Rapport technique TR98–139, NECI*, 31.

Lesire, C., Infantes, G., Gateau, T., and Barbier, M. (2016). A distributed architecture for supervision of autonomous multi-robot missions. *Autonomous Robots*, 40(7):1343–1362.

Li, N., Cushing, W., Kambhampati, S., and Yoon, S. (2014). Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. *ACM Transactions on Intelligent Systems and Technology*, 5(2):1–32.

Liang, Y. S., Pellier, D., Fiorino, H., and Pesty, S. (2022). iRoPro: An interactive robot programming framework. *International Journal of Social Robotics*, 14(1):177–191.

Martínez, D., Ribeiro, T., Inoue, K., Alenyà, G., and Torras, C. (2015). Learning probabilistic action models from interpretation transitions. In *Proc. of the International Conference on Logic Programming*.

McCluskey, T. L., Richardson, N. E., and Simpson, R. M. (2002). An interactive method for inducing operator descriptions. In *Proc. of the International Conference on Artificial Intelligence Planning and Scheduling*, pages 121–130.

Miclet, L. and de Gentile, C. (1994). Inférence grammaticale à partir d'exemples et de contre-exemples: deux algorithmes optimaux:(big et rig) et une version heuristique (brig). *F1-F13*.

Miglani, S. and Yorke-Smith, N. (2020). Nltopddl: One-shot learning of pddl models from natural language process manuals. In *Prof of Workshop on Knowledge Engineering for Planning and Scheduling*.

Minton, S. (2012). *Learning search control knowledge: An explanation-based approach*, volume 61. Springer Science & Business Media.

Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

Mourão, K., Zettlemoyer, L. S., Petrick, R. P. A., and Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proc. of the Uncertainty in Artificial Intelligence*, pages 614–623.

Nargesian, F. and Ghassem-Sani, G. (2008). Lhtndt: Learn htn method preconditions using decision tree. In *Proc. of the ICINCO-ICSO*, pages 60–65.

Nau, D. S., Au, T., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J. W., Wu, D., and Yaman, F. (2005). Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41.

Neider, D. and Gavran, I. (2018). Learning linear temporal properties. In *Proc. of Conference on Formal Methods in Computer-Aided Design*, pages 1–10.

Nevill-Manning, C. G. and Witten, I. H. (1997). Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82.

Oncina, J. and Garcia, P. (1992). Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pages 49–61. World Scientific.

Oncina, J., García, P., and Vidal, E. (1993). Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):448–458.

Parekh, R., Nichitiu, C., and Honavar, V. (1998). A polynomial time incremental algorithm for learning dfa. In *Proc. of the International Colloquium on Grammatical Inference*, pages 37–49. Springer.

Pearl, J. (2014). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.

Pednault, E. P. (1994). ADL and the state-transition model of action. *Journal of Logic and Computation*, 4(5):467–512.

Pellier, D. and Fiorino, H. (2018). Pddl4j: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176.

Pellier, D. and Fiorino, H. (2020). Totally and partially ordered hierarchical planners in PDDL4J library. *CoRR*, abs/2011.13297.

Raffelt, H. and Steffen, B. (2006). Learnlib: A library for automata learning and experimentation. In *Fundamental Approaches to Software Engineering*, pages 377–380.

Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine learning*, 62(1-2):107–136.

Rodrigues, C., Gérard, P., and Rouveirol, C. (2010a). Incremental learning of relational action models in noisy environments. In *Proc. of the International Conference on Inductive Logic Programming*, pages 206–213.

Rodrigues, C., Gérard, P., Rouveirol, C., and Soldano, H. (2010b). Incremental learning of relational action rules. In *proc. of the International Conference on Machine Learning and Applications*, pages 451–458.

Russell, S. and Norvig, P. (2021). *Artificial intelligence: a modern approach, global edition 4th*, volume 19. Pearson.

Sakakibara, Y. (1992). Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60.

Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32:27.

Segura-Muros, J. Á., Pérez, R., and Fernández-Olivares, J. (2018). Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling*.

Shah, A., Kamath, P., Shah, J. A., and Li, S. (2018). Bayesian inference of temporal task specifications from demonstrations. In *Prof. of the Neural Information Processing Systems*, pages 3808–3817.

Shah, M., Chrpa, L., Jimoh, F., Kitchin, D., McCluskey, T., Parkinson, S., and Vallati, M. (2013). Knowledge engineering tools in planning: State-of-the-art and future challenges. *Knowledge Engineering for Planning and Scheduling*, 53:53.

Shahaf, D. and Amir, E. (2006). Learning partially observable action schemas. In *Proc. of the National Conference on Artificial Intelligence*, volume 21, page 913. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Shen, W. (1993). Discovery as autonomous learning from the environment. *Machine Learning*, 12:143–165.

Simpson, R. (2005). Gipo graphical interface for planning with objects. In *Proc. of the International Conference for Knowledge Engineering in Planning and Scheduling*. Citeseer.

Sommerville, I. (2011). *Software engineering*. Addison-Wesley.

Stein, S. E., Heller, S. R., and Tchekhovskoi, D. V. (2006). The iupac chemical identifier – technical manual. *National Institute of Standards and Technology*, pages 20899–8380.

Stolcke, A. and Omohundro, S. (1994). Inducing probabilistic grammars by bayesian model merging. In *proc. of the International Colloquium on Grammatical Inference*, pages 106–118. Springer.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Proc. of the Symposium on Principles of Programming Languages*, pages 372–382.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Tsang, E. P. K. (1993). *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press.

Urmson, C. and Whittaker, W. R. (2008). Self-driving cars and the urban challenge. *IEEE Intelligent Systems*, 23(2):66–68.

van Rijsbergen, C. J. (1979). *Information Retrieval*. Butterworth.

Vaquero, T. S., Romero, V., Tonidandel, F., and Silva, J. R. (2007). itsimple 2.0: An integrated tool for designing planning domains. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 336–343.

Vaquero, T. S., Silva, J. R., Tonidandel, F., and Beck, J. C. (2013). itsimple: towards an integrated design system for real planning applications. *Knowledge Engineering Review*, 28(2):215–230.

Verma, P., Marpally, S. R., and Srivastava, S. (2021a). Asking the right questions: Learning interpretable action models through query answering. In *Proc. of the AAAI Conference on Artificial Intelligence*, pages 12024–12033.

Verma, P., Marpally, S. R., and Srivastava, S. (2021b). Learning user-interpretable descriptions of black-box AI system capabilities. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling*.

Vilar, J. M. (2000). Improve the learning of subsequential transducers by using alignments and dictionaries. In *Proc. of the International Colloquium on Grammatical Inference*, pages 298–311. Springer.

Vodrázka, J. and Chrpa, L. (2010). Visual design of planning domains. In *Proc. of the workshop on Scheduling and Knowledge Engineering for Planning and Scheduling*, pages 68–69.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. In *Machine Learning Proceedings 1995*, pages 549–557. Elsevier.

Xiaoa, Z., Wan, H., Zhuoa, H. H., Herzigb, A., Perrusselc, L., and Chena, P. (2019). Learning htn methods with preference from htn planning instances. In *Proc. of the HPlan workshop*, page 31.

Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3):107–143.

Yoon, S. W., Fern, A., and Givan, R. (2006). Learning heuristic functions from relaxed plans. In *Proc. of the International Conference on Automated Planning and Scheduling*, pages 162–171.

Younes, H. L. and Littman, M. L. (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. *Technical Report CMU-CS-04-162*, 2:99.

Yu, S. (1997). Regular languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, pages 41–110. Springer.

Zhuo, H. H. (2015). Crowdsourced action-model acquisition for planning. In *Proc. of the AAAI Conference on Artificial Intelligence*.

Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., and Munoz-Avila, H. (2009). Learning htn method preconditions and action models from partial observations. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Zhuo, H. H. and Kambhampati, S. (2013). Action-model acquisition from noisy plan traces. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Zhuo, H. H., Muñoz-Avila, H., and Yang, Q. (2011). Learning action models for multi-agent planning. In *Proc. of the International Conference on Autonomous Agents and Multiagent Systems*, pages 217–224.

Zhuo, H. H., Nguyen, T., and Kambhampati, S. (2013). Refining incomplete planning domain models through plan traces. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Zhuo, H. H., Peng, J., and Kambhampati, S. (2019). Learning action models from disordered and noisy plan traces. *CoRR*, abs/1908.09800.

Zhuo, H. H., Yang, Q., Hu, D. H., and Li, L. (2010a). Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540–1569.

Zhuo, H. H., Yang, Q., Hu, D. H., and Li, L. (2010b). Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540–1569.