# University of Granada

Department of Computer Science
and Artificial Intelligence




PhD Program in Information and
Communication Technologies


**PhD THESIS DISSERTATION**

# LEARNING EXPRESSIVE NUMERICAL PLANNING DOMAINS BY INTEGRATING MACHINE LEARNING TECHNIQUES


**PhD STUDENT**

## JOSE Á. SEGURA MUROS

**PhD ADVISORS**
Juan Fernández Olivares
Raúl Pérez



Granada, November 2021

Editor: Universidad de Granada. Tesis Doctorales
Autor: José Ángel Segura Muros
ISBN: 978-84-1117-211-0
URI: http://hdl.handle.net/10481/72367

El doctorando / *The doctoral candidate* **José Á. Segura Muros** y los directores de la tesis / *and the thesis supervisors*: **Juan Fernández Olivares** and **F. G. Raúl Pérez Rodríguez**.

Garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis y hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se han utilizado sus resultados o publicaciones.

---

*Guarantee, by signing this doctoral thesis, that the work has been done by the doctoral candidate under the direction of the thesis supervisors and, as far as our knowledge reaches, in the performance of the work, the rights of the other authors to be cited (when their results or publications have been used) have been respected.*

Lugar y fecha / *Place and date*:

| The PhD student: | The PhD advisor: | The PhD advisor: |
|---|---|---|
| Sgd.: J. Á. Segura Muros | Sgd.: J. Fernández Olivares | Sgd.: F. G. R. Pérez Rodríguez |

Esta tesis doctoral ha sido financiada por los Proyectos Nacionales de Investigación Españoles TIN2015-71618-R y RTI2018-098460-B-I00. El doctorando José Ángel Segura Muros, ha disfrutado de la ayuda FPI con referencia BES-2016-076700 del Ministerio de Ciencia, Innovación y Universidades de España.

This doctoral thesis has been supported by the Spanish National Research Projects TIN2015-71618-R and RTI2018-098460-B-I00. The PhD Student, J. A. Segura Muros, holds an FPI scholarship from the Spanish Ministry of Science, Innovation and Universities with reference BES-2016-076700.

Intelligence is the ability to adapt to change

Stephen Hawking

It's the job that's never started as takes longest to finish

J.R.R. Tolkien, *The Lord of the Rings*

# Agradecimientos

Me gustaría empezar agradeciendo a Juan y Raúl por su trabajo en estos años. Sin vuestra ayuda, conocimientos y consejos este trabajo no habría llegado a buen puerto. Gracias por dedicarme vuestro tiempo y sabiduría a lo largo de tantas reuniones y desayunos en estos años.

Gracias también a mis padres, Ángel y Aurora, por todo lo que han hecho por mí durante estos 32 años. Por su amor, por su ayuda y por su apoyo incondicional. Gracias a mi hermana, Gema, y a mi cuñado, Víctor, por vuestro cariño y confianza. No puedo (ni quiero) olvidarme de mencionar a Esperanza y Victoria, las cuales han crecido al tiempo que desarrollaba este trabajo.

Pese a que se la mucha vergüenza que te da esto, Maribel, a ti también quiero darte las gracias por todo, en general. Cuando comencé esta aventura no éramos más que unos extraños el uno para el otro, y ahora te has convertido en uno de los pilares de mi vida. Gracias por apoyarme, gracias por aguantarme, gracias por tu paciencia, en resumen, gracias por permitirme ser parte de tu vida.

No me olvido aquellos que han compartido conmigo estos años en el Cerrillo de Maracena. A todos los compañeros del CITIC-UGR, los lista de nombres es larga (Jesús, Sergio, Nacho, Jorge, Elena, Gustavo, Fran, Germán, Nuria...), pero ya sean del DB-4 o de la Guardería, muchas gracias. Gracias por los juegos de mesa, por los cafés, por los paseos haciendo incursiones, por las risas... Muchas gracias a todos, gracias por crear un ambiente de trabajo maravilloso. Aunque no te haya mencionado antes, no me he olvidado de ti, Juanan, pero es porque a ti te reservo este hueco en exclusiva. Muchísimas gracias Juanan, de corazón te digo que sin tu saber y asistencia este trabajo no sería el que es, y sin tu amistad estos años tampoco lo habrían sido.

Este párrafo es para todos los que me he dejado en el tintero, no me olvido de ninguno de vosotros. Desde de los señores mayores a los que les doy la turra cada vez que quiero coger un pincel, hasta la gente de la Posada que me daban la cafeína necesaria para empezar la jornada laboral, pasando por todas de aquellas personas que están presentes en mi día a día. En resumen, Muchas gracias a todos los que hacen mi vida mejor.

Y, finalmente, muchas gracias a ti, lector, gracias por decidir que este trabajo es merecedor de tu tiempo. Espero que el esfuerzo puesto en el trabajo realizado a lo largo de todos estos años pueda ayudarte en tu trabajo.

MUCHÍSIMAS GRACIAS A TODOS

# Resumen

Esta tesis doctoral plantea una novedosa metodología para el aprendizaje de Modelos de Acción para Planificación Automática. Esta metodología se enmarca dentro del campo de la Ingeniería de Conocimiento, concretamente en el área de la Adquisición de Conocimiento. Específicamente, esta tesis doctoral presenta un proceso de aprendizaje que combina de forma jerárquica distintas técnicas de Aprendizaje Automático, con un especial enfoque en el uso de técnicas de Inteligencia Artificial Explicable.

Los objetivos considerados en el desarrollo de la tesis doctoral son:

- **Implementación de un algoritmo de aprendizaje capaz de aprender modelos de acción STRIPS cuyas precondiciones y efectos también contengan expresiones aritméticas y relacionales.** El estudio empírico realizado para la evaluación de este objetivo compara la solución propuesta con algoritmos de referencia con el fin de analizar su comportamiento y situarlo dentro del estado del arte.

- **Diseño de un algoritmo de aprendizaje que mejore las capacidades del anterior a la hora de trabajar bajo situaciones en las que la calidad de los datos de entrada es baja.** Para testear el nuevo procedimiento se implementa un proceso experimental con el objetivo de confrontarlo respecto al algoritmo desarrollado previamente usando datos de entrada con incertidumbre. Además, con afán de obtener una mejor visión de conjunto de la calidad de los métodos desarrollados, se compara la nueva metodología con algoritmos de referencia del estado del arte.

- **Desarrollo de un algoritmo de aprendizaje de modelos de acción capaz de funcionar a partir de datos obtenidos de un entorno simulado.** Para un correcto análisis de la metodología implementada se usan datos obtenidos de las ejecuciones de agentes sobre entornos simulados, comprobando que los procesos propuestos pueden replicar el comportamiento de dichos agentes.

Los tres objetivos han sido alcanzados de forma satisfactoria implementando una serie de algoritmos de aprendizaje que combinan de forma jerárquica técnicas de regresión, clasificación estadística y análisis de grupos. Cada uno de estos procesos de aprendizaje es el producto de la consecución de cada uno de los objetivos propuestos anteriormente y supone una contribución al estado del arte por sí mismo. Para la correcta implementación de cada algoritmo de aprendizaje se han diseñado diversos métodos de preprocesamiento de datos, de tratamiento de la incertidumbre y de adquisición de modelos de clasificación, usando para ello técnicas bien conocidas del campo del Aprendizaje Automático. Finalmente, indicar que las contribuciones presentadas en la tesis doctoral se han evaluado usando, no solo dominios de planificación de referencia tomados de la comunidad de Planificación Automática, si no también, del entorno de trabajo de videojuegos GVG-AI.

# Abstract

This doctoral thesis proposes a novel methodology for learning Action Models for Automatic Planning. This methodology is framed within the field of Knowledge Engineering, specifically in the area of Knowledge Acquisition. Particularly, this doctoral thesis presents a learning process that hierarchically combines different Machine Learning techniques, with a special focus on the use of Explainable Artificial Intelligence techniques.

The objectives considered in the development of the doctoral thesis are:

- **Implementation of a learning algorithm capable of learning STRIPS action models whose preconditions and effects also contain arithmetic and relational expressions.** The empirical study carried out for the evaluation of this objective compares the proposed solution with reference algorithms in order to analyse its behaviour and situate it within the state of the art.
- **Design of a learning algorithm that improves the capabilities of the previous one when working under situations where the quality of the input data is low.** To test the new procedure, an experimental process is implemented to compare it with the previously developed algorithm using input data with uncertainty. Furthermore, in order to obtain a better overview of the quality of the developed methods, the new methodology is compared with state-of-the-art reference algorithms.
- **Development of an action model learning algorithm capable of operating from data obtained from a simulated environment.** For a correct analysis of the implemented methodology, data obtained from the execution of agents in simulated environments are used, verifying that the proposed processes can replicate the behaviour of these agents.

All three objectives have been successfully achieved by implementing a series of learning algorithms that hierarchically combine regression, statistical classification and cluster analysis techniques. Each of these learning processes is the product of the achievement of each of the above objectives and is a contribution to the state of the art in its own right. For the correct implementation of each learning algorithm, several methods of data pre-processing, uncertainty treatment and acquisition of classification models have been designed, using well-known techniques from the field of Machine Learning. Finally, it should be noted that the contributions presented in the doctoral thesis have been evaluated using reference planning domains taken from the Automatic Planning community and, also, from the GVG-AI videogame working environment.

# Contents

# List of Figures

# List of Tables

# Listings

# List of Algorithms

# Chapter 1

# Introduction

## 1.1 Motivation

Since the term *Artificial Intelligence* was coined in 1956, the number of areas of knowledge and applications of Artificial Intelligence have multiplied over time. Of these areas, Automatic Planning [GNT04] is one of the oldest and has historically received the most attention from the scientific community. Automatic planning is the discipline that deals with the production of plans to achieve a given goal, in order to be executed by an agent (whether human or not). Unfortunately, the implementation of automatic planning techniques in the real world presents problems.

In order to use automated planning techniques, it is necessary to define in advance an ontology that represents the "world" on which one wants to work. This ontology (called planning domain) contains a definition of the objects of the world being represented and their relationships, but more importantly, it contains a formal definition of the activities that can be executed in that world. These activities (called action models) are the basic component that forms the plans generated by automatic planning techniques, and their ultimate purpose is to be executed by an agent. Defining a planning domain is a heavy task that requires time and expert knowledge. This issue is greatly exacerbated as the complexity of the world being modelled increases.

However, in the area of Knowledge Engineering [SBF98], solutions to this problem have been proposed [JRF+12]. These solutions are the so-called **Action Model Learning Techniques** [AFP+18]. These techniques aim to automatically obtain a planning domain from a set of input data extracted from pre-existing process executions, in such a way that the domain is able to reproduce the processes from which the input information has been obtained. In recent years, several superb solutions have been proposed which address this learning problem. These solutions usually emphasise the aspect of learning using incomplete information, with some of them being capable of learning under extreme input scarcity.

These approaches successfully learn planning domains, but from the point of

view of learning planning domains for real-world problems, they also have some major drawbacks.

- The first is that they neglect the improvement of the expressiveness of the learned models, a key aspect for the implementation of planning domains in this type of problem.
- The second is that they disregard other types of uncertainty in the input data (e.g., noise), one of the most widespread problems with information obtained from real-world applications.

In order to close the gap caused by these shortcomings, taking a step forward in the state of the art, the following line of work is presented: **To develop an action model learning algorithm with sufficient capacity to be implemented in real-world applications, and able to generate the most expressive planning domains while handling uncertainty in the input data**. To achieve this goal, this solution bases the operation of its different components on techniques from the field of Machine Learning [MMC13, Mit99].

The use of technology from the field of Machine Learning opens the door to the use of some of the most robust and well-tested techniques in the history of modern computing. Machine Learning is the most prolific field of Artificial Intelligence, and basing the development of the contributions proposed in this paper on it gives us access to a vast library of techniques, methods and technologies to meet the needs of the challenge at hand.

Among these techniques, those of **Explainable Artificial Intelligence** are of particular interest for the design of the methods proposed in this document. Explainable Artificial Intelligence is the discipline [BDRD+20] that encompasses a series of specific machine learning techniques and methodologies that aim to generate solutions that can be interpreted by humans. Usually, machine learning algorithms are governed by objective functions defined by a mathematical model, and, when maximised/minimised, the solutions they obtain tend to make little sense from a human perspective. Examples of this can be seen in a multitude of state-of-the-art techniques, such as reinforcement learning, deep learning, or support vector machines, which, although extremely competent in terms of performance, are also extremely opaque and do not allow to understand their internal models (the so-called black-box models); as a natural consequence, they make it difficult to extract new knowledge from them. Techniques developed under the precepts of explainable artificial intelligence allow a human operator to see and interpret their internal models (so-called white-box models) in order to evaluate them. This allows the user to weigh the decisions made by the algorithm and generate human-understandable knowledge.

From the point of view of the work presented in this paper, techniques based on explainable artificial intelligence present a great advantage, since their use allows access to the information that has been extracted during the development of the learning process, being able to obtain new knowledge from it with fewer impediments. Moreover, the interpretability of explainable artificial intelligence models

is crucial for the simplicity of the learning process, since without them, the translation process necessary to obtain the planning domains would be a much more complicated task.

## 1.2 Aims and Contributions

With this idea as a starting point, this thesis aims to develop a set of action model learning techniques that, on the one hand, put special emphasis on the expressiveness of the models they generate and, on the other hand, are able to handle low-quality input data. These techniques are the first steps towards a more ambitious long-term goal: to develop an action model learning technique with sufficient potential to be successfully implemented in the real world.

In order to achieve the proposed goal, 3 sub-objectives are posed throughout the development of this thesis:

- Propose a learning process capable of learning STRIPS action models whose preconditions and effects also contain arithmetic and relational expressions.
- Extend the above process to tackle the problem of learning action models in the presence of noise in the input data.
- Propose a learning process for action models capable of operating using data obtained in simulated environments.

As mentioned above, these objectives are intended to lay the foundations for more complex technology, but even so, they are designed to make a contribution to the state of the art of literature in their own right. Of the contributions that will be presented in this report, the first one is the foundation stone on which the other two are built, and is the first implementation of the learning process proposed in the previous section.

These contributions have been presented indicating that they emphasise the expressiveness of the models that can be obtained with them. In the environment of Automatic Planning, in which we are moving, when we talk about "the expressiveness of a model" we refer to the abstract concept that indicates the capacity of the model to represent the widest possible variety of situations. A planning model has to be able to deal with worlds of enormous potential complexity, and the expressiveness of these worlds is key to being able to represent them appropriately. This complexity may be due to the need to contemplate the handling of numerical information, the handling of timestamps or the design of non-deterministic behaviours. Given the final objective proposed above, and the fact that the real world is the greatest source of disparate situations that exist, a learning process with the potential to obtain the most expressive models possible is paramount.

To address the challenge of action model learning, an algorithm was designed to hierarchically combine different Machine Learning techniques (namely, statistical classification and regression methods) into a single method. This method aims to generate a set of descriptive models (using statistical classification) that represent both the world before and after the execution of a given action. Then, using

these models as a starting point, the proposed process extracts the conditions and effects of that action. To increase the expressiveness of the learned models, preprocessing techniques (based, among other things, on regression) are applied to the input data to generate new knowledge to enrich them. This new knowledge allows to increase the expressiveness of the descriptive models, and therefore, of the action models that can be obtained from them. The designed learning process is defined as a sequence of sub-processes, where each step will apply a different Machine Learning technique. Each sub-process generates a standardised intermediate model, providing it with modularity and therefore allowing part of it to be modified and even new procedures to be added, as long as the input and output requirements of each of the elements of the general workflow are met.

Since the ultimate goal is to obtain a procedure that can be used to learn planning domains that can be implemented in the real world, the treatment of uncertainty in the input data is crucial. Data extracted from real-world processes are far from perfect and, therefore, they tend to be of poor quality. Whether due to human error, problems with the sensors that record the data, or simple chance, data extracted from the real world is often affected by noise. This problem will be addressed during the development of the second sub-objective of this thesis. Although the originally implemented learning procedure showed some resilience to the lack of input data, it also had shortcomings when working with noisy data. Therefore, a set of methods was proposed for the learning process in order to "detect" and "clean" the intermediate models created in the algorithm from errors. Specifically, a procedure was designed, based on statistical and cluster analysis techniques.

As a preliminary step to implementing our learning processes in the real world, it was decided to face the challenge of designing a methodology capable of working in simulated environments. Solving this challenge involves increasing the expressiveness of the learned action models by one step so that they are capable of responding to the problem at hand. The simulated environments used as a testbed come from GVG-AI [PLST+15], an environment developed for the General Artificial Intelligence [GP07] agent competition. From the point of view of the learning task, the collection of environments proposed in GVG-AI is challenging, as it requires the modelling of planning actions with conditional effects. Solving this challenge involves improving the expressiveness of the learned action models so that they are capable of responding to the real problem. With this in mind, the design of the third, and final, contribution of this thesis was undertaken. This contribution is a learning algorithm capable of generating action models for such environments. These action models represent a qualitative leap in terms of expressiveness in relation to the models learned in the previous contributions, allowing multiple behaviours to be modelled in them. This new contribution shares the same core precepts of the original one, which has been evolved and reworked to fit the new learning challenge.

## 1.3 Description of the Document

This document is structured in 6 chapters. **Chapter 1** has presented the contributions designed and implemented throughout the development of the doctoral thesis, as well as the main motivations that led to their development.

**Chapter 2** details in depth the technical concepts necessary to understand the work developed in the subsequent chapters. Specifically, it describes the state-of-the art in the fields of Automated Planning, Action Model Learning and Machine Learning. Furthermore, the video game framework GVG-AI, a basic tool for the development of the last sub-objective of the thesis, is also described. Finally, this chapter includes an extensive bibliographical compilation of the state-of-the-art.

In **Chapter 3**, *PlanMiner*, the basic process for learning domains of action models, is described. This algorithm implements for the first time the learning philosophy proposed above. Throughout this chapter, each of its components is described in detail. In order to support its practical relevance, the chapter ends with extensive experimentation comparing *PlanMiner* with other relevant state-of-the-art algorithms.

The term "noise" refers to the problem that occurs with data, whereby the information it contains is distorted and erroneous. Noise is a widespread problem in all areas of learning, and dealing with it is no trivial task. During the development of *PlanMiner*, certain deficiencies were detected when working with noisy data, which made the task of learning more difficult. These shortcomings are discussed in depth throughout **Chapter 4**, as well as the presentation of *PlanMiner-N*, a direct evolution of *PlanMiner* specially designed to increase the latter's resilience to uncertainty. This chapter highlights in detail the methods added to the original learning process while indicating how they make up for the shortcomings of the original learning process. This chapter ends with the presentation of a series of experiments that demonstrate *PlanMiner-N*'s capabilities in learning action models when faced with situations where the quality of the input data is low. These experiments compare the quality of the models learned by *PlanMiner-N* with both *PlanMiner* and a set of state-of-the-art algorithms.

As previously mentioned, GVG-AI is a very profitable field of application for *PlanMiner*, serving as a springboard for the implementation of the solutions proposed in this document in real-world problems. In the final stages of *PlanMiner-N*'s development, the problem of learning action models in simulated environments began to be addressed. To this end, we proceeded to test whether *PlanMiner* or *PlanMiner-N* performed correctly and were able to obtain a set of action models that could be used to play a GVG-AI video game. This test was a resounding failure, but it allowed us to detect the weaknesses of our previous contributions, ultimately leading to the creation of *PlanMiner-C*, an action model learning algorithm with effects conditioned to the context of the world on which they are applied. **Chapter 5** presents in detail this new algorithm, an evolution of *PlanMiner* that reworks the main components of *PlanMiner* in order to allow the learning of the previously mentioned action models. Together with the relevant explanations, this chapter presents an experimental process that aims to validate the new learning process, obtaining information from an agent running on a series of simulated GVG-AI en-

vironments, in order to reproduce its behaviour in a planning domain.

Finally, **Chapter 6** contains the most interesting and informative observations extracted from this research, aggregating all the relevant insights from the experiments described in previous chapters into the final conclusions. In addition, a series of potentially useful lines of work are provided to improve the work developed throughout this thesis in the future.

## 1.4 Motivación

Desde que se acuñó el término *Inteligencia Artificial* en 1956, el número de áreas de conocimiento y aplicaciones de la misma se han ido multiplicando con el paso del tiempo. De estas áreas, la Planificación Automática [GNT04] es una de los más antiguas y que más atención ha recibido por parte de la comunidad científica históricamente. La planificación automática es la disciplina que se encarga de la producción de planes que permitan alcanzar un objetivo dado, con el fin de de ser ejecutado por un agente (ya sea humano o no). Por desgracia, la implantación de técnicas de planificación automática en el mundo real presenta problemas.

Para poder usar las técnicas de planificación automática hace falta definir con anterioridad una ontología que representa al "mundo" sobre el que se quiere trabajar. Esta ontología (llamada dominio de planificación) contiene una definición de los objetos del mundo que se están representando y sus relaciones, pero más importante, contiene una definición formal de las actividades que pueden ejecutarse en dicho mundo. Estas actividades (llamadas modelos de acciones) suponen el componente básico que forma los planes generados por las técnicas de planificación automática, y su fin último es ser ejecutadas por un agente. La definición de un dominio de planificación es una tarea pesada que requiere de tiempo y conocimiento experto. Esta problemática se agrava enormemente según se incrementa la complejidad del mundo que se esta modela.

Sin embargo, en el área de la Ingeniería de Conocimiento [SBF98], se han propuesto soluciones a esta problemática [JRF+12]. Estas soluciones son las llamadas **Técnicas de Aprendizaje de Modelos de Acción** [AFP+18]. Estas técnicas tienen como objetivo el obtener automáticamente un dominio de planificación a partir de un conjunto de datos de entrada extraídos de ejecuciones de procesos preexistentes, de forma que, dicho dominio, que sea capaz de reproducir los procesos de los que se ha obtenido la información de entrada. En los últimos años se han propuesto una serie de soluciones magníficas que abordan este problema de aprendizaje. Estas soluciones, por lo general, enfatizan el aspecto del aprendizaje utilizando información incompleta, siendo algunas de ellas capaces de aprender bajo una escasez extrema de insumos.

Estas aproximaciones consiguen aprender exitosamente dominios de planificación, pero desde el punto de vista del aprendizaje de dominios de planificación para problemas del mundo real, este enfoque presenta una serie de problemas bastante importantes.

- El primero es que se deja de lado la mejora de la expresividad de los modelos aprendidos, aspecto clave para la implementación de los dominios de planificación en este tipo de problemas.
- El segundo es que se desatiende otros tipos de incertidumbre en los datos de entrada (por ejemplo, el ruido), uno de los problemas más extendidos que acaecen a la información obtenida de aplicaciones del mundo real.

Con afán de cerrar la brecha provocada por estas carencias, dando un paso hacia delante en el estado del arte, se presenta la siguiente línea de trabajo: **Desarrollar**

**un proceso de aprendizaje de modelos de acción con capacidad suficiente para ser implementado en aplicaciones del mundo real, siendo capaz de generar los dominios de planificación más expresivos, al tiempo que maneja incertidumbre en los datos de entrada.** Para poder alcanzar este objetivo, dicha solución basa el funcionamiento de sus diferentes componentes en técnicas del campo del Aprendizaje Automático [MMC13, Mit99].

El uso de tecnología del campo del Aprendizaje Automático abre la puerta a la utilización de algunas de las técnicas más sólidas y bien probadas de la historia de la computación moderna. El Aprendizaje Automático es el campo más prolífico de la Inteligencia Artificial, y basar el desarrollo de las contribuciones propuestas en este documento en él, nos permite acceder a una biblioteca ingente de técnicas, métodos y tecnologías que satisfagan las necesidades del desafío que se presenta.

De entre estas técnicas, las de **Inteligencia Artificial Explicable** levantan un especial interés para el diseño de las técnicas propuestas en este documento. La Inteligencia Artificial Explicable [BDRD+20] es la disciplina que engloba una serie de técnicas y metodologías específicas de aprendizaje automático las cuales pretenden generar soluciones interpretables por los humanos. Usualmente, las técnicas de aprendizaje automático se rigen por funciones objetivo definidas por un modelo matemático, y que, al ser maximizadas/minimizadas las soluciones que generan suelen tener poco sentido desde el punto de vista humano. Ejemplos de esto lo vemos en multitud de técnicas del estado del arte, como las técnicas de aprendizaje de refuerzo, las de aprendizaje profundo, o las máquinas de soporte vectorial, que, pese a ser extremadamente competentes en términos de rendimiento, pero también son extremadamente opacas y no permiten entender sus modelos internos (los llamados modelos de caja negra) lo cual dificulta la extracción de nuevo conocimiento de los mismos. Las técnicas desarrolladas bajo los preceptos de la inteligencia artificial explicable permiten a un operador humano ver e interpretar sus modelos internos (llamados modelos de caja blanca) para poder evaluarlos. De cara al usuario, esto le permite sopesar las decisiones tomadas por el algoritmo y generar conocimiento comprensible para el ser humano.

Desde el punto de vista del trabajo presentado en este documento, las técnicas basadas en inteligencia artificial explicable presentan una gran ventaja, ya que su uso permite acceder a la información que se ha ido extrayendo durante el desarrollo del proceso de aprendizaje, pudiendo obtener conocimiento nuevo de la misma con menos impedimentos. Además, la interpretabilidad de los modelos de inteligencia artificial explicable es crucial para la simplicidad del proceso de aprendizaje, ya que sin ellos, el proceso de traducción necesario para obtener los dominios de planificación sería una tarea mucho más complicada.

## 1.5 Objetivos y Contribuciones

Tomando esa idea base, esta tesis se plantea con el objetivo de desarrollar un conjunto de técnicas de aprendizaje de modelos de acción que, por un lado, hagan especial hincapié en la expresividad de los modelos que generan y que, por otro lado, sean capaces de manejar datos de entrada de baja calidad. Estas técnicas son los

primeros pasos de un objetivo final a largo plazo más ambicioso: desarrollar una técnica de aprendizaje de modelos de acción con potencial suficiente para poder ser implementada en el mundo real de forma satisfactoria.

Para alcanzar el objetivo propuesto, a lo largo del desarrollo de esta tesis se plantean 3 subobjetivos:

- Proponer un proceso de aprendizaje capaz de aprender modelos de acción STRIPS cuyas precondiciones y efectos también contengan expresiones aritméticas y relacionales.

- Extender el proceso anterior para afrontar el aprendizaje de modelos de acción ante la presencia de ruido en los datos de entrada.

- Proponer un proceso de aprendizaje de modelos de acción capaz de funcionar a partir de datos obtenidos en entornos simulados.

Como se ha comentado, estos objetivos tienen como meta sentar las bases de una tecnología más compleja, pero aún así, han sido diseñados para suponer una contribución al estado del arte de la literatura por sí mismos. De las contribuciones que se presentarán a lo largo de esta memora, la primera de ellas constituye la piedra fundacional sobre la que se construyen las otras dos, suponiendo la primera puesta en práctica del proceso de aprendizaje propuesto en la sección anterior.

Estas contribuciones han sido presentadas indicando que hacen hincapié en la expresividad de los modelos que pueden obtenerse con ellas. En el entorno de la Planificación Automática, en el cual nos estamos moviendo, cuando se habla de "la expresividad de un modelo" se hace en referencia al concepto abstracto que indica la capacidad que tiene el mismo de representar la mayor variedad de situaciones posible. Un modelo de planificación debe enfrentar mundos que pueden llegar a ser extremadamente complejos, y la expresividad de los mismos es clave para poder representarlos correctamente. Dicha complejidad puede venir dada por la necesidad de contemplar el manejo de información numérica, el manejo de marcas temporales o el diseño de comportamientos no deterministas. Dado el objetivo final propuesto anteriormente, y que el mundo real es la mayor fuente de situaciones dispares que existe, un proceso de aprendizaje con potencial para obtener los modelos más expresivos posible es primordial.

Para afrontar el desafío del aprendizaje de modelos de acción, se diseñó un algoritmo que combinará jerárquicamente en un único método distintas técnicas de Aprendizaje Automático (concretamente métodos de clasificación estadística y regresión). Este método tiene como objetivo generar un conjunto de modelos descriptivos (usando clasificación estadística) que representen tanto al mundo previo a la ejecución de una acción dada, como al posterior. Después, usando esos modelos como punto de partida, el proceso que se propone extrae las condiciones y efectos de dicha acción. Para incrementar la expresividad de los modelos aprendidos, se aplican técnicas de preprocesamiento (basadas, entre otras cosas, en regresión) a los datos de entrada para generar nuevo conocimiento con el fin de enriquecerlos. Este nuevo conocimiento permite incrementar la expresividad de los modelos

descriptivos, y por lo tanto, de los modelos de acción que se pueden obtener a partir de ellos. El proceso de aprendizaje diseñado está definido como una secuencia de subprocesos, donde cada paso del mismo, aplicará una técnica de Aprendizaje Automático distinta. Cada subproceso genera un modelo intermedio estandarizado, dotándolo de modularidad y por consiguiente permitiendo modificar parte los mismos, e incluso, añadir procedimientos nuevos, siempre y cuando se cumplan los requisitos de entrada y salida de cada uno de los elementos del flujo de trabajo general.

Dado que el objetivo final es el de obtener un procedimiento que pueda ser usado para aprender dominios de planificación que puedan ser implementados en el mundo real, el tratamiento de la incertidumbre en los datos de entrada es de extrema importancia. Los datos que se extraen de procesos del mundo real distan mucho de ser perfectos, por lo que, normalmente, tienden a ser de baja calidad. Ya sea por error humano, problemas con los sensores que toman dichos datos, o por simple azar, los datos que se extraen del mundo real suelen estar afectados por el ruido. Este problema será abordado durante el desarrollo del segundo subobjetivo de esta tesis. Pese a que el procedimiento de aprendizaje implementado originalmente presentaba cierta resiliencia a la falta de datos de entrada, también mostraba carencias a la hora de trabajar con datos ruidosos. Por ello se propuso un conjunto de métodos al proceso de aprendizaje con el objetivo de "detectar" y "limpiar" de errores los modelos intermedios que se crean en el algoritmo. En concreto, se diseñó un procedimiento, basados en técnicas estadísticas y de análisis de grupos.

Como paso previo a la implementación de nuestros procesos de aprendizaje en el mundo real, se decidió afrontar el desafío de diseñar una metodología capaz de trabajar en entornos simulados. Resolver este desafío supone incrementar en un paso la expresividad de los modelos de acción aprendidos, para que sean capaces de dar respuesta al problema que se presenta. Los entornos simulados usados como banco de pruebas provienen de GVG-AI [PLST+15], un entorno desarrollado para la competición de agentes de Inteligencia Artificial General [GP07]. Desde el punto de vista de la tarea de aprendizaje, la colección de entornos propuestos en GVG-AI supone un reto, ya que requiere del modelado de acciones de planificación con efectos condicionales. Resolver este desafío supone mejorar la expresividad de los modelos de acción aprendidos, para que sean capaces de dar respuesta al problema reales. Con esto en mente, se emprendió el diseño de la tercera, y última, contribución de esta tesis. Dicha contribución es un procedimiento de aprendizaje capaz de generar modelos de acciones para dichos entornos. Estos modelos de acciones suponen un salto cualitativo en términos de expresividad en referencia a los modelos aprendidos en las contribuciones anteriores, permitiendo modelar múltiples comportamientos en ellos. Esta nueva contribución se basa en los mismos preceptos que la original, los cuales evolucionan, rehaciéndolos para adecuarlos al nuevo desafío de aprendizaje.

## 1.6 Descripción del Documento

Este documento se estructura en 6 capítulos. El **Capitulo 1** presenta las contribuciones diseñadas e implementadas a lo largo del desarrollo de la tesis doctoral, así como las principales motivaciones que llevaron al desarrollo de las misma.

El **Capítulo 2** detalla pormenorizadamente los conceptos técnicos necesarios para comprender el trabajo desarrollado en los capítulos posteriores. Concretamente, se describe el estado del arte en los campos de Planificación Automática, el Aprendizaje de Modelos de Acción y el campo del Aprendizaje Automático. Además se describe el entorno de trabajo de videojuegos GVG-AI, una herramienta base para el desarrollo del último subobjetivo de la tesis. Finalmente, este capítulo contará con una extensa recopilación bibliográfica del estado del arte.

En el **Capítulo 3** se describe *PlanMiner*, el proceso base para el aprendizaje de dominios de modelos de acción. Este algoritmo implementa por primera vez la filosofía de aprendizaje propuesta anteriormente. A lo largo de este capítulo se describe detenidamente cada uno de sus componentes. El capítulo termina con una extensa experimentación donde se compara *PlanMiner* con otros algoritmos relevantes en el estado del arte.

El termino "ruido", hace referencia a la problemática que acaece a los datos, por los cuales, la información que contienen esta distorsionada y es errónea. El ruido es un problema muy extendido en todos los ámbitos del aprendizaje, y abordarlo no es trivial. Durante el desarrollo de *PlanMiner* se detectaron ciertas deficiencias a la hora de trabajar con datos ruidosos, lo cual dificulta la tarea del aprendizaje. Estas deficiencias son discutidas en profundidad a lo largo del **Capitulo 4**, así como la presentación de *PlanMiner-N* un procedimiento de aprendizaje, evolución de *PlanMiner*, el cual ha sido diseñado especialmente para incrementar la resiliencia del segundo a la incertidumbre. En este capítulo se detallan pormenorizadamente los métodos añadidos al proceso de aprendizaje original, al tiempo que se indica como suplen las carencias del mismo. Este capítulo finaliza con la presentación de una serie de experimentos que ponen de manifiesto las capacidades de *PlanMiner-N* a la hora de aprender modelos de acción al enfrentar situaciones donde la calidad de los datos de entrada es baja. Estos experimentos comparan la calidad de los modelos aprendidos por *PlanMiner-N* tanto con *PlanMiner* como con un conjunto de algoritmos del estado del arte.

Como se ha comentado previamente, GVG-AI es un campo de aplicación de *PlanMiner* muy provechoso, sirviendo de trampolín para la implementación de las soluciones propuestas en este documento en problemas del mundo real. En los estadios finales del desarrollo de *PlanMiner-N*, se comenzó a abordar la problemática del aprendizaje de modelos de acciones en entornos simulados. Para ello, se procedió a comprobar si *PlanMiner* o *PlanMiner-N* se desenvolvían correctamente y eran capaces de obtener un conjunto de modelos de acción que pudieran ser usados para jugar a un videojuego de GVG-AI. Esta prueba fue un rotundo fracaso, pero permitió detectar los puntos débiles de nuestras contribuciones anteriores, llevando finalmente a la creación de *PlanMiner-C*, un procedimiento de aprendizaje de modelos de acción con efectos condicionados al contexto del mundo sobre el que se aplican. El **Capítulo 5** presenta en detalle a este nuevo algoritmo, una evolución de

*PlanMiner* que rehace los componentes principales del mismo, con el fin de permitir el aprendizaje de los modelos de acción previamente mencionados. Junto a las explicaciones pertinentes, en este capítulo se presenta un proceso experimental que tiene como objetivo validar el nuevo proceso de aprendizaje, obteniendo información de un agente que se ejecuta sobre una serie de entornos simulados de GVG-AI, con el fin de reproducir su comportamiento en un dominio de planificación.

Finalmente, en el **Capitulo 6** se presentan las observaciones finales más interesantes extraídas de la investigación desarrollada, exponiendo las conclusiones extraídas de los diferentes procesos experimentales propuestos en los capítulos que forman este manuscrito. Además se aportan una serie de líneas de trabajo potencialmente provechosas para mejorar el trabajo desarrollado a lo largo de esta tesis en un futuro.

# Chapter 2

# Background and Related Work

## 2.1 Introduction

The contributions proposed in this dissertation integrate several fields of artificial intelligence. Throughout this report, a series of solutions are presented that integrate knowledge engineering techniques with machine learning techniques with the aim of learning planning domains. This chapter presents technologies and concepts from these three fields, with the goal of letting the reader understand the technical contributions proposed in later chapters. Additionally, it is pertinent to introduce GVG-AI, the video games simulation environment used in the experimentation of Chapter 5, which is of great interest for the last results presented in this manuscript.

Therefore, we will split the rest of the chapter into four sections: One devoted to diving into the details of automated planning, a second one to present the most significant techniques in the field of machine learning for the work here presented, a third block with a description of the issues about action model learning, and a final fourth block detailing the particularities of the video games framework.

(i) In the automated planning section, we will explain the main planning paradigms as well as the most widespread automated planning language.

(ii) Then, in the section devoted to machine learning we will briefly explain this knowledge area, while also offering an overview of core concepts needed to understand the contributions presented in this document.

(iii) Thirdly, in the part dedicated to action model learning, the most common learning strategies will be presented along with a description of their most typical input data formats; in addition, an in-depth revision of the state-of-the-art of action model learning techniques will also be included.

(iv) Finally, the chapter will conclude with a historical review of GVG-AI and a description of the framework and its components.

## 2.2 Automated Planning

Automated Planning and Scheduling [GNT04] (called AI planning, AI P&S, or AP) is the discipline of artificial intelligence that aims at the production of action plans, usually with the objective of having an agent execute them. AP is one of the oldest and most prolific artificial intelligence disciplines in artificial intelligence. The seed of what is now AP was planted during the Dartmouth conference in 1956 along with the founding of AI itself [MMRS56]. The work of Newell, Shaw, Calman and Herbert [NSS58] in the field of symbolic reasoning and the imitation of human reasoning laid the foundations for what decades later became modern AP. From the first steps of the robot SHAKEY [N+84] —the first automaton to reason using a planner — 50 years ago to the present day, AP has proven to be a discipline capable of being successfully employed in a multitude of different problems. Over the last few years, we have seen how AP has been used for space mission control [NKD+99, MNPW98], emergency management [FOCGPP06], orbital telescope operation [MLJ+88] or underwater vehicle guidance [BR07].

AP addresses the problem of defining strategies to obtain action sequences that solve a problem. These action sequences are designed to be executed by an autonomous entity (a human or an intelligent agent) in a timely manner (actions must be executed in a given order). Prof. Austin Tate wrote in the MIT Encyclopedia of Cognitive Science the following definition: "Planning is the process of generating (possibly partial) representations of future behaviour prior to the use of such plans to constraint or control that behaviour. The outcome is usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents. As a core aspect of human intelligence, planning has been studied since the earliest days of AI and cognitive science. Planning research has led to many useful tools for real-world applications, and has yielded significant insights into the organization of behaviour and the nature of reasoning about actions."

Thus, in order to develop AI planning systems, planning algorithms take a description of the current state (initial state) of the world, a goal and a set of actions as input. The initial state of the world defines the starting point from which the planning system begins to work, and the goal constitutes the desired state of the world. The defined set of actions transforms the world from its current state to the goal. An action is a piece of work that forms one logical step within a process. An action has conditions under which it can be executed, called preconditions, and its impact on the world, called effects. Finally, the description of this set of actions and their ordering (a plan) is the output of the planning algorithm [SW03]. These are the main aspects of planning problems, although even more complex problems can be found when dealing with real planning scenarios, as shown next.

### 2.2.1 PDDL: The Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) was designed to be a neutral specification of planning domains and problems, with neutral meaning that it does not favour any particular planning system [McDoo]. Since then, it has become a community standard for the representation and exchange of planning models. The idea behind this language was to create a core representation of planning problems which was generic enough to be implemented in the highest number of planners possible. The second most important desideratum in the design of PDDL was for it to resemble existing input notations by that time. Readers familiar with PDDL may skip this section if they wish.

```
;; ———————————————————————————— domain ————————————————————————————
(define (domain rovers)
    (:requirements
        :strips
        :typing
    )
    (:types
        waypoint rover - object
    )
    (:constants
        rock soil - sampleType
    )
    (:predicates
        (in ?r - rover ?wp - waypoint)
        (connected ?wp1 ?wp2 - waypoint)
        (sample ?wp - waypoint ?s - sampleType)
        (storedSample ?r - rover ?s - sampleType)
    )
    (:action move
        :parameters (?r - rover ?wp1 ?wp2 - waypoint)
        :precondition (and
            (in ?r ?wp1)
            (connected ?wp1 ?wp2)
        )
        :effect (and
            (not (in ?r ?wp1))
            (in ?r ?wp2)
        )
    )
    (:action sampleRock
        :parameters (?r - rover ?wp - waypoint)
        :precondition (and
            (in ?r ?wp)
            (sample ?wp rock)
            (not (storedSample ?r rock))
        )
        :effect (and
            (not (sample ?wp rock))
            (storedSample ?r rock)
        )
    )
    (:action sampleSoil
        :parameters (?r - rover ?wp - waypoint)
        :precondition (and
            (in ?r ?wp)
            (sample ?wp soil)
            (not (storedSample ?r soil))
        )
        :effect (and
            (not (sample ?wp soil))
            (storedSample ?r soil)
        )
    )
)
```

Listing 2.1: The Rovers PDDL planning domain

The basic version of PDDL is a standardisation of the syntax for expressing STRIPS actions, using preconditions and effects to describe its applicability. The syntax is inspired by LISP [Ste90], and the structure of a domain description is a LISP-like list of parenthesised expressions [FL03]. An early design decision in the language was to separate the description about the ontology of the planning domain from the description of specific objects, initial literals and goals that characterise a given problem. This decision led to the implementation of the planners

that accept two files as input described earlier. The main components of a planning domain will be described in the following lines. To illustrate this, Listing 2.1 shows an example of a domain and a problem written in the PDDL format for the Rovers planning problem.

- **Preamble.** Labelled with `domain`. This section names the domain and serves as a unique identifier which the planner can reference. In our example, the name of the domain is "rovers".
- **Requirements.** Labelled with the tag `:requirements`. This section specifies the requisites that a planner must satisfy in order to be able to solve problems for a given domain. Every requirement must be specified by a certain tag, in the example; the planner must be able to deal with STRIPS-like actions (`:strips`) and object types (`:typing`).
- **Types.** Labelled with the tag :types. This section indicates the different types of objects in the world, categorising them. The different types may be organised in a hierarchical way with a child-parent relation, where child types inherit parent types. `Object` is the default type of the domain, is always defined and is the parent of every other type of the domain. The example domain presents two different types: one to represent rovers and other to represent waypoints. Both are child types of the default type `Object`.
- **Constants.** Labelled with the tag `:constants`. Constants are special objects that do not need to be explicitly included in the different steps of the domain as they are present across the whole planning problem. In our example two constants are defined: rocks and soil, of type `sampleType`.
- **Predicates.** Labelled with the tag `:predicates`. This section defines the properties of the objects of the world and the relations among them. These predicates are literals, as explained earlier, with some (if any) arguments that reference declared objects. Arguments may have types to restrict the objects that instantiate them. In the example of Listing 2.1, a number of literals are defined to determine: the location of a rover (*in* ?*r* − *rover* ?*wp* − *waypoint*), if there is a sample in a waypoint (*sample* ?*wp* − *waypoint* ?*s* − *sampleType*) or stored in a rover (*storedSample* ?*r* − *rover* ?*s* − *sampleType*) or to indicate if two waypoints are connected (*connected* ?*wp1* ?*wp2* − *waypoint*).
- **Actions.** Labelled with the tag `:action`. This section defines an action of the problem; it can be defined multiple times (once for every action) and can be broken down into three subsections: Parameters, Preconditions and Effects.
  - The parameters of the action (tagged with `:parameters`) define the objects of the world that are performing the action, filtering which type of objects can be used with it.
  - The preconditions of the actions (tagged as `:precondition`) are defined as a logical expression (in is most basic version, PDDL only supports conjunction of literals) that must be true in the world state to allow the execution of the action.

– The effects of the action on the world (defined with the tag `:effect`), like the preconditions, are defined as a conjunction of literals to be added to or deleted from (marked with the logical connective not) the world state. An important syntactic restriction in PDDL is that *literals* in the world state can only be accessed in the preconditions/effects block if its arguments are fully defined in the parameters of the action. For example, the move action shown in Listing 2.1 needs as input a rover (the one to be moved) and two waypoints (the starting and ending waypoints respectively). To allow the execution of the action, the rover must be in the starting waypoint and a viable path must exist between it and the ending waypoint. The effects of the action on the world imply the addition of a literal that contains the new location of the rover and the deletion of the literal that indicates its old location.

```
;; ———————————————————————— problem ————————————————————————
(define
    (problem roversProb1)
    (:domain rovers)
    (:objects
        rov1 - rover
        wp1 wp2 wp3 - waypoints
    )
    (:init
        (in rov1 wp1)

        (connected wp1 wp2)
        (connected wp1 wp3)
        (connected wp2 wp1)
        (connected wp2 wp3)
        (connected wp3 wp2)
        (connected wp3 wp1)

        (sample wp2 soil)
        (sample wp3 rock)
    )
    (:goal
        (and
            (storedSample rov1 soil)
            (storedSample rov1 rock)
        )
    )
)
```

Listing 2.2: A Rovers PDDL planning problem

The elements of a PDDL planning problem (Listing 2.2) are the following:

- **Preamble.** Labelled with problem and `:domain`. This section names the problem file —assigning it an identifier —and pairs it with a given planning domain. The name in the domain must be the same in the preambles of both the domain file and the problem file. In the example problem file, the problem is named `roversProb1`, and it needs the domain named rovers.

- **Objects.** Labelled with `:objects`. This section defines the objects which exist within the problem's world. Each object must be named with a unique alphanumeric identifier. Objects must be typed (if necessary). If a type is not given, but the typing requirement is set, the planner will consider the

object as a default type object. We can see in the example that the world of the problem has a rover (`rov1`) and three waypoints (`wp1`, `wp2`, `wp3`).

- **Initial state.** Labelled with `:init`. This section describes the initial state of the world. PDDL forces to write specifically which literals are true in the initial state. Every literal of the initial state must be already defined in the domain file paired with the problem file. Since PDDL follows the CWA, there is no need to specifically state those facts which are initially false, and thus no need to list the negative literals, as typically planners follow the "Closed World Assumption" (CWA) [Kee13]. The CWA considers missing literals as negative (i.e. false), in contrast with the "Open World Assumption" (OWA) [RN16] that interprets missing elements in a world state as unknown and not evaluable instead of false. The initial state of the problem file of Listing 2.2 indicates the starting location of `rov1` (*in rov1 wp1*) , the connectivity grid of the waypoints (*connected wp1 wp2*) and the location of the samples (*sample wp3 soil*).

- **Problem goal.** Labelled with `:goal`. This section defines the goals of the problem. The goal is a logical expression that must be satisfied in the final state to consider a plan as a solution to the problem. The goal must explicitly indicate a given number of literal and its values. Literals that do not appear in the goal are not considered important will be ignored when determining if the goal is reached. The goal of the example problem is for `rov1` to pick up a rock and a soil sample.

### Improvements of PDDL over time

The characteristics of PDDL mentioned here are the characteristics of version 1.2 of PDDL. Over time, new versions of PDDL were published to improve the expressivity of the domains that can be coded, thus improving its capabilities to deal with more complex problems too. There were 5 major iterations of PDDL: ADL, PDDL 2.1, PDDL 2.2, PDDL 3.0 and PDDL+. Those iterations added extra functionalities to the previous versions of the language, with the exception of PPDDL, which is an alternative research line parallel to the main PDDL development. The main contributions of these versions of PDDL are the following:

- **ADL** is the acronym of Action Definition Language, a direct successor of STRIPS. ADL is not a new version of PDDL but an extension of the original PDDL which introduced conditional effects and logical quantifiers. Conditional effects are a new type of effects that are executed in a given action if certain preconditions are met, in addition to the rest of the action's preconditions. The new quantifiers enrich the logical expressions defined in PDDL with $\exists$, $\forall$ and the *or* operator.

- **PDDL 2.1** introduced two key elements: time and numbers. Time is included with a new type of actions called durative actions. These actions include a new section in addition to the classic sections: the duration of the action.

Numbers are managed with a new type of literal, the fluent. Fluents are literals with an associated numerical value. A fluent can be modified (incremented, decremented or assigned) and compared to other fluents, thus allowing a planning domain to deal with arithmetic and relational expressions. The addition of numerical information in PDDL also introduced the ability to track the costs of the domain's actions.

- **PDDL 2.2** included two straightforward improvements to PDDL 2.1: derived predicates and timed initial literals. The first improvement allows for the definition of a relational expression outside an action by assigning it to a named (and thus reusable) literal, which avoids repeating the expression multiple times in the domain. The second improvement enables the definition in the initial state of the moment when a literal will be available.

- **PDDL 3.0** introduced soft constraints into the actions. Soft constraints are a new type of precondition that the planner's user would like to see satisfied in the world state but whose fulfilment is not mandatory.

- **PDDL+** introduced the concepts of Processes and Events. Both elements represent exogenous changes in the world state that happen when a certain precondition is met, without the intervention of an action. The difference between Processes and Events is that Events can only happen once.

- **PPDDL** enables the codification of stochastic behaviours in the PDDL format. A PPDDL (Probabilistic PDDL) action may have an effect (or set of effects) with an associated probability, so that when a planner is building a plan for a given problem, these effects may (or may not) occur. At the time of writing, PPDDL is somewhat obsolete and is being displaced by other technologies such as RDDL[San10].

There are many more versions of PDDL [JSAJ19] that are not listed here. These versions are more niche and have more specific uses so most of the planners in the state-of-the-art does not support them. The most important features in PDDL for this work are the conditional effects and the management of numerical resources, and therefore, for the sake of comprehensibility of this document, we will focus on these in the following pages.

### Planning with numerical information

The management of numeric information included in version 2.1 of PDDL was a significant milestone, as the introduction of this new characteristic broadened the horizon of approachable problems using AP. One of the main contributions of this work is that it can learn PDDL 2.1 planning domains but enriched with this kind of information. The following lines will delve into this characteristic, explaining its different components. This explanation will be illustrated by expanding the rover examples of Listings 2.1 and 2.2 with numerical information.

**Fluents.** PDDL 2.1 introduced a new type of predicates to code numerical information. This new type of predicates are called Fluents (also known as functions). A fluent is a literal that has an associated numerical value instead of a Boolean value. In order to use numerical fluents, the planner must comply with the `:fluents` requirement. Fluents are defined in a different section in the planning domain as typical predicates. The following example presents three new fluents (*battery* ?*r* − *rover*), (*bat_usage* ?*r* − *rover*) and (*distance* ?*wp1* ?*wp2* − *waypoint*) to represent the level of battery remaining in a given rover, its consumption and the distance between waypoints, respectively.

```
(:fluents
    (battery ?r - rover)
    (bat_usage ?r - rover)
    (distance ?wp1 ?wp2 - waypoint)
)
```

For the initial state, fluents are instantiated in the `:init` section of the problem file. In contrast with the predicates, fluents are defined in their own way. Following the prefix notation, fluents are instantiated as (= < *literal* >< *value* >) where < *literal* > is the instantiated literal and < *value* > a number. Below this paragraph an extract of the initial state of a rovers problem can be found, where the fluents of the example above are instantiated. In this example, we can see the battery and consumption of the rover `rov1`, as well as the different distances among the waypoints set in the world.

```
(:init
    (in rov1 wp1)
    (sample wp2 soil)
    (sample wp3 rock)

    (= (battery rov1) 100)
    (= (bat_usage rov1) 3)

    (= (distance wp1 wp2) 10)
    (= (distance wp1 wp3) 5)
    (= (distance wp2 wp1) 10)
    (= (distance wp2 wp3) 15)
    (= (distance wp3 wp1) 5)
    (= (distance wp3 wp2) 15)
)
```

**Relational and Arithmetic Expressions.** PDDL implements the arithmetic and relational expressions using prefix notation. While this notation can be cumbersome to read by a human, it relieves the task of parsing the expression by a computer. Expressions in prefix notation are encoded as:

$$(operator\ operand1\ operand2)$$

where *operator* may be an arithmetic or relational operator, and the operands can be fluents or other arithmetic expressions. The arithmetic operators defined in PDDL are the following:

$$(+\ 4\ (battery\ rov1))$$
$$(-\ 4\ (battery\ rov1))$$
$$(*\ 4\ (battery\ rov1))$$
$$(/\ 4\ (battery\ rov1))$$

Relational operators can be used as any other literal in the logical expressions of the planning problem (i.e. the actions' preconditions or the problem's goal). The relational operators defined in PDDL are the following:

$$(=\ 4\ (battery\ rov1))$$
$$(>\ 4\ (battery\ rov1))$$
$$(\geq\ 4\ (battery\ rov1))$$
$$(<\ 4\ (battery\ rov1))$$
$$(\leq\ 4\ (battery\ rov1))$$

**Increase, decrease and assignment of fluents.** PDDL 2.1 implements a new set of effects to modify a fluent. These new effects complement the addition/deletion lists of the actions. The new effects can be used to assign a value to a fluent or to either increase or decrease its current value:

```
(assign (battery rov1) 4)
(increase (battery rov1) 4)
(decrease (battery rov1) 4)
```

The value to be assigned/increased/decreased can be a constant number or an arithmetic expression that depends on other fluents. An updated version of the move action of Listing 2.1 can be found below. In this action, there is a new precondition that checks if the battery level of the rover that carries out the action is higher than the consumption of the rover multiplied by the distance separating the two waypoints it will travel between. Then, once the action has been executed, the battery is lowered by that amount.

```
(:action move
        :parameters (?r - rover ?wp1 ?wp2 - waypoint)
        :precondition (and
            (in ?r ?wp1)
            (connected ?wp1 ?wp2)
            (>
                (battery ?r)
                ( *
                    (bat_usage ?r)
                    (distance ?wp1 ?wp2)
                )
            )
        )
        :effect (and
            (not (in ?r ?wp1))
            (in ?r ?wp2)
            (decrease
                (battery ?r)
                ( *
                    (bat_usage ?r)
                    (distance ?wp1 ?wp2)
                )
            )
        )
)
```

Listing 2.3: Rovers domain's move action with numerical information

### Conditional Effects

As noted above, conditional effects were included with the ADL version of PDDL. An action with conditional effects has the usual preconditions and effects blocks of STRIPS actions —which work as expected— along with a series of special structures added to the effects. These structures implement the conditional effects, defining a new block of effects that is only triggered if certain extra conditions are met in the world's state. The conditional structures have the following structure:

(*when* (*precondition*) (*effects*))

*precondition* is a logical expression that must be true in the world to allow the execution of the *effects* block. An action with conditional effects can have several *when* blocks, each with its own effects and conditions. Both the conditions and the effects of this block can use features provided by other PDDL contributions (namely increases, decreases, or arithmetic expressions).

Listing 2.4 presents an example of a conditional action developed on the move action (Listing 2.3). In this action, besides performing the tasks seen so far (moving from a point `?wp1` to a point `?wp2`, checking and updating its battery level), the move action keeps track of how many actions the rover has performed in order to request a check and maintenance of its systems after a certain number of actions have been performed. This conditional subroutine is intended to prevent breakage due to the wear and tear of the rover's components.

```
(:action move
        :parameters (?r - rover ?wp1 ?wp2 - waypoint)
        :precondition (and
            (in ?r ?wp1)
            (connected ?wp1 ?wp2)
            (>
                (battery ?r)
                ( *
                    (bat_usage ?r)
                    (distance ?wp1 ?wp2)
                )
            )
            (checked ?r)
        )
        :effect (and
            (not (in ?r ?wp1))
            (in ?r ?wp2)
            (decrease
                (battery ?r)
                ( *
                    (bat_usage ?r)
                    (distance ?wp1 ?wp2)
                )
            )
            (increase (timeout ?r) 1)
            (when
                (and
                    (=
                        (timeout ?r)
                        5
                    )
                )
                (and
                    (not (checked ?r))
                    (assign (timeout ?r) 0)
                )
            )
        )
)
```

Listing 2.4: Rovers domain's move action with conditional effects

### 2.2.2 Planning Strategies

While planning systems address the problems described earlier in this section, different planning strategies have been developed for representing and reasoning about these scenarios. In this section, the planning strategies mentioned and used throughout this dissertation are described. Particularly, the Classical Planning paradigm is fully detailed, since it has been the main paradigm used in this research work.

#### Classical Approach

The basic principle of Classical Planning, also commonly known as STRIPS or STRIPS-like planning [FN71], is finding a sequence of actions, which will modify the initial state of the world into a final state where the goal holds. A state is a set of atoms or literals that define how the objects of the model relate to each other and their traits. A literal is an instantiated predicate of the form $p(arg_1, arg_2, ..., arg_n)$ where $p$ is the name of the literal and $arg_i$ an object of the world. The planner adds

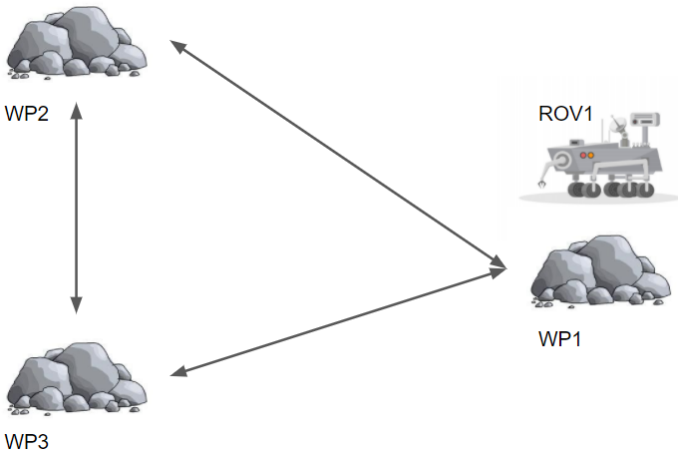



Figure 2.1: The Rovers problem

actions incrementally to the plan, trying to create the correct transformation from the initial to the final state. The STRIPS planning is based on actions defined as a tuple $< Header, Pre, Eff >$ where *Header* contains the name and parameters of the action, *Pre* are the preconditions that must be true to allow the execution of the action —that is, what elements must hold in the state to be able to apply the action— and *Eff* the effects of the action in the world after being executed —namely, what elements change as a result of applying that action.— Preconditions are defined as a conjunction of literals, while Effects can be represented as an Addition list (things to include in the state) and a Deletion list (things to delete from the state) of literals. An action is called a *Grounded Action* if its header is instantiated with objects of the world (and therefore added to a plan).

An example of a classical planning problem is the so-called "Rovers", that simulates the behaviour of a rover in the planet Mars that must move between a set of predefined waypoints to take rock and soil samples. Using the example from Figure 2.1, we can describe a world where there are three waypoints (`wp1`, `wp2`, `wp3`) and a rover (named `rov1`) in `wp1`. The situation where the rover is in a given waypoint is represented with the literal (*in rov1 wp1*). The locations of the samples are determined through the literals (*rockSample wp3*) and (*soilSample wp2*), for the rock samples and the soil samples respectively. In order to travel between two waypoints, they must be explicitly connected using the literal (*connected wp1 wp2*).

Typically, classical planners [Hofo1] take two separate files as input:

- **Problem file.** Encodes the set of objects in the world, the initial state of the problem (i.e. the literals that are true at the beginning of the problem) and the goal state of the problem (i.e. the selection of literals, and their values, which must be in a certain state to resolve the problem).

- **Domain file.** Describes the ontology of the problem. The ontology of a planning domain contains information about the type of the objects described in

the problem file, the predicates that represent the relations among the objects and the actions that can modify these predicates.

The basic planning algorithms of STRIPS-like domains are based on a method called forward-chaining planning (FCP) [CS07]. FCP is described as an informed search method that explores a search space of states in order to reach the objective of the problem. This search space is composed of a set of nodes such that each node is a set of predicates representing a state of the world. Each of the nodes that form the search space is generated after applying an action to a previous node, so that the algorithm generates new nodes from those already explored until a given objective state is reached. To do this, FCP makes use of a heuristic that measures how "close" a given node is to the objective that guides the whole search process; accordingly, planning engines that are based on it are called Heuristic Search planners [BG01]. Some of the best-known Heuristic Search planners in the state-of-the-art are OPTIC [BCC12], POPF [CCFL10], ENHSP [SHTR16], Dino [PFL+16], SMT-Plan+ [CFLM16], COLIN [CCFL09], Fast Forward [Hof01] and MetricFF [Hof03]. Fast Forward and its improved version, Metric-FF, are specially relevant for this work, since they are used during the experimental process of the contributions proposed in Chapters 3 and 4.

### The Fast Forward planner

The Fast Forward planner [Hof01] (FF from now on) was originally proposed by Jörg Hoffmann and Bernhard Nebel of the University of Freiburg in 2001. It was the winner of the International Planning Competition in 2000, and the ideas it originally put forward have had a major impact on the development of planning engines in recent years.

**Algorithm 1** Pseudocode of FF algorithm

**Input** $I$ : initial state, $G$ : problem's goal
**Output** $P$ : plan

```
1:  Initialise P as a empty plan
2:  s ← I
3:  while h(s) ≠ 0 do
4:      s' ← search state from s s that h(s') < h(s)
5:      if s' is ∅ then
6:          return FAILPLAN
7:      end if
8:      P ← path from s s to s'
9:      s ← s'
10: end while
11: return P
```

FF implements a method called Enforced Hill-Climbing. This method combines the well known Hill-Climbing method for local search with a breadth-first search algorithm. At each step of the algorithm, FF selects the first element $s'$ in

the neighbourhood of state $s$ that improves its heuristic value $h(s)$, updating $s$ by said element and repeating the process until the stopping criterion is met. When generating and exploring the neighbourhood of $s$, not only those nodes that arise from applying an action to it (i.e. the immediate neighbourhood) are taken into account, but the whole tree of nodes produced from $s$ is explored using a breadth-first search algorithm. This combination of technologies leads to the search process consisting of prolonged periods of exhaustive search, bridged by relatively quick periods of heuristic descent. After a certain number of nodes have been explored, in the absence of an element with a higher heuristic value than the current one, the algorithm finally stops and indicates that there is no plan to solve the problem. In order to speed up the search process, FF makes use of an abstraction widely adopted by state-of-the-art planners called "relaxed actions", which makes the planner ignore those effects of actions that eliminate elements of the states. The use of these actions has a positive effect on the performance of heuristic computations, as it greatly reduces their computational cost. Metric-FF, on the other hand, extends the original FF concept to allow the original planner to work with advanced PDDL concepts such as numerical information handling, conditional actions or the ADL extension.

## 2.3 Machine Learning

Machine Learning (ML) [MMC13, Mit99] is the discipline of Artificial Intelligence that studies computer algorithms able to improve their performance by themselves. Stanford University defines it as "the science of getting computers to act without being explicitly programmed" emphasising the automated element of the techniques developed in the field. Over the course of time, ML has become a pillar of modern aspects of science, engineering and business. The beginnings of this discipline can be traced back to the 1950s when Arthur Samuel and Frank Rosenblatt made pioneering advances in the field. Arthur Samuel wrote the first program capable of improving its execution, coining the term Machine Learning in the process. This program was an AI capable of playing checkers [Sam59] and was able to study past moves to predict future ones. Frank Rosenblatt designed the technology of the perceptron [Ros58], the most basic component of all existing neural networks. A decade later, Thomas M. Cover and Peter E. Hart published their paper "Nearest neighbour pattern classification", which introduced the world to the K-nearest-neighbour (kNN) algorithm [CH67], one of the most influential and fundamental algorithms in the history of computing. After these first initial steps, machine learning has advanced by leaps and bounds, tackling countless problems while enriching itself with knowledge from other fields.

ML algorithms gain experience from a set of sample data passed as input (called "training data"). This experience is used by the ML algorithms to learn (train) a mathematical model that can be used in several ways. The most usual applications of these models are: making predictions, making decisions or improving algorithm performance. As said earlier, the biggest advantage of ML learning algorithms is that they can achieve this automatically, without the need to be programmed ex-

plicitly to do so. Another strength of ML algorithms is that they can face problems intractable by a human, as they are able to process huge amounts of data without external intervention.

When designing the contributions implemented during the doctoral work, it was determined that the detection and modelling of patterns in the input data was crucial to perform the proposed action model learning work. For this purpose, it was decided to turn to machine learning, as it is the most prolific field of research in these topics. Later in this document, we will extensively explain how the field of machine learning merges with the field of action model learning; for now, we will focus only on presenting the technical concepts necessary to develop the rest of the manuscript.

### 2.3.1 Machine Learning techniques classification

While all ML systems share a common goal —the learning of a model that fits the available data and can generalise its performance to new, unseen data—, how they approach this problem and the use of the learned model determine the nature of the ML algorithms. ML algorithms can be sorted into three categories according to how they approach the learning process: supervised learning, unsupervised learning and reinforcement learning.

- **Supervised learning** algorithms train the models with a set of sample inputs (training data) paired with knowledge about the desired output of the samples. The goal of supervised learning algorithms is to fit a model that learns to correctly map inputs to outputs in future unknown data.
- **Unsupervised learning** algorithms lack the prior information about desired algorithm output. These algorithms take input examples and try to find patterns among them, aiming to group and model the inputs given their similarities by searching for a "hidden structure" that connects them.
- **Reinforced learning** algorithms try to maximise a given reward function that guides the learning process. Starting from a collection of input data, these algorithms predict an output and evaluate it with the reward function. From the feedback of this measure they learn a strategy that maximises it after successive iterations of the learning process.

Each of these paradigms addresses different problems [MMC13], namely Classification, Regression, Clustering and Association. Supervised Learning focuses on the first two, while Unsupervised Learning addresses the second two. On the other hand, Reinforcement Learning does not have such a sharp distinction for the problems it solves. Summarising, we can describe these problems as:

- **Classification.** Statistical classification is the problem of categorising a collection of examples into a set of finite discrete values. This is achieved by quantifying and categorising the different attributes of the data instances (called features) and finding a pattern in the attributes of those instances

that share a group. In supervised learning, the techniques that tackle this problem are simply called classification [KZP07] techniques, and rely on the use of the so-called class labels to group instances and find patterns among elements with the same label.

- **Regression.** Regression analysis [CH15] is the problem of estimating the relationship between a given continuous attribute (called dependent variable) and a set of input attributes (called independent variables). Regression techniques achieve this by learning the mathematical relation $f(X) = Y$ such that $X$ is the set of independent variables and $Y$ the dependent variable.

- **Clustering.** Cluster analysis [Rom04] (simply called clustering) is the ML problem of labelling data according to their characteristics. The clustering problem is very similar to the classification problem but, unlike classification, the label assigned to each piece of data is not known in advance, and must therefore be assigned dynamically given the similarity of the data.

- **Association.** Association rule learning [PS91] is the ML problem of finding a relation between the elements of a data collection. An association rule models an unknown hidden correlation among several features of the data collection.

Of these problems, the most important for our contributions are Classification, Regression and Clustering. The first two are used throughout the work done in this doctoral thesis, while the last one is a key concept in the development of the solution proposed in Chapter 4 of this document.

Finally, ML techniques are based on two main different learning methods: induction and deduction reasoning. Deduction reasoning [SW10] starts from a set of given axioms and infers new knowledge that connects them. Deduction algorithms usually follow a top-down approach by trying to learn a conclusion that must be true if the axioms are valid. The algorithms based on the induction learning [Mic83] paradigm usually follow a bottom-up approach to generalise the initial facts. To do this, inductive learning techniques detect patterns in the input data, generalise them, and try to make them fit as many data observations as possible. These patterns are obtained by selecting a set of hypotheses which are true for the input data in a way that tautologically encompasses them. This second paradigm is the core concept of the learning techniques designed in this document.

### 2.3.2 Datasets

Regardless of the paradigm on which an ML algorithm is based, the problem it solves or the reasoning method it uses, the standard format for representing an input data instance $x$ is an array of size $n$ $x = (x_1, x_2, ..., x_n)$ where $x_i$, $i < n$, are the values of its features, and $x_n$ the value of the dependent variable or class label (whichever applies). Arrays of examples are structured in an attribute-value matrix (see example Table 2.1). This matrix, called dataset, displays the examples in its rows, and the features of those examples in the columns.

| *Length* (*cm*) | *Weight* (*kg*) | *Age* | *Fur length* | *Fur coat* | *Barks*? | **Class** |
|---|---|---|---|---|---|---|
| 36.65 | 4.75 | 13 | Short | Tortoiseshell | *False* | Cat |
| 40.31 | 7.49 | 5 | Long-haired | Calico | *False* | Cat |
| 20.38 | 1.54 | 6 | Heavy | Cream | *False* | Dog |
| 39.53 | 18.88 | 2 | Short | Black Saddle | *True* | Dog |
| 30.86 | 2.07 | 2 | Short | Tabby | *False* | Cat |
| 70.22 | 64.38 | 12 | Double | Brindle | *True* | Dog |

Table 2.1: Example dataset

The class or dependent variable corresponding to each example is defined in the last column on the dataset, and is represented with a given categorical label (*Dog* or *Cat* in the example) or a continuous value, according to whether the dataset is modelling a classification or regression problem respectively. Table 2.1 shows an example of the different types of features that can be encoded in a dataset: a) categorical (*Fur length* and *Fur coat*), b) Boolean (*Barks*?), c) integer-valued (*Age*) o d) real-valued (*Length* and *Weight*).

#### Uncertainty

There are exogenous factors that can affect the quality of the data contained in the datasets. These factors, commonly referred to as "uncertainty", have a major impact on the performance of the various learning algorithms in the literature, making their treatment an important topic of current research. These factors can be classified into two different types: incompleteness and noise. We can summarise these factors as:

- An incomplete dataset may contain no information about the value of an attribute for a given instance, or it may contain no information about that instance at all. Incompleteness can also affect datasets by causing class labels associated with examples to disappear.

- A noisy dataset contains as much information as possible, but some of it may be wrong. For a given attribute, the value that appears in an instance may not be correct, either because there is some fuzziness (the value is not exactly what it should be) in it, or because it has been replaced by a totally random element (called outlier). In the case of class labels, noise is affected by assigning the wrong label to a given example.

### 2.3.3 Explainable Artificial Intelligence

As previously stated, the interpretability of the models used throughout the learning process is paramount. Therefore, the techniques and methodologies of eXplainable Artificial Intelligence (XAI) [BDRD+20] are highly valuable for the work developed in this manuscript. Throughout the development of this thesis, extensive use is made of these techniques in both classification and regression problems.

On the one hand, to work with classification problems, the contributions proposed in this dissertation use a type of white-box classification model called "rules". The rules [GGGP15] are the classification model that best fit our work, and presents the following structure:

**IF** $A_1$ and $A_2$ and ... and $A_m$ **THEN** *Class* is $CL$

with **weight w**

A rule is defined by an antecedent constructed as a conjunction of elements $A_i$ and a consequent that contains information about a class label $CL$. The elements $A_i$ are pairs "*Attr* is *val*", where *val* is a value of the domain of the variable *Attr*. The labels $A_i$ of each variable depends on the type of attribute. In our particular case, they can be True or False for logical attributes or a number for numerical attributes. The degree of confidence in the rule is defined by a weight. In our case, the weight reflects the percentage of examples that support the truth of the rule. This model is used irrespective of the classification algorithm that is implemented throughout the learning process (these algorithms will be presented in the following pages).

On the other hand, the implemented solutions make use of certain regression strategies to do their job properly. The most common regression techniques fit a line that linearly combines a number of variables given as input. Although the knowledge they contain is easily accessible to a human, these functions artificially combine the variables and generate models that are not easily interpretable by a human operator. As the complexity of the problem to be solved increases, so does the complexity of the linear model being fitted. This leads to the need to look for alternative methods that fit non-linear functions or that base their learning on black-box techniques (such as the aforementioned neural networks).

The concept of Symbolic Regression (SR) [Var98] was born in order to provide an interpretable alternative to classical regression models. A symbolic regressor [Koz94, WC13] explores the entire space of mathematical formulae to find one that best fits a given set of values. The search space consists of a set of basic mathematical operators (addition, multiplication and exponent) and a set of operands representing the attributes of the problem. The output of the symbolic regressors is a single arithmetic expression that combines these elements into a single regression model, usually represented in a tree structure (see Figure 2.2).

### 2.3.4 Related ML algorithms

To conclude, in the following lines we will describe briefly some algorithms from the literature used along of this document. These algorithms are ID3, C.45, RIPPER, NSLV and K-means. The first four algorithms are classification algorithms used in the experimentation process of the contributions presented in Chapters 3 and 4, while the last algorithm is a clustering algorithm used extensively in the solution described in Chapter 4.

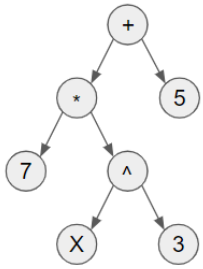



Figure 2.2: Symbolic regression tree of the formula $7 * X^3 + 5$

### Iterative Dichotomiser 3 (ID3)

ID3 [Qui86] is a classification algorithm developed at the University of Sydney in 1975 by J. R. Quinlan. ID3 is a supervised learning algorithm that constructs a decision tree from a set of data points. It constructs decision trees based on an information-gaining principle by measuring the entropy of the decision trees during the construction process. ID3 is restricted to binary classification problems.

The classification algorithm follows an iterative approach: A subset of the input elements is chosen at random, and ID3 fits a decision tree for this subset, validating the model with the rest of the elements. If every element is correctly classified, the algorithm stops. Otherwise, ID3 selects a random number of incorrectly classified elements and adds them to the input subset of elements used to build the decision tree; then, the process is repeated. This approach ensures two things: i) that ID3 fits the decision tree as fast as possible and ii) that ID3 does not overfit the decision tree to the input data.

The component that fits the decision trees in ID3 (Algorithm 2) proceeds by following a greedy strategy: Starting from a set of data points and a collection of attributes, ID3 selects one of those attributes, and, using a divide-and-conquer technique, splits the data points into two subsets. Each of these subsets separates those examples whose values meet a condition imposed on the selected attribute, from those examples that do not (e.g. that the values of an element must be greater than a certain threshold). ID3 stops when every subset contains data points with the same class label. In each step, ID3 selects a set of data points and takes the attribute that provides the largest information gain, and then splits the data points into subsets by their attribute values, thus creating a decision tree node for every subset.

The information gained by splitting a set of data points using the attribute *attr* is computed as:

$$gain(attr) = I(x, y) - E(attr)$$

where $I(x, y)$ is the information required to correctly classify $x$ objects that belong to the class label X and $y$ objects of class Y:

$$I(x, y) = -\frac{x}{x+y} log_2 \frac{x}{x+y} - \frac{y}{x+y} log_2 \frac{y}{x+y}$$

**Algorithm 2** Pseudocode of ID3 algorithm

**Input** $T$ : Training input data, $A$ : set of data attributes
**Output** $DT$ : Decision Tree

```
1: Initialize DT as a node with every element in T
2: if The elements of DT have same goal value then
3:    return DT
4: end if
5: if T is ∅ then
6:    Erase every element in DT, except those with most common goal value
7:    return DT
8: end if
9: end ← False
10: while end = False do
11:    if DT has no leaf nodes with elements with different goal labels. then
12:       end ← True
13:    else
14:       node ← a leaf node with elements with different goal labels.
15:       Select attr, the attribute with highest information gain of A :
16:       for all a in attr do
17:          subset ← elements covered by a
18:          if subset is not ∅ then
19:             Add successor to node connected with an arc labelled by a whose el-
                ements are subset
20:          end if
21:       end for
22:    end if
23: end while
24: return DT
```

Finally, $E(attr)$ is the entropy of *attr*, defined as:

$$E(attr) = \sum_{i=1}^{|attr|} \frac{x_i + y_i}{x + y} I(x_i, y_i)$$

with $x_i$ and $y_i$ as the number of instances covered by $vals_i$ of class $x$ and $y$ respectively.

ID3 has two main limitations: i) It only deals with nominal attributes, limiting the type of problems that ID3 can tackle. ii) It outputs an oversized tree when there are attributes with a large number of different values, as its metric favours this kind of attributes. The first limitation restricts the scope of application of ID3 to STRIPS planning domains. Finally, the decision trees generated by ID3 must be encoded in the form of rules (the reason why this encoding is necessary will be presented in Chapter 3) as defined in the previous section. This is done by creating a rule for each leaf node of the tree so that the antecedent of the rule is formed by all the conditions of the intermediate nodes en route to a given leaf node, and its class label is the one contained in that leaf node.

### C4.5

C4.5 [Qui14] was proposed in 1993 by J. R. Quinlan as an improvement to ID3, aiming to overcome its limitations. C4.5 tackles the issues present in ID3 by including several methods into its structure (see Algorithm 2): I) a new evaluation formula for the attributes; II) a procedure to process continuous values; and III) a pruning step to reduce the size of the output decision tree.

The attribute evaluation function in C4.5 deprecates entropy in favour of the information of the split. Information gain ratio *IG* for a given attribute *attr* is defined as:

$$IG(attr) = \frac{gain(attr)}{split\_info(attr)}$$

where *gain* is the gain measure of *attr* as computed by ID3 and *split_info* is the information contained in the split, measured as:

$$split\_info(attr) = -\sum_{i \in attr} P(\frac{i}{|attr|}) * log_2(P(\frac{i}{|attr|}))$$

with $P(\frac{i}{|attr|})$ as the proportion of elements $i$ in *attr*. As noted, unlike entropy in ID3, information gain in C4.5 is independent of the distribution of the classes in the examples being split. This metric penalises the most those attributes with a high number of different values, reducing the overall size of the learned decision tree.

C4.5 manages attributes with continuous values by implementing a process of synthetic attribute creation. For a given attribute with continuous values, the values are sorted in ascending order and C4.5 selects one as the threshold. It then creates a new attribute to discern if a value of the given attribute is higher or lower than the chosen threshold. C4.5 repeats this process by selecting every value of the original

attribute as the threshold of a new synthetic attribute. These new attributes are added to the set of nominal attributes and used during the tree building process.

Once the decision tree has been learned, C4.5 checks it trying to find branches to prune. Pruning reduces tree over-fitting. The heart of the pruning process is statistical confidence estimates. For a given near-leaf node (an internal node that only contains leaf nodes as successors), C4.5 substitutes it for a leaf node formed by adding together its leaf nodes. C4.5 calculates the confidence interval of the pruned and unpruned tree, and the one with the lower upper limit in the confidence interval is selected. C4.5 selects another near-leaf node iteratively until there are no more left, following a bottom-up strategy. The confidence interval is calculated as follows:

$$CI = x \pm \alpha * \sqrt{x * \frac{1-x}{x+y}}$$

where $x$ is the number of examples of one class, $y$ is the number of examples of the other class and $\alpha$ is a factor that depends on the desired level of confidence of the interval (typically 90% confidence, namely $\alpha = 1.64$).

#### RIPPER

RIPPER [Coh95] is the evolution of the IREP [FW94] rule learning algorithm (see Algorithm 3). Without modifying the main functionality of IREP, RIPPER included several minor improvements that greatly enhanced its performance. RIPPER divides the training set into two sets: the "growing set" and the "pruning set". These sets contain 2/3 and 1/3 of the data of the training set respectively. In each step of the algorithm, RIPPER implements a greedy strategy to build a rule from the data contained in the growing set. This rule is simplified by using the examples of the pruning set to tune them.

The rules are built in an iterative way: starting from an empty rule, RIPPER selects the pair <attribute, value> that covers more examples of the growing set of class $C_i$ and includes it as a condition in the rule. Then, it repeats the process until the rule covers only examples of class $C_i$. The rule learned is largely overfitted, and RIPPER tries to simplify it before including it in the ruleset. This simplification aims to diversify the rule (and hence the ruleset) when facing non-present examples. RIPPER applies the pruning operator (removing a condition from the rule) using the pruning set to measure the impact of the pruning. RIPPER selects the condition that most reduces the error rate of the rule in the pruning set and then removes it. The process is repeated, selecting another condition until no further reduction in the error rate can be achieved. The error rate is computed with the expression

$$pruningError(rule, PruneSet) = \frac{p-n}{p+n}$$

where $p$ is the number of examples of class $C_i$ covered by the rule and $n$ is the number of examples of other classes covered by the rule. The examples covered by the rule are removed from the training set for further iterations, and the rule is included in the output ruleset. The algorithm stops adding rules when there are no

**Algorithm 3** Pseudocode of RIPPER algorithm

**Input** $T$ : Training input data
**Output** $RS$ : Ruleset

```
1:  Initialize RS as an empty ruleset
2:  for all Class C_i in T do
3:     while Number of examples of class C_i in T != 0 do
4:        Split T into GrowSet and PruneSet
5:        Builds rule from GrowSet
6:        Prune rule using PruneSet
7:        if Error rate of rule in PruneSet > 50% then
8:           return RS
9:        else
10:          Add rule to RS
11:          Delete examples of T covered by rule
12:       end if
13:    end while
14: end for
15: Optimize RS
16: return RS
```

more examples of class $C_i$ in $T$ or when the description of the rule surpasses certain length.

The biggest improvement of RIPPER on IREP was the inclusion of an optimisation step of the output ruleset. The optimisation is carried out in the following way: For each rule $R_i$ in $RS$ two new rules are created: the revision rule and the replacement rule. The revision rule is made by adding new conditions to $R_i$ (and then pruning it) using the whole dataset as input instead of just the growing set. The replacement rule is built in an analogous way, but starting from an empty rule instead of $R_i$. Once these two new rules for $R_i$ are created, RIPPER selects one from among the three to be the final rule in $RS$ according to the MDL criterion (Minimum Descriptor Length).

### NSLV

NSLV [GGGP15] is a enhanced version of SLV [GGP14]. SLV was designed by A. González and R. Pérez in 1999 at the University of Granada. NSLV (Algorithm 4) is a rule-based classification algorithm. It makes use of a genetic algorithm to fit the classification rules to the input examples. In each iteration of the algorithm, SLV learns a rule that covers a set of examples from the dataset.

In each iteration of the algorithm, NSLV takes a set of input data, and proceeds to adjust the best rule from a collection of possible rules. The best rule is defined as the one that supports the largest number of positives examples (i.e. examples whose class concurs with the rule's consequent) of a given class and the smallest number of negative examples (i.e. examples whose class does not concur with the rule's consequent). Both components are evaluated asymmetrically when determining

**Algorithm 4** Pseudocode of SLV algorithm

**Input** $T$ : Training input data
**Output** $RS$ : Ruleset

1: Initialize $RS$ as an empty ruleset
2: *rule* ← learning a single rule from $T$ using a genetic algorithm
3: **while** Performance of *rule* $> 0$ **do**
4: Add *rule* to *ruleset*
5: Penalize examples covered by *rule* in $T$
6: *rule* ← learn a single rule from $T$
7: **end while**
8: **return** $RS$

the quality of a rule, with positive examples having more weight in the decision than negative ones. Once a rule is set, it determines which examples in the dataset are covered by it and marks them. Marked examples are penalised in later iterations of the algorithm, which means that they will not be treated as positive examples for a rule but as negative ones. This procedure ensures that the next rules that are adjusted for the same class in the future will contain new information. Once all the examples of a class are characterised, SLV will set a different class label. The whole process is repeated as long as the Performance of the learned rules is greater than zero. Roughly speaking, Performance measures the completeness of the learned rules (i.e. the number of examples that are covered by them). Finally, another particularity about NSLV rules is that they are descriptive, meaning that they contain all key features of the examples, rather than the minimum number of features that form other kinds of models.

The process that adjusts the rules for a given data set and a given class is implemented as a genetic algorithm [Dav91]. This algorithm learns the antecedent of a rule for a given class, encoding in every element of the population a candidate antecedent for the target rule. Each element of the population encodes a chromosome in two parts: the variable level and the value level. The variable level indicates whether an attribute from the examples is selected to be part of the antecedent. Each element of the variable level represents a threshold value that determines whether an attribute deserves to be in the antecedent or not based on its relevance. The value level indicates the value assignments that attributes can have. For each attribute, the value level establishes the presence or absence of a value. This encoding allows the genetic algorithm to modify both the attributes and their values through its genetic operators. The genetic algorithm is guided by a fitness function that measures the quality of an antecedent using a lexicographic evaluation function that considers its completeness (the number of examples it covers), its simplicity (the number of attributes it deems relevant) and its comprehensibility (the number of values assigned to each attribute). This fitness function assigns the highest scores to those antecedents that cover the most examples and have the fewest attributes and values marked as relevant.

In the literature, there are enhancements to NSLV proposed by its authors. Of

all of them, the most significant one is INSLV. INSLV improves upon NSLV by applying a window to select a given number of data from the dataset that is used to generate a set of rules. This process is repeated, increasing the size of the window in each step, so that the rules learned in previous steps are refined in successive stages. This process is repeated until the window size covers the entire input dataset.

### K-means

The K-means algorithm [M+67] was originally proposed by James MacQueen of the University of California in 1967. From the start, this clustering algorithm stood out in the literature due to its popularity and simplicity. K-means partitions a dataset passed as input into $k$ groups, so that each element is assigned to the group whose mean value is closest. This dissertation uses the revision of the K-means algorithm [HW79] presented by Hartigan and Wong in 1975, which is more efficient than the original proposal.

**Algorithm 5** Pseudocode of K-means algorithm

**Input** $T$ : Training input data, $k$ : number of desired clusters
**Output** $K$ : Set of clusters

1: $K \leftarrow$ Initialize $k$ clusters
2: **while** ¬ Convergence criterion is met **do**
3: Assign every element of $T$ to the closest cluster in $K$
4: Recalculate the mean value of each cluster in $K$
5: **end while**
6: **return** $K$

K-means iteratively distributes the different elements $T_i$ of $T$ among the clusters in $K$. In each run of the algorithm (see Algorithm 5), K-means selects each data point $T_i$ and calculates its distance to the centre of each cluster in $K$, assigning it to the nearest cluster. The centre of each cluster is called the centroid and is calculated as the average value of all the elements assigned to it. In other words, each element $T_i$ is grouped with those elements closest to it. This concept of closeness between two points is computed as the Euclidean distance between them. Once all the data points have been assigned, K-means updates the centroids of each cluster with the information from the new points assigned to them. This process is repeated until the convergence criterion of the algorithm is met, which is that, after one run, all clusters have stabilised. A stable cluster is one where, after one run, no element assigned in the previous run has been added or removed. Finally, it should be noted that the set of clusters $K$ must be initialised at the start before entering the main loop of the algorithm. This can be done randomly (the approach used in this work) or by following a greedy strategy.

**Cluster quality measures.** Throughout the development of this PhD thesis, a series of metrics are used to evaluate and measure the quality of the clusters obtained by K-means. These metrics, taken from the state-of-the-art of the field in

question, are the silhouette index [KR09] and the normalised standard deviation of a cluster. Summarising:

- The **silhouette index** of a cluster measures how close the elements of a cluster are among themselves and how far they are to the elements of the other clusters. A cluster with a good silhouette index contains well-matched elements which would be bad matches for the elements of the rest of the clusters.
- The **normalised standard deviation** quantifies how "wide" a given cluster is. A very wide cluster implies that its elements are dispersed in it; thus, its centre is not representative of them. On the other hand, a very tight cluster would imply that its elements are near the centroid and thus they are well represented by it.

On the one hand, the silhouette index of a cluster is computed as the average silhouette index of every data point assigned to it. For a given element $T_i$, its silhouette index is calculated as:

$$s(T_i) = \begin{cases} \frac{b(T_i)-a(T_i)}{max(a(T_i),b(T_i))} & \text{if } |k| > 1 \\ 0 & \text{if } |k| = 1 \end{cases}$$

where $a(T_i)$ is the average distance from the element $T_i$ to the rest of the elements of the cluster:

$$a(T_i) = \frac{1}{|k|-1} \sum_{i,j \in k, i \neq j} distance(T_i, T_j)$$

and $b(T_i)$ is the average distance from $T_i$ to the nearest element of the other clusters.

$$b(T_i) = \min_{k \neq k'} \frac{1}{|k'|} \sum_{j \in k'} distance(T_i, T_j)$$

with $k$ and $k'$ being elements of the set of clusters $K$.

The *distance* between two points is calculated by using the Euclidean distance (as in K-Means). The metric $s(T_i)$ ranges from -1 to 1. A value of 1 means that the $i$-th element is perfectly grouped in its assigned cluster; conversely, a value of -1 implies that the element is wrongly classified. The average silhouette score of all points indicates how well grouped the data are in their clusters.

On the other hand, a cluster's normalised standard deviation is calculated as:

$$nSTD(k) = \frac{\sqrt{\frac{1}{|k|} \sum_{T_i \in k} (T_i - \mu)^2}}{\mu}$$

where $k$ is a cluster of $K$ and $\mu$ its centroid. A cluster's normalised standard deviation quantifies how disperse the elements of the cluster are in relation to the cluster's centroid. The $nSTD(k)$ score has a range of $[0,\infty)$. The higher the value of $nSTD(k)$, the wider the cluster, and hence, the more separated the data points are from the centre of the cluster.

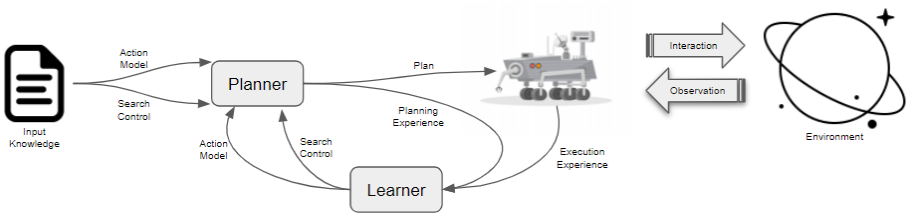



Figure 2.3: Integrated Planning-Execution-Learning architecture

## 2.4 Action Model Learning

AP is a powerful technology that can deal with a wide variety of situations and environments, but this capability comes with some drawbacks when trying to introduce AP systems into complex applications. Chief among these problems is the need to gather and maintain knowledge about the problem to be solved to enable AP techniques to solve it. These tasks (acquisition and maintenance of knowledge) are very burdensome and can require a great deal of time and effort. Over time, the different techniques for dealing with these tasks were formalised with Knowledge Engineering [SBF98] (KE). The KE field is an AI discipline dedicated to formalising knowledge using a series of multidisciplinary techniques. These techniques are responsible for modelling, acquiring, measuring and evaluating any kind of knowledge.

The application of KE techniques in AP not only allows to better encode the aforementioned information about the problems to be addressed, but also, knowledge acquisition techniques can alleviate the process of implementing AP techniques in more complex problems. As seen earlier in this chapter, the knowledge of the problems is usually encoded in a document called planning domain. The codification of these documents is a cumbersome and resource-intensive task and usually requires an expert to bring in extensive knowledge of the problem. This inconvenience grows with the complexity of the problem being tackled, as there are more situations to contemplate and address explicitly. In addition, the search methods on which the planning engines are based are often PSPACE-Complete [Byl94], which, together with generic and uninformed heuristics implemented in them, makes solving slightly large problems extremely costly. Hence, a poorly designed domain would further increase this issue to the point of rendering the problem unsolvable in practice [MRS02]. All these problems cause the task of hand-coding a planning domain infeasible under certain circumstances and relying on techniques that automatically realise this work is a must.

Knowledge acquisition techniques applied to AP can be divided into two main areas given the problem they aim to solve [JRF+12]. These fields are: action model acquisition, and search control learning. Action model acquisition techniques are concerned with gathering information about the operations that must be executed to solve a given problem, while search control learning is concerned with determining the optimal search strategies to address the problem. These KE techniques

can be merged with AP techniques in a constant cycle of Planning → Execution → Learning (see Figure 2.3), where a system would be able to learn from its own executions, iteratively refining the knowledge used to solve the problems, and consequently, gradually generating better solutions. Starting from a solution obtained using an AP technique (the planner), information about its execution on the problem's environment is collected. This information is then fed to a knowledge acquisition technique (the learner) that improves (or substitutes) the original knowledge about the problem contained in the planner. For the sake of comprehension and brevity in the rest of this work, we will only discuss the Action Model Learning techniques since they are those that occupy the scope of this PhD thesis.

If we briefly analyse the characteristics of PDDL we can see why programming a planning domain is extremely tedious. The need to explicitly define any kind of action required to solve a problem is preceded by a detailed study of it. This study must detect any possible situation that may be encountered during problem-solving, and define a path to obtain a valid plan to solve the problem. As previously mentioned, this task is very costly, both in terms of time and resources, and requires an expert with extensive knowledge of the problem being addressed. From a computational standpoint, an error in determining the number of actions (or their parameters) can negatively impact the performance of the planning engine. This is due to the way these engines tend to work: their almost exclusive reliance on heuristic search hinders their performance when a large number of elements (either actions or action parameters) are involved.

These problems were highlighted by Shen and Simon [SS89] in their work on the LIVE system in 1989. Throughout the 1990s new approaches to this problem appeared in the literature [Gil94, Ben96, Wan95, OC96], but it was not until 2008 that a milestone in this field was reached. The milestone we are talking about was the creation of the "Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)" [BM08] within the "International Conference on Automated Planning and Scheduling" [RNBH08] in 2008. In the words of its promoters, the aim of KEPS was "to promote the knowledge-based and domain modelling aspects of AP, to accelerate knowledge engineering research in AP and to encourage the development and sharing of prototype tools or software platforms that promise more rapid, accessible, and effective ways to construct reliable and efficient AP systems". In short: the aim of KEPS is to formalise the application of KE techniques in the field of PA. In the field at hand (knowledge acquisition), the advent of KEPS led to the emergence of multiple learning techniques for planning domains.

The present section will delve deeper into the aforementioned techniques and the challenges of applying KE methods to AP. As said earlier, we will specifically focus on AML techniques, presenting and highlighting its particularities, but will not go in depth into Search Control techniques in favour of a more comprehensive exposition of the topic. First, this section will present a categorisation of the different AML techniques. This categorisation will explain their characteristics, and how they influence the learning process. Second, the different technologies that can be implemented in an AML technique will be presented. For each technology, we will provide a description of the most successful state-of-the-art AML techniques, especially those closely related to the technique presented in this dissertation.

### 2.4.1 AML techniques classification

Although the goal of AML techniques is to learn action models, the way in which they address the problem differs markedly among different techniques [JCKV14]. Learning requirements vary greatly from technique to technique depending on the characteristics of (i) the models to be learned and (ii) the quality (and quantity) of the available input data [AFP+18]. For example, (i) could limit the application of certain techniques or impose certain requirements on the learning process, while (ii) directly influences the type of domains that can be obtained or whether extra techniques need to be implemented to perform the learning task correctly.

#### Model characteristics

Since the ultimate goal of AML techniques is to obtain a planning domain model, it is safe to assume that taking into account a model's characteristics is fundamental when designing them. The characteristics of the learned domains not only define the scope of application of a given AML technique, but they also delimit the different techniques that it will implement internally during its learning process. For AML techniques, planning domains are defined by two viewpoints: the expressiveness of the domain's actions and whether they are deterministic or not. These characteristics are independent of the format in which the domain is written (i.e. STRIPS-like, PDDL, etc.).

1. **Expressiveness.** This characteristic measures the ability of the AML technique to output a planning domain that encodes the widest range of situations. In the lower spectrum of the domain's expressivity are planning domains that only contemplate STRIPS actions; on the contrary, a highly expressive domain will handle numerical information, time, resources and/or conditional effects.

2. **Determinism.** A deterministic domain will always get the same result for the execution of a plan from the same initial state. The assumption that each action does not always behave the same may heavily impact the design process of AML techniques. The existence of non-deterministic behaviours raises doubts about the correctness of scarce data patterns during the learning process, since it is not possible to determine whether or not anomalous behaviour is erroneous or not.

The characteristics of the learned domains affect the design of AML techniques by promoting (or directly forcing) the use of certain learning methodologies. For example, AML techniques that are designed to deal with resources, time or numerical fluents cannot be based on technologies that only handle boolean information. This may suboptimally work in the opposite way: if a given problem will only deal with logical predicates, an excessively powerful AML technique will solve the problem but lack the efficiency of a learning algorithm optimised for that kind of information. A deterministic domain will force the AML technique to make some assumptions about the input when it encounters unusual input data. These

assumptions may lead to marking uncommon input data as "noise" and therefore ignoring it during the execution. With a non-deterministic domain, the task of separating uncommon valid examples from noise is, at least, harder, and in certain situations impossible. This issue forces the implementation of different procedures in the AML technique to deal with it properly. Table 2.2 shows an overview of the strengths and weaknesses of the different AML approaches, sorting them by the characteristics of their learned domains.

### System input

Input data is of vital importance in any AI technique based on knowledge extraction and learning [Pyl99]. AML techniques are no exception, as input data must be carefully considered during development. The overall performance of the learning process depends on the quality and quantity of the available input data. When the input data does not reach the expected quality standards, it may be necessary to implement certain strategies and techniques to enable the learning task to work correctly. In this dissertation, we will distinguish between two types of input data: plan traces and prior knowledge of the model being learned. The next lines will explore, for these two types of input, the repercussions of the aforementioned standards on the entire learning process. But, first of all, let us define what a plan trace is.

**Input plan traces** AML techniques usually base their learning on the use of plan traces [JRF+12]. A plan trace is a proof of the execution of a plan on an environment. The traces not only contain information about the list of actions that make up the plan, but also a summary of the state of the environment as these actions were executed. Regardless of the format in which the traces are written, the traces are defined as a list of tuples

$$< s_{i-1}, a_i, s_i >$$

where $a_i$ is an action of the plan executed in the state $s_{i-1}$ —named *pre-state*—, and $s_i$ is the state resultant of the execution of the action —the *post-state*—. In a plan trace with $n$ tuples, $s_o$ would be the initial state of the problem, while $s_n$ would be the state in which the goal of the plan was achieved.

Some variations on this model have been defined to cover the particularities of certain techniques. For example, it is possible to find traces that do not represent the states at all [CMW13] or traces that are missing some (or all) actions of the plan[ACO19]. The way in which states are coded may also vary, as they may follow, for example, the CWA or the OWA.

As well as with datasets (see Section 2.3.2 of this chapter), the quality of a plan trace may be affected by two different factors: Incompleteness and noise.

- An incomplete plan trace is a trace where some elements are missing. These elements can be actions of the plan, predicates of some state, or whole states.
- A noisy plan trace is a trace where the observations have been wrongly performed. Errors in the observations can be produced by a predicate with a

| | | **Determinism** | |
|---|---|---|---|
| | | **Deterministic** | **Non-Deterministic** |
| **Expressiveness** | **High** | Strengths:<br>• Covers all domains usable by top-of-the-line planning competitions<br>• Good coverage of learning techniques of diverse fields (e.g. machine learning)<br>Weaknesses:<br>• Fails to represent the uncertainty of complex real-world problems | Strengths:<br>• Can virtually handle every possible planning situation<br>• Learns the most expressive domains possible<br>Weaknesses:<br>• Requires highly complex learning and planning algorithms<br>• Has a tough time facing typical problems of real-world data like noise |
| | **Low** | Strengths:<br>• Can work with extremely efficient learning and planning techniques<br>• Requires fewer input data than other approaches to work properly<br>Weaknesses:<br>• Poor coverage of complex planning problems | Strengths:<br>• Fairly efficient learning algorithms<br>• Is able to represent simple random exogenous events of the planning world<br>Weaknesses:<br>• Uncertainty has a considerable impact even when learning simple problems |

Table 2.2: Characteristics of the AML approaches according to their output

value that is different from the real one, or by the appearance of an action that is different from the one executed in a given point.

Incompleteness and noise are not mutually exclusive and may simultaneously affect a plan trace. The uncertainty of the data —as this is called—leads to the use of algorithms with high fault-tolerance that may ignore missing or erroneous data. This is in addition to the necessity of implementing procedures to pre-process the input data in order to relieve some uncertainty issues.

Table 2.3 shows an example of a complete and non-noisy plan trace. This plan trace presents a plan where a rover `rov1` moves from `wp1` to `wp3`, traversing `wp2`, and then returning to `wp1`. In `wp2` and `wp3`, `rov1` takes samples of the soil and the rocks respectively. The first part of the table (sub-table 2.3a) lists the state transitions of the trace, where the second part (sub-table 2.3b) presents the intermediate states observed during the execution of the plan. The format used to display the state transitions had three different parts: two indexes to link with their associated pre-states and post-states, and a task header. The plan traces used in this work follow the OWA. In the following chapters we will define how we pose the learning problem and we will properly explain the reason behind this issue, but it can be summarised as: the more information contained in the plan traces, the better. This issue will be expanded on in further chapters of this manuscript.

**Plan traces and static relations.** This plan trace representation format is the one used in the first two contributions presented in this manuscript (described in Chapters 3 and 4). The experimentation in these contributions is performed on benchmark planning domains from the automated planning literature, which implies that the domains have a moderate size and the volume of information contained in the states is reduced. When it comes to representing the information of other sources, these premises are no longer fulfilled, since they can contain a huge volume of information. When tackling the task of encoding this information into a plan trace for our solution, we may find that it is not feasible due to the volume of data to be processed. For example, in the plan trace model presented above, the connection between locations must be encoded. For each pair of connected elements, a predicate must be necessarily included in a given state and, as the original plan traces follow the OWA, a predicate for every unconnected pair of elements must also be included. This causes a combinatorial explosion of predicates that needlessly increases the size of the traces, especially when this information must be repeated in each state of the trace. In the benchmark planning domains used in the experimental process of Chapters 3 and 4 this issue is negligible, but when facing the challenges presented in Chapter 5 it leads to an unmanageable problem due to the source used to get the experimental data. The experimental data is obtained from videogames observations (detailed later in section 2.5 of the present chapter) in which the world is defined as a grid where the different cells are connected. The volume of this information is huge, but it must be stored for further use. Before developing the techniques presented in the aforementioned chapter of this dissertation, it was mandatory to tackle this problem and design a new model of the plan traces capable of processing all the information and storing it more efficiently in a

| **Pre-state** | **Post-state** | *Action* |
|---|---|---|
| **0** | **1** | *(move rov1 wp1 wp2)* |
| **1** | **2** | *(sampleSoil rov1 wp2)* |
| **2** | **3** | *(move rov1 wp2 wp3)* |
| **3** | **4** | *(sampleRock rov1 wp3)* |
| **4** | **5** | *(move rov1 wp3 wp1)* |

(a) State transitions

| **Index** | *Predicates* |
|---|---|
| **0** | *(in rov1 wp1) ∧ (¬ (in rov1 wp2)) ∧ (¬ (in rov1 wp3)) ∧ (sample wp2 soil) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (¬ (storedSample rov1 soil)) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 100) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15) ∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |
| **1** | *(¬ (in rov1 wp1)) ∧ (in rov1 wp2) ∧ (¬ (in rov1 wp3)) ∧ (sample wp2 soil) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (¬ (storedSample rov1 soil)) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 70) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15)∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |
| **2** | *(¬ (in rov1 wp1)) ∧ (in rov1 wp2) ∧ (¬ (in rov1 wp3)) ∧ (¬ (sample wp2 soil)) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (storedSample rov1 soil) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 65) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15)* |
| **3** | *(¬ (in rov1 wp1)) ∧ (¬ (in rov1 wp2)) ∧ (in rov1 wp3) ∧ (¬ (sample wp2 soil)) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (storedSample rov1 soil) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 20) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15)∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |
| **4** | *(¬ (in rov1 wp1)) ∧ (¬ (in rov1 wp2)) ∧ (in rov1 wp3) ∧ (¬ (sample wp2 soil)) ∧ (¬ (sample wp3 rock)) ∧ (storedSample rov1 rock) ∧ (storedSample rov1 soil) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 15) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15)∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |
| **5** | *(in rov1 wp1) ∧ (¬ (in rov1 wp2)) ∧ (¬ (in rov1 wp3)) ∧ (¬ (sample wp2 soil)) ∧ (¬ (sample wp3 rock)) ∧ (storedSample rov1 rock) ∧ (storedSample rov1 soil) ∧ (= (bat_usage rov1) 3) ∧ (= (battery rov1) 0) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15)∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |

(b) States list

Table 2.3: Extract of a rover's domain plan trace.

limited space.

This new format for plan traces (see Table 2.4) is similar to those initially proposed (in that it is composed of a list of actions followed by a list of states), but it also contains a new block that stores information about the static relationships of the problem's world. In a planning problem, a static relationship is defined as a relation that remains unchanged throughout the execution of the plan, as opposed to a dynamic relationship, which evolves. The new plan traces separate all the static relationships from the states, including them in a new information block at the end of the file. This eliminates predicate redundancy, substantially reducing the size of the traces as a lot of repeated information is erased. Given that the static relations represent constants, we can assume that, if they do not appear in the initial state of the problem, they do not exist. This leads to the possibility of encoding them according to the CWA. As indicated earlier, the CWA considers all predicates that are not defined explicitly in a state as false. For practical purposes, this leads to only defining in the new plan traces those static relations that are true or have a numeric value in the initial state, which further reduces the number of predicates contained in them. Compared to the trace in Table 2.3, the newly formatted plan trace presented in Table 2.4 takes up 50.4% less space. This proportion of space saved increases as the size of the plan trace increases.

Finally, when creating the plan traces used in the different experimental processes presented in this document, the following procedure is used: Starting from an initial state, a planning domain and a plan, we take as state $[0]$ of the trace the aforementioned initial state, and we create the rest of the states $[i]$ sequentially on the basis of applying action $a_i$ to state $[i-1]$. To do this, the elements of $[i-1]$ are carried over to $[i]$ and modified according to the addition and deletion lists of the action $a_i$. In the specific case of the traces defined with the second proposed format, an extra step is taken to detect and separate static relations from dynamic relations. This is done by observing all the states of the trace in search of those elements whose value does not vary throughout the trace, and then extracting them from the states and including them in their own block in the plan trace.

**Background knowledge.** Along with the plan traces, an AML technique may be fed with information about the model to be learned. This information is the so called "background knowledge". The concept of background knowledge [JCKV14] is heterogeneous and encompasses a wide array of different information. Roughly speaking, all knowledge related to the model to be learned is considered "background knowledge", including information about the ontology. For the sake of understanding, we will sort the background knowledge into two categories according to its origin:

1. Implicit background knowledge. This kind of knowledge is contained in the plan traces and, after some processing, can be extracted and used by the AML techniques. Examples of implicit information are task headers, predicates, or the type of those parameters (if any).

2. Explicit background knowledge. This kind of knowledge is exogenous to the

| **Pre-state** | **Post-state** | *Action* |
|---|---|---|
| **0** | **1** | *(move rov1 wp1 wp2)* |
| **1** | **2** | *(sampleSoil rov1 wp2)* |
| **2** | **3** | *(move rov1 wp2 wp3)* |
| **3** | **4** | *(sampleRock rov1 wp3)* |
| **4** | **5** | *(move rov1 wp3 wp1)* |

(a) State transitions

| **Index** | *Predicates* |
|---|---|
| **0** | *(in rov1 wp1) ∧ (¬ (in rov1 wp2)) ∧ (¬ (in rov1 wp3)) ∧ (sample wp2 soil) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (¬ (storedSample rov1 soil)) ∧ (= (battery rov1) 100)* |
| **1** | *(¬ (in rov1 wp1)) ∧ (in rov1 wp2) ∧ (¬ (in rov1 wp3)) ∧ (sample wp2 soil) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (¬ (storedSample rov1 soil)) ∧ (= (battery rov1) 70)* |
| **2** | *(¬ (in rov1 wp1)) ∧ (in rov1 wp2) ∧ (¬ (in rov1 wp3)) ∧ (¬ (sample wp2 soil)) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (storedSample rov1 soil) ∧ (= (battery rov1) 65)* |
| **3** | *(¬ (in rov1 wp1)) ∧ (¬ (in rov1 wp2)) ∧ (in rov1 wp3) ∧ (¬ (sample wp2 soil)) ∧ (sample wp3 rock) ∧ (¬ (storedSample rov1 rock)) ∧ (storedSample rov1 soil) ∧ (= (battery rov1) 20)* |
| **4** | *(¬ (in rov1 wp1)) ∧ (¬ (in rov1 wp2)) ∧ (in rov1 wp3) ∧ (¬ (sample wp2 soil)) ∧ (¬ (sample wp3 rock)) ∧ (storedSample rov1 rock) ∧ (storedSample rov1 soil) ∧ (= (battery rov1) 15)* |
| **5** | *(in rov1 wp1) ∧ (¬ (in rov1 wp2)) ∧ (¬ (in rov1 wp3)) ∧ (¬ (sample wp2 soil)) ∧ (¬ (sample wp3 rock)) ∧ (storedSample rov1 rock) ∧ (storedSample rov1 soil) ∧ (= (battery rov1) 0)* |
| **Static Relations** | *(= (bat_usage rov1) 3) ∧ (= (distance wp1 wp2) 10) ∧ (= (distance wp1 wp3) 5) ∧ (= (distance wp2 wp1) 10) ∧ (= (distance wp2 wp3) 15) ∧ (= (distance wp3 wp1) 5) ∧ (= (distance wp3 wp2) 15) ∧ (connected wp1 wp3) ∧ (connected wp3 wp1) ∧ (connected wp1 wp2) ∧ (connected wp2 wp1) ∧ (connected wp2 wp3) ∧ (connected wp3 wp2)* |

(b) States list

Table 2.4: Updated rover's domain plan trace with static relations.

plan traces and is provided additionally to the plan traces. Explicit information provides extensive insight about the domain being learned that can be obtained from nowhere else. An incomplete action model (i.e. an action model that lacks some preconditions or effects) would be an example of explicit information.

All AML techniques need background information to a greater or lesser extent to work properly, and its availability (provided either by implicit or explicit sources) reduces the effort required for them to work. For example, an AML technique that receives plan traces without the task header information would need to find patterns among pre-states and post-states in order to infer them by itself, which is a harder learning task. Conversely, an AML that knows beforehand the goal of every action will only need to find a suitable collection of predicates that achieves it, thus easing the whole process.

### 2.4.2 Action model learning paradigms

In the previous sections, we have talked about how the characteristics of the inputs and outputs influence the learning process of AML techniques by either facilitating or complicating it, but without going into detail about these processes. This section aims to shed some light on the subject by explaining the most common learning paradigms used in the state-of-the-art, as well as showing practical examples of each of them. For each strategy, a bird's eye view will be given of its general operation, input data necessary to make it work and the type of output produced. In addition, we will relate the aforementioned learning paradigms to state-of-the-art AML techniques from the literature, highlighting their strengths and weaknesses in each case.

#### Inductive learning

Inductive learning-based AML techniques do not differ in their core functionality from the inductive learning ML techniques presented earlier. Inductive learning techniques need a set of hypotheses, a set of examples and, optionally, background knowledge; their output is a set of rules consisting of elements from the set of hypotheses that explain all input examples and do not contradict the background information. Inductive learning is one of the most important learning paradigms in machine learning, and a wide array of learning techniques are based on it.

Although this entirely depends on each particular inductive learning technique, we can generalise the strengths of the paradigm as:

- Able to learn models from scratch.
- Able to deal with logical, categorical and numerical information.
- High resilience to uncertainty.
- Given the existing number of approaches based on this paradigm, it is not difficult to find one that meets the requirements of a particular learning task.

In contrast, weaknesses include:

- Learned models may not make sense from a human point of view.
- Because of its blind learning, it is very dependent on the information contained in the input traces.

Inductive learning is the jack-of-all-trades paradigm, standing out when addressing learning problems about which we do not have prior information [ZK03]. It is useful when learning domains with high expressiveness, and its ability to find models for subsets of the input data allows it to detect non-deterministic effects. In addition, it allows incomplete or noisy data to be used as input in the absence of explicit information. As we noted earlier, inductive learning is one of the most widespread learning paradigms, and this is represented below with many different implementations:

- PELA [JFB08] is a technique based on the use of decision trees. For each action of the problem, PELA generates a decision tree from the plan traces used as input. From each of these decision trees, the algorithm extracts an action model. PELA can compile the decision trees into action models with conditional or probabilistic effects as needed. To obtain the models, PELA uses the TILDE algorithm [BDR01].
- Jiménez et al. [LJFB07] address the learning task using regression trees. The implementation of regression trees allows this technique to learn effects with numerical information. This is achieved by fitting a regression tree for each numerical fluent in the input data. Then, these trees are converted into conditional effects by assigning values to the modelled fluents according to the values of the rest of the input attributes.
- LOCM algorithms. The LOCM family of algorithms refers to 3 different algorithms, based on the same learning technique: LOCM [CMW13], LOCM2 [CG11] and NLOCM [GL16]. LOCM is an AML technique that learns from a set of plans without information about the intermediate states of the execution. LOCM relies on the use of finite state machines to learn static relations among the objects of the action headers of the input plans. For each object, a single finite state machine is fitted. These static relations are used to create the action models. This limits the complexity of the relations that can be learned using the algorithm. LOCM2 overcomes this by learning set finite state machines for each object, improving the coverage of possible domains that can be learned. NLOCM iterates on the solution proposed in LOCM2 to provide a technique capable of learning planning domains with action costs. Each plan is assigned a cost, and by using constraint programming techniques NLOCM is able to infer the cost of each separate action.
- Opmaker and Opmaker2. Opmaker [MRS02] implements a mixed learning process based on plan traces and user interaction. Given an action, the user must indicate the state resulting from applying it, and Opmaker calculates its

action model. To support this interaction, Opmaker makes use of the GIPO [JCKV14] algorithm. Opmaker2 [MCRW09] is the natural evolution of Opmaker and presents a set of procedures to automatically infer information that previously had to be supplied by the user. These procedures combine a series of heuristics to infer the resulting states from the input data.

- FAMA [AJO18] compiles the learning task as if it were a planning task. So, for each action, it defines a planning domain and a problem from the input plan traces. The domain contains a set of operators that add preconditions and effects to the action being learned and another set of operators to validate it. The plan obtained by solving the planning problem contains the information needed to build the desired action model.

- Pasula et al. [PZK04] approaches its learning process based on modelling the world using probabilistic relational rules. To this end, the learning algorithm is implemented in a three-level greedy search algorithm that i) learns a set of rules, ii) induces input/output relations between the rules learned in the previous step and iii) extracts the parameters of the action models from the output relations. In [PZK07], the authors present an evolution of this concept, improving the learning process to allow the inclusion of noise in the input data. The action models learned by both algorithms contain stochastic effects.

- Zettlemoyer et al. [ZPK05] proposes a learning method similar to that defined in Pasula et al. [PZK04] which, after learning the probabilistic relational rules, applies a process to learn background knowledge. This process iteratively builds increasingly complex concepts, testing their usefulness by comparing them with previously learned rules. This method can learn models of actions with probabilistic effects using noisy input data.

- Mourao et al. [MZPS12, Mou14] learn action models by fitting a collection of models using support vector machines and combining them in a single model. The learned models contain descriptions of the states on which a given action applies. Once all the different models are combined into a single model, it is used to generate the output STRIPS action model. This work is similar in its approach to the learning problem to the contributions proposed in this manuscript, and is able to work with uncertainty in the input data.

- AMAN [ZNK13] fits a set of models that represent all possible situations defined in the plan traces. Then, assuming that these models are noisy, it creates a graphical model that captures the relationships between the actions and states of the trace. This model is used to 'predict' what a noise-free trace would look like, and with it discard the erroneous elements of the model set. The authors presented a new version of the algorithm [ZPK19] capable of working with plan traces with parallel actions or disordered elements.

Special mention should be made of inductive learning methods that integrate reinforcement learning like LOPE [GMB00], a comprehensive planning/execution/

learning process (Figure 2.3) that employs reinforcement learning to learn the models of its actions. Starting from an uninformed model, LOPE generates a plan that it must execute to achieve a proposed goal. It then executes that plan and calculates the difference between the states predicted during the making of the plan and those it observes of the world as it goes about its work. These differences are used as a reward function and propagated to the action models, refining them step by step.

### MAX-SAT based learning

MAX-SAT [BHvM09] is a mathematical problem which consists in determining the maximum number of clauses in a logical formula that can be satisfied by assigning truth values to their variables. There is a variant of the problem that considers that each clause has a given weight, and the objective of the problem is to maximise the sum of the weights of the satisfied clauses. The benchmark MAX-SAT solving strategy is DPLL (Davis-Putnam-Logemann-Loveland) [NOT06], which solves this problem by selecting a variable, assigning it a value and checking the number of clauses satisfied by this choice. In case it finds a worse solution, the DPLL backtracks and selects another truth value until the whole solution space hsa been explored. In order to work, DPLL needs a logical formula in CNF (conjunctive normal form) format, and it returns the selection of pairs <variable, value> that satisfy the largest number of clauses in the formula (or the weight of the clauses in the case of the weighted MAX-SAT). Among their strengths are:

- Despite being an NP-hard problem, there is a rich literature on highly efficient approximations and variations of DPLL.
- It can work with certain levels of uncertainty, especially incompleteness.

Weaknesses include:

- It can only learn problems with logical information.

AML techniques based on this technique do their work by taking input traces and background knowledge, and creating from them a logical formula in CNF format. This formula is solved by a MAX-SAT solver, and its output is compiled into a valid action model.

- ARMS [YWJ07] presents an iterative process where in each step it creates a logical formula for each action in the domain. These formulas have as clauses the predicates of the intermediate states of the traces used as input. In addition, ARMS infers a number of extra constraints from implicit background information. These constraints represent the ordering relationship between actions, or the semantics of the STRIPS model when representing action models. Once the formulas are constructed, it solves them with a weighted MAX-SAT solver and builds the action models with those clauses that have the most coverage.

- RIM [ZNK13] is an algorithm that essentially works in the same way as ARMS. The main difference between the two approaches is that RIM starts from a partially learned model. From this model, it defines a series of constraints that, together with those it generates in a process similar to ARMS, allow it to be enriched and refined.

- LAMMAS [ZK13] is another algorithm like ARMS, but with the focus on traces obtained in a multi-agent environment. In addition to the limitations generated by ARMS, LAMMAS includes other limitations of its own to represent the coordination of agents to achieve a goal.

### Markov Logic Networks based learning

Markov logic networks are a form of knowledge base representation that combines first-order logic with probabilities. Using this knowledge base in this representation, a given world can be evaluated in such a way that the fewer elements of the knowledge base it contravenes, the more feasible it is. In this representation, each element of the knowledge base is a logical formula with an associated weight. This weight indicates how much a world is penalised if it does not meet the condition represented by the formula. Knowledge bases represented by a Markov logic network can be used in two ways: (i) to test the feasibility of a logical formula under certain circumstances or (ii) to extract a template of the "most feasible world" by taking those conditions with the highest weights. Among their strengths are:

- Ability to faithfully represent uncertainty and stochastic behaviour.

- Efficiency in the representation of very large knowledge bases.

- The internal presentation format of the formulas is very similar to STRIPS/PDDL.

Weaknesses include:

- It is limited to representing only logical information.

Similar to the way MAX-SAT-based techniques work, solutions based on Markov logic networks create a knowledge base from past information as input and extract information from it. The best example of these techniques is LAMP [ZYHL10]. LAMP takes the input plan traces and, as previously discussed, creates a knowledge base from them, encoding a logical formula for each state transition. Once the knowledge base is created, it generates a set of candidate formulas from implicit background knowledge (namely, the predicates and action headers) that are evaluated against the knowledge base. The formula with the highest probability is selected, and the set of action models is created from it.

### Transfer learning

Transfer learning [PY10] is a learning paradigm that studies how to apply learned knowledge from one problem to another similar problem. The philosophy behind this paradigm is to reuse previously acquired knowledge to solve problems where it is costly to obtain sufficient input data. Starting from a solved problem (called source) and a small set of input data from the problem to be solved (called target), transfer learning techniques apply a series of strategies to modify the source to be able to process the input data. This modification can be done in several ways: by assigning weights to the source elements and adjusting them, by selecting which characteristics or parameters of the source best represent the input data and creating a model from them, or by mapping the knowledge of the source with a translator. Among their strengths are:

- Requires a smaller amount of input data.
- Ability to work with uncertainty.

Weaknesses include:

- The overall quality of the learned domains depends on the quality of the source problem.
- It is necessary to have a similar problem previously solved.

The translation of a Transfer Learning problem into an AML problem is trivial: Given a source planning domain and a set of plan traces, a process must be found that allows the source domain to process these traces. Once the process is applied, the source domain can be easily transformed into the target domain. An example of such a technique is TRAMP [ZY14]. TRAMP encodes the input plan traces and the source domain as logical formulas. These formulas are used to create a knowledge base using Markov Logic Networks. Once the knowledge base is built, it generates a set of candidate formulas to link the logic formulas belonging to the plan traces with the formulas obtained from the source domain. After finding a suitable candidate formula, it proceeds to apply the changes encoded in it on the source domain in order to obtain the domain that encodes the plan traces.

## 2.5 The GVG-AI framework

Finally, the contribution presented in Chapter 5 makes use of the development environment created for the General Video Game Artificial Intelligence (GVG-AI) [PLST+15] competition as a source of input data to be used to generate action models. GVG-AI is framed within the research in Artificial General Intelligence [GP07], more specifically in the development of General Video-Game Playing techniques [LBCE+13, GT14], a collection of techniques which belong to the field of Artificial General Intelligence. Artificial General Intelligence studies the development of so-called "general" AI agents, i.e., agents whose intelligence can be applied to

many different types of problems. Extrapolating this to the sub-area of General Game Playing, we can define the area as a set of techniques that are concerned with developing artificial agents capable of playing a large number of different video games without a priori knowledge about them. The event that boosted this area of knowledge was the General Game Playing Competition, held from 2004 to 2012 [GT14]. In this competition, participants were provided with a low-level description of the games in the competition, and they had to design their own representation of the mechanics needed to play them, with the one who achieved the highest score emerging as the winner. Due to the restrictions on the representation used in the competition, the games used in the competition were mainly board games or puzzle games.

In 2013, Bellemere et al. [BNVB13] presented a framework for evaluating general AI agents using Atari 2600 video games. This framework significantly increased the complexity of the represented games and of the environment-agent interactions (mainly due to the fact that these must be in real-time). One year later, the first GVG-AI competition was held, presenting a new framework and a new standard for game representation that allows for the design of any classic video game up to Nintendo Entertainment System games. After this brief historical review, the next pages will present the GVG-AI environment and its particularities.

### 2.5.1 GVG-AI

The GVG-AI [PLST+15, PLST+16] competition was created by a team of researchers from the University of Essex, the New York University and Google DeepMind to evaluate general AI agents. Since its conception, the competition has been run as part of several international conferences and, to enter it, a competitor simply needs to implement a controller in the GVG-AI framework [1] and submit it to the competition web-page. In order to develop and test the controllers, a set of public games are included with the framework; however, when evaluating it, the evaluation committee will use a set of unpublished test games (which is renewed periodically). All games used are inspired by classic arcade games from the 80s/90s and have different victory conditions, scoring mechanisms, game-play and sprites. Currently, the game corpus amounts to more than 160 games and is distributed among several tracks. Each track presents different types of challenges. At the time of writing, the GVG-AI competition has multiple competition tracks available for single-player games [PLST+15], multiplayer games [GPLL16], game level generation [KPLLT16], game learning and game rule generation. PlanMiner-C draws its knowledge from an agent implemented for the first track, the single-player video games track. Over time, GVG-AI has become the benchmark for the development of general AI techniques, allowing a rapid advancement of the research field in recent times [PLLK+19] and even being used for teaching tasks [GCMS+21].

[1] Available at http://www.gvgai.net/

### Game Agents

An agent developed for the GVG-AI competition is a piece of JAVA software designed to return an action that tells the avatar what to do in each step of the game execution. During the competition, agents are allowed a certain amount of time to think about what action to perform, and, if the agent fails to return a valid action within that time, it is automatically disqualified. In the last competition held (in 2018), the time limit was 50 milliseconds, but if the agent did not return an action in less than 40 milliseconds their action was invalid and the avatar remained inactive until the next game step. Before the game starts, the agent is called for the first time (through its java constructor) to initialise its internal components; during this first step, the agent has 1 second to configure what it deems appropriate. The time limitation for determining the avatar's course of action makes an exhaustive search of the problem search space impractical, greatly limiting the techniques that can be implemented in the agent.

Each time an agent is called, it is provided with information about the world (called observation) and a time-stamp to manage its own execution). Since agents do not have access to the game definition, the framework encapsulates all the information they need in the observations of the world. At the time the agent is called, the observation given to it represents a snapshot of the world in a given moment. This observation contains statistics about the agent (its position, its score, its resources, its available actions...), about the game map (size, shape, topology...) and about the rest of the elements that conform the game (obstacles, NPCs, prizes...). In addition, an observation is a simulable model, so given an action, it can be applied to the observation of the world to check how it would change after executing it. An agent can perform this simulation as many times as it wants, as long as it does not exceed the maximum execution time set by the competition. Regardless of the game, an agent can always return the following actions: UP, DOWN, LEFT, RIGHT, USE and NULL. These actions are linked to the directional arrows and the space bar of the keyboard.

### VGDL

All games used in GVG-AI are written in the Video-Game Description Language (VGDL) [Sch13, ELL+13] format. VGDL is a declarative ASCII representation for 2D video games in the style of those programmed for the Atari 2600 and Commodore 64. VGDL has the power to express a wide variety of different games where an avatar (representing a player or agent) interacts with game elements to win. These interactions allow for the definition of action, adventure or puzzle games, both deterministic and stochastic. This separation allows to test artificial agents, but also to develop techniques of procedural generation of levels or game rules.

Similar to a planning problem that requires a domain and a problem separated in different files, a game defined in VGDL consists of a file with its description and one or more levels. On the one hand, the video game description is divided into 4 blocks: the SpriteSet, the InteractionSet, the LevelMapping and the TerminationSet. The SpriteSet describes all the objects in the video game, as well as their

```
wwwww
wgAww
ww..w
w.11w
wwk.w
wwwww
```

(a) ASCII representation.

(b) Graphical representation.

Figure 2.4: VGDL example.

properties and types. The InteractionSet determines the rules that govern the game, indicating how the objects defined in the previous block interact with each other. The LevelMapping indicates how to represent the SpriteSet objects in the level files, creating a map that links each object with an ASCII symbol. Finally, the TerminationSet defines the victory and defeat conditions of the game. On the other hand, the levels are composed of a 2D grid of ASCII symbols, formed according to the information contained in the LevelMapping of the definition file. Figure 2.4 presents an example of a level file together with its graphical representation.

# Chapter 3

# Learning Planning Domains from Plan Traces

## 3.1 Introduction

Despite the advances in Action Model Learning (AML), the techniques presented in the previous chapter are insufficient to learn planning domains with enough expressiveness to cover the capabilities of the more recent versions of PDDL, such as the handling of numerical information or the inclusion of arithmetical or relational expressions. This limits the application of these techniques, as they can not learn domains expressive enough to deal with real-world problems.

In the state-of-the-art there is only a handful of approaches [LJFB07, GL16] capable of working with numerical information, although they present quite important deficiencies that hinder their implementation. On the one hand, Lanchas et al. [LJFB07] is unable to generalise over the input data, creating artificial conditional structures which are difficult for a human being to interpret and which could be simplified by using a single more interpretable arithmetic expression. This issue provokes an increase in complexity and size of the actions learned as the number of distinct values that a given fluent has in the input data grows. On the other hand, Gregory et al. [GL16] can only learn costs associated with actions, and they must be fixed values. This restriction narrows the scope of application of the approach, invalidating the implementation of this technique in any problem with more than one fluent (cost of actions is represented as a single fluent in planning domains), problems whose fluents are not explicitly linked to actions, or in problems with actions that do not modify the fluents by a fixed value after execution.

From the point of view of the implementation of AML techniques in complex AP problems, these limitations are very problematic. The shortcomings in the expressiveness of the models that can be learned make the application of these techniques unfeasible, even for some benchmark domains from the literature. An example of an unlearnable domain would be the Rovers domain presented throughout Chapter 2 of this document. For example, the *move* action is impossible to learn

by the approaches named above. NLOCM would be unable to learn the decrement of the fluent (*battery* ?*r*) as it is not a fixed decrement and depends on other fluents. Lanchas et al. are able to learn the decrement, but replace it with a very large set of conditional effects that hinder the human interpretability of a domain. Finally, neither approach would be able to learn the relational expressions (explained later in this chapter). The system presented in this manuscript was conceived with the idea of overcoming these problems.

The aim of this chapter is to describe our proposal —called PlanMiner—in detail. The PlanMiner methodology is designed as a pipeline of machine learning techniques with the objective of learning STRIPS planning domains enriched with preconditions and effects with arithmetic and relational expressions supported by the PDDL 2.1 version.

The learning process of PlanMiner relies on fitting a series of classification models from which to extract the preconditions and effects of the actions being learned. These classification models are obtained by using a classifier, which is fed the information contained in plan traces used as input to the algorithm. PlanMiner preprocesses these traces with a two-step method: *(i)* formatting the information in the traces into a data structure understandable by the most widespread classification algorithms in the literature (i.e. attribute-value matrices as described in section 2.3.2), and *(ii)* inferring new information with which to enrich these matrices.

Because PlanMiner bases its learning process on the fitting of a series of classification models, it greatly benefits from the advantages of XAI techniques. Given the high interpretability of the models obtained in the intermediate steps of the learning pipeline, PlanMiner can directly represent the relevant information of the preconditions and effects of the action models. This considerably alleviates the work required to learn them. Moreover, in the information obtainment steps, XAI techniques allow for the generation of more interpretable expressions (via symbolic regression) that better represent the behaviours of the action models being learned.

The rest of the chapter is structured as follows: first, we will describe the algorithm presented in this chapter. This description will detail their most important components, depicting how they fit in the learning strategy presented in this document. In the same section (subsection 3.2.1), PlanMiner will be explained in-depth. Second, we will present the experimentation carried out to validate PlanMiner, in comparison with several state-of-the-art AML techniques. This experimentation shows that PlanMiner outperforms such AML techniques in the experiments proposed. Finally, the last section of this chapter contains the final remarks.

## 3.2 PlanMiner

The methodology presented in this chapter consists in a technique for learning planning domains from plan traces by using various machine learning techniques. The main idea behind this approach is to obtain the meta-state of the pre-states and post-states, and, from these meta-states generate the preconditions and effects of each action. A meta-state is a characterisation of the states associated with each action being learned. As previously mentioned, this whole process is designed as a

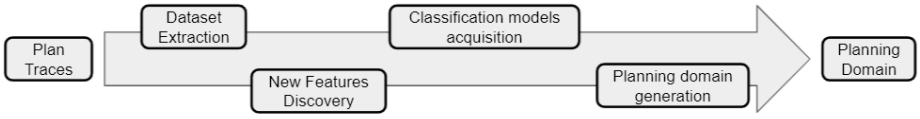



Figure 3.1: The learning pipeline of PlanMiner

pipeline (see Figure 3.1) that links the different techniques needed in the learning process.

In order to make use of machine learning techniques, the first step in the pipeline is to extract the information contained in the plan traces and format it into datasets. This step is essential, as plan traces are not a data structure that is understandable by the vast majority of ML techniques. These datasets are then pre-processed to infer new knowledge from them and enrich them. During this process, our methodology makes use of symbolic regression techniques to learn arithmetic-logical expressions that explain the evolution of the different fluents along with the execution of the input plan traces. These enriched datasets are used as the input to a classification algorithm, from which a series of classification models are extracted. These models contain the information about the meta-states of the pre-state and post-state actions, which are post-processed to translate them into PDDL format and generate the action schemes of the planning domain being learned.

This methodology turns the knowledge acquisition problem into a classification problem. This reduction is due to one of the basic design concepts that were taken into account when designing this solution. As we know, three elements are needed to define an action: a *Header* that indicates the name of the action and its parameters, a *Pre* block that represents those elements that must be true in a state to make an action applicable, and an *Eff* list that describes the changes that are applied. Assuming that we have a priori information about the *Header*, our methodology only needs to learn *Pre* and *Eff*. Learning *Pre* consists in detecting the common elements in all the states on which the action to be learned was applied. In other words, it consists in finding all the features that characterise the pre-state of a given action. Learning *Eff* is an identical process, but with an extra step: once the characterisation of the post-states is obtained, it must be compared with the characterisation of the pre-states. By comparing both meta-states it is possible to determine what steps must be taken to transform the pre-state into post-states, ergo what is the set of steps that must be taken to change the world once an action has been performed.

In chapter 2 of this dissertation report we have presented several solutions that employ a similar approach, but without the use of classification algorithms for the task of meta-state characterisation. Using classification techniques for this task has three major advantages for our methodology.

The first is that they allow us to deal with both categorical and numerical information, which increases the expressiveness of the learned models. The second is the resilience of these techniques to uncertainty, allowing our approach to work even with low-quality data. The third is that the use of these techniques makes it possible to obtain extra information that explicitly shows the reason why an action

model has certain preconditions and effects.

### 3.2.1 PlanMiner Overview

The methodology (see Algorithm 6) presented in this chapter was the first one to implement the concept of learning planning domains by applying classification techniques from plan traces, and the first contribution presented in this manuscript. The algorithm, named [SMPFO21], performs the following tasks in the different steps of its pipeline:

1. **Dataset Extraction.** Takes a set of input plan traces and outputs a collection of datasets. Data structures used as input to typical machine learning algorithms are not in PDDL format, so a translation process must be carried out in order to convert the input plan traces to datasets.

2. **Discovery of new features.** After taking the information contained in the datasets generated in the previous step, PlanMiner applies symbolic regression techniques to generate new features and enrich them. This step is crucial, because new knowledge must be explicitly encoded in the datasets to enable PlanMiner to learn arithmethic and relational expressions.

3. **Classification models acquisition.** With the datasets as input, PlanMiner relies on a classification algorithm to fit a classification model for each one. The hypotheses contained in the classification models define the key features to model a set of intermediate states.

4. **Planning domain generation.** Finally, the classification models are processed to get a set of action models. In order to obtain an action schema, preconditions and effects are extracted from each classification model. The final output of PlanMiner is a PDDL domain obtained by joining the learned action models.

In order to illustrate the whole learning process, throughout this section various examples will be shown based on *Rovers* domain (Table 2.3).

### 3.2.2 Data Set Extraction

In this first stage on the learning process (steps 3 and 4 in algorithm 6), PlanMiner takes the actions and states contained in each plan trace provided as input and adapts them to a typical classification input data structure. To achieve this, our procedure first takes each action $a$ contained in the input plan traces and extracts a *state transition* $(s_1, a, s_2)$. $s_1$ is a snapshot of the world just before executing the action (*pre-state*), while $s_2$ is an observation of the world just after executing the action (*post-state*). A given state can be the pre-state in a state transition and a post-state in a different one. In Listing 3.1 we show an example of a state transitions for the action *(goto rov1 wp1 wp2)* extracted the plan trace depicted in Table 2.3. In this example, we can see how the states 0 and 1 are related to the action 1 of the example plan trace.

**Algorithm 6** PlanMiner Algorithm overview

**Input**: PT: Set of Plan Traces
**Output**: AM: Set of Action Models

```
1: Initialises stDict as an empty dictionary
2: for all Plan trace pt in PTs do
3:    for all Different action act in the pt do
4:       Extract state transitions st of act in pt
5:       stDict[act] ← stDict[act] ∪ st
6:    end for
7: end for
8: for all key act in stDict do
9:    dat ← dataset created using stDict[act]
10:   Detect new features from dat and extend dat with them
11:   Fit a classification model cModel using dat as input
12:   Generate action model am from cModel
13:   AM ← AM ∪ am
14: end for
15: return AM
```

Action: *(goto rov1 wp1 wp2)*

- **pre-state:** *(at rov1 wp1) ∧ (¬ (at rov1 wp2)) ∧ (¬ (at rov1 wp3)) ∧ (¬ (scanned wp3)) ∧ (= (bat_usage rov1) 3) ∧ (= (energy rov1) 450) ∧ (= (dist wp1 wp2) 50) ∧ (= (dist wp2 wp3) 80)*
- **post-state:** *(¬ (at rov1 wp1)) ∧ (at rov1 wp2) ∧ (¬ (at rov1 wp3)) ∧ (¬ (scanned wp3)) ∧ (= (bat_usage rov1) 3) ∧ (= (energy rov1) 300) ∧ (= (dist wp1 wp2) 50) ∧ (= (dist wp2 wp3) 80)*

Listing 3.1: State transition of the *(goto rov1 wp1 wp2)* action.

Once the state transitions for a given action have been extracted, the algorithm calculates the schema form [YWJ07] of every state (Algorithm 7). Schema forms are calculated by selecting a state transition and taking each instance of the parameters in its action and replacing it with a given token every time it appears as an argument in any of the predicates of its associated states. When every parameter has been substituted by a token, *irrelevant predicates* are erased from the states. Irrelevant predicates are predicates that have not undergone at least one substitution during the translation into schema form. An exception to this rule are the predicates with no arguments, which are always considered relevant. As will be explained in section 3.2.4, the classification algorithm will choose from among the relevant predicates in order to keep the ones needed to model the states. The example shown in Listing 3.2 displays the schema form of the states associated with the generic action *(goto ?arg1 ?arg2 ?arg3)* before erasing irrelevant predicates (underlined). As can be seen, the actions have been equated with the independence of their parameters, and each occurrence of `rov1`, `wp1` and `wp2` has been erased. In

**Algorithm 7** Action schematisation process

**Input**: transSet: set of states transitions
**Output**: transSet: set of states transitions

```
1: for all sTrans ∈ transSet do
2:    actionH ← take the action header from sTrans
3:    pre-state ← take the pre-state from sTrans
4:    post-state ← take the post-state from sTrans
5:    for all parameter ∈ actionH do
6:       token ← create a token name for parameter
7:       for all predicate ∈ pre-state do
8:          substitute every occurrence of parameter in predicate with token
9:       end for
10:      for all predicate ∈ post-state do
11:         substitute every occurrence of parameter in predicate with token
12:      end for
13:   end for
14: end for
15: return sTrans
```

order to group all transitions according to the action to which they are linked, the state transitions in schema form are stored in a dictionary *stDict*, whose key is the name of the action appearing in said state transition and its value is the transition itself.

Action: *(goto ?arg1 ?arg2 ?arg3)*

- **pre-state:** *(at ?arg1 ?arg2)* ∧ *(¬ (at ?arg1 ?arg3))* ∧
  (¬ (at ?arg1 wp3)) ∧ (¬ (scanned wp3)) ∧
  *(= (bat_usage ?arg1) 3)* ∧ *(= (energy ?arg1) 450)* ∧
  *(= (dist ?arg2 ?arg3) 50)* ∧ (= (dist ?arg3 wp3) 80)

- **post-state:** *(¬ (at ?arg1 ?arg2))* ∧ *(at ?arg1 ?arg3)* ∧
  (¬ (at ?arg1 wp3)) ∧ (¬ (scanned wp3)) ∧
  *(= (bat_usage ?arg1) 3)* ∧ *(= (energy ?arg1) 300)* ∧
  *(= (dist ?arg2 ?arg3) 50)* ∧ (= (dist ?arg3 wp3) 80)

Listing 3.2: Schema form of a *(goto ?arg1 ?arg2 ?arg3)* action.

A dataset is created for every different action in the plan traces. Two actions are different if their headers (the action's name plus arguments after applying the schematisation process) are different. The datasets contain the information encoded in the states of the state transitions; this information (defined in predicates and fluents) is encoded in the datasets as attributes. Each instance of a dataset is a state, and its values are the values associated with each predicate: a logical value if the predicate is a logic value or a number if the predicate is numerical. The instances of the dataset are categorised by assigning them a class label given by the relation of the state they belong to with the action whose dataset is being modelled (i.e. pre-state or post-state), thus creating a binary classification problem. In order

| (at ?arg1 ?arg2) | (at ?arg1 ?arg3) | (bat_usage ?arg1) | (energy ?arg1) | (dist ?arg2 ?arg3) | (scanned ?arg3) | **Class** |
|---|---|---|---|---|---|---|
| *True* | *False* | 3 | 450 | 50 | *MV* | *pre − state* |
| *False* | *True* | 3 | 300 | 50 | *MV* | *post − state* |
| *True* | *False* | 3 | 300 | 80 | *False* | *pre − state* |
| *False* | *True* | 3 | 60 | 80 | *False* | *post − state* |
| *True* | *False* | 3 | 230 | 75 | *MV* | *pre − state* |
| *False* | *True* | 3 | 5 | 75 | *MV* | *post − state* |
| *True* | *False* | 3 | 400 | 35 | *True* | *pre − state* |
| *False* | *True* | 3 | 295 | 35 | *True* | *post − state* |
| *True* | *False* | 5 | 400 | 75 | *MV* | *pre − state* |
| *False* | *True* | 5 | 25 | 75 | *MV* | *post − state* |
| *True* | *False* | 5 | 500 | 50 | *False* | *pre − state* |
| *False* | *True* | 5 | 250 | 50 | *False* | *post − state* |
| *True* | *False* | 3 | 315 | 105 | *True* | *pre − state* |
| *False* | *True* | 3 | 0 | 105 | *True* | *post − state* |
| *True* | *False* | 5 | 500 | 80 | *MV* | *pre − state* |
| *False* | *True* | 5 | 100 | 80 | *MV* | *post − state* |
| *True* | *False* | 3 | 46 | 15 | *False* | *pre − state* |
| *False* | *True* | 3 | 1 | 15 | *False* | *post − state* |

Table 3.1: Dataset associated with the *(goto ?arg1 ?arg2 ?arg3)* action.

to fill each dataset, the state transitions associated with a given action are taken, and its states are stored as instances of the dataset. These instances are labelled according to their role in a certain state transition, which may lead to the appearance of the same state with slightly different information in several instances of the dataset. In those cases where a predicate that is modelled as an attribute in the dataset does not appear in a given state, the value assigned in its instance is a Missing Value (*MV*) token. For example, the predicate (scanned ?x) is part of the relevant predicates of the action *(goto ?arg1 ?arg2 ?arg3)*, but it is missing in many of the concrete state transitions of such action in the plan traces. Therefore, its absence in such states is represented as an MV. This leads to the adherence to the Open World Assumption for the interpretation of value absences in an instance [RN16]. Table 3.1 shows how the state transitions of the *(goto ?arg1 ?arg2 ?arg3)* actions defined in Table 2.3 are displayed as an attribute-value matrix. As can be seen, each state in the state transitions is included as an instance in the dataset, where its predicates are displayed as attributes; the class label defines its role in the state transition. Note that state 1 (Table 2.3) appears twice in the dataset (instances 2 and 3) with different values and different class label.

### 3.2.3 Discovery of new features

At the previous stage of the methodology, PlanMiner created a set of datasets. As the only information accessible to PlanMiner about the ontology of the problem is the actions' headers contained in the plan traces, the options of the methodology are limited. In order to overcome this handicap, PlanMiner tries to acquire new knowledge from scratch. This stage (step 5 in the algorithm) is the most crucial in the whole learning process as, without this new knowledge, the process of learning expressions (whether arithmetical or relational) among the elements of the learning problem would be impossible. Creating new features using brute force is not a viable option because there is a risk of uncontrolled increase in the size of the

learning problem. Also, when trying to learn relations among the elements of the dataset (and hence the predicates of the planning problem) an excessive creation of knowledge will lead to the emergence of spurious relations. Spurious relations contain useless information that makes the resolution of the learning problem very difficult; for example, a spurious relation drawn from example Table 3.2 would be one that indicates that (*bat_usage* ?*arg1*) must be smaller than (*dist* ?*arg2* ?*arg3*) in the meta-states of the pre-states. To overcome this, we divide the process of discovering new features into 3 steps:

1. Calculation of the difference between the numerical attributes of the dataset before and after executing an action. This step will produce new synthetic attributes (containing information about how a variable changes throughout the execution of a plan) that will be added to the dataset.

2. Fitting of arithmetic expressions that model the differences of the numerical attributes.

3. Discovery of the relational expressions that link the different elements of the problem.

Before advancing to the next step, the new information is filtered to detect useless or redundant information. This helps to keep the over-information produced in every step under control.

### Changes in numerical attributes

As a beginning step, prior to being able to learn complex information, we calculate the set of $\Delta$ values associated with each numerical attribute of the dataset. $\Delta(attribute)$ is defined as

$$\delta_i \in \Delta : \delta_i = x_{pre,attribute} - x_{post,attribute}$$

where *pre* and *post* are the instances of the states associated with the *ith* state transition. If $x_{pre,attribute}$ or $x_{post,attribute}$ are missing, $_i$ cannot be calculated and is substituted by a Missing Value token.

Once the $\Delta$ values have been obtained, we can discriminate those attributes that contain helpful information and are worthy of further exploration during the information discovery process. A $\Delta(attribute)$ is irrelevant to the learning process if every $\delta_i$ is equal to 0. An irrelevant $\Delta(attribute)$ means that its associated attribute is not affected by a given action and is discarded before the next step begins. Relevant $\Delta$ values are included in the dataset as a new attribute. Table 3.2 shows an example of the calculation of $\Delta$ values. This table contains a description of how the fluents evolve after executing a *goto* action; note that, out of the three fluents considered, only $\Delta(energy?arg1)$ is relevant.

| $\Delta(bat\_usage?arg1)$ | $\Delta(energy?arg1)$ | $\Delta(dist?arg2\ ?arg3)$ |
|---|---|---|
| $3-3=0$ | $300-450=-150$ | $50-50=0$ |
| $3-3=0$ | $60-300=-240$ | $80-80=0$ |
| $3-3=0$ | $5-230=-225$ | $75-75=0$ |
| $3-3=0$ | $295-400=-105$ | $35-35=0$ |
| $5-5=0$ | $25-400=-375$ | $75-75=0$ |
| $5-5=0$ | $250-500=-250$ | $50-50=0$ |
| $3-3=0$ | $0-315=-315$ | $105-105=0$ |
| $5-5=0$ | $100-500=-400$ | $80-80=0$ |
| $3-3=0$ | $1-46=-45$ | $15-15=0$ |

Table 3.2: Δ values extracted from the dataset contained in Table 3.1

### Fitting of arithmetic expressions

At this point in the process of knowledge acquisition, by using the calculated Δ values, PlanMiner would be able to infer how a fluent varies after executing a given action, but only if it changes by a fixed value (linear functions). More complex changes in the attributes (those that, for example, are explained by algebraic functions) require new and more complex solutions to be identified. As explained earlier (Section 2.3.1), regression analysis is the discipline of machine learning devoted to predicting the correlation of a numerical value with the other variables of a given problem. In this context, it can be used by PlanMiner to set a Δ value as the target variable and, using regression techniques, create an expression that fits it from the rest of predictive variables in the dataset.

PlanMiner implements the symbolic regression algorithm through an informed graph search algorithm [HNR68] with the objective of incrementally building a valid expression that fits the problem's goal. An overview of the symbolic regression algorithm implemented can be seen in Algorithm 8. In short, the algorithm creates the root node from an empty expression (Ø) and a set of numerical attributes passed as input (dataset *dat*) and, by adding new operators and operands to existing nodes, creates new states that represent new formulas. New formulas are added to a pool of created formulas. The algorithm selects a new formula from the pool and repeats the process until a formula that is suitable for the set of target values is found.

In order to guide the search, for a state that represents the arithmetic expression $x$, the algorithm uses a heuristic function

$$h(x, Goal) = 100\% * MAPE(pred(x), Goal) * |x|$$

, where MAPE (Mean Absolute Percentage Error) is a measure of difference between two continuous variables (pred(x) and Goal) and is calculated as

$$MAPE(pred(x), Goal) = \frac{\sum_{i=1}^{|Goal|} |pred(x)_i - goal_i|}{Goal_i}$$

, where $pred(x)$ are the forecast values obtained with the arithmetic expression $x$ paired with the *Goal* set of values. Finally, $|x|$ is the size of the arithmetic expression (the number of operands and operators in the expression).

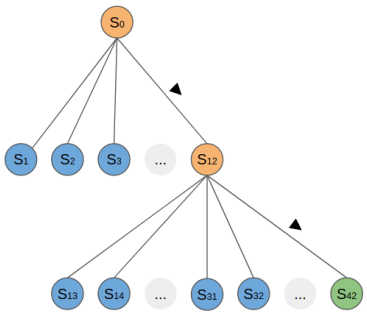
**Algorithm 8** Symbolic regression algorithm

**Input**: dat: Dataset
**Output**: f: arithmetic expression

```
1:  Set f as Ø arithmetic expression
2:  open ← generate successor nodes from f
3:  end ← False
4:  while end = False do
5:      successor ← node in open with the lowest h function value
6:      open ← open - successor
7:      if h(successor) < threshold ∨ timeout then
8:          end ← True
9:      else
10:         if h(successor) < h(f) then
11:             f ← successor
12:         end if
13:         open ← open ∪ generate successor nodes from successor
14:     end if
15: end while
16: return f
```

New states are created from a parent state by adding an operand paired with an operator to the arithmetic expression defined in it. Arithmetic operators used by the regression algorithm are $\{+, -, *, /\}$, while operands may be a constant integer or an attribute from the dataset. For each iteration of the algorithm, every possible combination of operators and operands is used to create new successor states. The search ends when a state with a heuristic value close zero is found or a certain amount of time has passed. The stop value for the heuristic function and the timeout settings can be found in the experimental setup section (i.e. Section 3.3).

If the search algorithm is not able to find a suitable expression within the allotted time, the $\Delta$ value associated is erased from the dataset. This is due to the supposition that, if the search algorithm is incapable of finding a viable pattern in the elements of the $\Delta$ value, then it is full of arbitrary values and no useful information can be extracted from it. If a suitable formula is found, the $\Delta$ value used as the goal is updated with it in the dataset. Figure 3.2 presents a brief example of a search process for $\Delta(energy\ ?arg1)$, where visited nodes (orange), expanded nodes (blue) and goal nodes (red) are shown in the figure. The formulas created in each node are displayed in the table on the right, as well as their heuristic values. Those heuristic values are calculated by using the information contained in Table 3.1. As can be seen, the heuristic values decrease as the candidate formulas are increasingly similar to the target formula.




| *State* | *Expression* | *h(node)* |
|---|---|---|
| $S_0$ | Ø | ∞ |
| $S_1$ | 1 | 298.375 |
| $S_2$ | 2 | 296.75 |
| $S_3$ | 3 | 295.125 |
| ... | ... | ... |
| $S_{12}$ | (*distance?arg1?arg2*) | 200 |
| $S_{13}$ | (*distance?arg1?arg2*) + 1 | 198.375 |
| $S_{14}$ | (*distance?arg1?arg2*) + 2 | 196.75 |
| ... | ... | ... |
| $S_{31}$ | (*distance?arg1?arg2*) − 9 | 214.625 |
| $S_{32}$ | (*distance?arg1?arg2*) ∗ 2 | 100 |
| ... | ... | ... |
| $S_{42}$ | (*distance?arg1?arg2*) ∗ (*bat_usage?arg1*) | 0 |

Figure 3.2: Search graph of the Δ(*energy*?*arg*1) expression.

#### Creation of relational expressions

As the final step of the new features discovery process, the relational expressions are created through a straightforward procedure. This procedure takes two numerical attributes of the dataset and creates a new one by pairing them with a relational operator. The relational operators used by our algorithm are {=, <, >, }.

Finally, the relevance of the new logical attributes must be checked. The relevance of a logical attribute is calculated by testing if every value in the new attribute is different. If the truth value of the relational expression remains constant, i.e., all rows in the attribute have the same value (either true or false), the attribute can be discarded, as it contains no useful information. Relevant attributes are included in the dataset. In Table 3.3, we present a new attribute created from the (*energy*?*arg1*) and Δ(*energy*?*arg*1) fluents contained in the dataset of Table 3.1. This Table shows how this new feature evolves over successive executions of the *goto* action.

### 3.2.4 Classification models acquisition

In order to characterise an action, PlanMiner needs to know *(i)* which elements hold in the state to be able to apply the action (preconditions) and *(ii)* which preexisting state elements change as a result of applying the action (effects).

(i) Obtaining the preconditions can be approached as a classification problem, since what PlanMiner looks for is which elements are common to all the states where that action was applied, i.e., to all the pre-states. Therefore, it consists in finding all features that characterise the pre-states of a given action.

(ii) Obtaining the effects also poses a problem of classification since we want to find which features belong to the post-states. This reduces the problem to fitting a model with every shared feature of the instances labelled as pre-states in the dataset, and consequently, fitting a model with every shared feature of the instances labelled as post-states in the dataset.

| *(energy ?arg1)* | Δ(*energy*?*arg*1) | (>*(energy ?arg1)* Δ(*energy*?*arg*1)) |
|---|---|---|
| 450 | 150 | *True* |
| 300 | 150 | *True* |
| 300 | 240 | *True* |
| 60 | 240 | *False* |
| 230 | 225 | *True* |
| 5 | 225 | *False* |
| 400 | 105 | *True* |
| 295 | 105 | *True* |
| 400 | 375 | *True* |
| 25 | 375 | *False* |
| 500 | 250 | *True* |
| 250 | 250 | *False* |
| 315 | 315 | *True* |
| 0 | 315 | *False* |
| 500 | 400 | *True* |
| 100 | 400 | *False* |
| 46 | 45 | *True* |
| 1 | 45 | *False* |

Table 3.3: Example attribute (>*(energy ?arg1)* Δ(*energy*?*arg*1))

PlanMiner boasts complete independence of the algorithms used in model learning. For the sake of the usability of the methodology, PlanMiner imposes a restriction on the format of the models learned. This restriction is to use rules in the format presented in section 2.3.3, and is intended to enable the definition of a proper process to translate a classification model into an action model. Throughout the rest of the document, whenever a reference is made to "classification models", we will refer to a set of rules and vice versa.

**IF**
(at ?arg1 ?arg2) = *True* ∧ (at ?arg1 ?arg3) = *False* ∧
(bat_usage ?arg1) = 3 ∧
(> (energy ?arg1) Δ(energy ?arg1)) = *True*
**THEN** *pre-state*

**IF**
(at ?arg1 ?arg2) = *False* ∧ (at ?arg1 ?arg3) = *True* ∧
(bat_usage ?arg1) = 3 ∧
Δ(energy ?arg1) =
(dist ?arg2 ?arg3) ∗ (bat_usage ?arg1)
**THEN** *post-state*

Listing 3.3: Classification models of the *(goto ?arg1 ?arg2 ?arg3)* action.

PlanMiner requires that there be a single rule for modelling pre-states and a single rule for post-states. In case any classification algorithm generates more than one rule for a label, these rules must be combined into a single rule. This is done by applying a proprietary statistical process, which is defined in section 4.2.3, as it is a fundamental procedure of the contribution proposed in Chapter 4. In addition, every classification algorithm uses its own internal models and, in order to continue the execution of PlanMiner, they must be transformed into rules. The process of

converting the different classification models is an *ad-hoc* procedure that depends on the classification algorithm used. These processes have already been described in the relevant subsections of Section 2.3.

Listing 3.3 presents the classification model of the *(goto ?arg1 ?arg2 ?arg3)* action learned using the NSLV classification algorithm. These models are fitted to the information shown in Table 3.1 plus the information added in the different processes of the previous subsection. In this example, the "*Attr* is *val*" pairs that define the antecedents of the example rules can be seen, as well as the class labels that represent the meta-states of the pre-states and post-states.

### 3.2.5 Planning domain generation

The datasets extracted from the plan traces, enriched with the newly discovered knowledge and added to them, contain information about the examples that represent states of the world. These examples have been generalised into a collection of features (defined as a rule) with the information needed to represent all of them. Classification rules define a meta-state with every shared feature from each state. Nonetheless, to be able to use these meta-states to learn the preconditions and effects of a given action, PlanMiner must change their format into PDDL.

In accordance with the way PlanMiner is designed, the methodology must translate a classification rule into a description of the world written in PDDL. Given the similarities of both models (recall Section 3.1), this process is trivial. On one hand, a rule's antecedent contains a set of tuples <*X*, *A*> linked by a conjunction operator. Each tuple represents the value A of the problem's attribute X. On the other hand, PDDL displays world states as a set of predicates linked by a conjunction operator.

**IF**
(at ?arg1 ?arg2) = *True* ∧
(at ?arg1 ?arg3) = *False* ∧
(bat_usage ?arg1) = 3 ∧
(>(energy ?arg1) Δ(energy ?arg1)) = *True*
**THEN** *pre-state*

→

(at ?arg1 ?arg2) ∧
(¬ (at ?arg1 ?arg3)) ∧
(= (bat_usage ?arg1) 3) ∧
(>(energy ?arg1)
(*
(distance ?arg1 ?arg2)
(bat_usage ?arg1)
)
)

**IF**
(at ?arg1 ?arg2) = *False* ∧
(at ?arg1 ?arg3) = *True* ∧
(bat_usage ?arg1) = 3 ∧
Δ (energy ?arg1) =
(dist ?arg2 ?arg3) * (bat_usage ?arg1)
**THEN** *post-state*

→

(¬ (at ?arg1 ?arg2)) ∧
(at ?arg1 ?arg3) ∧
(= (bat_usage ?arg1) 3) ∧
(decrease (energy ?arg1)
(*
(distance ?arg1 ?arg2)
(bat_usage ?arg1)
)
)

Listing 3.4: Pre-state and Post-state meta-state from the rules of Listing 3.3.

An example of the translation process can be seen in Listing 3.4. In this Listing we can see how the different elements of the rules are represented in the form of a predicate, linked by the conjunction operators. In addition, when creating the meta-states, the $\Delta(attribute)$ is also translated by the expression calculated in the previous features discovery step.

As said earlier, tuples <*X, A*> represent the essential predicates that define a state of the world. Those states contain the minimum essential information to model all the pre- or post-states for every appearance of the action of the plan traces. From these states, the preconditions and effects of the action are extracted.

Action preconditions are the set of features of the world that must hold in order to apply the action, and so this set can be obtained directly by displaying the information of the pre-state model as a conjunction of predicates. Action effects represent how the action changes the world, and thus they must be obtained by computing the steps necessary to transform the pre-state into the post-state. These steps are the addition and deletion list of logical predicates and the assignment/increment/decrement of continuous values of numerical predicates. By comparing the pre-state and post-state we can check which logical predicates must be added (false in the pre-state but true in the post-state) and deleted (true in the pre-state but false in the post-state). Including increments and decrements of numerical fluents is a straightforward process as $\Delta$ values explicitly contain this information, and so they only need to be translated into the PDDL format. In Listing 3.5 we illustrate this whole process: for example, you can see how (*bat_usage?arg1*) does not appear in the effects of the action, as it is the same in both the meta-state of the pre-states and the post-states.

**Preconditions:**
(at ?arg1 ?arg2)) ∧
(¬ (at ?arg1 ?arg3)) ∧
(=(bat_usage ?arg1) 3) ∧
(>(energy ?arg1)
(* (distance ?arg1 ?arg2) (bat_usage ?arg1)))

**Effects:**
(¬ (at ?arg1 ?arg2)) ∧
(at ?arg1 ?arg3) ∧
(decrease (energy ?arg1)
(* (distance ?arg1 ?arg2) (bat_usage ?arg1)))

Listing 3.5: Preconditions and Effects learned from the models of Figure 3.4.

## 3.3 Experiments and Results

This section is devoted to the experimental process through which PlanMiner has been validated. The aim of this experimentation is to demonstrate the usefulness of PlanMiner in learning planning domains, as well as to test its robustness to incompleteness. A full description of the results, along with the measurements and metrics outlined below, can be found in the Appendix B. From here on, *PlanMiner(X)* will denote a version of PlanMiner that uses a classification algorithm *X*.

This section is divided into 4 subsections. In the first one, the experimental setup is defined: it details the structure of the experimentation, the data used and the evaluation metrics. The second subsection defines the algorithms employed during the experimentation, either as state-of-the-art benchmark algorithms or classification algorithms to test PlanMiner. Finally, the last two subsections contain the results of the experimentation and the discussion of those results using STRIPS and numerical domains, respectively. The section that contains the results of the STRIPS domains is split into two parts: the first part draws a comparison among the different classification algorithms of PlanMiner, while the second part compares the best version of PlanMiner with the state-of-the-art algorithms.

### 3.3.1 Experimental Setup

To test the usability of PlanMiner, two batteries of experiments are defined. The first one is designed to test PlanMiner's capabilities by trying to learn STRIPS planning domains, while the second one is designed to test PlanMiner's ability to learn planning domains with numerical information domains. Each experiment uses its own input data. The domains used and their characteristics can be seen in Table 3.4. This Table shows the domains used in the experimental process. These domains have been selected from the third International Planning website Competition [1]. On this site, additional information about the different versions of the problems (included STRIPS and Numerical) can be found. The original description of the domains (in STRIPS and numerical versions) used in this experimentation has been transcribed in Appendix A since they are used to establish the metric that indicates the degree to which the extracted domain matches the original. The domains' characteristics (from left to right) are the number of actions of the domains, the number of parameters of the actions, the number of logic predicates, the number of numerical predicates (if any), and the maximum number of parameters of the predicates.

The data used as input in the experimentation has been generated from the implementation of the domains of Table 3.4 of the International Planning Competition [McD00]. For each domain, a set of 100 problems is generated, which are solved using a state-of-the-art planner —specifically, the FF planner [Hof03] —. Plan traces are generated from the plans that solve these problems, and are then used as input to the algorithms tested in the experimentation. The plan traces are modified by removing some of their elements in order to perform the experimentation with incomplete input data. These elements are predicates and fluents of the randomly selected trace states.

For each battery of experiments, the domains learned with the different algorithms are evaluated both syntactically and semantically. On the one hand, the syntax of the learned domains is evaluated by comparing them with the original domains and measuring their differences. On the other hand, the semantics of the learned domains are measured by checking whether they can correctly perform the same job as the original domains.

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

| Domain | \|Actions\| | max action arity | \|logical\| \|predicates\| | max predicate arity |
|---|---|---|---|---|
| BlocksWorld | 4 | 2 | 5 | 2 |
| Depots | 5 | 4 | 6 | 2 |
| DriverLog | 6 | 4 | 6 | 2 |
| ZenoTravel | 5 | 3 | 4 | 2 |

(a) STRIPS domains.

| Domain | \|Actions\| | max action arity | \|logical \| \|predicates\| | \|numerical\| \| predicates\| | max predicate arity |
|---|---|---|---|---|---|
| Depots | 5 | 4 | 6 | 4 | 2 |
| DriverLog | 6 | 4 | 6 | 4 | 2 |
| Rovers | 10 | 4 | 26 | 2 | 3 |
| Satellite | 5 | 4 | 8 | 6 | 2 |
| ZenoTravel | 5 | 3 | 2 | 8 | 2 |

(b) Numerical domains.

Table 3.4: Input domains characteristics

The syntactic evaluation of the domains is carried out by calculating the F-Score, precision and recall of the domains. The F-Score of a domain is a measure of the average accuracy of the domain, is calculated from the precision and recall of the domain, and indicates how similar the learned domain is to the reference domain. The precision of a domain indicates how many elements are left over compared to the original planning domain, while the recall measures how many elements are left over. The aggregate value of these metrics for a particular domain is measured by calculating their average value for each action in that domain. Formally, these metrics are defined as follows:

$$F\text{-}Score(a_i) = 2 * \frac{Precision(a_i) * Recall(a_i)}{Precision(a_i) + Recall(a_i)}$$

$$Precision(a_i) = \frac{tp(a_i)}{tp(a_i) + fp(a_i)}$$

$$Recall(a_i) = \frac{tp(a_i)}{tp(a_i) + fn(a_i)}$$

where *F-Score* is the harmonic mean between precision and recall for an action $a_i$, *tp* is the number of elements appearing correctly in $a_i$, *fp* is the number of elements left over in $a_i$, and *fn* is the number of elements missing in $a_i$. These elements can be either preconditions or effects.

The semantic evaluation of domains is tested by assessing whether they are valid. A domain is valid if, after reproducing a plan over an initial state, it obtains the target state of the problem solved by that plan. In order to make the assessment of domain validity as unbiased as possible, the problems used during the validation process —called test problems —are distinct from the problems defined to generate

the input plan traces. To determine if a domain is valid, it must be validated with all test problems. During this experimentation process, validation is performed with the VAL [HL03] tool of the International Planning Competition. Finally, it is worth noting that, for every experiment, a comparative graph of the results of the algorithms used in it will be included, as well as a table detailing the validation results of the algorithms used in the experimental process.

The randomised inclusion of incompleteness can artificially affect the results. To prevent this from altering the results excessively during experimentation, 10-Fold Cross-Validation [Sto74] (10F-CV) has been used. 10F-CV is a technique that consists in segmenting a dataset into several partitions (named folds), creating 10 subsets of training data from them and fitting a model to each of them, so that the overall model performance fitted to the overall dataset is computed as the average performance of each of the models fitted to each training subset. In addition to the training subsets, 10 validation subsets are generated to compute the performance of the models independently. The generation of the aforementioned subsets is done by splitting into 10 chunks, taking 1 of them as a validation subset and combining the rest to form a training subset. This process is repeated 9 more times, each time selecting a different partition as a validation subset until the desired ten pairs (training subset, validation subset) are obtained. In this experimentation, the training subsets are used to obtain planning domains (which are evaluated syntactically), while the validation subsets are used during the semantics evaluation process.

### 3.3.2 Algorithms used in the experiments

During the experimental process, different versions of PlanMiner have been evaluated, where each one uses a different algorithm to generate the classification models. The aim of this is to test the feasibility and robustness of the learning pipeline regardless of the learning engine used in it. The classification algorithms used in this experimentation are ID3 [Qui86], C4.5 [Qui14], RIPPER [Coh95] and NSLV [GGGP15]. A brief description of these algorithms can be found in Chapter 2.

In addition to classification algorithms, a set of state-of-the-art AML algorithms was selected as reference algorithms for the experimental process. These algorithms are ARMS [YWJ07], AMAN [ZNK13], OpMaker2 [MCRW09] and FAMA [ACO19]. Due to the limitations of these algorithms for learning planning domains with numerical information, they are only used during experimentation with STRIPS domains.

In these experiments, the parameters of PlanMiner, FF-Metric, VAL and the reference algorithms are set as default as noted by its authors in their reference works. Table 3.5 displays these parameters and their impact on a given algorithm's performance. If an algorithm does not appear in the mentioned table it is because it requires no parameter setup at all before execution. With regard to the different parameter settings, we include a brief description of them and their impact on the performance of the given algorithms (except for PlanMiner, as this information has already been given in the relevant sections above). That said, for each algorithm the parameter settings are:

| **PlanMiner parameters** | Value |
|---|---|
| Symbolic Regression acceptance threshold | 0.02 |
| Symbolic Regression timeout | 300 |
| **MetricFF parameters** | |
| H weight | 1 |
| G weight | 1 |
| **ARMS parameters** | |
| Probability threshold | 0.7 |
| **AMAN parameters** | |
| Number of iterations | 1500 |

Table 3.5: Settings of the different algorithms during the experimentation process.

- **Metric-FF.** MetricFF's parameters weigh the components of the heuristic that governs the internal search process of the planner. These weights change the importance that MetricFF gives to the estimated cost to the goal ($h$) and the cost of the current explored path ($g$) when guiding the search process. These parameters have been set to 1 to avoid interference with the search process, giving the same weight to both elements of the heuristic.

- **ARMS.** The *probability threshold* of ARMS is used to filter which information contained in the input plan traces is considered as a constraint when building the logic formulas used to learn action models. The lower the threshold, the more information is considered. This increases the computation time of ARMS, but it may consider in the learning process information with a low appearance rate in the input data, which can sometimes be useful. The probability threshold value (0.7) is recommended by the authors of ARMS as the best value that balances the amount of information processed and the results of the algorithm.

- **AMAN.** AMAN iteratively builds a set of action models from a partial set data extracted from the input plan traces. Increasing the number of *iterations* leads to an increase in the amount of sets of action models generated. This naturally increases the probability of finding the best set of action models possible, but at the cost of a longer computation time. The authors recommend 1500 iterations as the most efficient way to obtain good results.

### 3.3.3 STRIPS domains results

#### Comparison of classification algorithms

The experimental process begins by studying how the selected classification algorithm affects the performance of PlanMiner. Figure 3.3 shows these performances in terms of F-Score, while Table 3.6 shows the validity results of the battery of experiments. Figure 3.3 represents the degree of incompleteness of the plan traces in

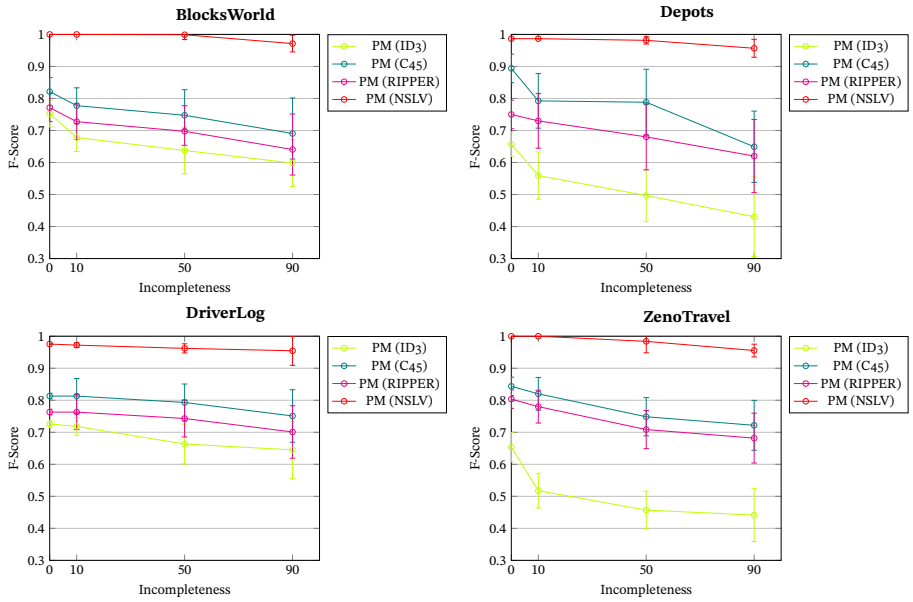



Figure 3.3: Performance comparison of PlanMiner using different classification algorithms on STRIPS domains.

the X axis, and the Tables are sorted by domain in order to improve their readability. This section presents a summarised version of the results. An extended version of this can be found in Appendix B (specifically in Tables B.5, B.6, B.7 and B.8).

If we look closely at the results the figure 3.3 we can see the following:

- **BlocksWorld.** PlanMiner (NSLV) consistently produces the best results throughout the experimentation, with perfect results until those more complex experiments where it drops slightly, but still remains above 98% F-Score. PlanMiner (ID3), (C45) and (RIPPER) show similar behaviour among themselves, although inferior to NSLV. Their performance gradually but steadily falls as the complexity of the experiments increases. Initially, the F-Score of these algorithms is between 80-75%, dropping to 70-60 points in the more complex experiments.

- **Depots.** As with BlocksWorld, PlanMiner (NSLV) leads the performance ranking with results above 95% F-Score with the lowest possible amount of data. PlanMiner (ID3) and (RIPPER) suffer a sharp initial F-Score drop of around a 10% of performance when incompleteness is found in the plan traces; the behaviour of the algorithms leads to final results of 65% F-Score for the experiments with 90% of predicates missing. Finally, PlanMiner (C45) has an initial F-Score of 74% which drops below 61% in the last experiments, with a constant performance drop throughout the whole experimentation.

- **DriverLog.** PlanMiner (C45), (RIPPER) and (NSLV) show similar behaviour,

dropping slightly throughout the experimentation as input data is removed. PlanMiner (C45) and (RIPPER) show a slightly steeper drop in the final experiments, showing a 5% F-Score difference in both classifiers, in contrast to the 2% lost by NSLV. Finally, PlanMiner (ID3) shows a stronger drop initially, which slows down slightly with the more complex experiments.

- **ZenoTravel.** PlanMiner (NSLV)'s results are perfect at the beginning of the experimentation, remain unchanged with certain levels of incompleteness, and drop slightly in the experiments with the minimum possible information (around 2%). The rest of the algorithms perform similarly, with large drops in F-Score throughout the experimentation (especially when some incompleteness is found in the plan traces) that cause them to lose around 25% F-Score between the results with complete data and the results with more incomplete data. PlanMiner (ID3) starts the experimentation with a 65% F-Score, dropping to 45% in the more complex experimentation.

PlanMiner is strongly influenced by the classification algorithm used in its core, with large differences in performance depending on which algorithm is used. These differences between the best and worst results can be as large as 60 percentage points in some experiments. On the one hand, we find that PlanMiner struggles to maintain an F-Score above 80% when incompleteness is found in the input data. Moreover, ID3, C45 and RIPPER present performance drops when a percentage of predicates are erased. This seriously hinders the performance in the rest of the experiments, leading to results below 50% in some experiments. On the other hand, the best classification algorithm —i.e. NSLV— clearly outperforms its rivals. The results show that, when learning STRIPS-like domains, PlanMiner (NSLV) exhibits high resilience to incompleteness, maintaining stable F-Score values, as opposed to the performance of the other algorithms. This gap in performance between NSLV and the other classification algorithms can be attributed to one particularity of NSLV: its ability to generate descriptive rules. As previously mentioned, descriptive rules contain all the information necessary to model the examples of a given class. The ability of NSLV to obtain complete representations of the meta-states greatly improves the performance of PlanMiner, as we have seen in this experimentation. On the other hand, this quality comes with a drawback: in case a state element always appears, it will be considered as necessary for the modelling of the state. These errors are called over-information errors and can be seen in Depots or DriverLog, where, despite complete data availability, the F-Score results are not 100%. Over-information errors occur when there are predicates that can be inferred from other elements. For example, in the aforementioned domains, we find that the locations that make up the world are bidirectionally connected; therefore, for a given pair of elements A and B, there exist two equivalent predicates to indicate that A is connected to B. NSLV's advantage of obtaining all the elements that define the meta-states causes it to learn both predicates —as one of them is redundant in most cases, this lowers the accuracy of the learned actions.

The validity results can be found in Table 3.6. Recapping, Validity is the metric (calculated using the tool from the International Planning Competition named

| Domain | Incompleteness | Algorithm | | | |
|---|---|---|---|---|---|
| | | PM ($ID_3$) | PM ($C_{45}$) | PM (RIPPER) | PM (NSLV) |
| Blocksworld | 0% | ✓ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✓ |
| Depots | 0% | ✗ | ✓ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✓ |
| DriverLog | 0% | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✓ |
| ZenoTravel | 0% | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✓ |

Table 3.6: Validity Results

VAL) that measures whether domains are semantically correct, i.e., whether they can resolve the plans from which they emerged.

This metric presents results in agreement with those seen for the F-Score. Validity is a very demanding metric, as a single wrong precondition or effect in an action can cause the whole domain to be invalid. Moreover, the fault tolerance of the validity metric is asymmetric, punishing errors in effects much more than in preconditions, and recall problems much more than precision problems. That is, a surplus precondition is much less damaging to validity than a missing effect. PlanMiner suffers from the problem described above and, depending on the classification algorithm used, obtains very different outcomes. While the classification algorithm RIPPER is unable to obtain a single valid domain, PlanMiner ($ID_3$) and ($C_{45}$) learn valid domains when no incompleteness is included in the plan traces; regarding PlanMiner (NSLV), it outperforms all the classification algorithms as it obtains valid domains in every single situation. The clear superiority of NSLV will lead to it being chosen as the classification algorithm in the following experiments.

### State-of-the-art algorithms comparison

The next aspect to be studied in the experimentation is how PlanMiner (NSLV) —the highest-performing version of PlanMiner— compares to the reference algorithms. Figure 4.6 presents a comparative graph that displays the F-Score of these algorithms. Additionally, in Table 4.4 the validity results of the battery of experiments are shown. For the sake of readability, it is worth saying that the X-axis of the Figure 4.6 represents the degree of incompleteness of the input plan traces and that the Tables group the data displayed by the planning domain being learned. The next lines are a summarised version of the experimental results; the full version can be found in Appendix B (specifically in Tables B.1, B.2, B.3 and B.4).

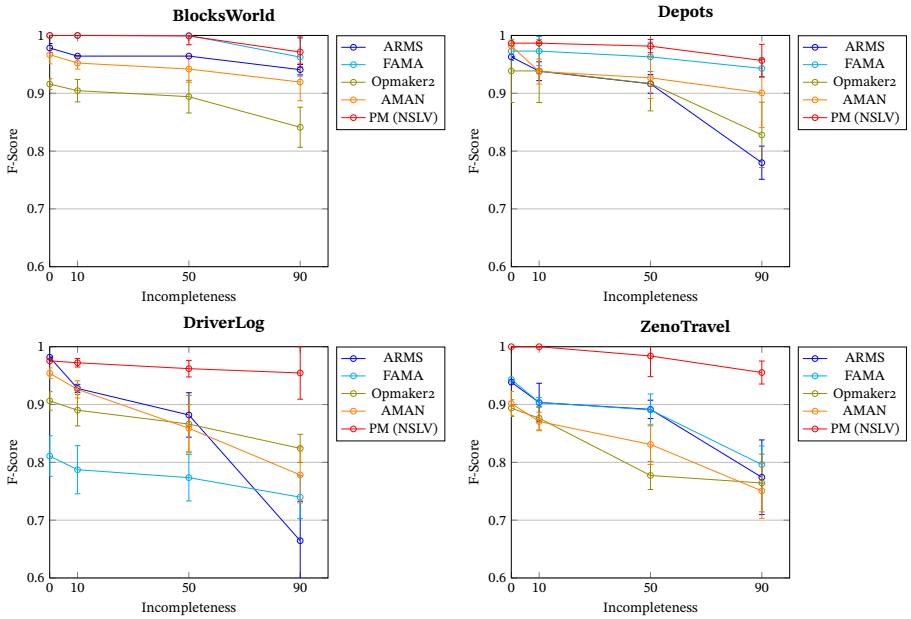



Figure 3.4: Performance comparison of between PlanMiner and state-of-the-art algorithms on STRIPS domains.

If we look closely at the results the figure 3.4 we can see the following:

- **BlocksWorld.** PlanMiner (NSLV) and FAMA achieve the best results throughout the experimentation, presenting only a small drop in the most complex experiments where the results presented are above 98% F-Score. The AMAN and ARMS benchmark algorithms remain above the 90% F-Score threshold even in the most complex experimentation with similar behaviour, while OPMaker-2 performance drops below the 85% mark.

- **Depots.** PlanMiner (NSLV) and FAMA lead the performance ranking with results above 95 percent F-Score on the lowest possible amount of data. AMAN does not drop below the 90 percent threshold in the most complex cases, but Opmaker2 and ARMS (especially ARMS) suffer a sharp drop in quality with 50 percent of predicates missing, bringing their F-Score to around 80%.

- **DriverLog.** The quality of the results for PlanMiner (NSLV) remains almost unchanged throughout the experimentation (losing only 2 points of F-Score at 90% of incompleteness), with results of 98%. The rest of the algorithms (except ARMS) show similar behaviour, with a stable drop in performance of 10% between the initial result with complete data and the more complex experiments. ARMS, on the other hand, shows an F-Score loss of 20 percentage points at 90% incompleteness compared to its value with half of the missing data.

| Domain | Incompleteness | Algorithm | | | | |
|---|---|---|---|---|---|---|
| | | ARMS | FAMA | OpMaker2 | AMAN | PM (NSLV) |
| Blocksworld | 0% | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 10% | ✓ | ✓ | ✗ | ✓ | ✓ |
| | 50% | ✓ | ✓ | ✗ | ✓ | ✓ |
| | 90% | ✓ | ✓ | ✗ | ✓ | ✓ |
| Depots | 0% | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 10% | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 50% | ✓ | ✓ | ✗ | ✓ | ✓ |
| | 90% | ✗ | ✓ | ✗ | ✓ | ✓ |
| DriverLog | 0% | ✓ | ✗ | ✓ | ✓ | ✓ |
| | 10% | ✓ | ✗ | ✗ | ✓ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✗ | ✓ |
| ZenoTravel | 0% | ✓ | ✓ | ✗ | ✓ | ✓ |
| | 10% | ✗ | ✓ | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ | ✗ | ✓ |

Table 3.7: Validity Results

- **ZenoTravel.** PlanMiner (NSLV)'s results are perfect at the beginning of the experimentation, remain unchanged with certain levels of incompleteness, and drop slightly in the experiments with the minimum possible information. ARMS and FAMA experience a slight drop of 2-3 F-Score percentage points when including some incompleteness in the input data, but then remain stable until the last experiment, where they lose 10 percentage points of F-Score (a little more in the case of ARMS).

As evidenced by the results, PlanMiner (NSLV) boasts a higher overall performance than the reference algorithms. In the worst cases, PlanMiner (NSLV) matches the performance of the benchmark algorithms, but, in the best situations, the F-Score gap in favour of PlanMiner grows as wide as 30 percentage points. Among the reference algorithms, we can see that FAMA presents the best results overall. This is due to the high resilience of the algorithm to incompleteness, which, faced with missing input data, is able to obtain good results (an exception to this rule is DriverLog, where the algorithm makes many over-information errors that negatively impact its F-Score). On the other side of the coin are the other state-of-the-art algorithms, which, depending on the problem, obtain different results (and usually below the performance of FAMA or PlanMiner (NSLV)).

The validity results can be found in Table 3.7 overall, we can see that the benchmark algorithms ARMS, FAMA, and AMAN have no trouble learning valid domains except in certain experiments with a certain set of domains or with high levels of incompleteness. The exception to this rule is OPMaker2, which is rather lacking in this regard, and FAMA, which struggles in the DriverLog domain. As previously discussed, validation measures whether domains are semantically correct, i.e., whether they can correctly reproduce plans of a given domain. A syntactic error may not invalidate a domain if it still allows the reproducibility of the

plans (even if the domain has lost "quality" by missing or excess elements). This is what happens with ARMS, FAMA and AMAN, which, although the F-Score of their domains is far from perfect, are able to obtain valid domains. OpMaker2 (and FAMA with the DriverLog domain), on the other hand, suffers from too many problems during the learning process (highlighted by a lower F-Score than the rest of the reference algorithms) that do influence this reproducibility, thus rendering the learned domains invalid.

### 3.3.4 Numerical domains results

Finally, the experimentation also studies how the chosen classification algorithm affects the performance of PlanMiner when the planning domains have numerical information. The classification algorithms used in these experiments are shared with the previous set of experiments with the exception of ID3. The results of this experimentation are divided between Figure 3.5 and Table 3.8. First, Figure 3.5 contains a graph that compares the algorithms in terms of F-Score, while Table 3.8 contains the validity results of PlanMiner with the different classification algorithms. In order to improve the readability of the results, Figure 3.5 displays the degree of incompleteness of the plan traces in the X-axis, and Table 3.8 sorts the results by the domain from which they were obtained. This section presents a summarised version of the results. An extended version can be found in Appendix B (specifically in Tables B.9, B.10 and B.11).

If we look closely at the results the figure 3.5 we can see the following:

- **Depots.** PlanMiner (NSLV) shows results above 96% F-Score throughout most of the experimentation, with no noticeable loss of performance until the most complex experimentation. PlanMiner (C45) starts the experimentation with results above 90 F-Score percentage points but quickly drops off when some incompleteness is encountered. PlanMiner (RIPPER) behaves similarly to PlanMiner (NSLV), but with a much worse overall performance, with results below 80%.
- **DriverLog.** PlanMiner (NSLV)'s results remain stable at around 95% F-Score, with little variation despite the levels of incompleteness; however, with the increased scarcity of input data, it suffers a drop of about 10 percentage points. PlanMiner (C45) loses 25 percentage points as soon as incompleteness is included in the input data, but, after this initial drop, it stabilises during the rest of the experimentation. PlanMiner (RIPPER) does not suffer such a sharp drop, but its results in general are below those of the other algorithms.
- **Rovers.** PlanMiner (NSLV) and (C45) start from the same point when learning the Rovers domain. Both approaches fare similarly, with PlanMiner (NSLV) performing slightly better overall. PlanMiner (RIPPER) has worse initial results but ends up matching PlanMiner (C45) with 50% of missing data.
- **Satellite.** PlanMiner (NSLV) shows a stable 99% F-Score performance against certain levels of incompleteness; nonetheless, as the complexity of the experiments increases this performance drops by 10%, but stabilises again on

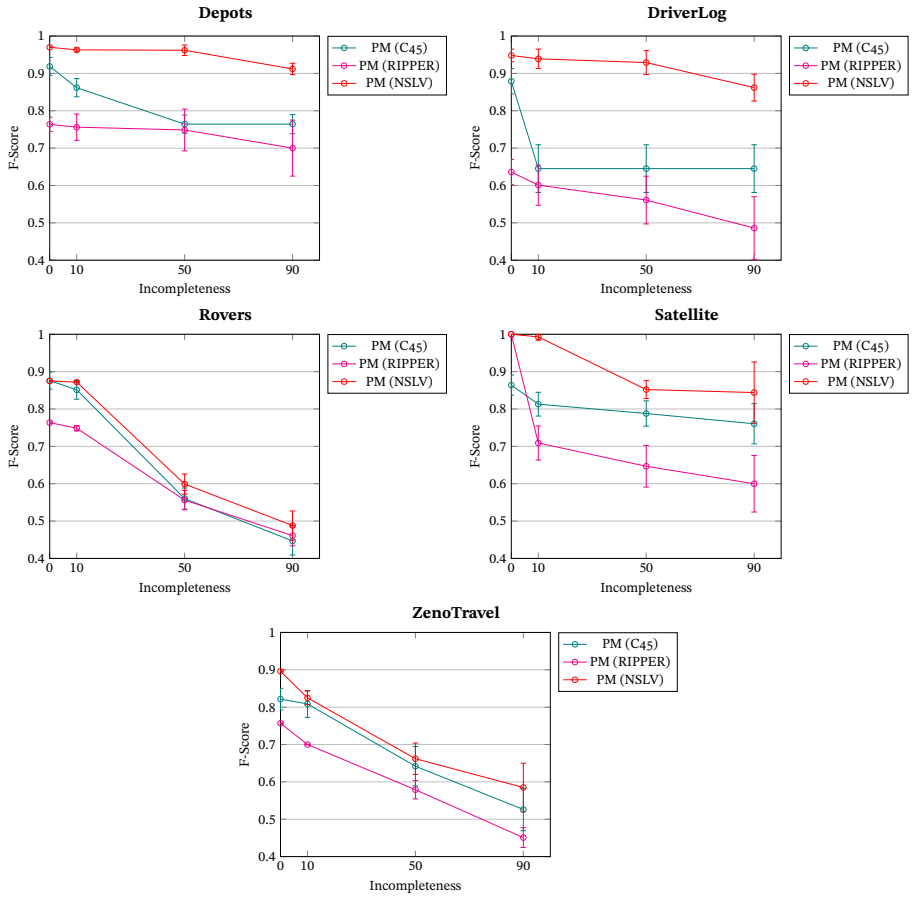



Figure 3.5: Performance comparison of PlanMiner using different classification algorithms on Numerical domains.

the more complex experiments. PlanMiner (C45) has an initial score of 88%, which gradually drops to 78 in the most complex experiment. Finally, PlanMiner (RIPPER) loses 30 percentage points of F-Score when confronted with noise, and this drop continues in the rest of the experimentation until the results of the approximation reach 60 percent with the most complex experimentation.

- **ZenoTravel.** With some differences in their performance, PlanMiner (NSLV), (C45) and (RIPPER) perform similarly, with a steady drop in performance as the experimental conditions become more severe. The difference between the best performing approach (90% F-Score, NSLV) and the worst (75%, RIPPER), of 15 percentage points, remains constant throughout the experiment.

Similar to the experimentation with STRIPS domains, NSLV demonstrates a clear dominance over the other classification algorithms, and, except for specific experiments, shows a clear resilience to incompleteness. Even so, all classifiers present a similar problem in the experiments: the rapid deterioration (and poor performance) of the precision metric. This is due to problems of (i) over-information of the data (mentioned earlier) and (ii) data bias. (i) Over-information affects domains where there are predicates that can be inferred from other predicates. These predicates are redundant and therefore erroneous, which reduces the performance of the domains even though they do not affect their validity. Depots, DriverLog, and ZenoTravel are the domains most affected by this problem. For example, in Zenotravel we have a situation where, when a person is on a plane, he cannot be in a city, and vice versa. This means that by only stating that the person is on a plane, the information about not being in a city can be omitted. When learning the different elements of the action models, PlanMiner cannot automatically perform this concept inference (as it does not have the necessary expert knowledge to do so), so it would learn both relationships. (ii) The problem of data bias occurs when the data does not present a certain diversity of information, which causes PlanMiner to work with spurious relationships. This problem is exacerbated as access to the data is restricted, and when taken to the extreme it causes actions to lose their generalisation, thus rendering them useless. Rovers, Satellite, and ZenoTravel are the domains where this problem is most prevalent. For example, locations representing the world are often related in a bidirectional way. As discussed in previous chapters, this means that to represent a link between location A and B, two predicates ((*connected A B*) and (*connected B A*)) can be used. During the learning process, PlanMiner obtains both, although from a human standpoint one of them is redundant. These issues hinder the overall performance of the algorithm, whose weakness surfaces through the precision metric, which shows results one step below the recall metric.

| Domain | Incompleteness | Algorithm | | |
|---|---|---|---|---|
| | | PM (C4.5) | PM (RIPPER) | PM (NSLV) |
| Depots | 0% | ✓ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✓ |
| DriverLog | 0% | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ |
| Rovers | 0% | ✗ | ✓ | ✓ |
| | 10% | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ |
| | 90% | ✗ | ✗ | ✗ |
| Satellite | 0% | ✗ | ✓ | ✓ |
| | 10% | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✓ |
| | 90% | ✗ | ✗ | ✗ |
| ZenoTravel | 0% | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✓ |
| | 50% | ✗ | ✗ | ✗ |
| | 90% | ✗ | ✗ | ✗ |

Table 3.8: Validity Results

In terms of validity, the problems presented earlier only affect PlanMiner (NSLV) in more complex experiments, i.e., experiments with a 90% incompleteness or experiments with the Rovers and ZenoTravel domains. In the former, the lack of data prevents PlanMiner (NSLV) from distinguishing between correct and erroneous predicates and ends up causing the loss of validity of the learned domains. In the latter, the large number of predicates and fluents defined in the domains causes precision bias errors even with complete data. By removing information from the input data, these errors are triggered and ultimately lead to the loss of validity of the learned planning domains.

## 3.4 Conclusions

This chapter has presented PlanMiner, a novel AML technique capable of learning planning domains that make use of preconditions and effects with arithmetic and relational expressions. This technique has been implemented as a pipeline of machine learning methods, which aims to obtain a set of classification models from which the preconditions and effects of an action model can be extracted. To achieve this goal, PlanMiner is given a set of plan traces from which it extracts information about the states of the world. This information is enriched through various ma-

chine learning processes and then used as the input of a classification algorithm that generates the models mentioned above. PlanMiner has been validated with domains obtained from the IPC, comparing its results with other existing state-of-the-art solutions. In the proposed experimentation, PlanMiner, when NSLV was used in its core as the classification algorithm, has demonstrated clear superiority in terms of performance over state-of-the-art algorithms, as well as a high capacity to learn planning domains under conditions of high input data sparsity.

# Chapter 4

# Learning Planning Domains from Noisy Plan Traces

## 4.1 Introduction

Noise is one of the key elements that most frequently affects data taken from real-world sources [WSF95]. This is because it is very common for the measurement tools from which such data is obtained to take incorrect data, either because they suffer from random malfunctions, have measurement errors due to hardware limitations or the conversion of analogue to digital signals, or simply due to human error. These errors are, in many cases, unavoidable, and algorithms that attempt to use data obtained from these sources must be designed to cope with this.

PlanMiner makes a number of assumptions during the learning process that hinder its work dealing with noisy data and, hence, learning planning domains using input data extracted from the real world. The first of these assumptions is that the input data is always correct (i.e. noise-free) so the regression models that are fitted during the information discovery step are also correct (if they exist). This assumption leads to assume that the datasets are always correct, and, in case an anomalous element is found among them, it can be accepted without the need to evaluate it to discern whether it is noise or an uncommon example. The third and final assumption is that classification algorithms only return a single classification rule for the pre-state and post-state meta-states (that is to say, a characterisation of the pre-states and post-states of a single action), which may not be true when noisy data is used as input. These assumptions improve the efficiency of the algorithm and increase the performance of PlanMiner when dealing with incomplete data, but when it comes to using it on noisy problems, they hinder its usage for learning planning domains.

In order to improve PlanMiner's resilience to noisy data, PlanMiner-N has been designed. The purpose of this chapter is to define PlanMiner-N and its components. The main contribution of PlanMiner-N with respect to PlanMiner is the suppression or adaptation of the previously detailed assumptions, implementing

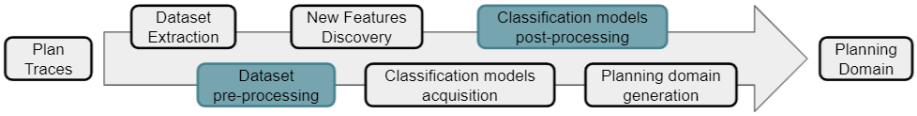



Figure 4.1: The learning pipeline of PlanMiner-N

a series of methods to make up for the shortcomings caused by these alterations. These methods slightly modify the PlanMiner pipeline (see example in Figure 4.1) to include two extra steps. These steps involve (i) pre-processing of the input data to detect and remove noise from the input data and (ii) post-processing of the obtained classification rules to detect inconsistencies between them and refine them. These processes are implemented using a number of statistical and unsupervised learning techniques. In the state-of-the-art, we can find several solutions capable of learning action models from noisy input data. Examples of these solutions are AMAN [ZNK13, ZPK19], Pasula et al. [PZK07], Zettlemoyer et al. [ZPK05] or Mourao et al. [Mou14], which, although they face the challenge of learning planning domains when there is noise in their input data, they generate planning domains that have little expressiveness (they are mainly only able to learn STRIPS domains). PlanMiner-N solves this by implementing PlanMiner's ability to learn planning domains whose actions contain arithmetic and relational expressions in their preconditions and effects.

The rest of the chapter is organised as follows: In the next section, the changes that PlanMiner-N entails regarding the solution proposed in the previous chapter will be briefly explained and their impact on the learning process. Then, within the same section, each of these new methods will be explained in detail and illustrated with comprehensive examples of them. Finally, the chapter will end with a presentation of the experimentation carried out to validate the methods described above.

## 4.2 PlanMiner-N

PlanMiner-N [SMFOP21] includes two new processes to the pipeline presented in the previous section. These processes presented extend the PlanMiner algorithm to improve its resilience against noise. Such methods were designed with the philosophy of enriching the learning pipeline without modifying its key parts. These new steps developed are the filtering of the noisy input data and the refinement of the meta-states. In the following lines, we will explain these steps in detail, illustrating the whole process with examples taken from domain *Rovers* (defined in Listing 2.1). This plan traces is a modified noisy trace from the example trace of Listing 2.3. In this example noisy trace, the erroneous elements are highlighted in bold.

**Algorithm 9** PlanMiner-N Algorithm overview

**Input**: PT: Set of Plan Traces
**Output**: AM: Set of Action Models

```
1:  Initializes stDict as an empty dictionary
2:  for all Plan trace pt in PTs do
3:      for all Different action act in the pt do
4:          Extract state transitions st of act in pt
5:          stDict[act] ← stDict[act] ∪ st
6:      end for
7:  end for
8:  for all key act in stDict do
9:      dat ← dataset created using stDict[act]
10:     Filter noise from dat #Step added in PlanMiner-N
11:     Detect new features from dat and extend dat with them
12:     Fit a classification model cModel using dat as input
13:     Refine cModel #Step added in PlanMiner-N
14:     Generate action model am from cModel
15:     AM ← AM ∪ am
16: end for
17: return AM
```

### 4.2.1 PlanMiner-N Overview

Algorithm 9 shows PlanMiners-N's general workflow, highlighting the changes introduced to PlanMiner's original workflow. PlanMiner-N modifies PlanMiner original contribution to enable the latter to operate under noisy input data situations. As said earlier, PlanMiner-N implements two new steps in the original pipeline of PlanMiner, performing in each one the following tasks:

1. **Plan traces noise filtering.** Just after storing the information of the plan traces in a dictionary whose keys are the action names and their values a set of associated state transitions, this process is applied. This procedure (step 10 of Algorithm 9) aims to clean the input information. Depending on the type of data found in a dataset being addressed (either predicates or numerical fluents), a different process is applied. A noise filter based on statistical filtering is applied to the predicates, which depends on a frequency threshold that determines whether a predicate is noisy or not. Regarding numerical fluents, we assume that the noise produced is random, so they are discretised and smoothed to reduce fluctuations in their values. This must be done so that the other components of the learning pipeline can perform their work to prevent the influence of the noisy values in its output.

2. **Meta-state refinement.** As explained above, a meta-state is a characterization of the predicates/fluents that can be found in either the pre-state or the post-state of an action. In the case of information without noise, each characterization is directly obtained and represented as the antecedent of a single

rule (see Section 3.2.4). However, when addressing noisy datasets, the classification model initially obtained (step 12 of Algorithm 9) may contain several characterizations (i.e. it can found several rules) for either the pre-state or post-state of an action. The Meta-state refinement (step 13 of Algorithm 9) solves this, enforcing the constraint that only two meta-states are needed, by combining the elements of the rules intially learned.

### 4.2.2 Plan traces noise filtering

The noise filtering process of PlanMiner-N aims to detect and delete anomalous erroneous information from the input data. The original PlanMiner algorithm takes the input data and preprocesses it to adjust the format of the data (i.e. dataset extraction) and to enrich it (i.e. discovery of new knowledge), PlanMiner-N introduces a new element to that preprocessing with the filtering of the input data. This new element is included between the two previous ones, and, as previously mentioned, its purpose is to reduce or alleviate as much as possible the noise problems that the input data may have.

Due to the nature of the information contained in the input data, we need to apply different techniques to it, since the noise treatment for the nominal values of the predicates is different from the noise treatment of the real values of the fluents. The only type of noise that affects predicates is outliers, while fluents are also affected by random noise. In the example noisy trace of Listing 4.1 we can see an outlier in the logical predicates in predicate (*at rov1 wp2*) of state `[0]`, an example of random noise in the numerical predicates in fluent (= (*at rov1*) *3.25*) of the same state, and an example of outlier in the fluents in element (= (*dist wp2 wp3*) *380*) of state `[2]`. This problem with the different types of noise conditions the techniques defined in PlanMiner-N, causing the input data preprocessing step to be applied differently for each type of input data, as there is no jack-of-all-trades preprocessing technique to deal with noise.

On the one hand, for noise in the logical predicates, PlanMiner-N implements statistical filtering in order to detect outliers and eliminate them. This filtering studies the distribution of the different truth values of each predicate along the traces, counting their frequency of occurrence. Then, if a truth value has an anomalously low frequency of occurrence, it is marked as noisy and removed. This process can be implemented because there are no conditional or stochastic behaviours in the learned actions. If such behaviours existed, we would not be able to discern between those outlier truth values or those that are related to an infrequent, but correct, non-deterministic behaviour (this issue will be tackled in Chapter 5).

#Actions
[0][1] *(goto rov1 wp1 wp2)*
[1][2] *(goto rov1 wp2 wp3)*

#States
[0] *(at rov1 wp1)* ∧ ***(at rov1 wp2)*** ∧
*(¬ (at rov1 wp3))* ∧ *(¬ (scanned wp3))* ∧
***(= (bat_usage rov1) 3.25)*** ∧ *(= (energy rov1) 450)* ∧
*(= (dist wp1 wp2) 50)* ∧ *(= (dist wp2 wp3) 80)*

[1] *(¬ (at rov1 wp1))* ∧ *(at rov1 wp2)* ∧
*(¬ (at rov1 wp3))* ∧ ***(scanned wp3)*** ∧
*(= (bat_usage rov1) 3)* ∧ ***(= (energy rov1) 299)*** ∧
*(= (dist wp1 wp2) 50)* ∧ *(= (dist wp2 wp3) 80)*

[2] ***(¬ (at rov1 wp3))*** ∧ *(¬ (at rov1 wp1))* ∧
*(¬ (at rov1 wp2))* ∧ *(¬ (scanned wp3))* ∧
*(= (bat_usage rov1) 3)* ∧ *(= (energy rov1) 60)* ∧
*(= (dist wp1 wp2) 50)* ∧ ***(= (dist wp2 wp3) -8000)***

Listing 4.1: Example of a noisy plan trace

On the other hand, for numerical predicates, PlanMiner-N bases the noise filtering on a discretisation technique [GLH15] that groups the different elements of a fluent under a series of discrete labels that replace them. In ML, discretisation is the process by which a set of continuous variables is transformed into a finite set of discrete variables. The benefits of discretisation [LHTD02] include categorising data for the sake of understandability, reducing the number of possible values of an attribute to improve the performance of ML algorithms, or, most relevant to the solution presented in this chapter, "smoothing" the discretised data. This smoothing process reduces fluctuations in the input data caused by random noise. By gathering similar elements under a single label, we do not only reduce the random noise of the data, but we can also isolate those data that are not similar to any other, i.e., the outliers.

This new preprocessing step is a powerful tool that can improve greatly the performance of the whole learning process, but, its major drawback is that it directly affects the execution of PlanMiner-N, increasing the amount of time that it requires to work. The preprocessing step is applied directly to the datasets extracted from the plan traces. The examples designed to illustrate the new methods start from the dataset of Table 4.1, a dataset created from the plan trace presented in Listing 2.1. In this table, in the *(bat_usage ?arg1)* attribute, examples of outliers (-4) or random noise (5.05) can be observed.

### Logical values noise treatment

Once the datasets have been created, the first action done by PlanMiner-N is to tackle noise in the logical attributes in the dataset. This is performed by PlanMiner-N by implementing a statistical filter to detect anomalies in the data. This filter is implemented over a collection of frequency tables that survey how the information is distributed in the dataset. These frequency tables measure the importance that

| (at ?arg1 ?arg2) | (at ?arg1 ?arg3) | (bat_usage ?arg1) | (energy ?arg1) | (dist ?arg2 ?arg3) | (scanned ?arg3) | *Class* |
|---|---|---|---|---|---|---|
| *True* | *True* | 3.25 | 450 | 50 | *MV* | *pre − state* |
| *False* | *True* | 3 | 299 | 50 | *MV* | *post − state* |
| *True* | *False* | 3 | 300 | -8000 | *True* | *pre − state* |
| *False* | *False* | 3 | 6000 | 86 | *False* | *post − state* |
| *True* | *False* | 3 | 230 | 75 | *MV* | *pre − state* |
| *False* | *True* | 3 | 5 | 75 | *MV* | *post − state* |
| *True* | *True* | 2.97 | 400 | 35 | *True* | *pre − state* |
| *False* | *True* | 3 | 295 | 33 | *False* | *post − state* |
| *True* | *False* | 5 | 400 | 75 | *False* | *pre − state* |
| *False* | *True* | 5.05 | -50 | 75 | *True* | *post − state* |
| *True* | *False* | 5 | 500 | 50 | *False* | *pre − state* |
| *True* | *True* | 5 | 250 | 50 | *False* | *post − state* |
| *True* | *False* | 3 | 315 | 1005 | *True* | *pre − state* |
| *False* | *True* | 3 | -0.5 | 105 | *True* | *post − state* |
| *True* | *False* | -4 | 500 | 80 | *MV* | *pre − state* |
| *False* | *True* | 5 | 100 | 80 | *MV* | *post − state* |
| *True* | *False* | 3 | 46 | 15 | *False* | *pre − state* |
| *False* | *True* | 3 | 10001 | 15 | *False* | *post − state* |

Table 4.1: Noisy dataset associated with the *(goto ?arg1 ?arg2 ?arg3)* action.

each value has in a given attribute. In the case of finding an anomaly in these measures, it is filtered out and erased from the dataset. An anomaly is a value with an abnormally low importance.

The filter (Algorithm 10) implemented in PlanMiner-N operates as follows: Starting from a single dataset, the filter fixes a class label (either pre-state or post-state) and creates a sub-dataset with only the information of the fixed label. Using this sub-dataset, the filter creates a frequency table for each different logical attribute. These frequency tables measure the number of times a given attribute takes a certain value. For a certain attribute, if the relative number of times a value appears is below a threshold, then it is considered *irrelevant*. Irrelevant attributes are considered as there is no useful information to be extracted from it, thus they are counted as noise. The deletion of a value is realised by selecting its appearances in the attribute's column and replacing them for missing values tokens. Since internally the pipeline components follow the OWA, the inclusion of a non-determined missing value does not influence the method of operation of the algorithm. Once a sub-dataset has been processed, the other class label is selected and the procedure is repeated. Figure 4.2 presents an example with the frequencies of the predicates from the example dataset presented in Table 4.1. In this example, it can be seen graphically the frequency rate of their values. Those values that do not exceed the threshold set by PlanMiner-N (defined by the green bar) will be eliminated.

The threshold set to filter noisy elements of the dataset can impact heavily on the performance of the process. In case of dealing with a situation where an attribute displays a non-noisy value with a low appearance rate, a very high threshold value may lead to detecting it as noise, and, therefore, erasing it. The absence of this value may hinder the later learning processes, including, in fact, an artificial extra noise. On the other side, a low threshold value may work the other way around, setting as "uncommon examples", noisy values. This issue would make the filtering process useless.

**Algorithm 10** Statistical noise filter overview

**Input** dat: Dataset
**Output** dat: Dataset

1: **for all** Class label *cLabel* in *dat* **do**
2: Initializes *subDat* as empty Dataset
3: **for all** Instance *i* in *dat* whose class is *cLabel* **do**
4: Include *i* in *subDat*
5: **end for**
6: **for all** Instance *i* in *subDat* **do**
7: **for all** attribute *attr* in *subDat* **do**
8: tCount → count number of True values of *attr*
9: fCount → count number of False values of *attr*
10: **if** $\frac{tCount}{tCount+fCount} <$ threshold **then**
11: Erase all True values in *attr*
12: **end if**
13: **if** $\frac{fCount}{tCount+fCount} <$ threshold **then**
14: Erase all False values in *attr*
15: **end if**
16: **end for**
17: **end for**
18: Update *dat* with the information of *subDat*
19: **end for**
20: **return** dat

### Numerical values noise treatment

After filtering the logical outliers, PlanMiner-N proceeds to process the numerical information in the input data. The variety of data contained in a noisy numerical attribute makes the use of a filter process like the one shown before inviable.

Even in a noise-free environment, a numerical continuous attribute may display wide range of different values. A statistical filter would not work correctly in this kind of situation, marking as noise the majority of values. PlanMiner-N implements an alternative filtering process (Algorithm 11) to deal with the noise in the numerical values. This filter takes as input a dataset, selects a numerical attribute, and extracts every element from it. The values extracted are then used as input of a discretisation algorithm that processes and groups them. Finally, the discretised values substitute the original values of the attribute.

The discretisation algorithm must produce a set of finite discrete elements from the collection of values used as input. PlanMiner-N achieves this using a clustering technique. The discretisation process groups the input elements into a set of clusters, and calculates the output discrete set of values as the mean element of each cluster. Given that the characteristics of the data are unknown a priori (the number of data points, their distribution, ...), the algorithm can not use a predefined number of clusters to fit the data. This provokes that PlanMiner-N must find the best

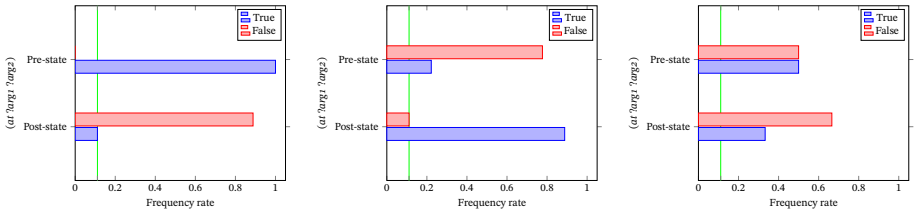



Figure 4.2: Frequency rates of the values of each predicate of the dataset presented in Table 4.1. Each graph represent a predicate and its values. The green line represents the threshold that determines whether a value is considered noisy.

**Algorithm 11** Numerical noise filter overview

**Input** dat: Dataset
**Output** dat: Dataset

1: Initializes *elems* as empty list of numbers
2: **for all** attribute *attr* in *dat* **do**
3: *elems* ← *elems* ∪ values list of *attr*
4: **end for**
5: *newAttr* ← discretize *elems*
6: Update *dat* with the information of *newAttr*
7: **return** dat

number of clusters automatically. PlanMiner-N implements a top-down divisive hierarchical clustering technique [LT06] to achieve this. This type of clustering methodology starts from every value in a single cluster and divides it recursively into smaller clusters. The principle on which these techniques are based is to incrementally create a larger number of clusters that better fit the data.

The clustering technique defined in PlanMiner-N (see Algorithm 12) follows a hierarchical, recursive divide-and-conquer strategy that works as follows:

1. Taking a single cluster with every element, the discretisation technique measures its quality.

2. If the quality of the cluster does not meet a certain acceptance criterion, the cluster is split into two new smaller clusters.

3. Then, the process of measuring and splitting of the clusters is applied to the new clusters.

This process is repeated recursively until there's no more clusters to split. If, during the process a cluster with a single element is found, the process marks it as outlier, and thus, the algorithm discards it. Cluster splitting is realised using a classical clustering algorithm. This algorithm will try to separate the elements of the cluster in two different groups. This task is realised in PlanMiner-N with the K-means clustering algorithm (explained in Section 2.3.4).

**Algorithm 12** Discretization algorithm overview

**Input** Dat: Set of data points, AC: acceptance criterion
**Output** C: Set of Clusters

```
1:  split ← Halve Dat using K_means algorithm
2:  C ← {}
3:  for all cluster in split do
4:     if Quality(cluster) >= AC then
5:        if |cluster| > 1 then
6:           C ← C ∪ cluster
7:        end if
8:     else
9:        C ← C ∪ divide cluster calling the discretisation process recursively
10:    end if
11: end for
12: return C
```

The quality of a cluster *quality*(*cluster*) measures how cohesive the values of a cluster are, as well as the size of the clusters. Cluster's quality is defined as the weighted arithmetic mean of both metrics:

$$quality(C_i) = \alpha * silI(C_i) + \beta * nSTD(C_i)$$

where *silI* is the index of the silhouette of the cluster $C_i$, and *nSTD* is the normalised standard deviation of $C_i$. Both functions are standard cluster quality measures, explained in Section 2.3.4 of this manuscript. These two metrics separately provide useful information about the quality of a cluster, but using them alone would be counterproductive. On the one hand, the silhouette index tends to benefit evenly distributed clusters. For example, when dealing with data that is homogeneously distributed, focusing only on the silhouette index would make our process to promote fit a set of wide clusters covering the entire range of data, but, that would represent poorly the data assigned to them. On the other hand, the single use of the standard deviation would lead to the generation of a large set of very tight clusters, which, taken to the extreme, could generate a cluster for each different value passed as input. These clusters would represent perfectly the datapoint assigned to them but would make the whole discretisation process useless too. Finally, $\alpha$ and $\beta$ are weights for the measures. These weights allow the fine-tuning of the quality function, giving more asymmetric importance to the measures when calculating the quality of a given cluster.

The combination of both metrics allows PlanMiner-N to obtain the best possible clusters, avoiding the problems described above. The main impediment to this combination of metrics is that there are differences between the behaviour of the metrics and their values ranges. Cluster standard deviation is a metric whose scores range in the interval $(0, \infty]$, while the silhouette index is bounded in the interval $[-1, 1]$. In addition, the best values of Cluster standard deviation are the values close to zero, this is opposite to silhouette index where the highest values are

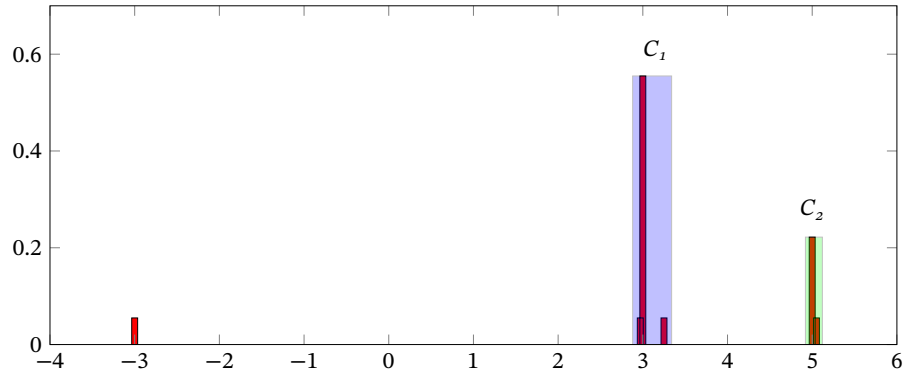



Figure 4.3: Example of discretisation. The x-axis shows the different values taken by (*bat_usage* ?*arg1*) in the dataset presented in Table 4.1, while the y-axis shows the frequency of occurrence of these values.

the bests scores. Given the behaviour of the metrics, a slightly worse score in the cluster's standard deviation may override a much better silhouette index. In order to lessen this issue and combine both metrics, PlanMiner-N adjusts the silhouette index by taking the opposite of the score obtained, and adding 1 to it. This transformation changes silhouette index results to the interval $[0, 2]$ where zero is the best possible value. This makes both indexes combinable in a single metric whose range is $[0, \infty]$, where Zero is the best result. Figure 4.3 shows the data points and clusters of the (*bat_usage* ?*arg1*) fluent. This example presents graphically how the values closer to "3" are grouped in the same cluster, while those closer to "5" are grouped in another different cluster. The value near "-3" is marked as an outlier and therefore is ignored in the next stages of PlanMiner-N's execution.

### 4.2.3 Meta-states refinement

As we have stated before, when dealing with noisy data, we cannot be sure that after extracting a classification model for a given dataset we will get a unique characterisation for the pre-states and another for the post-states (i.e. a rule set with two rules, one for each meta-state). Given the characteristics of the AML process presented in this document, it is necessary that this condition is always fulfilled. This is because a deterministic action model has only one rule for pre-state and one for post-state. In order to fulfil this constraint, PlanMiner-N implements this classification model refinement process, which merges a ruleset obtained from a classification algorithm, that characterises a single action, into a a pair of rules (one for the pre-state and other for the post-state). This process is performed before attempting to extract the preconditions and effects of the actions of the domain being learned. In addition, it presents an extra difficulty, as the multiple rules that can be learned by classification algorithms for each action may present inconsistencies between them. This is because the classification algorithms may have adjusted elements of

some of the rules to noisy information, detecting this is paramount for the correct learning of the planning domains.

The process implemented in PlanMiner-N to deal with this issue aims to (i) obtain the number of meta-states needed by PlanMiner to function correctly and (ii) detect inconsistencies in them. Prior to the application of said process, PlanMiner-N divides the ruleset in two sets, separating the rules that represent the meta-states of the pre-state from those that model the post-state. It then applies the refinement process to each collection of rules separately, obtaining a single rule for each.

This procedure decomposes the model (the ruleset obtained by the classification algorithm) to evaluate its elements (which are rules) using a statistical method. Once decomposed the rulesets, PlanMiner-N creates a repository with elements obtained, and evaluates them, allowing the detection of strange elements that can be considered noise and detecting inconsistencies between these elements. The Meta-state refinement method implements a set of procedures to study and resolve these issues automatically.

The evaluation method is implemented a as a two-step strategy that:

1. An anomaly detection strategy is applied to the elements of the rules' antecedents of a given ruleset, filtering those atypical elements found.

2. The remaining elements are studied in order to detect conflict between them, combining them in a single rule in the process.

Finally, to clarify that, in the end, the meta-state refinement method generates only the rules necessary for the learning process to proceed (i.e. two rules).

### Filtering of irrelevant features

The main objective of the filtering procedure (Algorithm 13) is to detect those elements of the rule antecedents that have an anomalously low frequency of occurrence. Marking them as noisy and discarding them for the rest of the procedure. Briefly recalling, each rule in the ruleset is defined as a conjunction of features $< attr, val >$ that has a weight associated with it. This weight indicates the percentage of examples of a given class covered by the rule in question.

That said, the implemented in PlanMiner-N to filter irrelevant features of the rules works as follows: Starting from a ruleset that contains several rules, it decompose the antecedents of each rule by separating the features that define them and computing its support value. The support of a feature is calculated by adding the weight of every rule in which it appears.

Once each feature has been extracted, all those that do not exceed a certain threshold are filtered out. Filter(*rules_feat*) marks and discards those features that are considered *irrelevant*. The criterion for marking a feature as irrelevant is given by the feature whose support value is higher, which is taken as the reference value. Finally, the filtering process sorts the set of features from highest to lowest according to their support value.

**Algorithm 13** Irrelevant features filtering Algorithm

**Input**: rules: Ruleset
**Output**: rules_feat: Ordered set of features

```
1: rules_feat ← {}
2: for all Rule rule in rules do
3:     for all Features elem in the antecedent of rule do
4:         rules_feat ← rules_feat ∪ elem
5:     end for
6: end for
7: Filter(rules_feat)
8: Short(rules_feat)
9: return rules_feat
```

**Algorithm 14** Meta-state Refinement Algorithm

**Input**: rules_feat: Ordered set of features
**Output**: rule: Rule

```
1: rule ← {}
2: while rules_feat is not empty do
3:     elem ← Top feature of rules_feat
4:     Delete elem from rules_feat
5:     if stat does not conflicts with some element of rule then
6:         Add elem to rule antecedent
7:     else
8:         SolveConflict(rule,elem)
9:     end if
10: end while
11: return rule
```

### Conflict resolution

Once the irrelevant attributes have been filtered and ordered out, the rest of them are combined in a single rule (Algorithm 14). This is done by taking the features in descending order and including them in an empty rule incrementally. During this process, PlanMiner-N may find a situation where a feature being included shares the attribute *attr* with another feature already included in the rule, but with a different value *val*.

This is called a *feature conflict* and occurs when the classification algorithms fits a rule to a specific set of data that is inconsistent with other rule. When a conflict is found, PlanMiner-N may face two course of actions: delete the feature with the lowest support (by considering that feature noisy) or delete both features (as considering that the whole feature is irrelevant to model the meta-state).

SolveConflict(*rule*,*elem*) determines said course of action by calculating a confidence interval and checking how the difference between the features' support value –namely $\Delta(s_1, s_2)$ –interact with it. The confidence interval is calculated as:

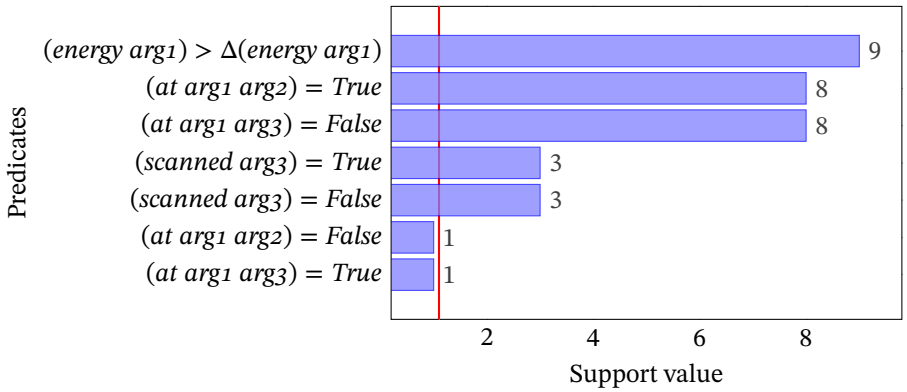



Figure 4.4: Support values of goto action's classification model.

$$[-0.1 * \bar{S}, 0.1 * \bar{S}]$$

where $\bar{S}$ as the mean support value between $s_1$ and $s_2$.

If $\Delta(s_1, s_2)$ is within that interval the difference is considered significant, and then, PlanMiner-N can conclude that the feature with the less support can be discarded. Otherwise, both characteristics are discarded and are not included in the rule that is being built. If no conflict of features arises, the feature is added to the antecedent of the rule being constructed.

In the example presented in Figure 4.4 we can see the support values of the classification rules that describe the pre-states of the (*goto* ?*arg1* ?*arg2* ?*arg3*) action. Those elements which support value is lesser than the threshold (defined by the red line) are erased. In this example we find a *feature conflict* with the elements (*scanned arg3*) = *False* and (*scanned arg3*) = *True*, in which both would be erased.

## 4.3 Experiments and Results

This section is devoted to the presentation of the experimental process of PlanMiner-N's validation. The aim of this experimentation is to demonstrate the robustness of the new methods implemented in PlanMiner to deal with noisy input data, as well as compare the performance of PlanMiner-N with state-of-the-art AML algorithms. A full description of the results, as well as the measurements and metrics outlined above, can be found in the Appendix C. In the following lines, when referring to a version of PlanMiner or PlanMiner-N that uses a given classification algorithm, we will refer to it as *PlanMiner(X)* or *PlanMiner-N(X)* respectively, where $X$ denotes the classification algorithm used.

The section is divided into 3 subsections: In the first one, the experimental setup will be described. Finally, the last subsections shows the results of the experimentation and the discussion of these results. Furthermore, the first experimental subsection is divided into two parts, the first part contains a comparison

| **PlanMiner-N parameters** | Value |
|---|---|
| Statistical noise filtering threshold | 5% |
| Cluster's quality *alpha* | 0.6 |
| Cluster's quality *beta* | 0.4 |
| Cluster's acceptance criterion | 0.05 |
| Irrelevant features detection threshold | 0.05 |

Table 4.2: Settings of the different algorithms during the experimentation process.

between the different classification algorithms of PlanMiner-N, while the second part compares the best version of PlanMiner-N with the state-of-the-art algorithms. For each block, a comparative graph of the results of each of the algorithms used in it is included, as well as a table detailing the validation results of the algorithms.

### 4.3.1 Experimental Setup

The experimentation presented here has been set identically to the experimentation of the previous chapter, and aims to validate the processes implemented in PlanMiner-N in order to design a noise-resistant action model learning algorithm. Both experimentations consist of two batteries of experiments, with the same input data, metrics and reference algorithms. The only two differences between the experimentations are (i) the inclusion of noise and (ii) the setting of new parameters added to PlanMiner-N.

#### Noise inclusion

The inclusion of noise differs between the experiments, as STRIPS-only domains only contemplate the noise produced by outliers, this type of noise will be the only one included in them. On the other hand, in the domains with numerical information, we must include both outliers and random noise. Similarly to the process of incompleteness inclusion, an element of a state is selected randomly and is modified. If the selected element is a predicate, its truth value is substituted by its contrary. If the selected element is a fluent, there's a 50% chance to substitute it by a random value or modified using a Gaussian distribution. The Gaussian distribution used for each fluent is centred in the value of the fluent and has variance 1.

#### New parameters settings

A number of new parameters governing the operation of the new PlanMiner-N components have been presented throughout the presentation of the technical contribution made in this chapter. For the sake of reproducibility of the experiments performed to evaluate the performance of PlanMiner-N, we provide the parameter settings used in this section. These parameters can be found in Table 4.2, and a detailed explanation of them can be found in the relevant sections presented earlier in this chapter.

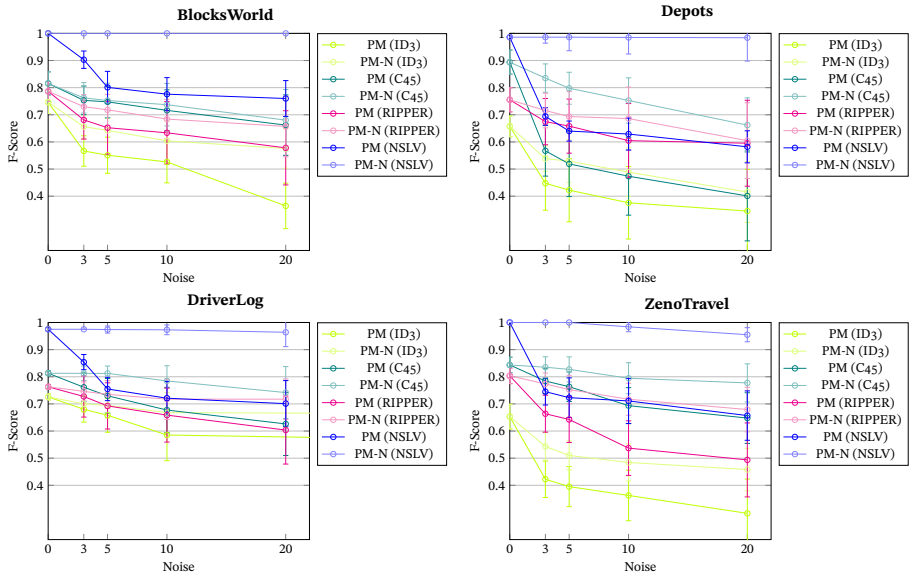



Figure 4.5: Performance comparison of PlanMiner-N using different classification algorithms on STRIPS domains.

## 4.3.2 STRIPS domains

### Comparison of classification algorithms

The experimental process begins by studying how the selected classification algorithm affects the performance of PlanMiner. Figure 4.5 shows these performances in terms of F-Score, while Table 4.3 shows the validity results of the battery of experiments. The Figure 4.5 fixes in X-axis the incompleteness degree of the plan traces, and Tables are shorted by domains in order to improve their readability. This section presents a summary version of the results. An extended version of this can be found in Appendix C (specifically in the Tables C.5, C.6, C.7, C.8, C.10, C.11, C.12 and C.13).

If we look closely at the results the figure 4.5 we can see the following:

- **BlocksWorld.** PlanMiner-N (NSLV) shows perfect results throughout the whole experimentation and is impervious to the effects of noise. Compared to PlanMiner (NSLV), PlanMiner-N (NSLV) shows almost 25% higher F-Score in the noisiest experiment. These differences also occur between PlanMiner-N (ID3) and PlanMiner (ID3), PlanMiner-N (C45) and PlanMiner (C45), and between PlanMiner-N (RIPPER) and PlanMiner (RIPPER), with an average variation of 10% throughout the experimentation. In the most complex experiments, NSLV performs better than ID3, C45 and RIPPER.

- **Depots.** As with BlocksWorld, PlanMiner-N (NSLV) has a high resistance to noise, which is unchanged throughout the experimentation. PlanMiner

(NSLV), on the other hand, suffers a severe drop in performance when some noise is included, which puts its F-Score almost 30 points below PlanMiner-N. The rest of the algorithms have similar behaviour to PlanMiner (NSLV), suffering from a performance drop when noise is included in the plan traces. PlanMiner-N shows values around 15-20% higher than PlanMiner using the same classification algorithm.

- **DriverLog.** The difference between PlanMiner-N (NSLV) and PlanMiner (NSLV) when dealing with some noise is 15% F-Score, a difference that increases to 30% using data with 20% noisy elements. PlanMiner-N (ID3) and PlanMiner(ID3) suffer an initial drop in performance (more pronounced in PlanMiner), but then stabilise and remain unchanged even at the highest noise levels. Using the C45 and RIPPER classification algorithms, similar behaviour to NSLV is observed, but not as marked with PlanMiner-N showing unchanged in the initial experiments, but dropping slightly in the final ones. Even so, the drops are much smaller than those seen with PlanMiner.

- **ZenoTravel.** PlanMiner-N (NSLV) obtains perfect results until it encounters 10% and 20% noise, where it drops to 98% and 96% F-Score respectively. PlanMiner (NSLV) drops from 100% F-Score to 74% when encountering some noise, this drop continues (although somewhat more controlled) throughout the experimentation, presenting results below 77% F-Score in the more complex experimental assumptions. The results of PlanMiner-N with ID3, C45 and RIPPER behave similarly to PlanMiner-N (NSLV), showing some noise resilience compared to their PlanMiner counterparts which lose a lot of performance when encountering some noise.

In general, all algorithms exhibit identical behaviour with a large drop in performance when noise is included in the plan traces with the exception of PlanMiner-n (NSLV). Since the PlanMiner algorithm is not designed to work with noisy information, severe performance losses are to be expected for it. On the other hand, PlanMiner-N, the approach tested in this experimentation, presents far better results than PlanMiner as expected too. PlanMiner-N shows some resistance to noise, but, as seen in the experimentation of the previous chapter, its performance is highly dependent on the classification algorithm used in the learning pipeline. This difference in the performance of PlanMiner-N can be seen in that, while experiments performed with NSLV remain somewhat stable throughout the experiment, those using the ID3 classifier, for example, show a drop of 10-12 points when noise is included in the plan strokes. This is due to the fact that, because of NSLV's ability to obtain descriptive rules from the datasets (i.e. rules that contain all the information necessary to represent a set of examples), the algorithm is able to generate sets of rules that fully explain the input information (including noisy examples). What is a priori a loss of generalisation of the algorithm is a blessing for PlanMiner-N, as it gives it extensive knowledge about the data, helping it to filter out the noise in the data. The other algorithms do not enjoy this advantage, providing less information to the learning algorithm. This means that PlanMiner is not able to correctly refine the models obtained with them. Nevertheless, if we compare PlanMiner and

| Domain | Noise | Algorithm | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | PM (ID3) | PM-N (ID3) | PM (C45) | PM-N (C45) | PM (RIPPER) | PM-N (RIPPER) | PM (NSLV) | PM-N (NSLV) |
| Blocksworld | 0% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Depots | 0% | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| DriverLog | 0% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ZenoTravel | 0% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

Table 4.3: Validity Results

PlanMiner-N using the same classifier, we see the clear superiority of the latter over the former regardless of the classification algorithm used. Using the ID3 classification algorithm, PlanMiner shows F-Score values below 30% in some experiments, while PlanMiner-N obtains 15 points more F-Score in the same experiments. With C.45 and RIPPER, PlanMiner-N shows an improvement of around 20% over PlanMiner throughout the experimental process. Although the biggest difference in performance can be observed with NSLV, since while PlanMiner obtains 58% F-Score results, PlanMiner-N obtains perfect results. This indicates that the changes made in PlanMiner-N are effective in addressing the learning problem using noisy input data.

In the experiments without noisy elements, the validity results are identical to those of the experiments in the previous chapter: those algorithms able to obtain valid planning domains with complete data can learn planning domains with noise-free data. This is because without noise and incompleteness the input data are essentially the same, and PlanMiner-N performs the identically as PlanMiner in the absence of noise in the plan traces. It is when noise is included in the input data that these results begin to diverge, as steep performance drops cause the invalidity of the domains learned by the bulk of the learning algorithms. As a reminder, validity is very sensitive to some domain deficiencies. The lack of a single effect renders the domain totally invalid, and the performance drops of the algorithms are so pronounced that these shortcomings arise everywhere. The exception to this rule is PlanMiner-N (NSLV) which demonstrates its supremacy over all other approaches by learning valid planning domains even in the most complex experiments. PlanMiner-N (NSLV) only fails to obtain valid domains with DriverLog and ZenoTravel in the experiments with the highest percentage of noise. In the case of the DriverLog domain, the invalidity is caused by the creation of a series of spurious preconditions that prevent the correct replication of the test problems, while in the case of ZenoTravel the problem that causes the invalidity is the lack of an effect with a low occurrence rate, which is erroneously detected as noisy by the algorithm and eliminated.

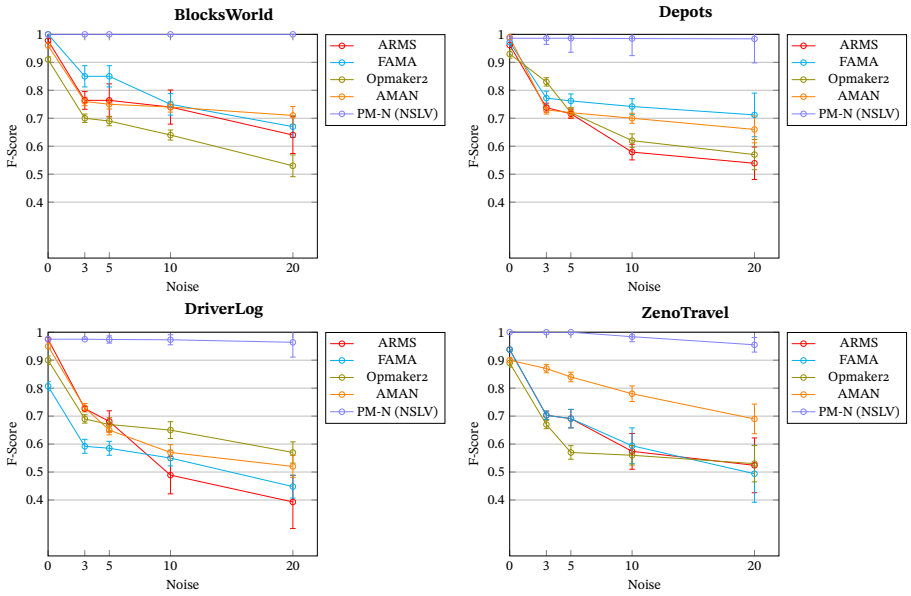



Figure 4.6: Performance comparison of between PlanMiner-N and state-of-the-art algorithms on STRIPS domains.

Finally, in terms of time efficiency, PlanMiner-N is around 10-20% slower than PlanMiner. This is due to the need to study and apply the noise filter to all predicates, which consumes computational resources.

### State-of-the-art algorithms comparison

Next, the experimental process is going to study how PlanMiner-N (NSLV) –the version of PlanMiner with the highest performance– performs in comparison to the reference algorithms. Figure 4.6 presents a comparative graph that displays the F-Score of these algorithms. Additionally, in Table 4.4 the validity results of the battery of experiments are shown. For the sake of readability, to say that the X-axis of the Figure 4.6 represents the degree of incompleteness of the input plan traces and that the Tables group the data displayed by the planning domain being learned. The next lines are a summarised version of the experimental result, the full version can be found in Appendix C (specifically in the Tables C.1, C.2, C.3 and C.4).

If we look closely at the results the figure 4.6 we can see the following:

- **BlocksWorld.** The benchmark algorithms suffer a steep drop throughout the experimentation, with results 20 to 30 points below the initial values with noise-free data. Of these, OPMaker2 is the worst performer, while ARMS and AMAN maintain similar performance. FAMA, on the other hand, suffers the least from the initial performance drop, but its results worsen severely as the complexity of the experiments increases. PlanMiner-N (NSLV), on the

other hand, performs better than all of them, obtaining 100% F-Score results regardless of the experiment performed.

- **Depots.** Starting from similar initial results, when including noisy data in the input data, the performance of the benchmark algorithms drops by almost 25 points. FAMA and AMAN show some stability in subsequent experiments, while the rest of the state-of-the-art algorithms continue to lose performance down to around 50% F-Score. In contrast, PlanMiner-N (NSLV) suffers a negligible drop in F-Score throughout the experimentation, maintaining a much higher performance than the benchmark algorithms.

- **DriverLog.** The benchmark algorithms show differences of up to 50 points between their initial results and those obtained in the more complex experiments. As usual, the steepest drops are found in the first experiments, when noise is included in the plan traces. In these experiments, the benchmark algorithms lose 20-25% F-Score. PlanMiner-N (NSLV) maintains a clear superiority over all benchmark algorithms.

- **ZenoTravel.** The benchmark algorithms present behaviour with large initial drops and constant, but more controlled drops as the experimental conditions are tightened. Of these, AMAN maintains a somewhat more stable behaviour, with more constant drops than the other benchmark algorithms. In the experiments with a higher percentage of noisy elements in the input data, the benchmark algorithms show results close to 50% F-Score, with the exception of AMAN, whose final results are around 70 points. However, PlanMiner-N (NSLV) shows results almost 30 points higher than AMAN.

The benchmark algorithms perform poorly when faced with noise in the input data. This behaviour was to be expected, since, with the exception of AMAN, none of these algorithms was expressly designed to work under noisy situations. Therefore, PlanMiner-N (NSLV) outperforms these algorithms in all experiments. The difference in performance between PlanMiner-N (NSLV) and the reference algorithms ranges from 10-30 points in the experiments with the lowest percentage of noise, to 30-60 points in the experiments with the highest percentage of noise. In terms of performance, second place is held by FAMA and AMAN. When faced with noise, both algorithms suffer a significant loss in performance. AMAN maintains a certain resilience to noise from that point on, showing little variability in results since then, while FAMA does not enjoy this benefit, but its initial performance loss is smaller than AMAN's. This causes FAMA to show slightly better average results. This causes FAMA to show somewhat better average results than AMAN.

As in the previous experimentation with the different versions of PlanMiner, the generalised loss of performance of the algorithms when noise is included has a negative impact on the validity results of the domains learned with them. Without noise, the benchmark algorithms perform similarly to what we saw in Chapter 3 of this document, but when noise is included, this is no longer true, as the multiple errors in the planning domains negatively affect the validity of the domains.

| Domain | Noise | Algorithm | | | | |
|---|---|---|---|---|---|---|
| | | ARMS | FAMA | OpMaker2 | AMAN | PM-N (NSLV) |
| Blocksworld | 0% | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✓ |
| Depots | 0% | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✓ |
| DriverLog | 0% | ✓ | ✗ | ✓ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ |
| ZenoTravel | 0% | ✓ | ✓ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✓ |

Table 4.4: Validity Results

### 4.3.3 Numerical domains

Finally, the experimental process studies how the selected classification algorithm affects the performance of PlanMiner-N when learning planning domains with numerical information. The classification algorithms used in this experimentation are the same used in the last experimental process, except for ID3. The results of this experimentation are divided between the Figure 4.7 and the Table 4.5. First, Figure 4.7 contains a graph that compares the algorithms in terms of F-Score, while, Table 4.5 contains the validity results of PlanMiner with the different classification algorithms. In order to improve the readability of the results, Figure 4.7 set the incompleteness degree of the plan traces in the X-axis, and Table 4.5 shorts the results given the domain from which they were obtained. This section presents a summary version of the results. An extended version of this can be found in Appendix C (specifically in the Tables C.14, C.15, C.16, C.17, C.18 and C.19).

If we look closely at the results in Figure 4.7 we can see the following:

- **Depots.** PlanMiner-N (NSLV) maintains 88% F-Score levels even in the complex experiments, losing less than 10% F-Score compared to the noiseless results. PlanMiner (NSLV) on the other hand suffers a drop of 20 using 3% of noisy elements, its final performance is below 70% F-Score in the experiments with 20% of noisy information. PlanMiner-N (C45) suffers a large F-Score loss of 13 points in the experiment with 5% noisy data, which stabilises somewhat in the following experiments. PlanMiner (C45), on the other hand, suf-

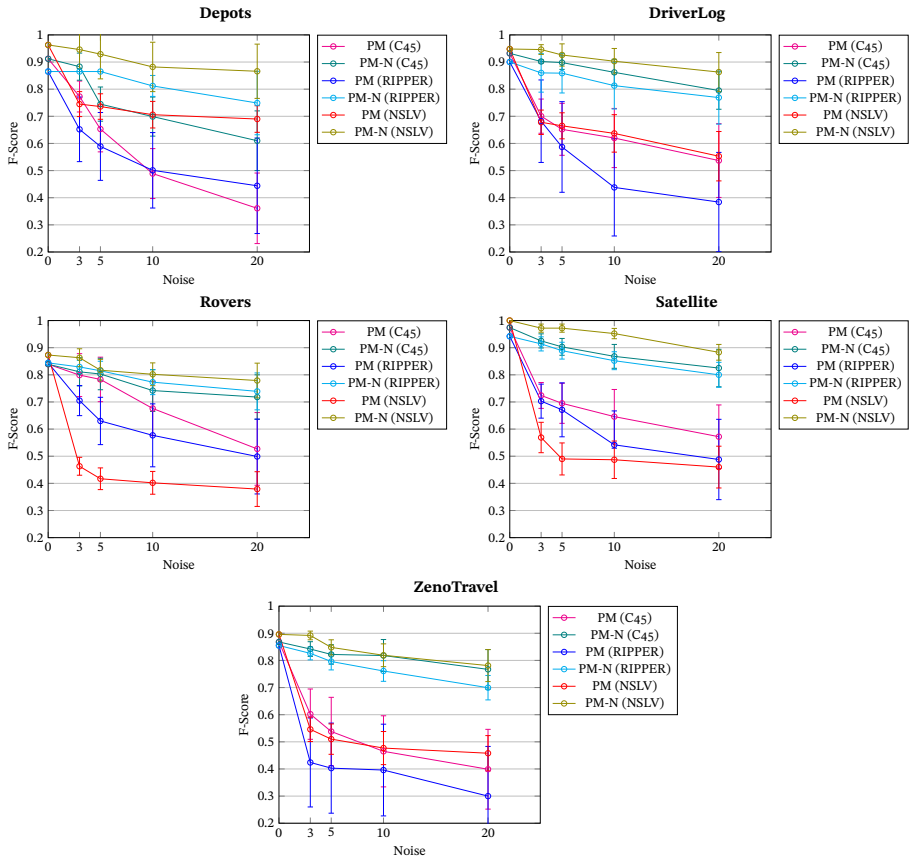



Figure 4.7: Performance comparison of PlanMiner-N using different classification algorithms on Numerical domains.

fers a steady drop in performance, bringing its final F-Score below 40 points. PlanMiner-N (RIPPER) and PlanMiner (RIPPER) show the biggest difference of all the algorithms, since, where PlanMiner-N (RIPPER) remains constant throughout the experimentation, suffering only F-Score losses in the more complex experiments, PlanMiner (RIPPER) drops steadily.

- **DriverLog.** The results of PlanMiner-N (C45), PlanMiner-N (RIPPER) and PlanMiner-N (NSLV) is similar regardless of the classification algorithm used. All approaches show a slight drop in F-Score throughout the experimentation, with intervals where the F-Score loss is zero. In general, these approximations lose on average 6 F-Score points between the experimentation without noisy data and the one with more noisy data. On the other hand, the experimentation on PlanMiner (C45), PlanMiner (RIPPER) and PlanMiner (NSLV) shows a steep F-Score drop of more than 20 points when noise is included. PlanMiner (C45) and PlanMiner (NSLV) moderate this tendency a little to reach an F-Score of around 55% in the final experiments, PlanMiner (RIPPER) on the other hand does not, reaching values below 40 F-Score points.

- **Rovers.** PlanMiner-N (NSLV) is little affected by noise, suffering a significant F-Score loss of 5 points in the experimentation with 5% noisy data. In the rest of the experimentation, the F-Score loss is minimal. PlanMiner (NSLV) loses 40% F-Score when noise is found in the input traces, a big contrast when compared to the results of PlanMiner-N (NSLV). Although the other proposals behave similarly, the F-Score loss of PlanMiner (NSLV) is the most pronounced in the whole experiment. PlanMiner-N (C45) performs at 72 points, compared to PlanMiner (C45) at 51 points, and PlanMiner-N (RIPPER) with a final F-Score of 73%, 20 points higher than PlanMiner (RIPPER).

- **Satellite.** Similar to DriverLog, the results of PlanMiner-N (C45), PlanMiner-N (RIPPER) and PlanMiner-N (NSLV) show a similar trend: a slight but steady drop in F-Score as the experimental assumptions become more complicated. These algorithms show F-Score drops of 11% in the more complex experiments compared to the no noise experiment. The results of the PlanMiner experiment show drops of 25 points with the C45 classifier and RIPPER, and more than 40 points with NSLV when faced with some noise. PlanMiner (NSLV) maintains the results in the rest of the experimental assumptions, leaving its final performance at 48 points, while PlanMiner (C45) and PlanMiner (RIPPER) obtain an F-Score of 58% and 49% respectively.

- **ZenoTravel.** PlanMiner-N (NSLV) remains unchanged at certain noise levels, with an F-Score of 89% at 3% noise. PlanMiner-N (NSLV) loses around 11% F-Score in the more complex experimentation. PlanMiner (RIPPER) and (C45) lose some F-Score until experimentation with 5% noise, but while PlanMiner (C45) shows some resistance to noise, (RIPPER) loses performance steadily. On the other hand, PlanMiner (NSLV), PlanMiner (C45) and PlanMiner (RIPPER) show a large initial drop, followed by a certain stabilisation

of the results. The F-Score value in the most complex experiments is 30 points PlanMiner (RIPPER), 40 points PlanMiner (C45) and 47 points PlanMiner (NSLV).

PlanMiner demonstrates some "natural resistance" to noise in this experiments, showing somewhat better results than the experiments with STRIPS domains. Still, the difference in performance between the version of PlanMiner presented in this chapter and the previous version is remarkable. Using the C.45 classifier, PlanMiner-N performs around 30 points better than PlanMiner, which can reach up to 50 points difference in certain planning domains. On the other hand, with the RIPPER classification algorithm, it maintains in all experiments a higher performance than 70% with PlanMiner-N, while with PlanMiner it reaches values around 30 F-Score points. Finally, with NSLV the same process is repeated (i.e. F-Score differences close to 4%), although aggravated with very significant decreases in performance simply by including noise. This contrasts sharply with PlanMiner-N (NSLV) which remains relatively unperturbed by noise, far outperforming the other approaches tested in the experiment. The aforementioned "natural resistance" to noise is given by the way the feature discovery component works, namely it is a product of the design of the symbolic regressor. Since the symbolic regressor does not seek exact results, but rather takes approximate (but as close to exact as possible) results as correct, the effect of the inclusion of certain noisy elements is diluted. Although these elements increase the error of the expression that is being learned, it is possible that the expression meets the acceptance criteria of the algorithm and is accepted by the learning algorithm. Unfortunately, this method is not infallible, and if insufficient data is available or if there are outliers with the potential to greatly perturb the error calculation, PlanMiner is unable to correctly learn the target expression. The inclusion of noise treatment methods increases this resistance, not so much because they influence the behaviour of the regression algorithm, but because they alter the noise problem and shift it to an incompleteness problem. As seen in the empirical studies in Chapter 3, incompleteness is highly tolerated by PlanMiner (and thus by PlanMiner-N) improving greatly the performance of the algorithm. Looking at the evolution of the accuracy and recall metrics of the algorithms throughout the experimentation (these results can be consulted in Appendix C), we can see how noise affects the first metric much more than the second. This is because any amount of noisy elements not addressed by the noise filtering processes triggers the bias problems described in the previous chapter. As we have seen in previous experiments, the property of NSLV to generate descriptive rules plays in its favour against bias problems. This particular case is no exception, and it is the main reason why NSLV performs better than the other classifiers.

Validity experiments show again the effectiveness of the methods implemented in PlanMiner-N for dealing with noise. If we compare the results of PlanMiner and PlanMiner-N, we see that PlanMiner's "natural resistance" to noise when learning numerical domains is not infallible and does not guarantee obtaining valid domains. The reason for this is as previously indicated in the experimentation of this and the previous chapter: the validity criteria are very demanding, and a single erroneous effect causes the entire planning domain to be invalid. PlanMiner can meet

| Domain | Noise | Algorithm | | | | | |
|---|---|---|---|---|---|---|---|
| | | PM (C4.5) | PM-N (C4.5) | PM (RIPPER) | PM-N (RIPPER) | PM (NSLV) | PM-N (NSLV) |
| Depots | 0% | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| DriverLog | 0% | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Rovers | 0% | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Satellite | 0% | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ZenoTravel | 0% | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| | 3% | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| | 5% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 10% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 20% | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

Table 4.5: Validity Results

these criteria in some experiments (using the RIPPER classification algorithm), but generally, this is not true. PlanMiner-N on the other hand does, at least under certain noise levels.

In terms of time efficiency, PlanMiner-N is 2% slower than PlanMiner, even outperforming it in some experiments. This is because, in the face of noise, PlanMiner may spend much more time trying to fit an arithmetic expression (even using all the time allowed for this and reaching the timeout threshold), which compensates for the time spent by PlanMiner-N in applying the different anti-noise processes implemented.

## 4.4 Conclusions

This chapter has presented PlanMiner-N, an AML technique built over the PlanMiner algorithm with the objective of implementing its learning capabilities under situations of noisy input data. Several methods have been developed over the original PlanMiner's learning pipeline with the aim of automatically detect and treat noisy elements in both the input data and the intermediate models used during the learning process. These new methods i) preprocess the input data of PlanMiner-N and ii) post-process the meta-states generated by the classification algorithms used during the learning process. i) The first method studies the information contained in the input data aiming to discern those erroneous elements contained in the plan traces in order to filter them. And ii) the second method refines the meta-states trying to find inconsistencies between them and implementing a process to clean them. PlanMiner-N has been validated with domains obtained from the IPC, com-

paring its results with other existing state-of-the-art solutions and with PlanMiner. In the proposed experimentation, PlanMiner-N has demonstrated clear superiority in terms of performance to state-of-the-art algorithms, as well as a high capacity to learn planning domains under conditions of high input data sparsity. In comparison to PlanMiner, the new methods have been proven useful to deal with noise input data, improving largely the algorithms performance.

# Chapter 5

# Learning conditional action models from plan traces

## 5.1 Introduction

The PlanMiner methodology was created with the goal of learning as expressive planning domains as possible, in order to implement them in real-world problems. The PlanMiner-N research presented in the previous chapter, explored the development of techniques to improve PlanMiner's resilience to poor quality input data, a common problem when extracting data from real-world sources, but, not exploring new ways of improving the expressiveness of domains learned by PlanMiner (with respect to numerical and relaltional expressions). This issue greatly limits the ability to implement the domain learner on problems closer to the real world.

The work presented in this chapter tackles this issue, introducing a novel action model learning technique able to learn planning domains a step ahead in terms of expressivity in comparison to those learned by PlanMiner. This work is a new variant of the original version of PlanMiner called PlanMiner-C, a learning algorithm for action models with conditional effects. Conditional actions are a type of planning action whose effects depend on the context in which they are applied, varying the result of their execution depending on certain aspects of the world to which they are applied.

As a preliminary step to implementing this technique in real-world problems, PlanMiner-C is designed to learn action models from information extracted from virtual videogame environments (specifically the GVG-AI environment [PLST+15] presented in Section 2.5 of Chapter 2). Our solution can use logs of agent executions trying to solve different scenarios (different game levels for many videogames), with the aim of being able to replicate their behaviour in a planning domain that could be used by a deliberative agent controlled by a planning engine. In GVG-AI, it is easy to find a conditional behaviour, for example, when pressing the `USE` key. Once this key is pressed, the consequences of the action executed by the automated player depends on the object the agent holds. Moreover, the vast majority of real-

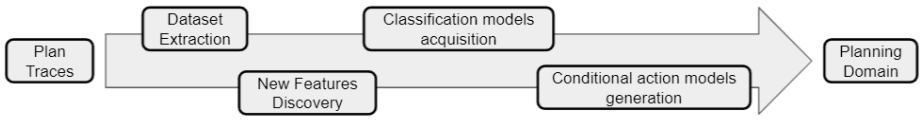



Figure 5.1: The learning pipeline of PlanMiner-C

world processes tend to have behaviours that vary depending on the context of the world or simple randomness, and designing a tool able to manage them is a major breakthrough in the action model learning field.

PlanMiner-C includes several modifications (outlined in Figure 5.1) to the original PlanMiner algorithm, overcoming its limitations, in order to handle the existence of actions with multiple behaviours depending on the context where they are applied. These modifications include i) a new method to handle input data, ii) a novel discovery of new features step, iii) a new classification model generation strategy and iv) a new PDDL domain generation process.

(i) The first modification is due to the need to deal with a larger volume of data than seen so far in PlanMiner and the use of a new encoding format for the input plan traces. In Chapter 2, Section 2.4.1 we already discussed this problem of data size, and how a better solution had to be found to optimise the format of the traces. To recapitulate, the execution logs extracted from GVG-AI are composed of the actions UP, DOWN, LEFT, RIGHT, and USE (of a concrete object), executed by the automated player in a game, with pre-states and post-states containing raw observations from this environment. The observations represent the game objects that can be found in each cell of the game. For example they provide information on the cell where the avatar is located, or the game object the avatar holds, or where the walls are, etc. This information can be straightly translated into instantiated literals (this translation is detailed in Section 5.4), and therefore into plan traces ready to be used by the PlanMiner's pipeline. But for a planner to properly work and solve problems in these types of domains (tile-based worlds) it is necessary to deduce the cells' connectivity relation from the observations for each concrete game level. Since PlanMiner assumes the OWA (Open World Assumption, if a literal is not present in a state its truth value is considered unknown) it would be necessary to encode in the plan traces additional literals to represent which cells are not connected with a given cell. This encoding would generate an extremely large volume of information that would make the processing of plan traces intractable. In order to improve this encoding, a new trace format was implemented (described in Chapter 2, Section 2.4.1 along with an example) that separated static relations (those elements that are immutable throughout an execution, for example the connectivity relation between tiles) from dynamic relations (those elements that vary throughout an execution) to handle and encoding them independently. The modifications made in PlanMiner-C are aimed at adapting the process of reading plan traces to this new reality on the input data.

(ii) Secondly, since we are addressing how to learn conditional actions, the different patterns that can be found in the datasets due to actions' situation-dependend effects must be taken into account when discovering new knowledge from the input data. PlanMiner-C rebuilds the original symbolic regression method implemented in the original algorithm with the goal of facing this issue and guide the learning process.

(iii) Thirdly, actions with multiple situation-dependent behaviours will have multiple meta-states associated with it (i.e. multiple rules associated with each action which explain its multiple situation-dependent effects). PlanMiner's original learning method was not designed for this reality. Therefore, we have redesigned the new classification models generation strategy, in order to optimise the rule generation method to handle several rules explaining the different behaviours of a single action model.

(iv) Finally, since an action with multiple behaviours is represented with more than one rule set (although obtained from a single dataset), it is necessary to implement a new PDDL generation process. This process has the goal of study the classification models obtained (i.e. the different rule sets) and outputs a single action model with conditional effects.

As previously mentioned, the GVG-AI environment has been used as the source from which to extract input data for PlanMiner-C. It should be noted that, due to the particularities of the GVG-AI environment, when measuring the quality of the data used, we find that we cannot apply the experimental methodology applied for PlanMiner and PlanMiner-N. This is due that this experimental processes need a reference planning domains to measure the learning domains. When learning from data extracted from a GVG-AI agent execution, there is no reference domains, so the mentioned experimental methodology can not be applied. This issue have led to the need to overcome the challenge of designing and proposing a new experimental process to measure PlanMiner-C performance.

Finally, it is worth noting that during the development of PlanMiner-C we discovered a possible way to learn action models with stochastic effects (such as those used by PPDDL). In due course, this way will be briefly pointed out and explained, but because it is outside the scope of the PlanMiner-C proposal, it will not be further explored or evaluated in the experimental sections (see Section 5.4) of this chapter).

The rest of the chapter is organised as follows: First, the learning requirements of the task ahead will be presented, illustrating them with several examples. Second, the PlanMiner-C algorithm will be presented, exposing the shortcomings of PlanMiner when dealing with the challenge ahead, in addition to the full description of the changes made in the solution proposed in this Chapter. Third, the experimentation will be presented to prove the capabilities of PlanMiner-C trying to learn planning domains from data extracted from the GVG-AI environment. And, fourth, the conclusions drawn during the whole chapter will be exposed.

| **Pre-state** | **Post-state** | *Action* |
|---|---|---|
| **0** | **1** | *(RIGHT a1 c16 c17)* |
| **1** | **2** | *(RIGHT a1 c16 c17)* |
| **2** | **3** | *(UP a1 c17 c12)* |
| **3** | **4** | *(USE a1 c17 c12)* |
| **4** | **5** | *(UP a1 c17 c12)* |

(a) State transitions

| **Index** | *Predicates* |
|---|---|
| **0** | *(= (posX a1) 3) ∧ ((= (posY a1) 1) ∧ (rockIn c12) ∧ (¬ (orien_U a1)) ∧ (orient_D a1) ∧ (¬ (orient_R a1)) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **1** | *(= (posX a1) 3) ∧ ((= (posY a1) 1) ∧ (rockIn c12) ∧ (¬ (orien_U a1)) ∧ (¬ (orient_D a1)) ∧ (orient_R a1) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **2** | *(= (posX a1) 3) ∧ ((= (posY a1) 2) ∧ (rockIn c12) ∧ (¬ (orien_U a1)) ∧ (¬ (orient_D a1)) ∧ (orient_R a1) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **3** | *(= (posX a1) 3) ∧ ((= (posY a1) 2) ∧ (rockIn c12) ∧ (orien_U a1) ∧ (¬ (orient_D a1)) ∧ (¬ (orient_R a1)) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **4** | *(= (posX a1) 3) ∧ ((= (posY a1) 2) ∧ (¬ (rockIn c12)) ∧ (orien_U a1) ∧ (¬ (orient_D a1)) ∧ (¬ (orient_R a1)) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **5** | *(= (posX a1) 2) ∧ ((= (posY a1) 2) ∧ (¬ (rockIn c12)) ∧ (orien_U a1) ∧ (¬ (orient_D a1)) ∧ (¬ (orient_R a1)) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)* |
| **Static Relations** | *(=(row c12) 2) ∧ (=(col c12) 2) ∧ (=(row c16) 3) ∧ (=(col c16) 1) ∧ (=(row c17) 3) ∧ (=(col c17) 2) ∧ (conn_U c12 c7) ∧ (conn_D c12 c17) ∧ (conn_R c12 c13) ∧ (conn_L c12 c11) ∧ (conn_U c16 c11) ∧ (conn_D c16 c21) ∧ (conn_R c16 c17) ∧ (conn_L c16 c15) ∧ (conn_U c17 c12) ∧ (conn_D c17 c22) ∧ (conn_L c16 c15) ∧ (conn_R c17 c18) ∧ (conn_L c17 c16)* |

(b) States list

Table 5.1: Extract from a Boulder Dash videogame plan trace.

## 5.2 Learning requirements

Initially, when we start to face the challenge of improving the expressiveness of learned patterns, an attempt was made using PlanMiner and PlanMiner-N with no success. During this attempt, information extracted from GVG-AI was used as input to the algorithms, but the action models obtained were useless. Despite the initial failure, a series of important requirements were detected that needed to be taken into consideration. These requirements were:

- **Handling of anomalous observations in the input data.** In a similar situation as the one that PlanMiner-N faces, there may be atypical observations in the input data taken from the GVG-AI environment. But, contrary to as it occurs with PlanMiner-N, these observations cannot be automatically flagged as erroneous and discarded, as they come from uncommon but valid behaviours. This means that all these behaviours must be taken into account, studied and, if necessary, modelled in the output domain of the algorithm.

- **Handling of multiple rules for each meta-state.** When faced with anomalous behaviour in the input data, a new requirement arose from the way in which the preconditions and effects of the action models are learned by PlanMiner (and PlanMiner-N). The requirement is that there can be several meta-states for the pre-state or post-state of the action models. This issue has already been discussed in the Chapter 4 and, also, the need for learning algorithms proposed throughout this manuscript to accept only one rule for each meta-state was commented. This requirement imposed by PlanMiner, which clashes head-on with the real needs of the learning problem presented by the data extracted from GVG-AI, was the main culprit for the initial failure of the learning algorithms.

- **Handling of static relations.** Last but not least, a rather large obstacle was found in the plan trace format initially proposed for PlanMiner and PlanMiner-N. This obstacle made the plan traces as they were conceived so far obsolete, leading to the design and development of a new plan trace format (both formats are defined in Chapter 2 of this document). The trigger for this error was the existence of large amounts of static relationships in the input data that led to even small runs of GVG-AI taking up huge amounts of disk space, making their processing intractable in practice.

The processes implemented during the development of PlanMiner-C, and presented in this chapter, aim to create a learning process able to learn action models with conditional effects. But, in order to do so, apart from satisfying the above described requirements, these new processes have to overcome a number of assumptions made in the original design of PlanMiner. In the previous chapter, we discussed the weaknesses of PlanMiner in dealing with noisy data, and that these weaknesses were caused by a number of assumptions made to improve the performance of the learning pipeline. To recap, these assumptions implied that: (i) the input data is always correct, and (ii) that, for each single action model, there is

only one rule that explains its preconditions and effects (i.e. that fits its pre-states and post-states). In the same way that under noisy scenarios these assumptions prevented the correct learning of the action models (discussed in depth in Chapter 4), under conditional behaviours such assumptions hinder the learning process, although for reasons different from those that affect the PlanMiner-N algorithm, which are explained in the following.

```
(:action RIGHT
        :parameters (?a - avatar ?c1 ?c2 - cell)
        :precondition (and
            (= (posX ?a) (col ?c1))
            (= (posY ?a) (row ?c1))
            (not (wallIn ?c2))
            (conn_R ?c1 ?c2)
        )
        :effect (and
            (when
                (and (orient_U ?a))
                (and (not (orient_U ?a)) and (orient_R ?a))
            )
            (when
                (and (orient_D ?a))
                (and (not (orient_D ?a)) and (orient_R ?a))
            )
            (when
                (and (orient_L ?a))
                (and (not (orient_L ?a)) and (orient_R ?a))
            )
            (when
                (and  (orient_R ?a))
                (and  (increase (posY ?a) 1))
            )
        )
)
```

Listing 5.1: Boulder Dash's videogame action with conditional effects

When learning planning domains with conditional actions, the first assumption implies in PlanMiner that the data can be prejudged, by supposing that anomalous data can be discarded as erroneous. This contradicts the premise that conditional effects may have an asymmetric (and very low) appearance rate and disrupts any attempt at learning such type of effects. For example, looking at the conditional action *RIGHT* from the Boulder Dash videogame (presented in Listing 5.1), we can see that the function (*posY* ?*arg1*) varies depending on the context of execution. The main function of this action is to increment (*posY* ?*arg1*) when it is executed, i.e. to update the position of the avatar indicating that it has moved to the left, but, as can be seen, this is not always the case. If the avatar is not facing left at the time the action is executed, its effect will be to orient it. This means that most of the time when this action is executed, its effects will have no repercussion on (*posY* ?*arg1*). Assuming that the anomalous behaviour (i.e. (*posY* ?*arg1*) is increased by 1) can be ignored without studying their nature, would mean that the conditional effect of the action would not be learned, since this observations would be detected as erroneous, and discarded. Furthermore, forcing that there is only one regression model that explains the behaviour of the function (*posY* ?*arg1*) would lead to the models of the two behaviours seen above being learned incorrectly, as the resulting model would mix both behaviours (i.e. resulting in a model that will try to represent

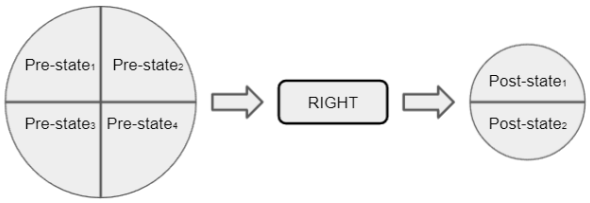



Figure 5.2: Scheme of the meta-states of the *move* action

both the increments of 1 and the situations where there is no changes).

On the other hand, the assumption that there is only one rule defining the pre-states or post-states forces PlanMiner to require neither more nor less than that this information in order to function properly. As explained in the last chapter, when dealing with noisy input data, this could not be assured, so PlanMiner-N implemented a meta-state refinement process to filter and combine several rules into a single rule, compelling this requirement to be met. When modelling conditional actions, it is possible to ensure 100% that this condition will be unfulfilled, since a single action will have several meta-states for either the pre-state or the post-state associated.

In the example seen above (in Listing 5.1), the *RIGHT* action would have associated four meta-states for the pre-state and two meta-states for the post-state (see scheme presented in Figure 5.2). On the one hand, the first collection of meta-states would represent the different situations provoked by the orientation of the avatar (each orientation has its own pre-state meta-state), while, on the other hand, the second pair of meta-states could represent those situations where (*posY ?arg1*) changes (or not). The meta-states of the same type would share many elements, but would contain key differences relative to the situations they represent.

Particularly, to note that the information about the values of (*posY ?arg1*) contained in the two meta-states of the post-states is crucial. From this information it can be extracted how said fluent changes, and, when used with the features discovery process, different regression models can be extracted as indicated in the previous paragraph. Trying to reduce the number of meta-states to fit the requirements of PlanMiner, as is done in PlanMiner-N, is counterproductive, as it would lead to the loss of important information of conditional action models.

## 5.3 PlanMiner-C

PlanMiner-C, follows the schema presented in Figure 5.1 and outlined above, meeting the requirements indicated in the previous section. In the following lines, we will explain the steps described earlier in detail, illustrating the whole process with examples taken from the videogame *Boulder Dash* (defined in Table 5.1).

### 5.3.1 Overview

Algorithm 15 shows PlanMiners-C's general workflow, highlighting the changes realised to PlanMiner's original workflow. PlanMiner-C redesigns PlanMiner original contribution to create a novel learning process able to learn action models with conditional effects. As said earlier, PlanMiner-N modifies four steps in the original pipeline of PlanMiner, performing in each one the following tasks:

- **Dataset Extraction.** Based on the same foundations as the original extraction process, PlanMiner-C extracts a set of state transitions with information about the actions and their associated states, and then, but augments them with relevant knowledge about the static relationships of the world (step 5 of Algorithm 15). The results of this process are then formatted and displayed as a dataset.

- **Discovery of new information.** Accounting for the diversity that can exist in the data, this process (Algorithm 15 step 11) is able to fit multiple regression models to a target dataset (PlanMiner fits a single regression model). Based on the original symbolic regression algorithm of PlanMiner, PlanMiner-C makes use of sequential covering techniques to find the different regression models in a collection of data. These models can be used then to feed the datasets, allowing the correct learning of situation-dependent arithmetic and relational expressions, which are crucial to maintain the expressiveness of the planning domains learned by PlanMiner-C.

- **Classification models acquisition.** As exposed earlier, due to the characteristics of the domains that PlanMiner-C tries to learn (domains with conditional actions), learning a single rule for the pre-state and another for the post-state makes the learning process unable to learn situation-dependent effects for the actions. The approach implemented in PlanMiner-C (step 12 of Algorithm 15) attempts to tackle this problem with a two-step learning process. This process first learns a single model with two rules, each explaining the common parts of both the pre-state and post-state of an action. Then, it proceeds to find the differences between the instances of the dataset, grouping the instances in different collections, and tries to adjust specific rules for each of them. This two-step method divides the final rule set obtained into two categories: the general rule model and the specific rules models, and improves the learning capabilities of PlanMiner-C avoiding the generalisation of the classification models and thus, the loss of important information.

- **Planning domain generation.** Finally, last but not least, the meta-states are processed to generate from them the action models in PDDL format. To realize this, PlanMiner extracts from them the preconditions and effects that define the goal action model. Since the restriction that indicates that there is only single a meta-state for the pre-states and the post-states is no longer met, PlanMiner-C (step 13 of Algorithm 15) must implement different a translation method than the one presented in PlanMiner. In this new method,

**Algorithm 15** PlanMiner-C Algorithm overview

**Input**: PT: Set of Plan Traces
**Output**: AM: Set of Action Models

```
1: Initializes stDict as dictionary
2: for all Plan trace pt in PTs do
3:    for all Different action act in the pt do
4:       Extract state transitions st of act in pt
5:       Include static relations of pt in st
6:       stDict[act] ← stDict[act] ∪ st
7:    end for
8: end for
9: for all key act in stDict do
10:   dat ← dataset created using stDict[act]
11:   Infer new knowledge from dat and add it
12:   Fit a set classification models cModel using dat as input
13:   Generate action model am from cModel
14:   AM ← AM ∪ am
15: end for
16: return AM
```

PlanMiner-C contemplates different translation schemes for each different combination of meta-states that can be found. The aforementioned translation schemes are selected after evaluating the rulesets learned in the previous step and seeing which one best fits the information contained in them.

### 5.3.2 Dataset Extraction

First, PlanMiner-C reads the input plan traces and generates a collection of datasets from them. The change in the format of the input plan traces, which allows for the differentiation between dynamic and statics relations, is a significant improvement in its space efficiency, and optimises the way in which the data is organised, but it requires an adaptation process in the original dataset extraction method in order to let the algorithm manage correctly the new plan traces.

The addition of static relations as a separate entity leads PlanMiner-C to define a new encoding process for the state transitions. In PlanMiner-C a state transition of an action is defined as a tuple $(s_1, a, s_2, r)$, where the elements $s_1$, a, $s_2$ denote respectively the pre-state, the action and the post-state. And $r$ is the set of static relations of the world as it appears in a given input plan trace. Listing 5.2 presents an example of the state transition of the action *(RIGHT a1 c16 c17))*, where we can see clearly its different elements.

Action: *(RIGHT a1 c16 c17)*

- **pre-state:** *(= (posX a1) 3) ∧ ((= (posY a1) 1) ∧ (rockIn c12) ∧ (¬ (orien_U a1)) ∧ (orient_D a1) ∧ (¬ (orient_R a1)) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)*

- **post-state:** *(= (posX a1) 3) ∧ ((= (posY a1) 1) ∧ (rockIn c12) ∧ (¬ (orien_U a1)) ∧ (¬ (orient_D a1)) ∧ (orient_R a1) ∧ (¬ (orient_L a1)) ∧ (= (hasGem a1) 0) ∧ (gemIn c7)*
- **Static Relations:** *(=(row c12) 2) ∧ (=(col c12) 2) ∧ (=(row c16) 3) ∧ (=(col c16) 1) ∧ (=(row c17) 3) ∧ (=(col c17) 2) ∧ (conn_U c12 c7) ∧ (conn_D c12 c17) ∧ (conn_R c12 c13) ∧ (conn_L c12 c11) ∧ (conn_U c16 c11) ∧ (conn_D c16 c21) ∧ (conn_R c16 c17) ∧ (conn_L c16 c15) ∧ (conn_U c17 c12) ∧ (conn_D c17 c22) ∧ (conn_L c16 c15) ∧ (conn_R c17 c18) ∧ (conn_L c17 c16)*

Listing 5.2: State transition of the *(RIGHT a1 c16 c17)* action showing static relations.

Once the states transitions have been extracted, PlanMiner-C proceeds to compute the schema form of each of them, following the same procedure as PlanMiner, but now considering the set of static relations. Recall that this process consists of the following steps:

1. For each instantiated parameter in action's header, replace its occurrences in the literals of the pre-state, post-state and static relations by a token representing a variable.
2. Remove irrelevant literals (i.e. those predicates and fluents whose parameters have not been fully substituted by a variable).

Once this action has been performed, PlanMiner-C includes all the remaining predicates in the static relations in both the pre-state and post-state, merging the information of the dynamic and static relations. After this, PlanMiner-C removes *r* from the state transition, resulting in the original schema form (see Listing 5.3) presented in Chapter 3.

Action: *(RIGHT ?arg1 ?arg2 ?arg3)*

- **pre-state:** *(= (posX ?arg1) 3) ∧ ((= (posY ?arg1) 1) ∧ (¬ (orien_U ?arg1)) ∧ (orient_D ?arg1) ∧ (¬ (orient_R ?arg1)) ∧ (¬ (orient_L ?arg1)) ∧ (= (hasGem ?arg1) 0) ∧ (=(row ?arg2) 3) ∧ (=(col ?arg2) 1) ∧ (=(row ?arg3) 3) ∧ (=(col ?arg3) 2) ∧ (conn_R ?arg2 ?arg3) ∧ (conn_L ?arg3 ?arg2)*
- **post-state:** *(= (posX ?arg1) 3) ∧ ((= (posY ?arg1) 1) ∧ (¬ (orien_U ?arg1)) ∧ (¬ (orient_D ?arg1)) ∧ (orient_R ?arg1) ∧ (¬ (orient_L ?arg1)) ∧ (= (hasGem ?arg1) 0) ∧ (=(row ?arg2) 3) ∧ (=(col ?arg2) 1) ∧ (=(row ?arg3) 3) ∧ (=(col ?arg3) 2) ∧ (conn_R ?arg2 ?arg3) ∧ (conn_L ?arg3 ?arg2)*

Listing 5.3: Schema form of a *(RIGHT ?arg1 ?arg2 ?arg3)* action with static relations highlighted in blue, after removing irrelevant literals.

Finally, as exposed in section 2.4.1, the new plan traces encode the dynamic and static relations follow different world assumptions. Dynamic relations follow the OWA, meaning that if a value of an element is not explicitly set, it is considered unknown. Static relations follow the CWA, meaning that those elements not set explicitly as true are interpreted as false . In the process of creating a dataset from the state transitions, this influences how the gaps produced in the columns of a dataset by the lack of information in a state are filled. When facing dynamic relations, if a literal is missing in a state, it is replaced by a "missing value" token, but when dealing with static relations, since they follow the CWA, PlanMiner-C fills the lack of information with `False`.

| $\Delta(row\ ?arg2)$ | $\Delta(col\ ?arg2)$ | $\Delta(row\ ?arg3)$ | $\Delta(col\ ?arg3)$ | $\Delta(posX\ ?arg_1)$ | $\Delta(posY\ ?arg_1)$ |
|---|---|---|---|---|---|
| $3-3=0$ | $4-4=0$ | $3-3=0$ | $5-5=0$ | $3-3=0$ | $4-4=0$ |
| $2-2=0$ | $2-2=0$ | $2-2=0$ | $3-3=0$ | $2-2=0$ | $3-2=1$ |
| $4-4=0$ | $0-0=0$ | $4-4=0$ | $1-1=0$ | $4-4=0$ | $0-0=0$ |
| $2-2=0$ | $0-0=0$ | $2-2=0$ | $1-1=0$ | $2-2=0$ | $0-0=0$ |
| $2-2=0$ | $0-0=0$ | $2-2=0$ | $1-1=0$ | $2-2=0$ | $1-0=1$ |
| $3-3=0$ | $1-1=0$ | $3-3=0$ | $2-2=0$ | $3-3=0$ | $1-1=0$ |
| $3-3=0$ | $2-2=0$ | $3-3=0$ | $3-3=0$ | $3-3=0$ | $3-2=1$ |
| $4-4=0$ | $3-3=0$ | $4-4=0$ | $4-4=0$ | $4-4=0$ | $3-3=0$ |
| $0-0=0$ | $3-3=0$ | $0-0=0$ | $4-4=0$ | $0-0=0$ | $3-3=0$ |

Table 5.2: $\Delta$ values extract

### 5.3.3 Discovery of new features

Once the datasets have been generated, a knowledge discovery process is performed. In chapter 3 it was explained that this process was divided into 3 sub-processes: the calculation of the $\Delta$ value sets associated with each dataset fluent, the fitting of symbolic regression models to discover arithmetic expressions that explain the $\Delta$ value sets, and the creation of relational expressions that relate the different attributes of the datasets. PlanMiner-C modifies only the symbolic regression models learning procedure, keeping the other two unchanged.

PlanMiner-C implements a sequential covering algorithm (see Algorithm 16) to, iteratively, adjusts arithmetic expressions that cover a subset of instances of the goal set of values. Starting from a collection of data extracted from a given dataset, and a set of $\Delta$ values defined as the target, the method proceeds as follows:

1. PlanMiner-C uses the symbolic regression algorithm originally implemented in PlanMiner, trying to find an arithmetic expression that perfectly matches a subset of the goal values.To do so, it explores the search space of the problem until it finds a formula with error 0 for at least 1 instance of the dataset (step 3 of Algorithm 16).

2. Once a candidate formula is found, it includes it in a set of formulas and deletes the examples covered by it from the dataset (steps 4 and 5 of Algorithm 16).

3. After adding an expression to the set of candidate formulas, the whole set is checked for duplicates within the candidate formulas (step 6 of Algorithm 16).

PlanMiner-C repeats this process until all instances of the dataset have been covered or a timeout of 300s has elapsed. In Table 5.2 we can see the goals sets extracted from the example plan trace presented in Table 5.1, as can be seen, $\Delta(posY\ ?arg_1)$) has two different behaviours: A more common one that keeps the fluent unchanged,

**Algorithm 16** Symbolic regression algorithm for conditional actions

**Input**: dat: Dataset
**Output**: f: set of arithmetic expressions

1: Set $f$ as Ø set of arithmetic expressions
2: **while not** ( *dat* totally covered) $\wedge$ **not** timeout **do**
3: *canditateF* $\leftarrow$ fit arithmetic expression for *dat*
4: $f \leftarrow f \cup$ *canditateF*
5: Delete instances covered by *canditateF* in *dat*
6: Update $f$ by removing redundant expressions
7: **end while**
8: **return** $f$

and a scarce one that increases it by 1 point. On the other hand, the rest of the fluents presents the same behaviour every time the action is executed (namely, not changing after executing the action).

The expressions obtained by the symbolic regressor are considered *candidates* until they are returned by the regression algorithm. This is because they are susceptible to being replaced by a better expression learned in later runs of the symbolic regression method. An expression $e_1(x)$ is better than an expression $e_2(x)$ if the following condition holds:

$$e_1(x), e_2(x) \in f \ : \ Supp(e_2(x)) \subset Supp(e_1(x))$$

where $Supp(e_1(x))$ and $Supp(e_2(x))$ are, respectively, the support set of the expressions $e_1(x)$ and $e_2(x)$ and $f$ is the collection of candidate expressions. The support set of an expression is computed by collecting all the examples covered by it.

Summarising, if the expression $Supp(e_1(x))$ from $f$ covers the same examples than $Supp(e_2(x))$ from $f$, it is considered better, and therefore $Supp(e_2(x))$ can be deprecated. The process is realised during the step 6 of Algorithm 16, and its goal is to avoid overfitting the expressions found in early runs of the method, by replacing very specific expressions with more generic ones.

Finally, in order to guide the symbolic regression algorithm, PlanMiner-C implements a new heuristic $h(x, goal)$ to measure the goodness of node $x$ with representing the *goal* set of objective values. This new heuristic is a two-components multi-criteria evaluation function guided by a lexicographer order.

That is to say, the first value of the heuristic guides the execution of the algorithm, indicating which nodes are the best (i.e. which expression being learned best represent the goal values), but in the case of a tie between some nodes, the second component of the evaluation function is taken into account to break the said tie. $h(x, goal)$ is computed as

$$h(x, goal) = (Supp(x(goal)), h'(x, goal))$$

where $Supp(x(goal))$ is the support of the expression represented in the $x$ node and $h'(x, goal)$ its heuristic value as defined in PlanMiner's discovery of new features

**Algorithm 17** PlanMiner-C rule extraction algorithm

**Input**: dat: Dataset
**Output**: RSL: set of classification rules

```
1: Set RSL as Ø ruleset
2: for all class label label in dat do
3:     subDat ← extract all examples of class label in dat
4:     genRule ← fit rule that cover every example of subDat
5:     Erase attributes of sDat modeled in genRule
6:     Fit a classification model espRules using sDat as input
7:     RSL ← RSL cup Combine genRule and espRules
8: end for
9: return RSL
```

step. The heuristic uses an optimisation criterion (max, min), i.e. it tries to maximise the first component and minimise the second. When selecting an expression during the symbolic regression procedure, that the heuristic will choose those functions with the highest support while trying to reduce its error.

The process returns an empty rule set in case one of the following two assumptions occurs: (i) the computation time allotted to the execution (300 seconds) runs out or (ii) the set of expressions obtained after covering all the objectives does not reach a minimum quality threshold. The quality measure set to determine the acceptance of an expressions candidate set is the mean support rate of the expressions. The support rate of an expression is calculated as the percentage of instances covered of the total by it. Given the case that the mean support rate of all expressions is below 25%, the whole candidate set is deprecated. Finally, in the case of obtaining a set of regression models, these are included in the initial dataset as new attributes of the same, in the same way as in PlanMiner. Once the attributes are included, the execution proceeds to the next step of the new information discovery method, following the original execution of the algorithm.

### 5.3.4 Classification models extraction

Similarly to the original PlanMiner algorithm, the entire learning pipeline showed so far is intended to prepare the datasets containing the input information for use in a classification algorithm capable of extracting a model representing the pre-state and post-states associated with the actions of the domain being learned. PlanMiner was designed to extract one single model for each state type, and as previously mentioned, it is not possible to correctly learn conditional actions with that constraint. Therefore, PlanMiner-C implements a new process (presented in Algorithm 17) capable of returning multiple models for each state type.

This new procedure consists of a two-step method that divides the learning process into (i) learning the common elements to each and every example of the dataset and b) learning the specific elements that differentiate those examples.

First, the common elements are defined in a classification model called *general*

*rule* that contains all the information shared between all instances of a given collection of meta-states of the same type (namely pre-state or post-states meta-states), in other words, the general rule contains those elements of the different meta-states of a given kind that are always equal among them. This is achieved by forcing the classification algorithm used in the learning process to output a single classification model for each class label in the dataset.

**General Rule**

**IF**
(posX ?arg1) = (row ?arg2) ∧ (posX ?arg1) = (row ?arg3) ∧
(orien_R ?arg1) = *True* ∧ (orien_L ?arg1) = *False* ∧
(orien_U ?arg1) = *False* ∧ (orien_D ?arg1) = *False* ∧
(conn_R ?arg2 ?arg3) = *True* ∧ (rockIn ?arg3) = *False* ∧
(wallIn ?arg3) = *False*
**THEN** *post-state*

**Specific Rule (I)**

**IF**
(posY ?arg1) = (col ?arg2) ∧ (posX ?arg1) ≠ (row ?arg3) ∧
Δ(posY ?arg1) = 0
**THEN** *post-state*

**Specific Rule (II)**

**IF**
(posY ?arg1) ≠ (col ?arg2) ∧ (posX ?arg1) = (row ?arg3) ∧
Δ(posY ?arg1) = 1
**THEN** *post-state*

Listing 5.4: Classification models of the *(RIGHT ?arg1 ?arg2 ?arg3)* action model post-states.

Secondly, said datasets are updated by deleting the components found in the general rule, this is achieved by selecting the attributes of the elements that form the general rule, and erasing from the datasets the information contained about them (namely, by deleting the columns representing these attributes). The remaining dataset is then used as input to the classification algorithm, with the goal to fit a series of classification models that sort its instances in different subsets. These models are called specific rules and contain the elements that make a meta-state of a given type, different from the other meta-states of the same type. A specific rule, combined with the common rule, creates a full meta-state for a pre-state or post-state. Listing 5.4 shows a set of rules of the post-state meta-states of the *(RIGHT ?arg1 ?arg2 ?arg3)* action. The general rule contains the information common to the post-states, as explained in section 5.2 of this chapter, while the specific rules contain the differences from those indicated in the aforementioned section.

Throughout the development of this method, we determined that the classification algorithm performed better if it only had to fit models for a single class at a

time. Therefore, a process was defined to separate the dataset samples into two data subsets (one for each class label in the problem) and run the method independently on each of them.

For a given dataset, the product of this method is a ruleset containing several rules of each class (pre-state or post-state). These rules are obtained by combining the general rule of a given class with the rules specific to that class. Each of these rules represents a meta-state for a particular behaviour of the action linked to the input dataset. In case that no specific rules are obtained for a given class, it will be represented in the output ruleset with its general rule. Finally, indicate that there is no limit to the number of specific rules that can be learned for a given class.

### 5.3.5 Conditional action models generation

Finally, PlanMiner-C takes the rulesets generated by the classification model extraction process, and creates a set of action models according to the information contained in them. Due to the possibility that more than one meta-state may exist for the pre-states or post-states, PlanMiner-C applies one of a variety PDDL translation schemes to the ruleset. In order to do so, the algorithm studies how the ruleset is defined, applying one PDDL translation scheme or another as needed. This task is easy to perform, as you only have to count the number of rules for the pre-state and post-state contained in the ruleset. As noted in the introduction of this chapter, when designing PlanMiner-C we detect the possibility of learning stochastic action models using it. In this section, we will highlight how this can be done, as there is a configuration of rules that represent these kinds of actions. To recap, saying that this way was not further explored in later stages of PlanMiner-C development, and that it remains as Future Work to improve the learning process performance. The possible situations that PlanMiner-C may encounter depend on the number of pre-states and post-states obtained in the classification models generation step:

- **Case 1:** If the number of rules for the pre-state meta-state is **1** and the number of rules for the post-state meta-state is **1**, the action being learned is considered to be **Non-conditional Deterministic**.
- **Case 2:** If the number of rules for the pre-state meta-state is **1** and the number of rules for the post-state meta-state is **n**, the action being learned is considered to be **Stochastic**.
- **Case 3:** If the number of rules for the pre-state meta-state is **n** and the number of rules for the post-state meta-state is **1**, the action being learned is considered to be **Conditional Deterministic of type I**.
- **Case 4:** If the number of rules for the pre-state meta-state is **n** and the number of rules for the post-state meta-state is **n**, the action being learned is considered to be **Conditional Deterministic of type II**.

These cases, and PlanMiner-C's model generation schemes, are as follows:

**Case 1: Non-conditional Deterministic action.** Starting from a single meta-state for the pre-state $R^1_{pre}$ and a single meta-state for the post-state $R^1_{post}$, PlanMiner-C creates a non-conditional deterministic action model (i.e. an action that always behave the same). To do so, it calculates the preconditions of the action by taking the pre-state meta-state, and the effects by calculating the difference $\Delta(R^1_{pre}, R^1_{post})$ between the pre-state and post-state meta-states (i.e. what things are missing or surplus in the pre-state to obtain the post-state). This is the base case of the learning approach of PlanMiner.

| | |
|---|---|
| **Pre-state rules** | $R^1_{pre}$ |
| **Post-state rules** | $R^1_{post}$ |
| **Preconditions** | $R^1_{pre}$ |
| **Effects** | $\Delta(R^1_{pre}, R^1_{post}) = (R^1_{pre} - R^1_{post}) \cup (R^1_{post} - R^1_{pre})$ |

**Case 2: Stochastic STRIPS action.** In the case of finding a ruleset with a single meta-state $R^1_{pre}$ for the pre-state but a collection of meta-states $R^1_{post}, R^2_{post}, \dots R^n_{post}$ for the post-state, PlanMiner-C considers it to be a non-deterministic action model. Regardless of the context of the world, these action models have random behaviour, making some changes in the world or others in an uncontrolled way. In order to obtain a stochastic action model, PlanMiner-C calculates the preconditions of the action by taking the meta-state of the pre-state, and the different sets of effects by calculating the difference $\Delta(R^1_{pre}, R^i_{post})$ between the meta-states of the post-states and those of the pre-state (i.e. what things are missing or surplus in the pre-state to obtain the post-state). Each set of effects is assigned a probability of running equal to the support of the post-state meta-state related to it.

| | |
|---|---|
| **Pre-state rules** | $R^1_{pre}$ |
| **Post-state rules** | $R^1_{post}, R^2_{post}, \dots R^n_{post}$ |
| **Preconditions** | $R^1_{pre}$ |
| **Effects** | • $\#R^1_{post}$: $\Delta(R^1_{pre}, R^1_{post}) = (R^1_{pre} - R^1_{post}) \cup (R^1_{post} - R^1_{pre})$<br>• $\#R^2_{post}$: $\Delta(R^1_{pre}, R^2_{post}) = (R^1_{pre} - R^2_{post}) \cup (R^2_{post} - R^1_{pre})$<br>...<br>• $\#R^n_{post}$: $\Delta(R^1_{pre}, R^n_{post}) = (R^1_{pre} - R^n_{post}) \cup (R^n_{post} - R^1_{pre})$ |

**Case 3: Conditional actions (I)** If during the meta-state fitting process, the classifiers return a ruleset with several models $R^1_{pre}, R^2_{pre}, \dots R^n_{pre}$ for the pre-state, but only one post-state $R^1_{post}$, PlanMiner-C faces a situation where a conditional action models must be extracted. To do this, it calculates the preconditions of the action by taking the meta-states of the pre-states and extracting their common elements. For the effects, the algorithm it calculates the difference $\Delta(R^i_{pre}, R^1_{post})$ between the meta-state of the post-state and each meta-state of the pre-states set (i.e. what things are missing or surplus in each pre-state to obtain the post-state). Each effects block is assigned a trigger condition calculated from the specific elements of the pre-state model used to calculate them. These specific elements are those

elements that differentiate the given pre-state meta-state from the other pre-state meta-states.

| | |
|---|---|
| **Pre-state rules** | $R^1_{pre}, R^2_{pre}, \dots R^n_{pre}$ |
| **Post-state rules** | $R^1_{post}$ |
| **Preconditions** | $R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}$ |
| **Effects** | • $R^1_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^1_{pre}, R^1_{post}) = (R^1_{pre} - R^1_{post}) \cup (R^1_{post} - R^1_{pre})$<br>• $R^2_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^2_{pre}, R^1_{post}) = (R^2_{pre} - R^1_{post}) \cup (R^1_{post} - R^2_{pre})$<br>...<br>• $R^n_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^n_{pre}, R^1_{post}) = (R^n_{pre} - R^1_{post}) \cup (R^1_{post} - R^n_{pre})$ |

**Case 4: Conditional actions (II)** The last situation contemplated by PlanMiner-C is the case where several meta-states of pre-states $R^1_{pre}$, $R^2_{pre}$, ... $R^n_{pre}$ and post-states $R^1_{post}$, $R^2_{post}$, ... $R^n_{post}$ exist in the same ruleset. In this situation, PlanMiner-C determines that the action to be learned is a conditional action model. Given that there is an unknown number of meta-states of both types, before performing the translation process, PlanMiner-C must match the pre-state and post-state models in order to link them and correctly extract the effects of the action model. Due to the processes that are implemented along the learning pipeline, information about the precedence of states is lost, and, in order to continue the process of generating the action models, this information must be retrieved, by matching the meta-states of the pre-states and post-states with each other. PlanMiner-C performs this matching by traversing the plan traces and looking for correspondences between pre-state and post-states. The correspondence of a pair meta-states is calculated by finding the number of state transitions of the plan traces are represented by them (i.e. if the pre-state meta-state fits the pre-state and post-state meta-state fits the post-state). Those pre-state and post-states with the highest correspondence rate are matched.

Once this process is done, PlanMiner-C calculates the preconditions $R^1_{pre} \cap R^2_{pre} \cap ... \cap R^n_{pre}$ of the action by taking the meta-states of the pre-state and extracting their common elements. For the effects, it first calculates all elements common to the post-states. With both the common elements of the pre-states and the post-states meta-states, PlanMiner-C proceeds to calculate the addition and subtraction lists $\Delta(R^1_{pre} \cap R^2_{pre} \cap ... \cap R^n_{pre}, R^1_{post} \cap R^2_{post} \cap ... \cap R^n_{post})$ of the common effects of the action model. Once the common effects have been generated, the algorithm calculates the difference $\Delta(R^i_{pre}, R^i_{post})$ between each meta-state pair of a post-state and a pre-state (i.e. what things are missing or surplus in each pre-state to obtain the post-state). Each effects block is assigned a trigger condition calculated from the specific elements of the pre-state model used to calculate them. These specific elements are those elements that differentiate the given pre-state meta-state from the other pre-state meta-states.

| | |
|---|---|
| **Pre-state rules** | $R^1_{pre}, R^2_{pre}, \dots R^n_{pre}$ |
| **Post-state rules** | $R^1_{post}, R^2_{post}, \dots R^n_{post}$ |
| **Preconditions** | $R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}$ |
| **Effects** | $\Delta(R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}, R^1_{post} \cap R^2_{post} \cap \dots \cap R^n_{post})$<br>• $R^1_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^1_{pre}, R^1_{post}) = (R^1_{pre} - R^1_{post}) \cup (R^1_{post} - R^1_{pre})$<br>• $R^2_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^2_{pre}, R^2_{post}) = (R^2_{pre} - R^2_{post}) \cup (R^2_{post} - R^2_{pre})$<br>...<br>• $R^n_{pre} - (R^1_{pre} \cap R^2_{pre} \cap \dots \cap R^n_{pre}) \rightarrow \Delta(R^n_{pre}, R^n_{post}) = (R^n_{pre} - R^n_{post}) \cup (R^n_{post} - R^n_{pre})$ |

## 5.4 Experimentation and Results

This section is dedicated to showing the experimental process that PlanMiner-C has undergone to demonstrate its viability. The aim of this experimentation is to demonstrate PlanMiner-C's ability to learn planning domains from data extracted from the GVG-AI videogame environment. In this experimentation, it has been decided to use only the NSLV as a classification algorithm in the learning process. NSLV has been selected among the classification algorithms presented along with this document as it is the one that has shown the best performance throughout the experimentation carried out in previous chapters. The rest of the section is divided into two blocks: the first one will present the experimental setup, indicating how the input data for PlanMiner-C has been extracted from GVG-AI, the quality metric applied as a measure and the games used in the experimental setup; the second block contains the results of and the evaluation of the experiment, describing in detail the domains learnt throughout the experimental process.

### 5.4.1 Experimental setup

In order to test the capabilities of PlanMiner-C, the experimental process presented in these lines measures the scenario solving capacity in video games of the GVG-AI environment. Due to the characteristics of the experimentation, when measuring the quality of the domains learned by PlanMiner-C we cannot apply the same methodology and metrics as those applied to PlanMiner and PlanMiner-N in the previous chapters. This is due to two limiting factors caused by using as source of the input data the GVG-AI environment. The first of these factors is the lack of a reference domain to compare the output of PlanMiner-C. The lack of a correct domain predefined by a human operator impedes the calculation of metrics used so far in this document (i.e. Precision, Recall and F-Score). The other factor that affects the experimental process here is heir to the first factor, and that is, that there are no plans to solve the scenarios of the GVG-AI videogames, which makes the use of standard validation tools (i.e. VAL) impossible. As a solution to this problem, in this experimentation, we propose to compare the success of the plans generated with directly in the GVG-AI environment.

### Evaluation process

The experimental evaluation process defined to compare the learned domains with the GVG-AI environment is implemented with the following philosophy:

1. Use the domains learned to obtain plans to solve a scenario.
2. Introduce the plans into a game agent and execute them step by step.
3. Observe if the agent solves the scenario successfully.

This process is implemented in a 2-step domain validation scheme: that first, makes a learning step, and, second, implements a validation method. On the one hand, to perform the domain learning process, a GVG-AI video game is taken and 15 scenarios are generated for it. Then, 1 execution trace is obtained for each scenario by observing an agent explore them. Finally, these traces are used as input for PlanMiner-C, which learns a planning domain. In order, to obtain the aforementioned plan traces, an agent has been implemented which tries to solve the proposed scenarios. This method records the executions of the agents, storing the list of actions carried by it, as well as a collection of observations of the game world. These observations are taken each time an action is executed by the agent. The agent implemented for experimentation has 3 different modes of operation:

- **Random walk.** At each tick of GVG-AI's execution, the agent returns a random action.
- **MCTS guided.** The agent's AI is governed by a Monte-Carlo search algorithm [BPW+12] that evaluates the current game state and chooses the best action at each tick.
- **Teleoperated.** The agent is controlled by a human operator who guides its movements and actions at each tick.

Each operation mode implemented has been used during the experimental process to obtain different plan traces, with the aim to obtain knowledge about a wide variety of standard behaviours. Additionally, in order to test whether the modifications made in PlanMiner-C make it retain its resilience to incompleteness, the plan traces obtained are stripped of a percentage of randomly selected elements, as seen in the experimental process of Chapter 3.

On the other hand, the domain validation process takes the domain learned using PlanMiner-C and obtains a plan for each of the game scenarios used in the previous step. These plans obtained solve the planning problems that represent the scenarios. Namely, they solve the scenarios. To solve the planning domains, the FF-Metric [Hof03] planning engine was used. Once the problems for each scenario have been solved, they are translated into a GVGAI interpretable format for a game agent to execute them step by step. If at the end of the execution the scenario has been solved correctly by the agent, the domain success is accounted.

The metric used to measure the quality of the domains is said domain success. For each scenario, this metric measures the percentage of correctly solved scenarios

for the domain being evaluated. In order to prevent elements such as randomness from affecting the results, five sets of game traces are generated for each video game. These traces are obtained by trying to solve each game scenario five times using the agent in a given operation mode. The domain success score is the average success score of the five subsets of traces.

### Translation Procedure of GVG-AI observations to PDDL

5.4 Throughout the experimental process, it is necessary to translate the information from the observations into PDDL. This is done by means of a set of ontologies designed ad-hoc for each video game, which aim to map the knowledge from one format to another. GVG-AI observations are formed by a collection of state variables that define the world and must be traversed one by one to put them into the appropriate format. Depending on the internal structure that each game follows to represent the knowledge of its observations, the translation process should be approached differently. The ontologies present a set of guidelines indicating how each of the state variables should be translated into PDDL format. This translation process must be performed to correctly encode the input plan traces and to generate the problems that represent the scenarios used in the experimentation. The translation process for each of these tasks is different, as each task receives a different input and output:

- **Plan traces** are created by taking the raw GVG-AI traces generated by the agent (recapitulating a list of actions performed plus a list of observations taken from GVG-AI), passing directly to PDDL format the actions of the trace and comparing the information contained in the observations with a given ontology.

- **Planning problems** are defined by taking the first observation of the game (i.e. the one defined in the VGDL file that defines the scenario) and collecting all the objects that appear in that initial state to define them in their own block. Finally, the objective is defined in a generic way depending on the videogame that is being addressed.

### GVG-AI games used

The videogames selected to study the capabilities of PlanMiner-C learning planning domains with conditional actions and numerical information are Bait, Zelda and Boulderdash. Tables 5.3, 5.4 and 5.5 contain information about these games and its ontologies. These tables present a small descriptive picture of the game and a description of the game, and also describe in detail said PDDL ontology. For the sake of readability, the ontology is divided in several blocks. These blocks describe i) the world's objects types, ii) the dynamic relations of the videogame, and iii) its static relations.

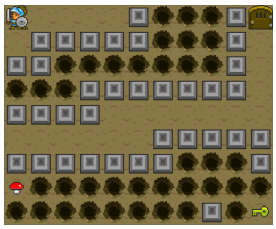
| | |
|---|---|
| |  |
| **Description and goals** | The objective of this game is to reach the goal, collecting a key first. The player can push boxes around to open paths. There are holes in the ground that kill the player, but they can be filled with boxes (and both hole and box disappear). The player can also collect mushrooms that give points. Directional actions move the avatar through the world, while the use action has no function. |
| | **Ontology** |
| | *Objects* |
| *Avatar* | Represents the player in the game world |
| *Cell* | Represents a location on the map |
| | *Dynamic Relations* |
| $(posX\ A_1)$ | Indicates the row in which the player is located in the game world |
| $(posY\ A_1)$ | Indicates the column in which the player is located in the game world |
| $(orient_X\ A_1)$ | Indicates the direction in which the avatar $A_1$ is facing (Up, Down, Left or Right). There are 4 different predicates, one for each orientation |
| $(hasKey\ A_1)$ | Indicates whether the avatar $A_1$ has a key |
| $(keyIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a key |
| $(wallIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a wall |
| $(holeIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a hole |
| | *Static Relations* |
| $(row\ C_1)$ | Indicates the row of the grid that the Cell $C_1$ occupies |
| $(column\ C_1)$ | Indicates the column of the grid that the Cell $C_1$ occupies |
| $(conn_X\ C_1\ C_2)$ | Indicates whether Cell $C_1$ and Cell $C_2$ are connected and in which direction the connection is made (Up, Down, Left or Right). There are 4 distinct predicates, one for each orientation |
| $(exitIn\ C_1)$ | Indicates if there is an output in Cell $C_1$ |

Table 5.3: Bait description and PDDL ontology

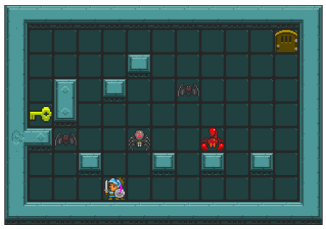
| | |
|---|---|
| |  |
| **Description and goals** | The avatar must find a key in a maze to open a door and exit. The player is also equipped with a sword to kill enemies existing in the maze. The player wins if it exits the maze, and loses if it is hit by an enemy. 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it. $-1$ point if the avatar is killed. The directional actions move the avatar through the world, while the use action is for using the sword. |
| | **Ontology** |
| | *Objects* |
| *Avatar* | Represents the player in the game world |
| *Monster* | Represents the monsters in the game world |
| *Cell* | Represents a location on the map |
| | *Dynamic Relations* |
| $(posX\ A_1)$ | Indicates the row in which the player or monster is located in the game world |
| $(posY\ A_1)$ | Indicates the column in which the player or monster is located in the game world |
| $(orient_X\ A_1)$ | Indicates the direction in which the avatar $A_1$ is facing (Up, Down, Left or Right). There are 4 different predicates, one for each orientation |
| $(hasKey\ A_1)$ | Indicates whether the avatar $A_1$ has a key |
| $(keyIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a key |
| | *Static Relations* |
| $(row\ C_1)$ | Indicates the row of the grid that the Cell $C_1$ occupies |
| $(column\ C_1)$ | Indicates the column of the grid that the Cell $C_1$ occupies |
| $(conn_X\ C_1\ C_2)$ | Indicates whether Cell $C_1$ and Cell $C_2$ are connected and in which direction the connection is made (Up, Down, Left or Right). There are 4 distinct predicates, one for each orientation |
| $(wallIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a wall |
| $(exitIn\ C_1)$ | Indicates if there is an output in Cell $C_1$ |

Table 5.4: Zelda description and PDDL ontology

| | |
|---|---|
| |  |
| **Description and goals** | The avatar must dig in a cave to find at least 10 diamonds, with the aid of a shovel, before exiting through a door. Some heavy rocks may fall while digging, killing the player if it is hit from above. There are enemies in the cave that might kill the player, but if two different enemies collide, a new diamond is spawned. 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned. $-1$ point is given if the avatar is killed by a rock or an enemy. The directional actions move the avatar through the world, while the use action is for using the shovel. |
| | **Ontology** |
| | *Objects* |
| *Avatar* | Represents the player in the game world |
| *Monster* | Represents the monsters in the game world |
| *Cell* | Represents a location on the map |
| | *Dynamic Relations* |
| $(posX\ A_1)$ | Indicates the row in which the player or monster is located in the game world |
| $(posY\ A_1)$ | Indicates the column in which the player or monster is located in the game world |
| $(orient_X\ A_1)$ | Indicates the direction in which the avatar $A_1$ is facing (Up, Down, Left or Right). There are 4 different predicates, one for each orientation |
| $(hasGem\ A_1)$ | Indicates the number of gems the avatar has taken $A_1$ |
| $(keyIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a key |
| | *Static Relations* |
| $(row\ C_1)$ | Indicates the row of the grid that the Cell $C_1$ occupies |
| $(column\ C_1)$ | Indicates the column of the grid that the Cell $C_1$ occupies |
| $(conn_X\ C_1\ C_2)$ | Indicates whether Cell $C_1$ and Cell $C_2$ are connected and in which direction the connection is made (Up, Down, Left or Right). There are 4 distinct predicates, one for each orientation |
| $(wallIn\ C_1)$ | Indicates whether in a cell $C_1$ there is a wall |
| $(exitIn\ C_1)$ | Indicates if there is an output in Cell $C_1$ |

Table 5.5: Boulderdash description and PDDL ontology

| | Agent Mode | Incompleteness | | | |
|---|---|---|---|---|---|
| | | 0% | 10% | 50% | 90% |
| **Bait** | Random walk | 98% | 96% | 89% | 80% |
| | MCTS guided | 100% | 100% | 95% | 88% |
| | Teleoperated | 100% | 100% | 99% | 97% |
| | Mixed | 100% | 100% | 100% | 100% |
| **Zelda** | Random walk | 100% | 73% | 0% | 0% |
| | MCTS guided | 100% | 82% | 0% | 0% |
| | Teleoperated | 100% | 92% | 86% | 81% |
| | Mixed | 100% | 98% | 91% | 85% |
| **Boulderdash** | Random walk | 89% | 81% | 72% | 63% |
| | MCTS guided | 94% | 86% | 78% | 70% |
| | Teleoperated | 100% | 99% | 94% | 89% |
| | Mixed | 100% | 100% | 97% | 91% |

Table 5.6: PlanMiner-C experimental results

### 5.4.2 Results and discussion

This section is subdivided into 3 blocks, one for each videogame used in the experimental process. In each block, a table with the results and a detailed description of them is included. The experiments are separated given the operation mode used by the agent that generates its input plan traces. Additionally, we include an extra experiment realised by gathering random plan traces of the later experiments (these experiments are tagged as "mixed operation mode").

- **Bait** Using data obtained with the agent in "Random Walk" mode, PlanMiner-C has problems in solving some scenarios, even with complete data. These problems are compounded by tightening the experimental conditions (i.e. increasing the percentage of missing data), failing to solve at least 3 scenarios out of the 15 proposed at 90% incompleteness. The other three operational methods show no problems in learning planning domains that solve the proposed scenarios, even under certain levels of incompleteness. At 50% incompleteness, the traces obtained with "MCTS guided" and "Teleoperated" prevent PlanMiner-C from obtaining a planning domain that solves certain scenarios, and as the percentage of elements removed from the traces increases, the number of unsolvable scenarios increases. On the other hand, the domains obtained using the "Mixed" mode of operation do not present problems in solving the proposed scenarios, regardless of the percentage of incompleteness of the input data.

- **Zelda** At the beginning of the experimentation, the data obtained with the "Random Walk" mode allows PlanMiner-C to generate domains that solve all the proposed scenarios, but the inclusion of incompleteness in the input data deteriorates the quality of the data very quickly. With 50 per cent missing data, the input data gaps lead to critical failures in the planning domains that

impede the scenarios from being solved. The traces obtained by the "MCTS guided" mode show a similar, but less severe, behaviour, losing success rate in a much smoother way as the experimental conditions are hardened. On the other hand, the domains obtained from the data generated using the "Teleoperated" and "Mixed" modes maintain a certain resilience to incompleteness (especially those generated with "Mixed"). At 10 per cent incompleteness, the domains show a performance of 92% and 98 per cent, which, although gradually degrading, does not fall below an 80% success rate in the most complex experiments.

- **Boulderdash** The initial results of PlanMiner-C using input data from the "Random Walk" and "MCTS guided" modes are not perfect even using data without incompleteness. As seen with the Zelda video game, PlanMiner-C results deteriorate severely when incompleteness is included in these data, but this time without reaching critical levels that prevent scenario resolution. The data generated by the "Teleoperated" and "Mixed" modes mean that the planning domains obtained with PlanMiner-C present more stable results, maintaining around a 100% success rate even with some levels of incompleteness. In the most complex experiments, the performance of PlanMiner-C with such data is above 90 points (solving 13/14 scenarios of the proposed ones).

### Discussion

Looking closely at the results, it is clear that the data obtained using the "Random Walk" mode of operation is of lower quality than that generated by the other modes of operation, causing PlanMiner-C to fail to correctly learn planning domains and to be more affected by incompleteness in more complex experiments. The random movements performed by "Random Walk" can explore options that would not normally be explored by chance alone, but are not certain to probe the options needed to correctly solve the scenarios. This makes the data obtained by this mode of operation more susceptible to incompleteness, as by removing elements from the traces, some vital (and already sparse) information may be lost.

The "MCTS guided" and "Teleoperated" modes of operation do not present this problem, since their movements are aimed at achieving the objective set by a given video game. In the case of the data obtained by the "Teleoperated" mode, the results of PlanMiner-C are qualitatively superior to those of "MCTS guided" due to the presence of a human agent defining the strategy of the game agent to solve the game. This human agent is able to design higher quality designs than those obtained by the MTCS algorithm, which positively affects the PlanMiner-C results. Unfortunately, both modes of operation have a deficiency that does not exist in the "Random Walk" mode, and that is that they do not explore some moves because they are "useless" for solving the scenarios. As previously mentioned, Random Walk could perform such moves simply by luck.

The use of plan traces that combine all implement agent behaviours in "Mixed" mode is the one that has given the best results. Combining the existence of se-

quences of actions aimed at accomplish the scenario's objectives (from the "MCTS guided" and "Teleoperated" modes) with a wider variety of behaviours in the traces (from the "Random Walk" mode) yields the best possible results. These results not only have the best average performance but are also relatively resistant to the inclusion of incompleteness in the input data. Appendix D presents a collection of planning domains learned using the "Mixed" mode of operation.

## 5.5 Conclusions

This chapter has presented PlanMiner-C, an AML technique that rebuilds the PlanMiner algorithm with the objective of learning planning domains with conditional effects. PlanMiner-C has been reworked several methods of the original learning pipeline with the aim of detecting different patterns of behaviours in the input data in order to code them correctly in conditional execution structures. The redesigned methods i) implement a new procedure for handling the input data, ii) change the feature discovery strategy, iii) alter the way in which the classification algorithms are used in the pipeline and iv) reformulate the process of generating planning domains. i) The new procedure was designed to accommodate the algorithm to the new plan trace format used to encode large volumes of input data. ii) The modified feature discovery process allows for the learning of multiple patterns that represent the varied behaviours that conditional effects can have. iii) The redesigned classification model generation method optimises the behaviour of PlanMiner to correctly handle meta-states represented by multiple rules. And, finally, iv) the redesigned PDDL generation strategy is able to handle those meta-states defined by multiple rules and obtain a valid action model. PlanMiner-C has been validated using data extracted from the GVG-AI environment, checking if it is able to generate planning domains able to solve its scenarios. In the proposed experimentation, PlanMiner-C has demonstrated its ability to learn planning domains with conditional effects that can solve videogame scenarios of the GVG-AI competition.

# Chapter 6

# Final Remarks

In the field of Knowledge Engineering, methods that apply knowledge acquisition techniques to automatic planning have gained relevance in recent years due to the complexity of defining a planning domain capable of modelling convoluted processes. In the initial chapters of this document, we provided a literature review of the Action Model Learning research area, showing its benefits and relevance within the broader discipline of AI while highlighting its problems and shortcomings.

Throughout this dissertation, we have presented a series of innovative contributions to the knowledge area of AML. These contributions are designed with the dual aim of (i) being able to increase the expressiveness of the domains that can be elicited with AML techniques, while (ii) creating an approach that is resilient to uncertainty in the input data. These contributions have achieved the proposed objectives by basing their learning process on the use of multiple well-known data mining techniques, with a special emphasis on XAI techniques. Our contributions have been tested with state-of-the-art models of the research areas we are concerned with, and, as far as possible, have been compared against cutting-edge AML algorithms. During these empirical studies, the contributions presented in this document have demonstrated a superb performance in the tasks at hand.

In Chapter 1, 3 objectives were proposed for the thesis, from each of which a significant contribution has been made. These three contributions have been presented throughout this manuscript and are PlanMiner and two sub-versions of it: PlanMiner-N and PlanMiner-C. The achievements of each of them (as well as a brief summary of each contribution) are:

**PlanMiner.** The PlanMiner methodology (presented in Chapter 3) is our primary contribution to the field of AML. It is implemented as a pipeline of machine learning techniques and is able to learn STRIPS planning domains that use preconditions and effects with arithmetic and relational expressions, even under conditions of high input data sparsity. PlanMiner's learning process is based on obtaining a set of classification models (which represent the pre-states and post-states of the actions.) from which the preconditions and effects can be extracted. To do so, the algorithm starts from a set of plan traces that are used (after being previously formatted as

a dataset) as input to a symbolic regressor that obtains new information usable to enrich the input data, then, by using a classification algorithm, the aforementioned classification models are extracted. The addition of new information to the input data allows the learning of arithmetic-relational expressions in the aforementioned classification models. PlanMiner was tested with domains extracted from the International Planning Competition, which are used in the proposed experimentation as a template to generate input data with which to attempt replication. During experimentation, PlanMiner demonstrates that it is able to learn both pure STRIPS domains and planning domains with numerical information. Furthermore, during the experimentation, PlanMiner shows a high tolerance to incompleteness, generating planning domains with high quality even under extreme input sparsity situations.

**PlanMiner-N.** In order to improve PlanMiner's resistance to uncertainty, the PlanMiner-N (detailed in Chapter 3 of this document) solution was designed as propossed as a contribution of the state-of-the-art of the Action Model Learning field. This solution takes the original PlanMiner pipeline and includes in it a number of extra methods to deal with noisy data. The task of these methods is to (i) pre-process the input data to detect and remove noise from it, and (ii) refine the classification models obtained during the learning process. These methods are designed to complement the other components of the PlanMiner learning process without altering them. To do their job properly, these new methods are implemented on top of a number of unsupervised learning techniques (e.g. cluster analysis) and statically methods. The performance of PlanMiner-N was tested in an experimental process similar to that used in PlanMiner, using input data extracted from the same planning domains, as well as compared to the same benchmark algorithms. From the experimental study, it can be seen that the changes made to the PlanMiner methodology have been effective in dealing with noisy information, as PlanMiner-N shows a significant performance improvement not only with respect to the original algorithm, but also with respect to the reference algorithms, and is able to learn valid planning domains even in the most complex experimental scenarios.

**PlanMiner-C.** The latest contribution presented is the PlanMiner-C algorithm (see Chapter 5). This solution is the product of an alternative line of work to the one that gave birth to PlanMiner-N, and, where PlanMiner-N focused on improving PlanMiner's capabilities in the face of uncertainty from low-quality input data, PlanMiner-C aimed to improve the expressiveness of the learned action models. With this end in mind, PlanMiner-C was designed to learn STRIPS and numerical planning domains with conditional and stochastic effects, modifying major components of PlanMiner to fit the new learning scenarios. These modifications involve updating the input data handling component, designing new discovery of features and classification model generation strategies, and implementing a new PDDL domain generation process. To test the capabilities of PlanMiner-C, a framework for the development of General Game Playing techniques called GVG-AI was used to obtain information about agent executions in a set of simulated environ-

ments. Then with this information PlanMiner-C tried to replicate the behaviour of the agents in a planning domain able to work correctly in the said environments. The results indicate that PlanMiner-N is able to learn planning domains capable of solving scenarios for the video games used in the experimentation, even under certain levels of incompleteness.

## 6.1 Future lines of research

As mentioned at the beginning of this document, the development of this doctoral thesis shows the first steps of a more ambitious work: to create an AML algorithm that can be implemented in the real world. To achieve this long-term objective, 3 sub-goals were defined at the beginning of the doctoral work, and, although the algorithms of the PlanMiner family have achieved them, there is still much room for improvement to reach the long term aforementioned objective. To improve the proposed contributions presented above, we envisage the following lines of future research:

**Line 1: Learning of planning domains with stochastic behaviours.** In Chapter 5, we briefly presented the possibility that PlanMiner-C can learn planning domains with stochastic behaviours. Since we have already implemented the procedures for this algorithm proposed to be able to learn these types of domains, we consider that it would be fruitful to propose an experimental process to measure the capabilities of PlanMiner-C in this topic. Depending on the results of this experimentation, we would study the possibility of moving forward in this line of development, defining a fine-tuning strategy to accommodate the learning process of the algorithm, or, if not, we would detect the weak points of our solution in order to define a strategy to tackle them.

**Line 2: Extension of the rule learning strategy of PlanMiner-C.** Throughout the experimentation presented in this document (especially in the experimentation of Chapters 3 and 4) it has been shown that the selection of the classification algorithm used within the learning pipeline greatly influences (for better and for worse) the final results of the solution. Therefore, we consider that it would be interesting to explore the a way to mix the method proposed in Chapter 5 to obtain classification rules and NSLV (the classification algorithm which performed the best in Chapters' 3 and 4 experimental processes), in order to expand their functionalities and obtain a new classifier designed entirely to fit PlanMiner's necessities. Throughout the experimentation carried out in this thesis, we have gathered a lot of information about the successes and failures of the classification algorithms used with the different versions of PlanMiner, that gave us a great insight to combine their virtues in a proprietary rule learning algorithm for PlanMiner in order to improve its overall performance and prepare it to more complex experimental scenarios.

**Line 3: Combination of the methods proposed in this document.** Each of the three approaches presented in this manuscript follows its own line of work, experimenting with its own data and under its own circumstances, albeit PlanMiner-N and PlanMiner-C are derived from PlanMiner. Because of this, the result of this thesis are 3 different algorithms, which work under different assumptions. Creating a line of work that allows converging the research carried out so far in a single algorithm is a priority for future research lines. The result of the merge of the three approaches would be a version of the PlanMiner that combines the properties of the different contributions, i.e. a solution capable of learning planning domains with arithmetic-relational expressions and conditional and stochastic effects and with resistance to both incompleteness and noise conditions.

**Line 4: Learning planning domains with temporal information.** Finally, in order to further increase the complexity of the planning domains obtained using the PlanMiner family of algorithms, the idea of studying the feasibility of learning temporal planning domains is proposed as new research line. In Chapter 2 the ability of PDDL 2.2 to manage temporal information was mentioned, but since the scope of application of PlanMiner lies outside this area of knowledge, it was not further explored. We consider that, in the future, it would be interesting to explore it and trying to learn temporal planning domains. A temporal planning domain is a domain formed by actions that take a given time to complete (called durative actions), in contrast to the actions learned in the domains proposed in this manuscript, which are considered to be executed instantaneously. This way of conceiving actions is a major paradigm shift for PlanMiner, but the development of a tool capable of obtaining this kind of planning domains would, in the long run, put the algorithm on the threshold of learning from real-world data.

## 6.2 Publications associated with the thesis

Throughout the development of the thesis, a series of scientific works have been developed, the full list of publications associated with the thesis is presented below.

### 6.2.1 Publications in international journals

- José Á Segura-Muros, Raúl Pérez, Juan Fernández-Olivares. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 51, 1-17, 2021.

- José Á Segura-Muros, Juan Fernández-Olivares, and Raúl Pérez. Learning Numerical Action Models from Noisy Input Data. *Preprint (arXiv:2111.04997, KBS:KNOSYS-D-21-04341)*, 2021.

### 6.2.2 Publications in national and international conferences

- José Á Segura-Muros, Raúl Pérez, Juan Fernández-Olivares. Learning HTN Domains using Process Mining and Data Mining techniques. In *Workshop on Generalized Planning*, ICAPS-17), 2017.
- José Ángel Segura-Muros, Juan Fernández-Olivares. Integration of an automated hierarchical task planner in ros using behaviour trees. In *6th International Conference on Space Mission Challenges for Information Technology*, SMC-IT 2017, 2017.
- José Á Segura-Muros, Raúl Pérez, Juan Fernández-Olivares. Using Inductive Rule Learning Techniques to Learn Planning Domains. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, IPMU 2018, 2018.
- José Á Segura-Muros, Raúl Pérez, Juan Fernández-Olivares. Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques. In *Knowledge Engineering for Planning and Scheduling*, KEPS-18, 2018.
- José Á Segura-Muros, Raúl Pérez, Juan Fernández-Olivares. Learning Planning Action Models with Numerical Information and Logic Relationships Using Classification Techniques. In *Conference of the Spanish Association for Artificial Intelligence*, CAEPIA 2018, 2018.

# Appendix A

# Domains used in the experimental setups

## A.1 STRIPS domains

### A.1.1 BlocksWorld

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
         (ontable ?x)
         (clear ?x)
         (handempty)
         (holding ?x)
         )

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

  (:action put-down
       :parameters (?x)
       :precondition (holding ?x)
       :effect
       (and (not (holding ?x))
       (clear ?x)
       (handempty)
       (ontable ?x)))
  (:action stack
       :parameters (?x ?y)
       :precondition (and (holding ?x) (clear ?y))
       :effect
```

```
      (and (not (holding ?x))
      (not (clear ?y))
      (clear ?x)
      (handempty)
      (on ?x ?y)))
  (:action unstack
      :parameters (?x ?y)
      :precondition (and (on ?x ?y) (clear ?x) (handempty))
      :effect
      (and (holding ?x)
      (clear ?y)
      (not (clear ?x))
      (not (handempty))
      (not (on ?x ?y)))))
```

Listing A.1: BlocksWorld STRIPS planning domain

### A.1.2 Depots

```
(define (domain Depot)
(:requirements :typing)
(:types place locatable - object
  depot distributor - place
        truck hoist surface - locatable
        pallet crate - surface)

(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface))

(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (at ?x ?y))
:effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (available ?x) (at ?y ?p) (on ?y ?z) (
    clear ?y))
:effect (and (not (at ?y ?p)) (lifting ?x ?y) (not (clear ?y)) (not (
    available ?x))
             (clear ?z) (not (on ?y ?z))))

(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (clear ?z) (lifting ?x ?y))
:effect (and (available ?x) (not (lifting ?x ?y)) (at ?y ?p) (not (
    clear ?z)) (clear ?y)
    (on ?y ?z)))

(:action Load
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (lifting ?x ?y))
```

```
:effect (and (not (lifting ?x ?y)) (in ?y ?z) (available ?x)))

(:action Unload
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (available ?x) (in ?y ?z))
:effect (and (not (in ?y ?z)) (not (available ?x)) (lifting ?x ?y)))
```

Listing A.2: Depots STRIPS planning domain

### A.1.3 DriverLog

```
(define (domain driverlog)
  (:requirements :typing)
  (:types         location locatable - object
    driver truck obj - locatable

  )
  (:predicates
    (at ?obj - locatable ?loc - location)
    (in ?obj1 - obj ?obj - truck)
    (driving ?d - driver ?v - truck)
    (link ?x ?y - location) (path ?x ?y - location)
    (empty ?v - truck)
)


(:action LOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action UNLOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (in ?obj ?truck))
  :effect
   (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action BOARD-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck))
  :effect
   (and (not (at ?driver ?loc)) (driving ?driver ?truck) (not (empty ?
       truck))))
```

```
(:action DISEMBARK-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (driving ?driver ?truck))
  :effect
   (and (not (driving ?driver ?truck)) (at ?driver ?loc) (empty ?truck)
       ))

(:action DRIVE-TRUCK
  :parameters
   (?truck - truck
    ?loc-from - location
    ?loc-to - location
    ?driver - driver)
  :precondition
   (and (at ?truck ?loc-from)
   (driving ?driver ?truck) (link ?loc-from ?loc-to))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action WALK
  :parameters
   (?driver - driver
    ?loc-from - location
    ?loc-to - location)
  :precondition
   (and (at ?driver ?loc-from) (path ?loc-from ?loc-to))
  :effect
   (and (not (at ?driver ?loc-from)) (at ?driver ?loc-to)))


)
```

Listing A.3: DriverLog STRIPS planning domain

### A.1.4 ZenoTravel

```
(define (domain zeno-travel)
(:requirements :typing)
(:types aircraft person city flevel - object)
(:predicates (at ?x - (either person aircraft) ?c - city)
             (in ?p - person ?a - aircraft)
       (fuel-level ?a - aircraft ?l - flevel)
       (next ?l1 ?l2 - flevel))


(:action board
 :parameters (?p - person ?a - aircraft ?c - city)

 :precondition (and (at ?p ?c)
                 (at ?a ?c))
 :effect (and (not (at ?p ?c))
```

```
            (in ?p ?a)))

(:action debark
 :parameters (?p - person ?a - aircraft ?c - city)

 :precondition (and (in ?p ?a)
                    (at ?a ?c))
 :effect (and (not (in ?p ?a))
              (at ?p ?c)))

(:action fly
 :parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 - flevel)

 :precondition (and (at ?a ?c1)
                    (fuel-level ?a ?l1)
    (next ?l2 ?l1))
 :effect (and (not (at ?a ?c1))
              (at ?a ?c2)
              (not (fuel-level ?a ?l1))
              (fuel-level ?a ?l2)))

(:action zoom
 :parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 ?l3 - flevel)

 :precondition (and (at ?a ?c1)
                    (fuel-level ?a ?l1)
    (next ?l2 ?l1)
    (next ?l3 ?l2)
   )
 :effect (and (not (at ?a ?c1))
              (at ?a ?c2)
              (not (fuel-level ?a ?l1))
              (fuel-level ?a ?l3)
  )
)

(:action refuel
 :parameters (?a - aircraft ?c - city ?l - flevel ?l1 - flevel)

 :precondition (and (fuel-level ?a ?l)
                    (next ?l ?l1)
                    (at ?a ?c))
 :effect (and (fuel-level ?a ?l1) (not (fuel-level ?a ?l))))


)
```

Listing A.4: ZenoTravel STRIPS planning domain

## A.2 Numeric domains

### A.2.1 Depots

```
(define (domain Depot)
(:requirements :typing :fluents)
```

```
(:types place locatable - object
  depot distributor - place
        truck hoist surface - locatable
        pallet crate - surface)

(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface)
)

(:functions
  (load_limit ?t - truck)
  (current_load ?t - truck)
  (weight ?c - crate)
  (fuel-cost)
)

(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (at ?x ?y))
:effect (and (not (at ?x ?y)) (at ?x ?z)
    (increase (fuel-cost) 10)))

(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (available ?x) (at ?y ?p) (on ?y ?z) (
    clear ?y))
:effect (and (not (at ?y ?p)) (lifting ?x ?y) (not (clear ?y)) (not (
    available ?x))
             (clear ?z) (not (on ?y ?z)) (increase (fuel-cost) 1)))

(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (clear ?z) (lifting ?x ?y))
:effect (and (available ?x) (not (lifting ?x ?y)) (at ?y ?p) (not (
    clear ?z)) (clear ?y)
    (on ?y ?z)))

(:action Load
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (lifting ?x ?y)
    (<= (+ (current_load ?z) (weight ?y)) (load_limit ?z)))
:effect (and (not (lifting ?x ?y)) (in ?y ?z) (available ?x)
    (increase (current_load ?z) (weight ?x))))

(:action Unload
:parameters (?x - hoist ?y - crate ?z - truck ?p - place)
:precondition (and (at ?x ?p) (at ?z ?p) (available ?x) (in ?y ?z))
:effect (and (not (in ?y ?z)) (not (available ?x)) (lifting ?x ?y)
    (decrease (current_load ?z) (weight ?x))))

)
```

Listing A.5: Depots numeric planning domain

### A.2.2 DriverLog

```
(define (domain driverlog)
  (:requirements :typing :fluents)
  (:types          location locatable - object
    driver truck obj - locatable)

  (:predicates
    (at ?obj - locatable ?loc - location)
    (in ?obj1 - obj ?obj - truck)
    (driving ?d - driver ?v - truck)
    (link ?x ?y - location) (path ?x ?y - location)
    (empty ?v - truck)
)
  (:functions (time-to-walk ?l1 ?l2 - location)
    (time-to-drive ?l1 ?l2 - location)
    (driven)
    (walked))


(:action LOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action UNLOAD-TRUCK
  :parameters
   (?obj - obj
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (in ?obj ?truck))
  :effect
   (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action BOARD-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :precondition
   (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck))
  :effect
   (and (not (at ?driver ?loc)) (driving ?driver ?truck) (not (empty ?
       truck))))

(:action DISEMBARK-TRUCK
  :parameters
   (?driver - driver
    ?truck - truck
    ?loc - location)
  :precondition
```

```
   (and (at ?truck ?loc) (driving ?driver ?truck))
  :effect
   (and (not (driving ?driver ?truck)) (at ?driver ?loc) (empty ?truck)
       ))

(:action DRIVE-TRUCK
  :parameters
   (?truck - truck
    ?loc-from - location
    ?loc-to - location
    ?driver - driver)
  :precondition
   (and (at ?truck ?loc-from)
   (driving ?driver ?truck) (link ?loc-from ?loc-to))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)
    (increase (driven) (time-to-drive ?loc-from ?loc-to))))

(:action WALK
  :parameters
   (?driver - driver
    ?loc-from - location
    ?loc-to - location)
  :precondition
   (and (at ?driver ?loc-from) (path ?loc-from ?loc-to))
  :effect
   (and (not (at ?driver ?loc-from)) (at ?driver ?loc-to)
  (increase (walked) (time-to-walk ?loc-from ?loc-to))))


)
```

Listing A.6: DriverLog numeric planning domain

### A.2.3 Rovers

```
(define (domain Rover)
(:requirements :typing :fluents)
(:types rover waypoint store camera mode lander objective)

(:predicates (at ?x - rover ?y - waypoint)
             (at_lander ?x - lander ?y - waypoint)
             (can_traverse ?r - rover ?x - waypoint ?y - waypoint)
       (equipped_for_soil_analysis ?r - rover)
             (equipped_for_rock_analysis ?r - rover)
             (equipped_for_imaging ?r - rover)
             (empty ?s - store)
             (have_rock_analysis ?r - rover ?w - waypoint)
             (have_soil_analysis ?r - rover ?w - waypoint)
             (full ?s - store)
       (calibrated ?c - camera ?r - rover)
       (supports ?c - camera ?m - mode)
             (available ?r - rover)
             (visible ?w - waypoint ?p - waypoint)
             (have_image ?r - rover ?o - objective ?m - mode)
             (communicated_soil_data ?w - waypoint)
```

```
            (communicated_rock_data ?w - waypoint)
            (communicated_image_data ?o - objective ?m - mode)
        (at_soil_sample ?w - waypoint)
        (at_rock_sample ?w - waypoint)
            (visible_from ?o - objective ?w - waypoint)
        (store_of ?s - store ?r - rover)
        (calibration_target ?i - camera ?o - objective)
        (on_board ?i - camera ?r - rover)
        (channel_free ?l - lander)
        (in_sun ?w - waypoint)

)

(:functions (energy ?r - rover) (recharges) )

(:action navigate
:parameters (?x - rover ?y - waypoint ?z - waypoint)
:precondition (and (can_traverse ?x ?y ?z) (available ?x) (at ?x ?y)
                (visible ?y ?z) (>= (energy ?x) 8)
      )
:effect (and (decrease (energy ?x) 8) (not (at ?x ?y)) (at ?x ?z)
    )
)

(:action recharge
:parameters (?x - rover ?w - waypoint)
:precondition (and (at ?x ?w) (in_sun ?w) (<= (energy ?x) 80))
:effect (and (increase (energy ?x) 20) (increase (recharges) 1))
)

(:action sample_soil
:parameters (?x - rover ?s - store ?p - waypoint)
:precondition (and (at ?x ?p)(>= (energy ?x) 3) (at_soil_sample ?p) (
    equipped_for_soil_analysis ?x) (store_of ?s ?x) (empty ?s)
    )
:effect (and (not (empty ?s)) (full ?s) (decrease (energy ?x) 3) (
    have_soil_analysis ?x ?p) (not (at_soil_sample ?p))
    )
)

(:action sample_rock
:parameters (?x - rover ?s - store ?p - waypoint)
:precondition (and  (at ?x ?p) (>= (energy ?x) 5)(at_rock_sample ?p) (
    equipped_for_rock_analysis ?x) (store_of ?s ?x)(empty ?s)
    )
:effect (and (not (empty ?s)) (full ?s) (decrease (energy ?x) 5) (
    have_rock_analysis ?x ?p) (not (at_rock_sample ?p))
    )
)

(:action drop
:parameters (?x - rover ?y - store)
:precondition (and (store_of ?y ?x) (full ?y)
    )
:effect (and (not (full ?y)) (empty ?y)
  )
)
```

```
(:action calibrate
 :parameters (?r - rover ?i - camera ?t - objective ?w - waypoint)
 :precondition (and (equipped_for_imaging ?r) (>= (energy ?r) 2)(
     calibration_target ?i ?t) (at ?r ?w) (visible_from ?t ?w)(on_board
      ?i ?r)
    )
 :effect (and (decrease (energy ?r) 2)(calibrated ?i ?r) )
)



(:action take_image
 :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m -
     mode)
 :precondition (and (calibrated ?i ?r)
       (on_board ?i ?r)
                     (equipped_for_imaging ?r)
                     (supports ?i ?m)
        (visible_from ?o ?p)
                    (at ?r ?p)
      (>= (energy ?r) 1)
               )
 :effect (and (have_image ?r ?o ?m)(not (calibrated ?i ?r))(decrease (
     energy ?r) 1)
    )
)

(:action communicate_soil_data
 :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y -
     waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_soil_analysis ?r ?
     p)
                   (visible ?x ?y)(available ?r)(channel_free ?l)(>= (
                       energy ?r) 4)
            )
 :effect (and (not (available ?r))(not (channel_free ?l))
(channel_free ?l) (communicated_soil_data ?p)(available ?r)(decrease (
    energy ?r) 4)
  )
)

(:action communicate_rock_data
 :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y -
     waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_rock_analysis ?r ?
     p)(>= (energy ?r) 4)
                   (visible ?x ?y)(available ?r)(channel_free ?l)
            )
 :effect (and   (not (available ?r))(not (channel_free ?l))
(channel_free ?l)
(communicated_rock_data ?p)(available ?r)(decrease (energy ?r) 4)
          )
)
```

```
(:action communicate_image_data
 :parameters (?r - rover ?l - lander ?o - objective ?m - mode ?x -
     waypoint ?y - waypoint)
 :precondition (and (at ?r ?x)(at_lander ?l ?y)(have_image ?r ?o ?m)(
     visible ?x ?y)(available ?r)(channel_free ?l)(>= (energy ?r) 6)
           )
 :effect (and (not (available ?r))(not (channel_free ?l))
(channel_free ?l)
(communicated_image_data ?o ?m)(available ?r)(decrease (energy ?r) 6)
         )
)

)
```

Listing A.7: Rovers numeric planning domain

### A.2.4 Satellite

```
(define (domain satellite)
  (:requirements :typing :fluents :equality)
 (:types satellite direction instrument mode)
 (:predicates
              (on_board ?i - instrument ?s - satellite)
        (supports ?i - instrument ?m - mode)
        (pointing ?s - satellite ?d - direction)
        (power_avail ?s - satellite)
        (power_on ?i - instrument)
        (calibrated ?i - instrument)
        (have_image ?d - direction ?m - mode)
        (calibration_target ?i - instrument ?d - direction))



  (:functions (data_capacity ?s - satellite)
       (data ?d - direction ?m - mode)
    (slew_time ?a ?b - direction)
    (data-stored)
    (fuel ?s - satellite)
    (fuel-used)
  )

  (:action turn_to
   :parameters (?s - satellite ?d_new - direction ?d_prev - direction)
   :precondition (and (pointing ?s ?d_prev)
                      (not (= ?d_new ?d_prev))
    (>= (fuel ?s) (slew_time ?d_new ?d_prev))
              )
   :effect (and  (pointing ?s ?d_new)
                 (not (pointing ?s ?d_prev))
    (decrease (fuel ?s) (slew_time ?d_new ?d_prev))
    (increase (fuel-used) (slew_time ?d_new ?d_prev))
           )
  )
```

```
  (:action switch_on
   :parameters (?i - instrument ?s - satellite)

   :precondition (and (on_board ?i ?s)
                      (power_avail ?s)
                 )
   :effect (and (power_on ?i)
                (not (calibrated ?i))
                (not (power_avail ?s))
           )

  )


  (:action switch_off
   :parameters (?i - instrument ?s - satellite)

   :precondition (and (on_board ?i ?s)
                      (power_on ?i)
                  )
   :effect (and (not (power_on ?i))
                (power_avail ?s)
           )
  )

  (:action calibrate
   :parameters (?s - satellite ?i - instrument ?d - direction)
   :precondition (and (on_board ?i ?s)
         (calibration_target ?i ?d)
                      (pointing ?s ?d)
                      (power_on ?i)
                 )
   :effect (calibrated ?i)
  )


  (:action take_image
   :parameters (?s - satellite ?d - direction ?i - instrument ?m - mode
       )
   :precondition (and (calibrated ?i)
                      (on_board ?i ?s)
                      (supports ?i ?m)
                      (power_on ?i)
                      (pointing ?s ?d)
                      (power_on ?i)
        (>= (data_capacity ?s) (data ?d ?m))
                 )
   :effect (and (decrease (data_capacity ?s) (data ?d ?m)) (have_image
       ?d ?m)
    (increase (data-stored) (data ?d ?m)))
  )
)
```

Listing A.8: Satellite numeric planning domain

### A.2.5 ZenoTravel

```
(define (domain zeno-travel)
(:requirements :typing :fluents)
(:types aircraft person city - object)
(:predicates (at ?x - (either person aircraft) ?c - city)
             (in ?p - person ?a - aircraft))
(:functions (fuel ?a - aircraft)
            (distance ?c1 - city ?c2 - city)
            (slow-burn ?a - aircraft)
            (fast-burn ?a - aircraft)
            (capacity ?a - aircraft)
            (total-fuel-used)
      (onboard ?a - aircraft)
            (zoom-limit ?a - aircraft)
            )

(:action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and (at ?p ?c)
                    (at ?a ?c))
 :effect (and (not (at ?p ?c))
              (in ?p ?a)
    (increase (onboard ?a) 1)))


(:action debark
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and (in ?p ?a)
                    (at ?a ?c))
 :effect (and (not (in ?p ?a))
              (at ?p ?c)
    (decrease (onboard ?a) 1)))

(:action fly
 :parameters (?a - aircraft ?c1 ?c2 - city)
 :precondition (and (at ?a ?c1)
                  (>= (fuel ?a)
                            (* (distance ?c1 ?c2) (slow-burn ?a))))
 :effect (and (not (at ?a ?c1))
              (at ?a ?c2)
              (increase (total-fuel-used)
                            (* (distance ?c1 ?c2) (slow-burn ?a)))
              (decrease (fuel ?a)
                            (* (distance ?c1 ?c2) (slow-burn ?a)))))

(:action zoom
 :parameters (?a - aircraft ?c1 ?c2 - city)
 :precondition (and (at ?a ?c1)
                  (>= (fuel ?a)
                            (* (distance ?c1 ?c2) (fast-burn ?a)))
                  (<= (onboard ?a) (zoom-limit ?a)))
 :effect (and (not (at ?a ?c1))
              (at ?a ?c2)
              (increase (total-fuel-used)
                            (* (distance ?c1 ?c2) (fast-burn ?a)))
              (decrease (fuel ?a)
```

```
                        (* (distance ?c1 ?c2) (fast-burn ?a)))
  )
)

(:action refuel
 :parameters (?a - aircraft ?c - city)
 :precondition (and (> (capacity ?a) (fuel ?a))
              (at ?a ?c)
    )
 :effect (and (assign (fuel ?a) (capacity ?a)))
)
)
```

Listing A.9: ZenoTravel numeric planning domain

# Appendix B

# PlanMiner's experimental results

## B.1 STRIPS domains

### B.1.1 ARMS

- **BlocksWorld.** ARMS is hardly affected by incompleteness. At 50% missing data the precision drops 2 points, and at 90% it drops 5 points. The recall does not vary in these experiments, which means that the F-Score only varies by 3 points throughout the experiment. This leads to the domains learned by ARMS always being valid.

- **Depots.** The algorithm remains resilient to incompleteness even in the most complex experimentation, with little or no variation in results. At 90% incompleteness, the algorithm loses 17% precision and 11% recall, which causes the F-Score to drop 11 points and the learned domains to become invalid.

- **DriverLog.** Up to 10 per cent incompleteness, the algorithm maintains the validity of the domains despite the 10 point loss of precision. In more complex experiments the metrics drop by more than 20 points, dragging the F-Score down to 66%. In earlier experiments, the domains are no longer valid due to recall drops to 50% missing data.

- **ZenoTravel.** Similar to what happens in the Driverlog domain, ARMS goes from perfect results to losing 10% precision and 2% recall when removing a certain number of data (which causes a drop in F-Score of 3 points when removing input data and the invalidity of the learned domains). The results remain stable until the last experiments where the precision and recall drop by 25 and 8 points respectively. In these experiments, ARMS shows an F-Score of 77

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.9576 | 0.0000 | 1.0000 | 0.0000 | 0.9783 | 0.0078 |
| | 10% | 0.9310 | 0.0000 | 1.0000 | 0.0000 | 0.9642 | 0.0000 |
| | 50% | 0.9310 | 0.0148 | 1.0000 | 0.0000 | 0.9642 | 0.0000 |
| | 90% | 0.8883 | 0.0159 | 1.0000 | 0.0000 | 0.9408 | 0.0089 |
| Depots | 0% | 0.9629 | 0.0000 | 0.9629 | 0.0000 | 0.9629 | 0.0000 |
| | 10% | 0.9214 | 0.0159 | 0.9556 | 0.0165 | 0.9381 | 0.0162 |
| | 50% | 0.9000 | 0.0159 | 0.9334 | 0.0165 | 0.9163 | 0.0162 |
| | 90% | 0.7372 | 0.0209 | 0.8296 | 0.0561 | 0.7798 | 0.0287 |
| DriverLog | 0% | 1.0000 | 0.0000 | 0.9642 | 0.0000 | 0.9818 | 0.0000 |
| | 10% | 0.8941 | 0.0129 | 0.9642 | 0.0000 | 0.9278 | 0.0070 |
| | 50% | 0.8405 | 0.0523 | 0.9285 | 0.0252 | 0.8819 | 0.0384 |
| | 90% | 0.6239 | 0.0708 | 0.7142 | 0.0874 | 0.6644 | 0.0683 |
| ZenoTravel | 0% | 1.0000 | 0.0000 | 0.9000 | 0.0000 | 0.9387 | 0.0000 |
| | 10% | 0.9070 | 0.0340 | 0.8846 | 0.0000 | 0.9034 | 0.0331 |
| | 50% | 0.8993 | 0.0329 | 0.8846 | 0.0344 | 0.8916 | 0.0158 |
| | 90% | 0.7525 | 0.0834 | 0.8000 | 0.0631 | 0.7741 | 0.0646 |

Table B.1: ARMS Results

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 50% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 90% | 0.9700 | 0.0319 | 0.9556 | 0.0405 | 0.9625 | 0.0327 |
| Depots | 0% | 0.9778 | 0.0202 | 0.9629 | 0.0000 | 0.9702 | 0.0099 |
| | 10% | 0.9640 | 0.0357 | 0.9629 | 0.0000 | 0.9632 | 0.0178 |
| | 50% | 0.9566 | 0.0396 | 0.9556 | 0.0165 | 0.9558 | 0.0244 |
| | 90% | 0.9483 | 0.0193 | 0.9481 | 0.0202 | 0.9481 | 0.0155 |
| DriverLog | 0% | 0.7812 | 0.0310 | 0.8428 | 0.0407 | 0.8108 | 0.0351 |
| | 10% | 0.7565 | 0.0360 | 0.8214 | 0.0564 | 0.7872 | 0.0416 |
| | 50% | 0.7429 | 0.0307 | 0.8071 | 0.0541 | 0.7735 | 0.0402 |
| | 90% | 0.7180 | 0.0261 | 0.7642 | 0.0597 | 0.7397 | 0.0371 |
| ZenoTravel | 0% | 0.9259 | 0.0000 | 0.9615 | 0.0000 | 0.9433 | 0.0000 |
| | 10% | 0.8718 | 0.0157 | 0.9384 | 0.0210 | 0.9036 | 0.0084 |
| | 50% | 0.8479 | 0.0354 | 0.9384 | 0.0210 | 0.8907 | 0.0275 |
| | 90% | 0.7722 | 0.0525 | 0.8230 | 0.0210 | 0.7961 | 0.0321 |

Table B.2: FAMA Results

### B.1.2 FAMA

- **BlocksWorld.** Even the most complex experimentation FAMA maintains its precision, recall and F-Score results up to the most complex experimentation. In these experiments, the algorithm loses around 4% F-Score, with a further drop in precision. These results make the learned domains valid.

- **Depots.** The algorithm maintains the precision results, with a slight drop throughout the whole experimentation. The recall on the other hand remains unchanged up to the experiments with the smallest amount of input data. This resilience to incompleteness is shown in the F-Score difference between the best and worst results, which is 94.8%, and in the validity of the learned domains.

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.9060 | 0.0181 | 0.9259 | 0.0000 | 0.9158 | 0.0092 |
| | 10% | 0.8983 | 0.0279 | 0.9112 | 0.0202 | 0.9044 | 0.0194 |
| | 50% | 0.8853 | 0.0425 | 0.9037 | 0.0202 | 0.8941 | 0.0282 |
| | 90% | 0.8382 | 0.0325 | 0.8445 | 0.0405 | 0.8411 | 0.0346 |
| Depots | 0% | 0.9230 | 0.0649 | 0.9556 | 0.0482 | 0.9386 | 0.0546 |
| | 10% | 0.9230 | 0.0649 | 0.9556 | 0.0482 | 0.9386 | 0.0546 |
| | 50% | 0.9026 | 0.0643 | 0.9334 | 0.0482 | 0.9167 | 0.0472 |
| | 90% | 0.8148 | 0.0492 | 0.8445 | 0.0844 | 0.8280 | 0.0565 |
| DriverLog | 0% | 0.9133 | 0.0172 | 0.9000 | 0.0298 | 0.9063 | 0.0159 |
| | 10% | 0.9104 | 0.0201 | 0.8714 | 0.0407 | 0.8902 | 0.0271 |
| | 50% | 0.8760 | 0.0476 | 0.8571 | 0.0564 | 0.8661 | 0.0496 |
| | 90% | 0.8289 | 0.0335 | 0.8214 | 0.0437 | 0.8241 | 0.0243 |
| ZenoTravel | 0% | 0.8538 | 0.0245 | 0.9384 | 0.0000 | 0.8938 | 0.0145 |
| | 10% | 0.8346 | 0.0361 | 0.9230 | 0.0210 | 0.8762 | 0.0203 |
| | 50% | 0.7639 | 0.0312 | 0.7923 | 0.0344 | 0.7773 | 0.0243 |
| | 90% | 0.7535 | 0.0401 | 0.7769 | 0.0688 | 0.7643 | 0.0499 |

Table B.3: OPMaker2 Results

- **DriverLog.** The algorithm starts from an F-Score metric of 81 points, which drops to 73% in the more complex experiments. These drops affect both precision and recall, and are constant throughout the experiment, but are most noticeable at the beginning and end of the experiment. Because of this, none of the domains learned by FAMA are valid.

- **ZenoTravel.** FAMA loses 5 and 3 percent precision in the experiments with 10 and 50 percent missing predicates, while its recall drops slightly at the beginning of the experimentation (3 percent) and is maintained until the experimentation with 90 percent incompleteness. In the final experiments, both metrics drop and end up placing the F-Score at 79%. These drops in precision end up hurting domain validity, and FAMA is unable to learn valid domains from 10 percent missing predicates.

### B.1.3 OPMaker2

- **BlocksWorld.** In the initial experiments, the algorithm shows good resistance to the incompleteness of the input data, losing around 2% precision and recall until the experiments with half of the input data. In those experiments with a higher level of incompleteness, these metrics show a large drop, with the F-Score at 84 points. Although the biggest performance loss of the algorithm occurs in the final experiments (and accounts for half of the total performance loss), OPMaker2 only manages to learn valid domains with complete data.

- **Depots.** The algorithm remains impervious to incompleteness up to the experiments with 90 percent missing data, losing neither precision nor recall except at 50 percent incompleteness where it drops by 2 percent. In the more complex experiments, its precision and recall suffer a severe 9 percent drop

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.9568 | 0.0144 | 0.9777 | 0.0331 | 0.9668 | 0.0159 |
| | 10% | 0.9434 | 0.02773 | 0.9629 | 0.0370 | 0.9522 | 0.0104 |
| | 50% | 0.9302 | 0.0448 | 0.9556 | 0.0309 | 0.9418 | 0.0228 |
| | 90% | 0.9136 | 0.0459 | 0.9259 | 0.0261 | 0.9194 | 0.0324 |
| Depots | 0% | 0.9642 | 0.0000 | 1.0000 | 0.0000 | 0.9818 | 0.0000 |
| | 10% | 0.9279 | 0.0236 | 0.9481 | 0.0331 | 0.9376 | 0.0216 |
| | 50% | 0.9138 | 0.0319 | 0.9407 | 0.0496 | 0.9266 | 0.0358 |
| | 90% | 0.8915 | 0.0570 | 0.9112 | 0.0721 | 0.9006 | 0.0601 |
| DriverLog | 0% | 0.9443 | 0.0182 | 0.9642 | 0.0000 | 0.9541 | 0.0092 |
| | 10% | 0.9112 | 0.0257 | 0.9428 | 0.0319 | 0.9262 | 0.0149 |
| | 50% | 0.8356 | 0.0648 | 0.8857 | 0.0298 | 0.8590 | 0.0407 |
| | 90% | 0.7479 | 0.0580 | 0.8142 | 0.0638 | 0.7782 | 0.0483 |
| ZenoTravel | 0% | 0.9207 | 0.0248 | 0.8846 | 0.0384 | 0.9017 | 0.0209 |
| | 10% | 0.8882 | 0.0294 | 0.8538 | 0.0172 | 0.8707 | 0.0160 |
| | 50% | 0.8482 | 0.0302 | 0.8153 | 0.0501 | 0.8309 | 0.0347 |
| | 90% | 0.7723 | 0.0445 | 0.7307 | 0.0543 | 0.7506 | 0.0475 |

Table B.4: AMAN Results

in performance, bringing the F-Score to 82 percent. The initial resistance to incompleteness makes the domains learned up to 10% of missing data valid.

- **DriverLog.** By removing input data, the algorithm is affected by a small percentage. In these experiments, it loses 3% performance in both precision and recall. Both metrics continue to drop throughout the experimentation, but where recall drops by 2 percent and 3 percent in later experiments, precision drops by 4 percent and 5 percent. This is reflected in the F-Score making it present at 82 points at 90% incompleteness. Although the algorithm shows some signs of resistance to incompleteness, it is only able to learn valid domains with complete data.

- **ZenoTravel.** The OPMaker2 metrics show stable results before and after experimentation with half of the data missing. In these experiments, precision and recall drop by about 10 points. The final F-Score of the algorithm is 75%, and due to the results of the initial metrics, OPMaker2 is not able to learn valid domains.

### B.1.4 AMAN

- **BlocksWorld.** AMAN shows resistance to incompleteness, losing 4% precision and 2% recall in total over the course of the experiment. This drop is gradual but increases slightly in the more complex experimentation. The F-Score is 4 points below the results obtained using complete data, while the domains are valid regardless of the quality of the input data used.

- **Depots.** In the first experiments, both precision and recall drop by 4 percent and 6 percent when incompleteness is included. These drops are reiterated throughout the experimentation but are much more moderate. In the final

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.8053 | 0.0331 | 0.8482 | 0.0527 | 0.8214 | 0.0438 |
| | 10% | 0.7036 | 0.0669 | 0.8482 | 0.0442 | 0.7775 | 0.0557 |
| | 50% | 0.6833 | 0.0880 | 0.8093 | 0.0782 | 0.7474 | 0.0798 |
| | 90% | 0.6309 | 0.1604 | 0.7451 | 0.8095 | 0.6905 | 0.1109 |
| Depots | 0% | 0.8139 | 0.0555 | 0.9739 | 0.0341 | 0.8939 | 0.0445 |
| | 10% | 0.7444 | 0.0888 | 0.8488 | 0.0839 | 0.7924 | 0.0852 |
| | 50% | 0.7217 | 0.1286 | 0.8488 | 0.0839 | 0.7881 | 0.1029 |
| | 90% | 0.6003 | 0.1119 | 0.6853 | 0.1111 | 0.6491 | 0.1112 |
| DriverLog | 0% | 0.7221 | 0.0223 | 0.7326 | 0.0034 | 0.7259 | 0.0115 |
| | 10% | 0.7047 | 0.0364 | 0.7283 | 0.0198 | 0.7182 | 0.0274 |
| | 50% | 0.6226 | 0.0765 | 0.7073 | 0.0516 | 0.6633 | 0.0623 |
| | 90% | 0.6028 | 0.191 | 0.6876 | 0.0823 | 0.6450 | 0.0900 |
| ZenoTravel | 0% | 0.6436 | 0.0555 | 0.6654 | 0.0387 | 0.6533 | 0.0456 |
| | 10% | 0.4978 | 0.0696 | 0.5327 | 0.0473 | 0.5177 | 0.0541 |
| | 50% | 0.4223 | 0.0683 | 0.4855 | 0.0483 | 0.4570 | 0.0589 |
| | 90% | 0.4074 | 0.0943 | 0.4803 | 0.0783 | 0.4414 | 0.0823 |

Table B.5: ID3 Results

experiments, the F-Score is 90 points. In terms of validity, despite the initial drop in metrics, all domains are valid.

- **DriverLog.** The learned domains maintain certain levels of resilience in the first experiments, losing 3 percent precision and 2 percent recall at 10 percent incompleteness. In the latter experiments, the metrics drop to a greater degree, especially precision, which is 20 percent below these initial experiments. This results in the F-Score dropping by 14% with respect to the initial result. After the metrics drop with half of the input data, the domains are no longer valid.

- **ZenoTravel.** The metrics fall steadily throughout the experimentation in steps from 4 percent precision and 3 percent recall to 90 percent missing data. In these experiments, both metrics drop 7 points. This leads to the algorithm losing a lot of performance, bringing its F-Score 25 points below its initial value. The validity of the domains is only true for those obtained from the complete data.

### B.1.5 PlanMiner (ID3)

- **BlocksWorld.** Although the precision drops by 10% when some incompleteness is encountered, recall is not affected by incompleteness until experimentation with half of the missing data (where it drops by 4 points). This brings the F-Score to 77 points. In more complex experiments both precision and recall continue to drop, bringing the overall performance of ID3 to 69% F-Score. Because of this, only domains obtained with complete data are valid.

- **Depots.** Similar to the previous domain, the algorithm suffers from the initial inclusion of incompleteness, initially dropping 7 points in precision and

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.8046 | 0.0545 | 0.8453 | 0.0384 | 0.8214 | 0.0438 |
| | 10% | 0.7457 | 0.0776 | 0.8027 | 0.0392 | 0.7775 | 0.0557 |
| | 50% | 0.7025 | 0.0999 | 0.7889 | 0.0532 | 0.7474 | 0.0798 |
| | 90% | 0.6463 | 0.1436 | 0.7400 | 0.0816 | 0.6905 | 0.1109 |
| Depots | 0% | 0.8864 | 0.0665 | 0.9024 | 0.0285 | 0.8939 | 0.0445 |
| | 10% | 0.7746 | 0.1073 | 0.8175 | 0.0599 | 0.7924 | 0.0852 |
| | 50% | 0.7523 | 0.1336 | 0.8110 | 0.0662 | 0.7881 | 0.1029 |
| | 90% | 0.5998 | 0.1526 | 0.6923 | 0.0782 | 0.6491 | 0.1112 |
| DriverLog | 0% | 0.8045 | 0.0100 | 0.8285 | 0.0078 | 0.8130 | 0.0089 |
| | 10% | 0.8045 | 0.0100 | 0.8285 | 0.0078 | 0.8130 | 0.0089 |
| | 50% | 0.7778 | 0.0667 | 0.8222 | 0.0423 | 0.7930 | 0.0577 |
| | 90% | 0.7142 | 0.103 | 0.7978 | 0.0656 | 0.7509 | 0.0818 |
| ZenoTravel | 0% | 0.8346 | 0.0322 | 0.8593 | 0.0242 | 0.8435 | 0.0287 |
| | 10% | 0.8023 | 0.0723 | 0.8456 | 0.0382 | 0.8201 | 0.0512 |
| | 50% | 0.7094 | 0.0783 | 0.7864 | 0.0398 | 0.7485 | 0.0596 |
| | 90% | 0.6885 | 0.0915 | 0.7664 | 0.0564 | 0.7217 | 0.0778 |

Table B.6: C45 Resultados

13 in the recall score, bringing the F-Score to 79 points. Both metrics stabilise slightly in later experiments but fall again in those experiments with the highest number of missing data. With these drops, the F-Score for the ID3 variant of PlanMiner is 64 points. PlanMiner's poor performance with the ID3 classification algorithm means that no domain learned is valid.

- **DriverLog.** The recall remains stable, even for the most complex experiments, losing only 3% of its original value up to that point. On the other hand, the precision drops the most in the experiments with 10 and 50% missing data, where it drops to 62% precision. Due to the resilience of the recall to incompleteness, the F-Score is maintained until the experiments with half of the missing data, but at 50% incompleteness its F-Score is 66% and at 90% incompleteness it is 64%. As with the Depots experiments, neither domain is valid due to the overall performance of the algorithm.

- **ZenoTravel.** The initial precision results are 64%, which is very low compared to the results with complete data from the rest of the approaches considered in this experimentation. Moreover, by including incompleteness, these results drop by an extra 15%. This behaviour is repeated with recall, which stands at 53 points only when some incompleteness is included. As the experimentation features get harder, the metrics drop even further, with the F-Score dropping to a minimum of 44 points at 90% incompleteness. This behaviour means that domain validity is not fulfilled in any learned domain.

### B.1.6 PlanMiner (C4.5)

- **BlocksWorld.** The performance of the algorithm drops by about 5% when incompleteness is included. Although this drop is repeated throughout the rest of the experimentation for precision, recall remains stable up to the most

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.7700 | 0.0573 | 0.7728 | 0.0373 | 0.7714 | 0.0438 |
| | 10% | 0.7023 | 0.0735 | 0.7486 | 0.0365 | 0.7275 | 0.0557 |
| | 50% | 0.6647 | 0.1027 | 0.7227 | 0.0513 | 0.6974 | 0.0798 |
| | 90% | 0.6055 | 0.1473 | 0.7040 | 0.0876 | 0.6405 | 0.1109 |
| Depots | 0% | 0.7365 | 0.0417 | 0.7765 | 0.0475 | 0.7523 | 0.0445 |
| | 10% | 0.7037 | 0.0975 | 0.7663 | 0.0775 | 0.7384 | 0.0857 |
| | 50% | 0.6467 | 0.1245 | 0.7294 | 0.0853 | 0.6885 | 0.1029 |
| | 90% | 0.5737 | 0.1476 | 0.6738 | 0.0834 | 0.6219 | 0.1142 |
| DriverLog | 0% | 0.7553 | 0.0167 | 0.7781 | 0.0078 | 0.7630 | 0.0089 |
| | 10% | 0.7553 | 0.0683 | 0.7781 | 0.0438 | 0.7630 | 0.0548 |
| | 50% | 0.7183 | 0.0782 | 0.7726 | 0.0448 | 0.7430 | 0.0577 |
| | 90% | 0.6649 | 0.0972 | 0.7488 | 0.0647 | 0.7009 | 0.0818 |
| ZenoTravel | 0% | 0.7823 | 0.0447 | 0.8298 | 0.0184 | 0.8035 | 0.0287 |
| | 10% | 0.7573 | 0.0703 | 0.8117 | 0.0368 | 0.7801 | 0.0512 |
| | 50% | 0.6548 | 0.0797 | 0.7540 | 0.0404 | 0.7085 | 0.0596 |
| | 90% | 0.6094 | 0.0974 | 0.7476 | 0.0584 | 0.6817 | 0.0778 |

Table B.7: RIPPER Results

complex experiments. This behaviour takes the F-Score from 82% in the experiments with complete data to 69% with 90% missing elements. Still, no learned domain is valid.

- **Depots.** In Depots, the C.45 variant of PlanMiner shows performance drops at 10 and 90 per cent incompleteness, with a levelling off in the experiments in between. These drops are 10 and 14 points on average and lead to an F-Score of 64%. Due to the drops, only the domain learned with complete data is valid.

- **DriverLog.** Up to the most advanced experimentation, the precision and recall metrics remain invariant. Precision drops by 3 points at 50 per cent incompleteness, and in more complex experiments it drops by 7 per cent, while recall drops by 3 per cent. The final F-Score in these experiments is 75 per cent, and, despite the stability of the algorithm in the initial experiments, no domain obtained with it is valid.

- **ZenoTravel.** The algorithm shows a significant drop in performance in the experiments with half the input data. In the previous and subsequent experiments, both precision and recall show slight losses but tend to remain stable. This leads to an initial F-Score of 84% and in the more complex experiments 72%, but experimentation at 50% incompleteness accounts for 83% of the total F-Score difference. The generalised performance of the algorithm leads to no domain being valid.

### B.1.7 PlanMiner (RIPPER)

- **BlocksWorld.** Throughout the experiment, there are repeated drops in the precision of 6 or 7 points and the recall of 2 or 3 points as the experimental conditions are tightened. Starting from an F-Score of 77%, this decreasing

performance behaviour causes the overall performance to decrease by 5% at each step of the experimental process. The domains learned by the algorithm are not valid.

- **Depots.** In initial experiments with 10 per cent incompleteness, the algorithm's performance suffers somewhat (precision drops by 3 per cent and recall suffer an ignorable drop), and presents some resilience to incomplete data. In the more complex experiments, this drop becomes more severe, with precision losses of 6 and 7 at 50 and 90 per cent incompleteness and recall losses of up to 3 per cent in the more complex experiments. The F-Score ranges from 75% in experiments with all data to 62% in those experiments with the highest level of incompleteness, with the greatest loss of performance in the final experiments. PlanMiner with the RIPPER classification algorithm is unable to learn valid planning domains.
- **DriverLog.** Up to experimentation with 50 percent incompleteness, the algorithm is resilient, after which point it loses almost 4 points of precision. With 90% missing data, its precision is 66%, and recall drops to 74%. Consequently, the F-Score shows little change throughout the experimentation, showing a difference of 6 points between the best and worst results. Unfortunately, due to the low performance of the algorithm initially, none of the domains learned are valid.
- **ZenoTravel.** The main loss of performance is in the experimentation with half of the missing data. In this experimentation PlanMiner using the RIPPER classification algorithm loses 10 percent precision and 6 percent recall. Throughout the rest of the experiment, precision drops steadily (but to a lesser extent), and recall remains almost unchanged. The F-Score only drops by 2 points with respect to the full data, but in the experiments, with half of the input data, these drops account for 8 percent of the final F-Score result. Of all the domains learned by the algorithm, none is valid.

### B.1.8 PlanMiner (NSLV)

- **BlocksWorld.** El algoritmo presenta resultados del 100% hasta la experimentación con el 50% de incompletitud. En esta experimentación se pierde un 2% de precisión, aunque el recall no se mantiene al 100%. El resultado final es de 96 de precisión y 98 de recall puntos al 90% de incompletitud. Estos buenos resultados llevan a que todos los dominios sean validos.
- **Depots.** Similar to the experimentation with the BlocksWorld domain, up to 50 percent incompleteness PlanMiner using the NSLV sorting algorithm. In this experiment, it loses 1 point of precision, and in more complex experiments it loses 5 points compared to experiments with complete data. On the other hand, recall remains at 100% regardless of the data used as input. These results mean that the F-Score goes from 98% in the experiments without incompleteness to 95% in the more incomplete experiments and that the domains learned by the algorithm are always valid.

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 50% | 0.9790 | 0.0309 | 0.9925 | 0.0165 | 0.9854 | 0.0150 |
| | 90% | 0.9596 | 0.0528 | 0.9851 | 0.0202 | 0.9713 | 0.0260 |
| Depots | 0% | 0.9736 | 0.0000 | 1.0000 | 0.0000 | 0.9867 | 0.0000 |
| | 10% | 0.9736 | 0.0000 | 1.0000 | 0.0000 | 0.9867 | 0.0000 |
| | 50% | 0.9639 | 0.0217 | 1.0000 | 0.0000 | 0.9815 | 0.0114 |
| | 90% | 0.9181 | 0.0506 | 1.0000 | 0.0000 | 0.9567 | 0.0279 |
| DriverLog | 0% | 0.9334 | 0.0000 | 1.0000 | 0.0000 | 0.9655 | 0.000 |
| | 10% | 0.9273 | 0.0134 | 1.0000 | 0.0000 | 0.9622 | 0.0071 |
| | 50% | 0.9146 | 0.0171 | 0.9928 | 0.0159 | 0.9520 | 0.0148 |
| | 90% | 0.8206 | 0.0686 | 0.9857 | 0.0195 | 0.8946 | 0.0453 |
| ZenoTravel | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 50% | 0.9916 | 0.0186 | 0.9769 | 0.0516 | 0.9840 | 0.0357 |
| | 90% | 0.9365 | 0.0492 | 0.9769 | 0.0210 | 0.9553 | 0.0198 |

Table B.8: NSLV Results

- **DriverLog.** precision drops by 10 percent precision in the more complex experiments, in stark contrast to the behaviour so far where it only loses one precision point in total. The recall of the algorithm suffers similar behaviour, only much lighter (only dropping by 2 percent in the final experiments). This causes the F-Score to be stable throughout the experimentation except at 90% incompleteness where it drops 6 points due to the algorithm's drop in precision. However, this drop in performance does not affect the validity of the domains.

- **ZenoTravel.** The algorithm maintains a high resilience to incompleteness, starting from perfect precision and recall results. In the most complex experimentation, the precision result is 93 points, while the recall result drops to 97%. This leads to an F-Score 4.5 points below the results obtained using complete data, but this drop does not preclude the validity of the learned domains.

## B.2 Numerical domains

### B.2.1 PlanMiner (C4.5)

- **Depots.** The algorithm exhibits a drop in precision at 10-point incompleteness, which is repeated in experiments with 50% missing data. Recall, on the other hand, is not affected by the first levels of incompleteness, dropping only at 50% incompleteness, where it loses 9 points. Both metrics stabilise in the more complex experiments and cause the F-Score to lose 10 points of performance in the experimentation with half the input data. Learned domains are only valid in experiments with complete data.

- **DriverLog.** The algorithm contemplates a serious drop in its metrics by in-

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.9055 | 0.0263 | 0.9295 | 0.0201 | 0.9187 | 0.0239 |
| | 10% | 0.8084 | 0.0285 | 0.9295 | 0.0201 | 0.8617 | 0.0243 |
| | 50% | 0.7137 | 0.0294 | 0.8140 | 0.0222 | 0.7642 | 0.0257 |
| | 90% | 0.7137 | 0.0294 | 0.8140 | 0.0222 | 0.7642 | 0.0257 |
| DriverLog | 0% | 0.8538 | 0.0453 | 0.8916 | 0.0273 | 0.8790 | 0.0338 |
| | 10% | 0.6004 | 0.0890 | 0.6899 | 0.0413 | 0.6451 | 0.0638 |
| | 50% | 0.6004 | 0.0890 | 0.6899 | 0.0413 | 0.6451 | 0.0638 |
| | 90% | 0.6004 | 0.0890 | 0.6899 | 0.0413 | 0.6451 | 0.0638 |
| Rovers | 0% | 0.8665 | 0.0340 | 0.8803 | 0.0109 | 0.8759 | 0.0229 |
| | 10% | 0.8345 | 0.0372 | 0.8756 | 0.0133 | 0.8513 | 0.0251 |
| | 50% | 0.5318 | 0.0466 | 0.5926 | 0.0198 | 0.5601 | 0.0283 |
| | 90% | 0.4073 | 0.0492 | 0.4895 | 0.0322 | 0.4468 | 0.0377 |
| Satellite | 0% | 0.8457 | 0.0307 | 0.8894 | 0.0246 | 0.8638 | 0.0270 |
| | 10% | 0.7735 | 0.0389 | 0.8535 | 0.0249 | 0.8129 | 0.0318 |
| | 50% | 0.7246 | 0.0482 | 0.8457 | 0.0258 | 0.7879 | 0.0338 |
| | 90% | 0.6838 | 0.0628 | 0.8283 | 0.0483 | 0.7604 | 0.0538 |
| ZenoTravel | 0% | 0.8172 | 0.0325 | 0.8353 | 0.0136 | 0.8215 | 0.0288 |
| | 10% | 0.7847 | 0.0472 | 0.8288 | 0.0273 | 0.8086 | 0.0359 |
| | 50% | 0.6183 | 0.0603 | 0.6753 | 0.0436 | 0.6419 | 0.0527 |
| | 90% | 0.4803 | 0.0692 | 0.5687 | 0.0473 | 0.5261 | 0.0568 |

Table B.9: C45 Results

cluding incompleteness. This drop is 25 points of precision and 21 points of recall, but from this experimentation, they remain unchanged. The F-Score ends up at 64 points in the most complex experiments. No domain learned by the algorithm is valid.

- **Rovers.** After a slight loss of performance in the experiments with 10 per cent incompleteness (3 per cent precision and 0.5 per cent recall), the algorithm is unable to maintain its results and loses 30 per cent performance when removing half of the input data. In the more complex experiments, it again suffers a 10-12% drop in metrics, which brings the F-Score to 44 points. Due to the poor performance of the algorithm, none of the domains learned by the algorithm are valid.

- **Satellite.** The algorithm shows a constant drop in performance throughout the experimentation. Throughout this experimental process, no noticeable drops are shown in any of the more complex experiments. The drop-in precision is more marked than the drop in recall, which remains more stable. This behaviour makes the F-Score difference between the simplest and the most complex experimentation 10 points. Finally, domain validity does not hold, regardless of the data used as input.

- **ZenoTravel.** Similar to Rovers, the algorithm shows some resilience to incompleteness initially but suffers severe performance losses in the final experiments. The precision of the domains learned by the algorithm is 61 points at 50% incompleteness and 48 points at 90%, while recall drops to 67 points and 56 points in these experiments. This causes the F-Score to drop by 15% in the experimentation with half of the data and by 12% in the experiments with

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.7552 | 0.0237 | 0.7787 | 0.0181 | 0.7636 | 0.0193 |
| | 10% | 0.7376 | 0.0398 | 0.7752 | 0.0317 | 0.7561 | 0.0353 |
| | 50% | 0.7139 | 0.0623 | 0.7710 | 0.0509 | 0.7486 | 0.0559 |
| | 90% | 0.6682 | 0.0858 | 0.7487 | 0.0694 | 0.7000 | 0.0749 |
| DriverLog | 0% | 0.6162 | 0.0355 | 0.6582 | 0.0311 | 0.6366 | 0.0338 |
| | 10% | 0.5835 | 0.0561 | 0.6271 | 0.0505 | 0.6012 | 0.0538 |
| | 50% | 0.5377 | 0.0676 | 0.5962 | 0.0595 | 0.5617 | 0.0638 |
| | 90% | 0.4472 | 0.0882 | 0.5272 | 0.0782 | 0.4861 | 0.0838 |
| Rovers | 0% | 0.7353 | 0.0020 | 0.7989 | 0.0010 | 0.7636 | 0.0015 |
| | 10% | 0.7162 | 0.0094 | 0.7323 | 0.0056 | 0.7486 | 0.0072 |
| | 50% | 0.5172 | 0.0373 | 0.5985 | 0.0201 | 0.5561 | 0.0258 |
| | 90% | 0.4146 | 0.0382 | 0.5108 | 0.0214 | 0.4619 | 0.0272 |
| Satellite | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 0.6651 | 0.0503 | 0.7452 | 0.0407 | 0.7099 | 0.0457 |
| | 50% | 0.6083 | 0.0617 | 0.6874 | 0.0490 | 0.6467 | 0.0557 |
| | 90% | 0.5440 | 0.0854 | 0.6449 | 0.0702 | 0.5999 | 0.0757 |
| ZenoTravel | 0% | 0.7450 | 0.0006 | 0.7672 | 0.0004 | 0.7573 | 0.0005 |
| | 10% | 0.6883 | 0.0008 | 0.7251 | 0.0004 | 0.7003 | 0.0006 |
| | 50% | 0.5437 | 0.0236 | 0.6005 | 0.0251 | 0.5795 | 0.0246 |
| | 90% | 0.4126 | 0.0242 | 0.4936 | 0.0259 | 0.4509 | 0.0266 |

Table B.10: RIPPER Results

90% incompleteness, in addition to the fact that none of the learned domains are valid.

### B.2.2 PlanMiner (RIPPER)

- **Depots.** Due to the stability of the precision and recall of even the most complex experiments, the F-Score of the learned domains only drops by 1% until they are reached. From 50% incompleteness in the input data, the precision drops by 2 points, while the recall remains almost unchanged. At 90% incompleteness, these drops become steeper and result in a loss of 5 points and 3 points of precision and recall respectively. The F-Score in the most complex experiments is 70 points, but no domain obtained with the RIPPER classifier is valid.

- **DriverLog.** Initially, the algorithm performs below 70% (61% precision and 65% recall). These values drop gradually over the course of the experiments, showing a large drop in the experiments up to those performed with the smallest possible number of data. This drop represents a loss of 9 points of precision and 6 points of recall and places the final F-Score at 48 points. Given the overall performance of the algorithm, no domain is valid.

- **Rovers.** At 50% incompleteness, the algorithm has a 20-point drop in precision and a 14-point drop in recall. This results in an F-Score drop of 20% in these experiments. At 90 per cent incompleteness, performance drops again, but only a 10 per cent loss in precision and an 8 per cent loss in recall. The final F-Score values are 46%, with the exception of the domain learned with the complete data, the domains learned by the algorithm are not valid.

| Domain | Incompleteness | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.9623 | 0.0000 | 1.0000 | 0.0000 | 0.9707 | 0.0000 |
| | 10% | 0.9539 | 0.0114 | 1.0000 | 0.0000 | 0.9637 | 0.0059 |
| | 50% | 0.9494 | 0.0223 | 0.9966 | 0.0102 | 0.9621 | 0.0148 |
| | 90% | 0.8842 | 0.0248 | 0.9690 | 0.0210 | 0.9125 | 0.0151 |
| DriverLog | 0% | 0.9184 | 0.0176 | 0.9804 | 0.0180 | 0.9486 | 0.0175 |
| | 10% | 0.9073 | 0.0322 | 0.9730 | 0.0273 | 0.9397 | 0.0264 |
| | 50% | 0.8954 | 0.0364 | 0.9662 | 0.0236 | 0.9290 | 0.0321 |
| | 90% | 0.7841 | 0.0508 | 0.9601 | 0.0278 | 0.8626 | 0.0365 |
| Rovers | 0% | 0.7756 | 0.0000 | 1.0000 | 0.0000 | 0.8730 | 0.0000 |
| | 10% | 0.7747 | 0.0039 | 1.0000 | 0.0000 | 0.8729 | 0.0022 |
| | 50% | 0.4588 | 0.0314 | 0.8671 | 0.0220 | 0.5995 | 0.0399 |
| | 90% | 0.3501 | 0.0458 | 0.8098 | 0.0361 | 0.4887 | 0.0274 |
| Satellite | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 0.9932 | 0.0146 | 0.9933 | 0.0145 | 0.9938 | 0.0095 |
| | 50% | 0.8576 | 0.0360 | 0.8530 | 0.0339 | 0.8522 | 0.0243 |
| | 90% | 0.8523 | 0.0796 | 0.8334 | 0.0866 | 0.8448 | 0.0826 |
| ZenoTravel | 0% | 0.8128 | 0.0000 | 1.0000 | 0.0000 | 0.8965 | 0.0000 |
| | 10% | 0.7037 | 0.0269 | 1.0000 | 0.0000 | 0.8252 | 0.0180 |
| | 50% | 0.6206 | 0.0750 | 0.7154 | 0.0217 | 0.6625 | 0.0422 |
| | 90% | 0.5212 | 0.0972 | 0.6761 | 0.0217 | 0.5858 | 0.0658 |

Table B.11: NSLV Results

- **Satellite.** With precision results of 100 per cent initially, the algorithm loses 44 performance points when incompleteness is included. This is repeated with recall, where it drops by 25% in these experiments. During the rest of the experimentation these drops are constant but in a much more moderate way. In the experiments with 90% incompleteness, domain precision is 54% and recall 64% (which translates into an F-Score of 59%). As expected, only the domain with complete data is valid.

- **ZenoTravel.** The precision and recall of the algorithm show similar behaviours: with a constant drop in performance with a particularly steep drop in the more complex experiments. This behaviour causes the difference between the best and worst results to be 33 points for precision and 27 points for recall, and the F-Score values in the most complex experiments to be 45%. No domain learned by this approach is valid.

### B.2.3 PlanMiner (NSLV)

- **Depots.** Depots results show a flawless recall score. In contrast, the precision drops by 1% in the experiments up to the most complex experiments where it drops by 8 points. This causes the F-Score to suffer from the precision values, dropping by 6 points throughout the experimentation (down to a 91% F-Score in the last experiments). However, these errors do not prevent all learned domains from being perfect.

- **DriverLog.** The algorithm shows resilience to incompleteness while maintaining a precision of around 90% and a recall of over 96% even for experiments with the lowest number of elements. In these experiments the precision drops by 11% and the recall by 2%, presenting an F-Score of 86% in them and making the learned domains invalid.
- **Rovers.** Precision and recall remain unchanged in the first experiments. In experiments with 50% incompleteness, the precision drops by 32 points and recall by 15 points, and in experiments, with 90% incompleteness, the metrics drop by 10 points and 6 extra points. The F-Score drops from 87% to 48% in the most complex experiments, with a drop of almost 30 points at 50% incompleteness, and this drop impedes the validity of the learned domains.
- **Satellite.** Starting from perfect precision and recall, the inclusion of incompleteness hardly affects the results. Only the most complex experiments present a stumbling block to the algorithm, where it drops by 15%. This hurts the F-Score with 90 per cent incompleteness, which drops to 84 per cent, and the validity, which presents valid domains up to that experimentation.
- **ZenoTravel.** At each step of the experimental process, the precision drops by 10%, in contrast to the recall of the algorithm which remains at 100% until the experiments with 50% incompleteness. From these experiments onwards the recall drops by almost 30 points to 71 points. This drop-in recall severely affects the F-Score, which suffers a serious loss of performance (from 82% to 66%) and the validity of the domains.

# Appendix C

# PlanMiner-N's experimental results

## C.1 STRIPS domains

### C.1.1 ARMS

- **BlocksWorld.** By including noise, the algorithm loses a not inconsiderable amount of performance. This loss is 25 per cent in precision and 17 per cent in recall, which materialises in a drop in F-Score from 97 to 76 points. The metrics remain more stable throughout the rest of the experimentation (losing only 2 per cent F-Score) until the one that uses as input data with 20 per cent noisy information. In these experiments, precision is 56 points and recall 72, 13 and 8 points lower respectively. ARMS is unable to learn valid domains except those obtained with complete data.

- **Depots.** Noise negatively affects the algorithm, causing a 30-point loss in precision and a 9-point loss in recall. This drop continues throughout the rest of the experimentation in a milder form, but with a spike at 10% noise. In this experiment, the performance of the algorithm is 44 points of precision and 71 points of recall. In the more complex experiments, again, the drop is much more moderate, with a 5 and 4 point drop in metrics, respectively. The F-Score in the final experiments is 53 points, and none of the domains learned by ARMS using noisy data are invalid.

- **DriverLog.** ARMS shows a difference of 56 precision and 47 recall points between the noiseless and the noisiest results. The biggest drop is between experimentation with 3 and 10 per cent noisy data, where precision drops by 30 and 14 points, and recall by 9 and 11 points. In the most complex experiments, the F-Score ends up with a value of 39%. Finally, only the domain obtained with complete data is valid.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.9636 | 0.0152 | 0.9858 | 0.0022 | 0.9781 | 0.0075 |
| | 3% | 0.7175 | 0.0476 | 0.8183 | 0.0256 | 0.7645 | 0.0326 |
| | 5% | 0.7102 | 0.0703 | 0.8130 | 0.0514 | 0.7648 | 0.0599 |
| | 10% | 0.6952 | 0.0738 | 0.8099 | 0.0553 | 0.7400 | 0.0615 |
| | 20% | 0.5647 | 0.0797 | 0.7243 | 0.0585 | 0.6402 | 0.0668 |
| Depots | 0% | 0.9564 | 0.0000 | 0.9762 | 0.0000 | 0.9624 | 7 0.000 |
| | 3% | 0.6574 | 0.0197 | 0.8853 | 0.0130 | 0.7383 | 0.0165 |
| | 5% | 0.5827 | 0.0214 | 0.8205 | 0.0138 | 0.7164 | 0.0170 |
| | 10% | 0.4482 | 0.0352 | 0.7182 | 0.0214 | 0.5792 | 0.028 |
| | 20% | 0.3936 | 0.0726 | 0.6726 | 0.0482 | 0.5396 | 0.058 |
| DriverLog | 0% | 0.8254 | 0.0000 | 0.9995 | 0.0000 | 0.9753 | 0.0000 |
| | 3% | 0.6488 | 0.0011 | 0.8193 | 0.0048 | 0.7278 | 0.0076 |
| | 5% | 0.5723 | 0.0457 | 0.7990 | 0.0313 | 0.6812 | 0.0384 |
| | 10% | 0.3637 | 0.0824 | 0.6001 | 0.0523 | 0.4896 | 0.0671 |
| | 20% | 0.2674 | 0.1243 | 0.5235 | 0.0715 | 0.3934 | 0.0950 |
| ZenoTravel | 0% | 0.8828 | 0.0000 | 0.9897 | 0.0000 | 0.9386 | 0.0000 |
| | 3% | 0.6265 | 0.0200 | 0.7848 | 0.0100 | 0.7033 | 0.0150 |
| | 5% | 0.5789 | 0.0395 | 0.7723 | 0.0273 | 0.6918 | 0.0334 |
| | 10% | 0.4262 | 0.0833 | 0.7284 | 0.0457 | 0.5743 | 0.064 |
| | 20% | 0.4052 | 0.1222 | 0.7059 | 0.0732 | 0.5240 | 0.098 |

Table C.1: ARMS Results

- **ZenoTravel.** With initial precision and recall results of 88% and 98%, when noise is included, these metrics drop 26 points in precision and 18 points in recall. When noise is added, the algorithm again suffers a severe drop, bringing precision to 36 points and recall to 60 points. In the final experiments, these drops continue, resulting in an F-Score of 52 points when noise is added. Due to the behaviour of the algorithm, only the domain obtained with complete data is valid.

### C.1.2 FAMA

- **BlocksWorld.** In the experiments with some noise (3% and 5% of erroneous elements), the algorithm stands at 72 points. The recall, on the other hand, maintains some resilience to noise, losing only 2% of performance in these experiments. In later experiments, the metrics drop to 59 points (precision) and 81 (recall), bringing the final F-Score to around 67%. Due to the poor performance of precision at the start, no domain obtained is valid, even at low noise levels

- **Depots.** It starts with an initial precision of 96 points, but drops to 70 points when some noise is included; this behaviour is repeated with recall, where it starts with a score of 98 points and drops to 86 points. These drops are repeated constantly and repeatedly throughout the experimentation. This means that, with 20% noisy data, precision and recall are 59% and 81% respectively, and therefore the F-Score is 71 points. As expected, the only domain that is valid is the one learned with complete data.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 0.7242 | 0.0526 | 0.9862 | 0.0244 | 0.8582 | 0.0387 |
| | 5% | 0.7242 | 0.0526 | 0.9862 | 0.0244 | 0.8582 | 0.0387 |
| | 10% | 0.6625 | 0.0676 | 0.8462 | 0.0286 | 0.7572 | 0.0383 |
| | 20% | 0.5990 | 0.0701 | 0.7553 | 0.0303 | 0.6762 | 0.0395 |
| Depots | 0% | 0.9642 | 0.0215 | 0.9856 | 0.0115 | 0.9725 | 0.0160 |
| | 3% | 0.7047 | 0.0317 | 0.8681 | 0.0169 | 0.7726 | 0.0258 |
| | 5% | 0.6783 | 0.0320 | 0.8400 | 0.0172 | 0.7623 | 0.0251 |
| | 10% | 0.6399 | 0.0462 | 0.8329 | 0.0267 | 0.7429 | 0.0285 |
| | 20% | 0.5962 | 0.0984 | 0.8101 | 0.0537 | 0.7126 | 0.0784 |
| DriverLog | 0% | 0.7625 | 0.0245 | 0.8467 | 0.0086 | 0.8073 | 0.0166 |
| | 3% | 0.5283 | 0.0314 | 0.6611 | 0.0193 | 0.5928 | 0.0252 |
| | 5% | 0.4977 | 0.0317 | 0.6580 | 0.0199 | 0.5859 | 0.0259 |
| | 10% | 0.4751 | 0.0399 | 0.6203 | 0.0213 | 0.5524 | 0.0282 |
| | 20% | 0.3463 | 0.0573 | 0.5461 | 0.0350 | 0.4485 | 0.0415 |
| ZenoTravel | 0% | 0.9051 | 0.0000 | 0.9690 | 0.0000 | 0.9387 | 0.0000 |
| | 3% | 0.5941 | 0.0232 | 0.8172 | 0.0779 | 0.7033 | 0.0157 |
| | 5% | 0.5988 | 0.0455 | 0.8062 | 0.0213 | 0.6918 | 0.0332 |
| | 10% | 0.4751 | 0.0786 | 0.7250 | 0.0567 | 0.5943 | 0.0647 |
| | 20% | 0.3892 | 0.1251 | 0.5941 | 0.0861 | 0.4940 | 0.1028 |

Table C.2: FAMA Results

- **DriverLog.** The algorithm has two major performance losses at 3% and 20% noise. The first large loss is a 24-point drop in precision and a 15-point drop in recall, which puts the F-Score 21 points below its initial experimental value. The second drop is a 13-point loss in precision and a 9-point loss in metrics, bringing the F-Score in the most complex experiments to 44 points. No domain learned by FAMA is valid.

- **ZenoTravel.** Similar to the other domains, the algorithm suffers severe performance losses initially and in the more complex experiments. When noise is included, precision drops 31 points, while recall drops to 81 points. At 10 and 10 noise, the precision drops to 47 and 38 points, while the recall drops to 72 and 59 points. The F-Score of the domains obtained in the more complex experimentation is 49%. Only the domain obtained with non-noisy data is valid.

### C.1.3 OPMaker2

- **BlocksWorld.** With a precision of 87% using data without noise, OPMaker2 loses 23 points of precision when noise is included. Similar behaviour can be observed for recall, which drops 19 points from its initial value. As the experimentation progresses, the precision drops repeatedly as the experimental conditions are tightened, while the recall shows some stability. Unfortunately, in the experiments with 20% noise, the recall suffers a serious loss of performance. In the latter experiments, the precision of the algorithm is 50 points, the recall 56 points and, consequently, the F-Score 53 points. The only

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.8748 | 0.0168 | 0.9584 | 0.0453 | 0.9199 | 0.0102 |
| | 3% | 0.6481 | 0.0237 | 0.7654 | 0.0073 | 0.7083 | 0.0154 |
| | 5% | 0.6320 | 0.0252 | 0.7500 | 0.0127 | 0.6972 | 0.0174 |
| | 10% | 0.5827 | 0.0292 | 0.7029 | 0.0138 | 0.6463 | 0.0188 |
| | 20% | 0.5061 | 0.0465 | 0.5659 | 0.0212 | 0.5361 | 0.0393 |
| Depots | 0% | 0.8926 | 0.0215 | 0.9736 | 0.0090 | 0.9376 | 0.0152 |
| | 3% | 0.7597 | 0.0228 | 0.9153 | 0.0097 | 0.8325 | 0.0158 |
| | 5% | 0.6425 | 0.0247 | 0.8006 | 0.0102 | 0.7249 | 0.0174 |
| | 10% | 0.5591 | 0.0323 | 0.6951 | 0.0165 | 0.6201 | 0.0247 |
| | 20% | 0.4872 | 0.0740 | 0.6606 | 0.0347 | 0.5749 | 0.0542 |
| DriverLog | 0% | 0.8787 | 0.0195 | 0.9300 | 0.0110 | 0.9372 | 0.0157 |
| | 3% | 0.6357 | 0.0200 | 0.7551 | 0.0118 | 0.6999 | 0.0155 |
| | 5% | 0.6202 | 0.0321 | 0.7261 | 0.0175 | 0.6752 | 0.0243 |
| | 10% | 0.5878 | 0.0386 | 0.7191 | 0.0223 | 0.6503 | 0.0386 |
| | 20% | 0.5453 | 0.0533 | 0.6172 | 0.0237 | 0.5763 | 0.0308 |
| ZenoTravel | 0% | 0.8562 | 0.0160 | 0.8932 | 0.0147 | 0.8972 | 0.0150 |
| | 3% | 0.6578 | 0.0166 | 0.6976 | 0.0141 | 0.6737 | 0.0158 |
| | 5% | 0.5428 | 0.0285 | 0.6079 | 0.0202 | 0.5746 | 0.0233 |
| | 10% | 0.5202 | 0.0463 | 0.6003 | 0.0226 | 0.5632 | 0.0357 |
| | 20% | 0.4921 | 0.0748 | 0.5752 | 0.0581 | 0.5303 | 0.0655 |

Table C.3: OPMaker2 Results

valid domain obtained by the algorithm is the one learned using noise-free input data.

- **Depots.** With 3% noisy input data the algorithm loses 24 precision points and 6 recall points. During the rest of the experimentation, the precision drops by 10% until the end, presenting results in the most complex experimentation of 48 points, while the recall behaves the same, presenting final results of 66 points. In the experiments with the highest number of noisy data, the algorithm presents an F-Score of 57%. After the severe drops in precision when noise is included, the domains of the algorithm are invalid.

- **DriverLog.** As in the previous domains, the algorithm shows a large drop in performance, with a 24% drop in precision and an 18% drop in recall. In subsequent experiments, the algorithm's performance continues to drop, but in a much more moderate way. This trend is broken with recall, which, in the more complex experiments, loses an extra 10% of performance. This leads to an F-Score of 57 points for the algorithm in these experiments. The performance of OPMaker3 prevents domains learned with noisy data from being valid.

- **ZenoTravel.** With an initial precision and recall of 85 and 89 points, the algorithm drops to 65 and 69 points when noise is included in the input data. Experimenting with 5 per cent noise, the algorithm's performance drops again to around 10 per cent on both metrics. After this experimentation, these drops moderate and stabilise. In the most complex experiments, the algo-

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.9282 | 0.0000 | 1.0000 | 0.0000 | 0.9663 | 0.0000 |
| | 3% | 0.7438 | 0.0175 | 0.7849 | 0.0135 | 0.7623 | 0.0156 |
| | 5% | 0.7275 | 0.0216 | 0.7849 | 0.0135 | 0.7560 | 0.0173 |
| | 10% | 0.7037 | 0.0221 | 0.7849 | 0.0135 | 0.7463 | 0.0187 |
| | 20% | 0.6628 | 0.0451 | 0.7684 | 0.0242 | 0.7178 | 0.0322 |
| Depots | 0% | 0.9899 | 0.0000 | 1.0000 | 0.0000 | 0.9955 | 0.0000 |
| | 3% | 0.7062 | 0.0181 | 0.7678 | 0.0129 | 0.7361 | 0.0153 |
| | 5% | 0.6437 | 0.0224 | 0.7501 | 0.0138 | 0.7281 | 0.0175 |
| | 10% | 0.6172 | 0.0236 | 0.7252 | 0.0142 | 0.7005 | 0.0187 |
| | 20% | 0.5426 | 0.0523 | 0.6867 | 0.0427 | 0.6627 | 0.0489 |
| DriverLog | 0% | 0.9241 | 0.0250 | 0.9889 | 0.0000 | 0.9529 | 0.0122 |
| | 3% | 0.6951 | 0.0186 | 0.7732 | 0.0128 | 0.7323 | 0.0156 |
| | 5% | 0.6068 | 0.0192 | 0.7097 | 0.0145 | 0.6503 | 0.0178 |
| | 10% | 0.5192 | 0.0336 | 0.6331 | 0.0247 | 0.5784 | 0.0282 |
| | 20% | 0.4501 | 0.0460 | 0.5959 | 0.0333 | 0.5290 | 0.0399 |
| ZenoTravel | 0% | 0.8201 | 0.0157 | 0.9882 | 0.0055 | 0.9001 | 0.0135 |
| | 3% | 0.8045 | 0.0128 | 0.9487 | 0.0188 | 0.8782 | 0.0158 |
| | 5% | 0.7881 | 0.0203 | 0.9085 | 0.0146 | 0.8403 | 0.0179 |
| | 10% | 0.7172 | 0.0335 | 0.8793 | 0.0238 | 0.7889 | 0.0283 |
| | 20% | 0.6275 | 0.0748 | 0.7674 | 0.0471 | 0.6959 | 0.0536 |

Table C.4: AMAN Results

rithm has an F-Score of 53%. Finally, domain validity is not satisfied in any of the domains learned using noisy input data.

### C.1.4 AMAN

- **BlocksWorld.** The precision of the algorithm drops by 18 when including some noise in the input data, and its recall by 22 points. Throughout the experimentation, the precision continues to drop repeatedly but much more slightly (in steps of 2%), while the recall stabilises. In the more complex experiments, both metrics suffer a small, somewhat steeper drop (6 per cent for precision and 2 per cent for recall), bringing the algorithm's F-Score to 71 points. Due to the performance of the algorithm, only the domain obtained with noise-free data is valid.

- **Depots.** The algorithm's metrics start with a value above 97% (specifically 98% for precision and 100% for recall), but when some noise is included in the input data its performance drops 28 points and 24 points respectively. As the complexity of the experiments increases, the overall performance of the algorithm drops by around 2 per cent in each experiment. This leads to an F-Score of 66% in the most complex experiment. AMAN is unable to learn valid domains except those obtained with complete data.

- **DriverLog.** Similar to the Depots domain, the behaviour of the algorithm can be summarised as follows: A large initial performance loss along with a series of much smaller drops in subsequent experiments. But, in contrast to

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.7102 | 0.0602 | 0.7778 | 0.0281 | 0.7463 | 0.0414 |
| | 3% | 0.5356 | 0.0647 | 0.5991 | 0.05591 | 0.5667 | 0.0560 |
| | 5% | 0.5102 | 0.0692 | 0.5990 | 0.0637 | 0.5510 | 0.0666 |
| | 10% | 0.4789 | 0.0885 | 0.5701 | 0.0680 | 0.5263 | 0.0772 |
| | 20% | 0.3035 | 0.0982 | 0.4283 | 0.0717 | 0.3642 | 0.0837 |
| Depots | 0% | 0.6362 | 0.0492 | 0.6785 | 0.0216 | 0.6574 | 0.0382 |
| | 3% | 0.4182 | 0.1198 | 0.4753 | 0.0701 | 0.4477 | 0.0992 |
| | 5% | 0.3836 | 0.1426 | 0.4602 | 0.0891 | 0.4223 | 0.1166 |
| | 10% | 0.3255 | 0.1602 | 0.4227 | 0.1071 | 0.3757 | 0.1336 |
| | 20% | 0.2872 | 0.1756 | 0.4072 | 0.1328 | 0.3452 | 0.1535 |
| DriverLog | 0% | 0.6973 | 0.0273 | 0.7534 | 0.0002 | 0.7259 | 0.0115 |
| | 3% | 0.6535 | 0.0663 | 0.7398 | 0.0271 | 0.6802 | 0.0472 |
| | 5% | 0.6102 | 0.0982 | 0.6952 | 0.0387 | 0.6582 | 0.0628 |
| | 10% | 0.5174 | 0.1206 | 0.6527 | 0.0604 | 0.5853 | 0.0945 |
| | 20% | 0.4277 | 0.1655 | 0.6298 | 0.0709 | 0.5287 | 0.1125 |
| ZenoTravel | 0% | 0.6262 | 0.0672 | 0.6882 | 0.0292 | 0.6533 | 0.0456 |
| | 3% | 0.3882 | 0.0891 | 0.4882 | 0.0454 | 0.4224 | 0.0674 |
| | 5% | 0.3387 | 0.0900 | 0.4579 | 0.0501 | 0.3952 | 0.0739 |
| | 10% | 0.2725 | 0.1126 | 0.4511 | 0.0794 | 0.3627 | 0.0932 |
| | 20% | 0.1781 | 0.1621 | 0.4192 | 0.0871 | 0.2967 | 0.1264 |

Table C.5: PlanMiner (ID3) Results

Depots, AMAN suffers another severe performance loss in the more complex experiments in the recall metric. That said, the algorithm's F-Score starts the experiment at 95 points and drops to 73 in the experiments with 3% noise. At 10% noise, its performance is 57 points, and in the experimentation with the highest number of noisy elements, it drops to 52 points. Given this behaviour, only the domain obtained with complete data is valid.

- **ZenoTravel.** Until experimentation with 10 per cent noise, the algorithm's precision drops by 2 per cent in each experiment. Recall starts with better results compared to precision, but in the previously named experiments, its performance loss is 4 points at each step. In the experiments with 10% noise, the precision drops by 7 points, in contrast to the recall performance loss of only 3 points. In the more complex experiments, the overall performance of the algorithm brings the F-Score of the algorithm to 69 points. As the only valid set of learned domains are those obtained using non-noisy data.

### C.1.5 PlanMiner (ID3)

- **BlocksWorld.** precision drops from 71 points to 53 when noise is included, while recall drops from 77 to 59 points in that experiment. In subsequent experiments, the results drop slightly (about 2 points performance in the metrics), but in the experimentation with 20% of noisy data, the performance of the algorithm is severely impaired. In these experiments, precision loses 17% of its performance, while recall loses 15%. The F-Score of the algorithm

ranges from 74 points in the noiseless experiments to 36 points in the noisy experiments, and no learned domain is valid.

- **Depots.** The precision suffers an initial drop in precision of 22 points, which stabilises slightly in the following experiments. This stabilisation is maintained until the experiments with 10% noisy data, after which it drops repeatedly to 28 points. The recall, on the other hand, starts the experimentation showing performance of 67%, but the loss of noise inclusion in the plan traces reduces it by 20 points. In later experiments the recall continues to drop steadily but much more slightly, showing results of 40% in the more complex experiments. The significant loss of performance at the beginning of the experiment causes the F-Score to drop by 21% when faced with noise. At 5% noise, this drop slows down, with a reduction in the metric of only 2 points, but is repeated in subsequent experiments until the value of the metric reaches 34 points. The behaviour of the algorithm leads to the fact that no learned domain is valid.

- **DriverLog.** In the initial experiments (3% and 5% of noisy elements in the plan traces) the algorithm suffers performance drops of 4% (in the case of precision) and 3% (in the case of the recall). In the more complex experiments (10 and 20% of noisy items) the metrics suffer steeper drops, which put the metrics at 42 and 62 points for precision and recall respectively. Due to these drops, the F-Score in the experimentation with 20% noise is 52 points. None of the learned domains are valid.

- **ZenoTravel.** Initial precision results drop by 24% when noise is included, while recall drops by 20%. On the one hand, precision continues to drop steadily throughout the experimentation, falling by 5 per cent until the most complex experimentation, where it drops by 10 points. On the other hand, recall suffers a 3 per cent drop in performance but shows some stabilisation in the experiments with 10 per cent of noisy elements. The behaviour of the algorithm means that, with an initial F-Score of 65 points, when noise is included in the plan traces, it drops by almost 20 points. As the experimental features get harder, the F-Score drops further to 29 points in the experiments with the highest number of noisy elements. These F-Score values indicate that none of the domains learned by PlanMiner (ID3) are valid.

### C.1.6 PlanMiner-N (ID3)

- **BlocksWorld.** precision drops from 70 points to 53 when noise is included, while recall suffers a performance loss of 7 points. In further experimentation, the algorithm continues to suffer constant drops in performance. In the case of precision, these drops are slightly aggravated in the experimentation with 10 per cent of noisy predicates, where it loses 10 per cent of its performance. In the case of recall, the drops are 1 or 2 per cent at most. This leads to the F-Score dropping by 9 per cent when noise is included, and then continu-

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.7072 | 0.0602 | 0.7891 | 0.0274 | 0.7463 | 0.0414 |
| | 3% | 0.5991 | 0.0664 | 0.7101 | 0.0302 | 0.6562 | 0.0432 |
| | 5% | 0.5701 | 0.0720 | 0.7100 | 0.0369 | 0.6425 | 0.0437 |
| | 10% | 0.5142 | 0.0969 | 0.6925 | 0.0538 | 0.6039 | 0.0721 |
| | 20% | 0.4782 | 0.1035 | 0.6732 | 0.0690 | 0.5758 | 0.0728 |
| Depots | 0% | 0.6174 | 0.0424 | 0.6974 | 0.0201 | 0.6574 | 0.0382 |
| | 3% | 0.4683 | 0.0681 | 0.6183 | 0.0291 | 0.5383 | 0.0472 |
| | 5% | 0.4493 | 0.0902 | 0.6093 | 0.0591 | 0.5293 | 0.0783 |
| | 10% | 0.3887 | 0.1009 | 0.5887 | 0.0684 | 0.4887 | 0.0892 |
| | 20% | 0.2954 | 0.1293 | 0.5254 | 0.0901 | 0.4154 | 0.1119 |
| DriverLog | 0% | 0.7082 | 0.0281 | 0.7491 | 0.0002 | 0.7259 | 0.0115 |
| | 3% | 0.6719 | 0.0357 | 0.7300 | 0.0086 | 0.7003 | 0.0297 |
| | 5% | 0.6502 | 0.0538 | 0.7238 | 0.0283 | 0.6945 | 0.0461 |
| | 10% | 0.6126 | 0.0981 | 0.7193 | 0.0548 | 0.6680 | 0.0737 |
| | 20% | 0.5983 | 0.1142 | 0.7083 | 0.0604 | 0.6537 | 0.0855 |
| ZenoTravel | 0% | 0.6229 | 0.0643 | 0.6848 | 0.0282 | 0.6533 | 0.0456 |
| | 3% | 0.5029 | 0.0682 | 0.5839 | 0.0348 | 0.5432 | 0.0509 |
| | 5% | 0.4583 | 0.0744 | 0.5613 | 0.0380 | 0.5101 | 0.0535 |
| | 10% | 0.4281 | 0.0795 | 0.5400 | 0.0397 | 0.4842 | 0.0682 |
| | 20% | 0.3891 | 0.1022 | 0.5139 | 0.0442 | 0.4582 | 0.0783 |

Table C.6: PlanMiner-N (ID3) Results

ing to lose up to 3 per cent in the more complex experiments. The behaviour of the algorithm results in no learned domain being valid.

- **Depots.** The performance of the algorithm suffers two large drops in performance in the experimentation, one in the experimentation with 3% noisy elements and one in the experimentation with 20% noisy elements. In these experiments, precision drops by 15 and 10 points respectively, while recall drops by 9 and 6 points. In the rest of the experiments performance drops, but in a much more controlled way. The initial drop in performance brings the F-Score to 53 points, 12 points less than in the experiments without noise, and in the more complex experiments it ends up at 41 points. The overall performance of the algorithm indicates that none of the learned domains are valid.

- **DriverLog.** PlanMiner-N (ID3) shows a constant performance loss behaviour. This performance loss does not have large peaks of precision and recall loss but remains stable and constant throughout the experimentation. Precision drops by around 3% in each experiment, while recall decreases at a rate of 1/2 points at each step of the experimental process. The difference between the F-Score result in the experiments with non-noisy data and those with a higher proportion of noisy items is 7%. Although the performance of the algorithm is better with respect to the other domains, no domain learned by it is valid.

- **ZenoTravel.** Similar to BlocksWorld, the algorithm shows severe drops in the first experiments, which are repeated more moderately in the more com-

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.8001 | 0.0649 | 0.8384 | 0.0229 | 0.8157 | 0.0425 |
| | 3% | 0.7278 | 0.0742 | 0.7862 | 0.0349 | 0.7532 | 0.0528 |
| | 5% | 0.7023 | 0.0784 | 0.7920 | 0.0398 | 0.7476 | 0.0582 |
| | 10% | 0.6655 | 0.0985 | 0.7749 | 0.0582 | 0.7163 | 0.0757 |
| | 20% | 0.5839 | 0.1325 | 0.7248 | 0.0974 | 0.6620 | 0.1114 |
| Depots | 0% | 0.8738 | 0.0552 | 0.9137 | 0.0391 | 0.8939 | 0.0445 |
| | 3% | 0.5082 | 0.1191 | 0.6239 | 0.0794 | 0.5672 | 0.0932 |
| | 5% | 0.4501 | 0.1584 | 0.5757 | 0.0994 | 0.5189 | 0.1202 |
| | 10% | 0.4001 | 0.1784 | 0.5484 | 0.1185 | 0.4736 | 0.1435 |
| | 20% | 0.3283 | 0.1929 | 0.4882 | 0.1309 | 0.4010 | 0.1656 |
| DriverLog | 0% | 0.7730 | 0.0175 | 0.8530 | 0.0002 | 0.8130 | 0.0089 |
| | 3% | 0.7114 | 0.0547 | 0.8114 | 0.0576 | 0.7614 | 0.0528 |
| | 5% | 0.6698 | 0.0644 | 0.7898 | 0.0682 | 0.7298 | 0.0638 |
| | 10% | 0.6074 | 0.0883 | 0.7474 | 0.0884 | 0.6774 | 0.0836 |
| | 20% | 0.5462 | 0.1184 | 0.7062 | 0.1142 | 0.6262 | 0.1163 |
| ZenoTravel | 0% | 0.8162 | 0.0361 | 0.8782 | 0.0158 | 0.8435 | 0.0287 |
| | 3% | 0.7349 | 0.0580 | 0.8301 | 0.0394 | 0.7842 | 0.0475 |
| | 5% | 0.6900 | 0.0621 | 0.8295 | 0.0402 | 0.7634 | 0.0545 |
| | 10% | 0.6028 | 0.0895 | 0.7847 | 0.0482 | 0.6936 | 0.0663 |
| | 20% | 0.5382 | 0.1256 | 0.7555 | 0.0393 | 0.6478 | 0.0934 |

Table C.7: PlanMiner (C45) Results

plex ones. This leads to a 12-point drop in precision when including noise in the input data and a 10-point drop in the recall. For the remainder of the experiment, the metrics drop-in steps of 5 and 2 respectively, as the experimental characteristics are tightened. The F-Score initially drops by 11%, and at each step of the subsequent experimental process, it drops by around 3%, while domain validity is not met for any domain learned by the algorithm.

### C.1.7 PlanMiner (C4.5)

- **BlocksWorld.** With an initial precision of 80 per cent, PlanMiner(C45) loses 8 per cent precision when some noise is encountered, while recall drops 4 points from its results on the noiseless data. Throughout the experimentation, the metrics continue to fall, with a spike in the performance loss of 8 per cent for precision and 5 per cent for recall in the more complex experiments. The F-Score in the more complex experiments stands at 66%, and none of the domains obtained using the algorithm are valid.

- **Depots.** precision suffers an initial drop of 37 points when noise is included in the plan traces, and recall is reduced by 29 points. In even the most complex experiments, the precision decreases in steps of 5 per cent, while recall decreases in steps of 4 per cent. In experiments with 20% of noisy elements, both metrics decrease by around 7%. In the experiments with more noise, the final result is 40 F-Score points, and the only valid domain set is the one obtained using non-noisy plan traces.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.8062 | 0.0542 | 0.8295 | 0.0382 | 0.8157 | 0.0425 |
| | 3% | 0.7398 | 0.0605 | 0.7902 | 0.0455 | 0.7624 | 0.0561 |
| | 5% | 0.7235 | 0.0894 | 0.7820 | 0.0592 | 0.7526 | 0.0638 |
| | 10% | 0.6901 | 0.0992 | 0.7777 | 0.0500 | 0.7372 | 0.0782 |
| | 20% | 0.6384 | 0.1485 | 0.7304 | 0.7239 | 0.6803 | 0.1127 |
| Depots | 0% | 0.8683 | 0.0672 | 0.9282 | 0.0222 | 0.8939 | 0.0445 |
| | 3% | 0.8091 | 0.0792 | 0.8693 | 0.0306 | 0.8352 | 0.0523 |
| | 5% | 0.7588 | 0.0784 | 0.8385 | 0.0384 | 0.7978 | 0.0589 |
| | 10% | 0.7135 | 0.1021 | 0.7900 | 0.0694 | 0.7527 | 0.0836 |
| | 20% | 0.6074 | 0.1382 | 0.7243 | 0.0785 | 0.6620 | 0.1003 |
| DriverLog | 0% | 0.8005 | 0.0150 | 0.8247 | 0.0029 | 0.8130 | 0.0089 |
| | 3% | 0.7982 | 0.0492 | 0.8211 | 0.0098 | 0.8125 | 0.0270 |
| | 5% | 0.7982 | 0.0492 | 0.8211 | 0.0098 | 0.8125 | 0.0270 |
| | 10% | 0.7639 | 0.0864 | 0.8093 | 0.0284 | 0.7845 | 0.0569 |
| | 20% | 0.7109 | 0.1094 | 0.7770 | 0.0882 | 0.7418 | 0.0957 |
| ZenoTravel | 0% | 0.8252 | 0.0353 | 0.8635 | 0.0185 | 0.8435 | 0.0287 |
| | 3% | 0.8091 | 0.0400 | 0.8580 | 0.0244 | 0.8352 | 0.0385 |
| | 5% | 0.7839 | 0.0549 | 0.8542 | 0.0378 | 0.8272 | 0.0461 |
| | 10% | 0.7458 | 0.0629 | 0.8376 | 0.0449 | 0.7946 | 0.0572 |
| | 20% | 0.7233 | 0.0984 | 0.8252 | 0.0585 | 0.7773 | 0.0703 |

Table C.8: PlanMiner-N (C45) Results

- **DriverLog.** Starting from an initial precision of 77 points and a recall of 85 points, when noise is included, both metrics are 71 points and 81 points respectively. Precision continues to decrease by 5 or 6 points throughout the rest of the experiment, reaching 54 points in the most complex experiment. Recall, on the other hand, drops by around 3 to 4 percentage points and presents final results of 70 percentage points. Consistent with the other two metrics, the F-Score suffers a very strong loss of performance initially, which moderates during the rest of the experimentation. The final F-Score value is 62%. None of the domains learned by the algorithm is valid.

- **ZenoTravel.** The algorithm shows a significant decrease in precision and recall in the initial and final experiment sections. When noise is included in the plan traces, the precision drops 8 points and the recall 4 points. In the experiments with 5% noise, the drops are slightly slowed down in both metrics but are accentuated again in the final experiments. The precision in the most complex experiments is 53 points and the recall is 75 points. This causes the final F-Score of the algorithm in these experiments to be 64%, and given its overall behaviour, no set of learned domains is valid.

### C.1.8 PlanMiner-N (C4.5)

- **BlocksWorld.** After the initial performance loss of 7 and 3 precision and recall points respectively, the algorithm greatly reduces the performance loss in subsequent experiments. The precision drops by around 2% until the most complex experiments, while the recall drops in steps of 1%. In experiments

with a higher percentage of noisy items, both metrics drop by about 5 per cent. The F-Score shows a difference of 13 points between the best and worst experimental results, even though none of the learned domains are valid.

- **Depots.** With an initial precision of 86% and an initial recall of 92%, both metrics drop by 6 points. In subsequent experiments, this behaviour is repeated with more moderate decreases of 4 and 3 per cent respectively for precision and recall. In the more complex experiments, the drops in performance worsen, bringing the performance of the metrics to 60 points (precision) and 72 points (recall). These final drops bring the F-Score of the algorithm to 66 points. Due to these initial drops, the only valid set of domains is the one obtained with non-noisy input data.

- **DriverLog.** PlanMiner-N (C45) has some noise resilience with the DriverLog domain. The precision drops 0.2 points when noise is included, maintaining these results until the experiments with 10% of noisy elements, and the recall drops 0.3 points until the aforementioned experiments. From these experiments onwards, the precision drops by 3% and 5% in the more complex experiments, while the recall drops by 2% and 3% in the more complex experiments. In the most complex experiments, the F-Score ends up at 74%. Of all the domains learned by the algorithm, none is valid.

- **ZenoTravel.** Initially, PlanMiner(C45) obtains 82% precision, which drops to 80% and 78% in the experiments with 3% and 5% noisy predicates. In these experiments, the recall decreases by 1 point. In the experiments with 10% noisy elements, the performance suffers a steeper drop of 4 and 2 points in the metrics. The metrics in more complex experiments, the value of the metrics leads to the F-Score ending up at 77%. Finally, domain validity is not met in any of the domains learned.

### C.1.9 PlanMiner (RIPPER)

- **BlocksWorld.** PlanMiner (RIPPER) loses 12 points of precision out of 76 points obtained with non-noisy input data. This drop is repeated repeatedly as data is removed from the input plan traces in 5% steps, down to results of 47 precision points with the noisiest input data. Meanwhile, recall suffers an 8-point decrease in performance when noise is included, with subsequent experiments losing 1 per cent performance, except in experiments with 20 per cent noisy elements, where it drops 3 points. Due to this behaviour, the F-Score is 57 points in the most complex experiments, and none of the learned domains are valid.

- **Depots.** In experiments with 3 per cent noisy elements, precision drops by 9 per cent, while recall decreases by 7 per cent. Up to the experimentation with 10 per cent noisy data, the metrics drop much more slightly, with a 3 per cent and 1 per cent loss in precision respectively. In the more complex experiments, the algorithm again suffers a significant loss of performance, with

Table C.9: PlanMiner (RIPPER) Results

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.7672 | 0.0453 | 0.8009 | 0.0284 | 0.7863 | 0.0361 |
| | 3% | 0.6492 | 0.0898 | 0.7212 | 0.0606 | 0.6811 | 0.0706 |
| | 5% | 0.5949 | 0.1024 | 0.7197 | 0.0858 | 0.6524 | 0.0972 |
| | 10% | 0.5584 | 0.1300 | 0.7034 | 0.0944 | 0.6336 | 0.1152 |
| | 20% | 0.4792 | 0.1549 | 0.6794 | 0.1132 | 0.5782 | 0.1367 |
| Depots | 0% | 0.7454 | 0.0532 | 0.7666 | 0.0384 | 0.7553 | 0.0445 |
| | 3% | 0.6559 | 0.0995 | 0.6985 | 0.0701 | 0.6747 | 0.0852 |
| | 5% | 0.6210 | 0.1006 | 0.6829 | 0.0882 | 0.6582 | 0.0997 |
| | 10% | 0.5655 | 0.1575 | 0.6443 | 0.1183 | 0.6047 | 0.1367 |
| | 20% | 0.5494 | 0.1794 | 0.6494 | 0.1344 | 0.5952 | 0.1584 |
| DriverLog | 0% | 0.7442 | 0.0103 | 0.7882 | 0.0006 | 0.7630 | 0.0089 |
| | 3% | 0.6833 | 0.0561 | 0.7603 | 0.0982 | 0.7273 | 0.0759 |
| | 5% | 0.6395 | 0.0685 | 0.7558 | 0.1058 | 0.6926 | 0.0852 |
| | 10% | 0.5772 | 0.0725 | 0.7285 | 0.1173 | 0.6587 | 0.0994 |
| | 20% | 0.5183 | 0.0998 | 0.6947 | 0.1590 | 0.6036 | 0.1253 |
| ZenoTravel | 0% | 0.7862 | 0.0333 | 0.8276 | 0.0045 | 0.8035 | 0.0287 |
| | 3% | 0.6282 | 0.0792 | 0.7181 | 0.0541 | 0.6637 | 0.0678 |
| | 5% | 0.5833 | 0.1063 | 0.7005 | 0.0657 | 0.6426 | 0.0849 |
| | 10% | 0.4598 | 0.1285 | 0.6182 | 0.0899 | 0.5371 | 0.1006 |
| | 20% | 0.3991 | 0.1636 | 0.5974 | 0.1085 | 0.4936 | 0.1363 |

Table C.10: PlanMiner (RIPPER) Results

metrics dropping to 54 (precision) and 64 (recall) points. The F-Score of the algorithm stabilises at around 59% in the experimentation with 10% of noisy elements and remains at these values until the end of the experimentation. The domains learned by PlanMiner (RIPPER ) are not valid.

- **DriverLog.** It shows a 23-point difference in precision between the results obtained with the noiseless data and those obtained with the noisier input data. The largest losses in performance are in the experimentation with 3% noisy data (a drop of 6 points) and the experimentation with 3% noisy data (a drop of 6 points). Recall shows drops of 2-3% across the experiments, but no noticeable decrease in any of the experiments. The behaviour of recall means that the F-Score buffers the performance drops suffered by precision. Nevertheless, its value in the most complex experiments is 60 points, and no domain obtained with the algorithm is valid.

- **ZenoTravel.** Throughout the experimentation, the precision of the algorithm drops steadily. Initially, it suffers a 16-point loss in precision when noise is included in the input data, but in later experiments, these drops amount to a loss of up to 8 points in some experiments. Recall, on the other hand, exhibits two large performance losses, followed by intervals of stabilisation of the metric. The drops are found in the experiments with 3% and 10% of noisy items. This behaviour leads to the F-Score dropping steadily throughout the experimental process, with two large drops accentuated in

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 0.7681 | 0.0462 | 0.8023 | 0.0285 | 0.7863 | 0.0361 |
| | 3% | 0.7098 | 0.0591 | 0.7640 | 0.0363 | 0.7300 | 0.0441 |
| | 5% | 0.6744 | 0.0788 | 0.7551 | 0.0383 | 0.7182 | 0.0571 |
| | 10% | 0.6329 | 0.0842 | 0.7494 | 0.0646 | 0.6843 | 0.0725 |
| | 20% | 0.5995 | 0.1323 | 0.7162 | 0.0938 | 0.6571 | 0.1117 |
| Depots | 0% | 0.7264 | 0.0561 | 0.7886 | 0.0362 | 0.7553 | 0.0445 |
| | 3% | 0.6694 | 0.0772 | 0.7840 | 0.0573 | 0.7157 | 0.0639 |
| | 5% | 0.6092 | 0.1003 | 0.7802 | 0.0888 | 0.6935 | 0.0925 |
| | 10% | 0.5993 | 0.1293 | 0.7782 | 0.1073 | 0.6861 | 0.1159 |
| | 20% | 0.5011 | 0.1485 | 0.7073 | 0.1246 | 0.6046 | 0.1396 |
| DriverLog | 0% | 0.7502 | 0.0093 | 0.7749 | 0.0074 | 0.7630 | 0.0089 |
| | 3% | 0.7385 | 0.0482 | 0.7559 | 0.0239 | 0.7462 | 0.0350 |
| | 5% | 0.7262 | 0.0649 | 0.7483 | 0.0494 | 0.7351 | 0.0512 |
| | 10% | 0.6895 | 0.0952 | 0.7445 | 0.0528 | 0.7175 | 0.0746 |
| | 20% | 0.6895 | 0.0952 | 0.7445 | 0.0528 | 0.7175 | 0.0746 |
| ZenoTravel | 0% | 0.7845 | 0.0362 | 0.8263 | 0.0146 | 0.8035 | 0.0287 |
| | 3% | 0.7482 | 0.0572 | 0.8085 | 0.0383 | 0.7736 | 0.0402 |
| | 5% | 0.7192 | 0.0603 | 0.7932 | 0.0473 | 0.7528 | 0.0526 |
| | 10% | 0.6500 | 0.0792 | 0.7782 | 0.0374 | 0.7172 | 0.0588 |
| | 20% | 0.6142 | 0.1092 | 0.7349 | 0.0602 | 0.6792 | 0.0821 |

Table C.11: PlanMiner-N (RIPPER) Results

the experiments indicated above. The F-Score of the domains obtained in the most complex experiments is 49%, and none of the domains learned by PlanMiner (RIPPER) is valid.

### C.1.10 PlanMiner-N (RIPPER)

- **BlocksWorld.** Throughout the experimentation, there are repeated drops in precision and recall of 4 or 5 points and 1 or 2 points, respectively, as the experimental conditions are tightened. The values of the metrics in the most complex experiments are 59 points (precision) and 71 points (recall), while the F-Score performs 65%. Of all the domains learned by the algorithm, none is valid.

- **Depots.** At the beginning of the experimental process, the algorithm suffers a 6-point loss of precision while recall remains unchanged. The metrics maintain some resilience to noise in later experiments up to the more complex ones. In these experiments, precision again suffers a 9-point drop, and recall suffers an 8-point drop. The F-Score ranges from 75% in the experiments with all data to 60% in the experiments with higher levels of noise. Due to the behaviour of the other metrics, the biggest loss of performance occurs in the final experiments and accounts for half of the lost F-Score. No domain learned by the algorithm is valid.

- **DriverLog.** Even in experiments with 10% of noisy elements, precision and recall drop by 3 points. In later experiments, precision drops by 5 points in

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 0.8923 | 0.0400 | 0.9164 | 0.0405 | 0.9034 | 0.0327 |
| | 5% | 0.7012 | 0.0602 | 0.8983 | 0.0805 | 0.8011 | 0.0595 |
| | 10% | 0.6677 | 0.0645 | 0.8787 | 0.0854 | 0.7766 | 0.0612 |
| | 20% | 0.6384 | 0.0723 | 0.8285 | 0.0851 | 0.7602 | 0.0666 |
| Depots | 0% | 0.9724 | 0.0000 | 1.0000 | 0.0000 | 0.9863 | 0.0000 |
| | 3% | 0.6856 | 0.0750 | 0.7211 | 0.1416 | 0.6947 | 0.0324 |
| | 5% | 0.6674 | 0.0919 | 0.7145 | 0.0397 | 0.6404 | 0.0378 |
| | 10% | 0.5645 | 0.0555 | 0.7074 | 0.0358 | 0.6294 | 0.0598 |
| | 20% | 0.4912 | 0.0289 | 0.6355 | 0.0814 | 0.5828 | 0.0596 |
| DriverLog | 0% | 0.9500 | 0.0000 | 1.0000 | 0.0000 | 0.9750 | 0.0000 |
| | 3% | 0.8642 | 0.0280 | 0.8381 | 0.0385 | 0.8544 | 0.0287 |
| | 5% | 0.7228 | 0.0438 | 0.7939 | 0.0428 | 0.7550 | 0.0433 |
| | 10% | 0.6430 | 0.0794 | 0.7821 | 0.0745 | 0.7217 | 0.0628 |
| | 20% | 0.6272 | 0.0828 | 0.7696 | 0.1154 | 0.7015 | 0.0853 |
| ZenoTravel | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 0.6925 | 0.0882 | 0.8679 | 0.0479 | 0.7451 | 0.0492 |
| | 5% | 0.6538 | 0.0659 | 0.8472 | 0.1236 | 0.7230 | 0.0739 |
| | 10% | 0.6382 | 0.0585 | 0.7358 | 0.0765 | 0.7111 | 0.0831 |
| | 20% | 0.6310 | 0.0582 | 0.6931 | 0.1194 | 0.6577 | 0.0918 |

Table C.12: PlanMiner (NSLV) Results

each set of experiments, while recall remains stable at around 74%. From the experiments with 10% of noisy elements onwards, the F-Score shows resilience to noise, standing at around 71%. The domains obtained by the algorithm are not valid.

- **ZenoTravel.** The difference in precision between experiments with non-noisy data and those with a higher percentage of noisy elements in the input plan traces is 17 points. During the experimentation, the greatest loss of precision is found in the experimentation with 10 per cent noise, where it drops by 6 per cent. The difference in recall between the noiseless and noisier experiments is 9 per cent, with the greatest loss of performance (4 per cent) found in the more complex experiments. The F-Score of the learned domains ends up at 67 points, 13 points lower than in the noiseless experiments. In no experiment with the ZenoTravel domain is the validity of the learned domains fulfilled.

### C.1.11 PlanMiner (NSLV)

- **BlocksWorld.** The precision drops from 100 to 89 when noise is included, while recall drops from 100 to 91. The precision again suffers a severe loss of performance in the experiments with 5% of noisy elements, which accounts for 19% of the algorithm's performance. In subsequent experiments, the algorithm drops by 3 per cent at each step of the experimental process. The recall, on the other hand, maintains certain stability until the most complex

experiments, decreasing only by 1/2% until this experimentation. In the experiments with a higher percentage of noisy elements, the recall loses 5% of its performance. The F-Score presents a difference of 24 points between the best results (100% F-Score using non-noisy elements) and the worst results (76% F-Score in the most complex experimentation), this drop is mainly concentrated in the experiments with 3% and 5% of noisy elements, where it drops by 10% in each of them. With non-noisy input data, the results of the perfect metrics, so the validity is fulfilled in them, but, from these experiments onwards the validity is not fulfilled.

- **Depots.** The algorithm loses 29 precision points and 27 recall points by including some noise in the input data. With 10 per cent of noisy data, the precision drops again by 10 per cent, while recall only drops significantly again (by 7 points) in the more complex experiments. After an initial drop of 28 F-Score points in the experiments using information with 3% noisy elements, the algorithm contains its performance loss, showing decreases of 4% in each step of the experimental process. The only domains learned by the algorithm that are valid are those learned with non-noisy data.

- **DriverLog.** Initially, precision and recall decrease by 9 points and 17 points respectively. In later experiments, precision decreases by 14 and 8 points, while recall maintains a more moderate decreasing behaviour (losing 7 of performance in total until the end of the experiment). In the most complex experiments, the F-Score stands at 70 points, 27 points below its initial experimental value. Due to this difference in performance, the validity of the domains is only fulfilled for those obtained from non-noisy data.

- **ZenoTravel.** PlanMiner(NSLV) obtains perfect results for both precision and recall with non-noisy input data, but, by including some noise, both metrics suffer a serious loss of performance. This drop means a 31-point decrease in precision and a 14-point decrease in recall. As the complexity of the experiments increases, the metrics continue to drop much more moderately. On the one hand, the precision of the domains drops by about 3% at each step of the experimental process, while, on the other hand, the recall drops by 2% (with the exception of a 9-point drop in the experiments with 10% noise in the plan traces). This leads to an F-Score in the most complex experiments of 65 points, 35 points below its initial value, and only domains obtained with non-noisy data are valid.

### C.1.12 PlanMiner-N (NSLV)

- **BlocksWorld.** PlanMiner-N (NSLV) presents no problem in learning planning domains from noisy data, presenting perfect results throughout the experimentation and obtaining valid domains at all stages of the experimental process.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| BlocksWorld | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 5% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 20% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| Depots | 0% | 0.9729 | 0.0000 | 1.0000 | 0.0000 | 0.9862 | 0.0000 |
| | 3% | 0.9726 | 0.0282 | 1.0000 | 0.0000 | 0.9867 | 0.0222 |
| | 5% | 0.9721 | 0.0434 | 1.0000 | 0.0000 | 0.9869 | 0.0501 |
| | 10% | 0.9702 | 0.0686 | 1.0000 | 0.0000 | 0.9852 | 0.0618 |
| | 20% | 0.9695 | 0.0688 | 1.0000 | 0.0000 | 0.9845 | 0.0862 |
| DriverLog | 0% | 0.9505 | 0.0000 | 1.0000 | 0.0000 | 0.9755 | 0.0000 |
| | 3% | 0.9505 | 0.0000 | 1.0000 | 0.0000 | 0.9755 | 0.0000 |
| | 5% | 0.9473 | 0.0223 | 1.0000 | 0.0000 | 0.9747 | 0.0132 |
| | 10% | 0.9396 | 0.0336 | 1.0000 | 0.0000 | 0.9733 | 0.0186 |
| | 20% | 0.9327 | 0.0411 | 0.9867 | 0.0983 | 0.9648 | 0.0537 |
| ZenoTravel | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 1.0000 | 0.0206 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 5% | 1.0000 | 0.0287 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 10% | 0.9604 | 0.0348 | 1.0000 | 0.0000 | 0.9846 | 0.0187 |
| | 20% | 0.9102 | 0.0572 | 1.0000 | 0.0000 | 0.9553 | 0.0262 |

Table C.13: PlanMiner-N (NSLV) Results

- **Depots.** Even in the most complex experimentation, the algorithm shows no noticeable loss of precision or recall (the latter with values of 100% throughout the experimental process). In the experimentation with 20% of noisy input data, the precision drops 0.3 points. This slight loss of precision affects the F-Score by causing it to lose 0.2 points from its initial value but does not affect the validity of the learned domains.

- **DriverLog.** With experimentation with 20% of noisy elements, the precision drops by 2%. This is not the only loss of performance that the algorithm suffers in the experimentation, but, in the previous experiments, the loss is only 1 point of the metric in total. The recall also drops by 2 points in the more complex experiments, but unlike the precision, the recall remains unchanged for the rest of the experiment. PlanMiner-N (NSLV) has no problems in learning valid domains, except in the experiments with a higher percentage of noisy items, where the decrease in recall negatively affects the validity of the items.

- **ZenoTravel.** The algorithm obtains perfect results in all experimentation up to the most complex experiments. In these experiments, the precision drops by 4 points, although the recall is not affected. Both F-Score and validity are slightly affected by the loss of precision, the former dropping 3 points in the most complex experiments, and making the domains obtained in these experiments invalid.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.8946 | 0.0000 | 0.9124 | 0.0000 | 0.9129 | 0.0000 |
| | 3% | 0.7471 | 0.0621 | 0.8007 | 0.0459 | 0.7738 | 0.0575 |
| | 5% | 0.6144 | 0.1013 | 0.6936 | 0.0669 | 0.6532 | 0.0848 |
| | 10% | 0.4363 | 0.1224 | 0.5340 | 0.0613 | 0.4894 | 0.0927 |
| | 20% | 0.3078 | 0.1561 | 0.4116 | 0.1156 | 0.3610 | 0.1308 |
| DriverLog | 0% | 0.9029 | 0.0000 | 0.9345 | 0.0000 | 0.9224 | 0.0000 |
| | 3% | 0.6506 | 0.0811 | 0.7088 | 0.0393 | 0.6706 | 0.0632 |
| | 5% | 0.5821 | 0.0117 | 0.6582 | 0.0755 | 0.6228 | 0.0957 |
| | 10% | 0.5108 | 0.1294 | 0.6290 | 0.0884 | 0.5703 | 0.1096 |
| | 20% | 0.4174 | 0.1577 | 0.5334 | 0.1113 | 0.4677 | 0.1361 |
| Rovers | 0% | 0.8196 | 0.0000 | 0.8592 | 0.0000 | 0.8395 | 0.0000 |
| | 3% | 0.7520 | 0.1044 | 0.8493 | 0.0423 | 0.7999 | 0.0791 |
| | 5% | 0.7248 | 0.1179 | 0.8430 | 0.0507 | 0.7836 | 0.0829 |
| | 10% | 0.5900 | 0.1319 | 0.7562 | 0.0710 | 0.6767 | 0.1002 |
| | 20% | 0.5411 | 0.1649 | 0.6273 | 0.1084 | 0.5287 | 0.1347 |
| Satellite | 0% | 0.9505 | 0.0000 | 0.1000 | 0.0000 | 0.9740 | 0.0000 |
| | 3% | 0.6933 | 0.0554 | 0.7648 | 0.0257 | 0.7243 | 0.0477 |
| | 5% | 0.6564 | 0.0959 | 0.7582 | 0.0557 | 0.6958 | 0.7386 |
| | 10% | 0.5927 | 0.1158 | 0.7098 | 0.8533 | 0.6462 | 0.1001 |
| | 20% | 0.5006 | 0.1351 | 0.6439 | 0.9565 | 0.5727 | 0.1172 |
| ZenoTravel | 0% | 0.8511 | 0.0000 | 0.8758 | 0.0000 | 0.8688 | 0.0000 |
| | 3% | 0.5799 | 0.1195 | 0.6346 | 0.0790 | 0.6020 | 0.0927 |
| | 5% | 0.4930 | 0.1444 | 0.5746 | 0.1031 | 0.5383 | 0.1263 |
| | 10% | 0.4172 | 0.1530 | 0.5183 | 0.1142 | 0.4655 | 0.1319 |
| | 20% | 0.3381 | 0.1622 | 0.4564 | 0.1200 | 0.3998 | 0.1478 |

Table C.14: PlanMiner (C45) Results

## C.2 Numerical domains

### C.2.1 PlanMiner (C4.5)

- **Depots.** The algorithm loses 15 per cent precision in both the experimentation with 3 per cent noisy elements and the experimentation with 5 per cent noisy elements. These drops remain constant, but slightly more moderate during the rest of the experimentation until results in the more complex experimentation reach 30 per cent. Recall falls by around 10 per cent at each step of the experimental process, with the exception of the experiments with 10 per cent noisy elements, where it falls by 16 per cent. These results lead to an F-Score in the final experiments of 36 points. In the case of validity, the algorithm is able to learn valid domains using data with up to 3% noisy elements.

- **DriverLog.** Starting from an initial precision of 90 points and a recall of 93, the inclusion of noise in the plan traces causes the metrics to drop to 65 and 70 points respectively. For the remainder of the experimental process, the metrics continue to fall steadily but in a much more controlled manner. These drops cause the precision in the more complex experiments to be 41

points, while the recall value is 53%. In these experiments the F-Score is 46 points, half the F-Score value obtained using non-noisy input data. No domain learned using the algorithm is valid.

- **Rovers.** By including noisy elements in the input plan traces, the precision drops from 81 points to 75 points. This contrasts with the recall behaviour, which is only affected by 1 point in these experiments. In the experiments with 10% noisy data, both metrics suffer a significant decrease of 13% in the precision of the algorithm and 9% in its recall. The F-Score suffers from severe performance drops in the more complex experiments, with a value of 52% in these experiments. Finally, the validity of the learned domains is not met, regardless of the level of noise contained in the input plan traces.

- **Satellite.** The precision of the algorithm ranges from 95 points to 50 points throughout the experimentation. The most remarkable loss of precision (of 26 points) is found at the beginning of the experiment when noise is included in the plan traces. In later experiments, the precision continues to drop, but more steadily. This behaviour is repeated with recall, which initially drops by 24 points, but subsequently moderates its decreases in 5-point steps. The F-Score difference between the experimentation without noisy elements and the more complex experimentation is 40 points, and no domain obtained by the algorithm is valid.

- **ZenoTravel.** Similar to Satellite, the algorithm shows a large drop in performance initially, which moderates in subsequent experiments. The initial precision of the algorithm is 85 points, which drops to 57 points when noise is included and stands at 41 points for input data with 10 per cent noise and 33 points for the noisier experiments. Recall, on the other hand, drops by 24 points initially, which gradually decreases to 45 points. The F-Score shows results consistent with the behaviour of the metric, with values of 39 points in the most complex experiments. The algorithm is unable to learn valid planning domains.

### C.2.2 PlanMiner-N (C4.5)

- **Depots.** The algorithm loses a large amount of precision in experiments with 5% and 20% noise. The recall is not affected by noise in the first experiments, but from the experiments with 5% of noisy elements onwards, it drops to 14%. In the more complex experiments, the precision is around 54 points, while the recall is around 68. Due to the behaviour of the metrics, the F-Score initially drops by 3 per cent, and in the experiment with 5 per cent noise, it drops by an extra 14 per cent. These drops are repeated throughout the experimental process until the metric is at 53%. Given the initial drop in performance, subsequently learned domains are not valid.

- **DriverLog.** Starting from an initial precision and recall of 90 and 96 points respectively, the algorithm loses 18 and 10 points in these metrics throughout

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.8928 | 0.0000 | 0.9336 | 0.0000 | 0.9121 | 0.0000 |
| | 3% | 0.8456 | 0.0419 | 0.9244 | 0.0475 | 0.8888 | 0.0507 |
| | 5% | 0.6998 | 0.0780 | 0.7872 | 0.0520 | 0.7493 | 0.0634 |
| | 10% | 0.6395 | 0.0828 | 0.7583 | 0.0670 | 0.6903 | 0.0729 |
| | 20% | 0.5400 | 0.1201 | 0.6855 | 0.1091 | 0.6150 | 0.1107 |
| DriverLog | 0% | 0.9091 | 0.0000 | 0.9615 | 0.0000 | 0.9328 | 0.0000 |
| | 3% | 0.8638 | 0.0062 | 0.9570 | 0.0009 | 0.9022 | 0.0064 |
| | 5% | 0.8430 | 0.0217 | 0.9478 | 0.0290 | 0.8982 | 0.0253 |
| | 10% | 0.8087 | 0.0446 | 0.9290 | 0.0474 | 0.8628 | 0.0487 |
| | 20% | 0.7255 | 0.0690 | 0.8657 | 0.0662 | 0.7956 | 0.0691 |
| Rovers | 0% | 0.8122 | 0.0000 | 0.8524 | 0.0000 | 0.8391 | 0.0000 |
| | 3% | 0.7831 | 0.0785 | 0.8491 | 0.0395 | 0.8110 | 0.0527 |
| | 5% | 0.7716 | 0.0823 | 0.8325 | 0.0383 | 0.8033 | 0.0582 |
| | 10% | 0.7153 | 0.1081 | 0.7762 | 0.0418 | 0.7425 | 0.0770 |
| | 20% | 0.6656 | 0.1283 | 0.7626 | 0.0430 | 0.7181 | 0.0823 |
| Satellite | 0% | 0.9723 | 0.0000 | 0.9748 | 0.0000 | 0.9745 | 0.0000 |
| | 3% | 0.9265 | 0.0371 | 0.9426 | 0.0186 | 0.9255 | 0.0262 |
| | 5% | 0.9061 | 0.0452 | 0.9069 | 0.0202 | 0.9034 | 0.0316 |
| | 10% | 0.8615 | 0.0601 | 0.8643 | 0.0221 | 0.8682 | 0.0448 |
| | 20% | 0.8289 | 0.0999 | 0.8193 | 0.0392 | 0.8259 | 0.0692 |
| ZenoTravel | 0% | 0.8606 | 0.0000 | 0.8682 | 0.0000 | 0.8686 | 0.0000 |
| | 3% | 0.8481 | 0.0494 | 0.8423 | 0.0061 | 0.8423 | 0.0272 |
| | 5% | 0.8268 | 0.0564 | 0.8236 | 0.0172 | 0.8229 | 0.0366 |
| | 10% | 0.8137 | 0.0827 | 0.8173 | 0.0115 | 0.8183 | 0.0593 |
| | 20% | 0.7677 | 0.1102 | 0.7605 | 0.0323 | 0.7675 | 0.0739 |

Table C.15: PlanMiner-N (C45) Results

the experimentation. The biggest drop in performance is suffered by the metrics in the most complex experiments, but throughout the rest of the experiment, each one shows different behaviour. Where precision loses around 4% of performance as the complexity of the experiment increases, recall shows some resilience to noise, decreasing by 1 point at most. The F-Score of the algorithm drops more moderately than precision, until the most complex experimentation where it loses 5 per cent of its value. Despite the good initial results, no domain learned by the algorithm is valid.

- **Rovers.** precision drops 3 points when noise is included in the plan traces, in subsequent experiments the algorithm maintains some resilience to more complex experiments. In these experiments, the metric drops 6 points at each step of the experimental process. Recall, on the other hand, shows resilience to noise up to the experimentation with 10% of noisy elements. In the more complex experiments, the metrics are at 76 and 66 points, leading to the F-Score ending up with values of 71%. No domain learned by the algorithm is valid.
- **Satellite.** With precision results of 97% initially, the algorithm loses 5 performance points and 3 recall points by including noise in the input data. In subsequent experiments, precision drops by 4 per cent at each step of the experimental process, while recall drops by around 5 per cent in these experiments. As with precision and recall, the F-Score starts with results above 95 per cent and drops by 5 points when noise is included. These drops are repeated throughout the experimentation until the final F-Score results are 82%. No domain learned by the algorithm is valid.
- **ZenoTravel.** The difference between the results using non-noisy data and the results using the noisiest data is 10 points. Precision drops by 2 points, until the most complex experimentation, where it drops by 5 points. The recall behaves in the same way, decreasing in steps of 2% until the experiments with the highest possible number of noisy elements, where it decreases by 10%. In terms of F-Score, the algorithm performs 76% in the experiments with the highest noise levels, and in terms of validity, no domain learned by the algorithm is valid.

### C.2.3 PlanMiner (RIPPER)

- **Depots.** The algorithm loses 22% precision by including noise, while recall drops by 20%. In subsequent experiments, precision decreases by around 9 percent at each step of the experimental process. Recall shows a similar behaviour, but with more moderate drops (6 percent). These drops lead to the F-Score of the algorithm showing values of 44% in the most complex experiments. Severe drops in performance at the beginning of the experimentation cause the domains learned by the algorithm to be invalid.
- **DriverLog.** The initial precision of the algorithm is 88% and drops to 64% as the complexity of the experiments increases. The recall starts at 90 percent,

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.8554 | 0.0000 | 0.8715 | 0.0000 | 0.8659 | 0.0000 |
| | 3% | 0.6340 | 0.1281 | 0.6756 | 0.1002 | 0.6522 | 0.1191 |
| | 5% | 0.5551 | 0.1420 | 0.6174 | 0.1076 | 0.5897 | 0.1250 |
| | 10% | 0.4677 | 0.1553 | 0.5435 | 0.1189 | 0.5018 | 0.1395 |
| | 20% | 0.3961 | 0.1938 | 0.4993 | 0.1578 | 0.4444 | 0.1761 |
| DriverLog | 0% | 0.8895 | 0.0000 | 0.9001 | 0.0000 | 0.9001 | 0.0000 |
| | 3% | 0.6460 | 0.1745 | 0.7274 | 0.1384 | 0.6826 | 0.1529 |
| | 5% | 0.5206 | 0.1899 | 0.6444 | 0.1409 | 0.5877 | 0.1677 |
| | 10% | 0.3709 | 0.1948 | 0.5133 | 0.1505 | 0.4389 | 0.1795 |
| | 20% | 0.3030 | 0.2024 | 0.4642 | 0.1613 | 0.3842 | 0.1832 |
| Rovers | 0% | 0.8382 | 0.0000 | 0.8591 | 0.0000 | 0.8449 | 0.0000 |
| | 3% | 0.6889 | 0.0634 | 0.7219 | 0.0485 | 0.7051 | 0.0554 |
| | 5% | 0.6011 | 0.0971 | 0.6685 | 0.0791 | 0.6306 | 0.0872 |
| | 10% | 0.5393 | 0.1386 | 0.6119 | 0.9653 | 0.5772 | 0.1167 |
| | 20% | 0.4407 | 0.1553 | 0.5447 | 0.1151 | 0.4992 | 0.1381 |
| Satellite | 0% | 0.9253 | 0.0000 | 0.9424 | 0.0000 | 0.9425 | 0.0000 |
| | 3% | 0.6670 | 0.0858 | 0.7553 | 0.0476 | 0.7038 | 0.0627 |
| | 5% | 0.5981 | 0.1135 | 0.7512 | 0.0773 | 0.6711 | 0.0994 |
| | 10% | 0.4259 | 0.1469 | 0.6665 | 0.1002 | 0.5427 | 0.1258 |
| | 20% | 0.3246 | 0.1868 | 0.6453 | 0.1195 | 0.4885 | 0.1482 |
| ZenoTravel | 0% | 0.8333 | 0.0000 | 0.8702 | 0.0000 | 0.8559 | 0.0000 |
| | 3% | 0.3829 | 0.1308 | 0.4650 | 0.1924 | 0.4243 | 0.1646 |
| | 5% | 0.3476 | 0.1348 | 0.4636 | 0.1969 | 0.4031 | 0.1664 |
| | 10% | 0.3130 | 0.1396 | 0.4595 | 0.1986 | 0.3966 | 0.1698 |
| | 20% | 0.2029 | 0.1516 | 0.4022 | 0.2114 | 0.3005 | 0.1834 |

Table C.16: PlanMiner (RIPPER) Results

but drops to 72 percent when it encounters noisy elements in the input plan traces. Both metrics continue to decrease throughout the experimentation, reaching 30 and 46 points for each metric respectively. The F-Score shows a difference of 52 points between experiments with non-noisy data and experiments with a higher percentage of noisy items. Finally, despite the initial values of the metrics, no domain learned by the algorithm is valid.

- **Rovers.** The algorithm shows a difference of 39 points between the best and worst precision results, with the largest decrease in precision (of 15%) being found when noise is included, and the precision keeps dropping throughout the experimentation. The recall suffers an initial drop of 13 points, which is repeated throughout the experimental process in steps of 7 points at each step. This leads to the F-Score oscillating from 84% to 49% throughout the experiment, and the only set of domains learned are those obtained with non-noisy data.
- **Satellite.** With precision results of 92% and recall of 94% initially, the algorithm loses 26 and 19 performance points respectively when noise is included. Although the subsequent performance drops are more moderate, in the more complex experiments the precision ends up at 32 points, while the recall shows results of 64 points. The F-Score starts with results of 94% and drops 24 points when noise is included, during the rest of the experimentation the decrease in performance causes the algorithm to show a value of 48%. The algorithm has no problem learning planning domains from non-noisy data, but when noise is included in the plan traces, it cannot maintain the validity of the domains learned.
- **ZenoTravel.** The metrics drop steadily throughout the experimentation. On the one hand, precision shows a difference of 63 points between domains learned with non-noisy data and those learned with data with the highest percentage of noisy information. Of these 63 points, 50 points are lost when noise is included in the plan traces. On the other hand, recall has an initial performance loss of 41 points, and in the more complex experiments, the value of the metric is 47%. Finally, the F-Score results drop 55 points throughout the experimentation, and the domains learned by the algorithm are not valid.

### C.2.4 PlanMiner-N (RIPPER)

- **Depots.** The algorithm remains stable until experimentation with 10% of noisy elements in the input data. From these experiments, the precision drops 6 points, and the recall drops 4 points. In the more complex experiments, the metrics are 70 and 78 points respectively. The F-Score shows a difference of 12 points with respect to its initial experimental results, but none of the domains learned by the algorithm are valid.
- **DriverLog.** The precision of the algorithm is 88% in the noiseless experiments, which drops to 71% as the complexity of the experiments increases,

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.8390 | 0.0000 | 0.8916 | 0.0000 | 0.8658 | 0.0000 |
| | 3% | 0.8390 | 0.0000 | 0.8916 | 0.0000 | 0.8658 | 0.0000 |
| | 5% | 0.8390 | 0.0000 | 0.8916 | 0.0000 | 0.8658 | 0.0000 |
| | 10% | 0.7764 | 0.0494 | 0.8555 | 0.0226 | 0.8125 | 0.0386 |
| | 20% | 0.7091 | 0.1220 | 0.7818 | 0.1047 | 0.7499 | 0.1178 |
| DriverLog | 0% | 0.8870 | 0.0000 | 0.9224 | 0.0000 | 0.9001 | 0.0000 |
| | 3% | 0.8315 | 0.1004 | 0.8908 | 0.0438 | 0.8609 | 0.0713 |
| | 5% | 0.8197 | 0.1057 | 0.8976 | 0.0472 | 0.8598 | 0.0737 |
| | 10% | 0.7719 | 0.1107 | 0.8408 | 0.0653 | 0.8139 | 0.0862 |
| | 20% | 0.7163 | 0.1281 | 0.8156 | 0.0685 | 0.7695 | 0.0984 |
| Rovers | 0% | 0.8248 | 0.0000 | 0.8697 | 0.0000 | 0.8448 | 0.0000 |
| | 3% | 0.8012 | 0.0316 | 0.8467 | 0.0163 | 0.8297 | 0.0247 |
| | 5% | 0.7867 | 0.0445 | 0.8454 | 0.0243 | 0.8153 | 0.0354 |
| | 10% | 0.7436 | 0.0617 | 0.8065 | 0.0271 | 0.7733 | 0.0460 |
| | 20% | 0.7045 | 0.0970 | 0.7667 | 0.0345 | 0.7397 | 0.0681 |
| Satellite | 0% | 0.9236 | 0.0000 | 0.9609 | 0.0000 | 0.9424 | 0.0000 |
| | 3% | 0.8903 | 0.0409 | 0.9328 | 0.0027 | 0.9147 | 0.0269 |
| | 5% | 0.8693 | 0.0572 | 0.9092 | 0.0540 | 0.8895 | 0.0319 |
| | 10% | 0.8274 | 0.0632 | 0.8719 | 0.0650 | 0.8523 | 0.0334 |
| | 20% | 0.7626 | 0.0718 | 0.8469 | 0.0935 | 0.8006 | 0.0460 |
| ZenoTravel | 0% | 0.8270 | 0.0000 | 0.8837 | 0.0000 | 0.8558 | 0.0000 |
| | 3% | 0.7984 | 0.0315 | 0.8514 | 0.0137 | 0.8263 | 0.0241 |
| | 5% | 0.7635 | 0.0497 | 0.8228 | 0.0232 | 0.7964 | 0.0312 |
| | 10% | 0.7132 | 0.0462 | 0.8116 | 0.0268 | 0.7619 | 0.0389 |
| | 20% | 0.6491 | 0.0525 | 0.7401 | 0.0353 | 0.6995 | 0.0457 |

Table C.17: PlanMiner-N (RIPPER) Results

with the greatest loss of precision concentrated in the last experiments, a drop of 6 points. Meanwhile, the recall shows a slight loss of 3% up to the experiments with the highest number of noisy elements. In the latter experiments, the drop in performance is 5 points and places the recall of the algorithm at 81%. The F-Score gradually drops 3/4 points until the experimentation with 20% of noisy elements in the input plan traces. Although the performance drops are slight compared to other experiments, no domain learned by the algorithm is valid.

- **Rovers.** PlanMiner-N (RIPPER) maintains a constant precision decrease of 2 points until the most complex experiments, where it drops by 6 points. Recall, on the other hand, loses 2 performance points but remains unchanged up to the experiments with 10% of noisy elements. In the more complex experiments, recall loses 4 performance points at each step of the experimental process, which brings the F-Score in these experiments to 73 points. Although the F-Score difference is less than 10%, once the noise is included, the domains obtained with the algorithm are no longer valid.

- **Satellite.** Initially, the algorithm starts with precision results of 92% and recall results of 96%. When noise is included in the plan traces, the precision and recall decrease by 3 points and these drops are reiterated throughout the rest of the experimentation up to 76 and 84 points in the most complex experiments. Consequently, the F-Score starts at 94 points, and at each step of the experimental process, it drops by 3% to 80 points. PlanMiner-N (RIPPER) is able to obtain valid domains with input data with 3% and 5% of noisy information.

- **ZenoTravel.** On the one hand, the algorithm shows a difference in precision between experiments with noise-free data and those with the highest percentage of noisy data of 18 points. Throughout the experimental process, the algorithm loses precision in steps of 3 points, except in the most complex experiments where it loses 6 points. On the other hand, the recall of the algorithm drops 14 points throughout the experimentation, with decreases of 3% as the complexity of the experiments increases. The F-Score stands at 69 points in the most complex experiments, and none of the domains learned by the algorithm is valid.

### C.2.5 PlanMiner (NSLV)

- **Depots.** PlanMiner's precision drops 22% when there is some noise in the input tracks. For the noisiest experiments, this figure drops to 64%. PlanMiner has a recall of 100 per cent with the noiseless data, but when there is some noise, that value drops to 78 per cent. The results remain stable up to 20% noise and fall to around 2% with the increasing complexity of the experiments. PlanMiner does not get valid planning domains when there is noise and presents F-Score results below 75 per cent when it encounters some noise.

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.9490 | 0.0000 | 1.0000 | 0.0000 | 0.9633 | 0.0000 |
| | 3% | 0.7244 | 0.0216 | 0.7812 | 0.0686 | 0.7450 | 0.0460 |
| | 5% | 0.7231 | 0.0281 | 0.7718 | 0.0723 | 0.7364 | 0.0475 |
| | 10% | 0.7102 | 0.0419 | 0.7679 | 0.0768 | 0.7067 | 0.0493 |
| | 20% | 0.6461 | 0.0478 | 0.6625 | 0.0787 | 0.6901 | 0.0491 |
| DriverLog | 0% | 0.9182 | 0.0000 | 0.9807 | 0.0000 | 0.9487 | 0.0000 |
| | 3% | 0.7677 | 0.0403 | 0.6402 | 0.0375 | 0.6788 | 0.0452 |
| | 5% | 0.6942 | 0.0625 | 0.6407 | 0.0601 | 0.6655 | 0.0484 |
| | 10% | 0.6354 | 0.0645 | 0.6139 | 0.0928 | 0.6374 | 0.0691 |
| | 20% | 0.5725 | 0.1158 | 0.5403 | 0.1023 | 0.5535 | 0.0912 |
| Rovers | 0% | 0.7756 | 0.0000 | 1.0000 | 0.0000 | 0.8734 | 0.0000 |
| | 3% | 0.3574 | 0.0401 | 0.7063 | 0.0317 | 0.4635 | 0.0336 |
| | 5% | 0.2968 | 0.0563 | 0.6601 | 0.0610 | 0.4174 | 0.0402 |
| | 10% | 0.2918 | 0.0249 | 0.6554 | 0.0617 | 0.4024 | 0.0427 |
| | 20% | 0.2711 | 0.0375 | 0.6341 | 0.0353 | 0.3796 | 0.0646 |
| Satellite | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 0.6818 | 0.0535 | 0.4930 | 0.0435 | 0.5694 | 0.0565 |
| | 5% | 0.6141 | 0.0732 | 0.4403 | 0.0686 | 0.4909 | 0.0594 |
| | 10% | 0.5553 | 0.0857 | 0.4072 | 0.0688 | 0.4872 | 0.0695 |
| | 20% | 0.4865 | 0.0895 | 0.4077 | 0.0896 | 0.4609 | 0.0777 |
| ZenoTravel | 0% | 0.8127 | 0.0000 | 1.0000 | 0.0000 | 0.8969 | 0.0000 |
| | 3% | 0.4928 | 0.0433 | 0.6157 | 0.0587 | 0.5464 | 0.0453 |
| | 5% | 0.4758 | 0.0439 | 0.5543 | 0.0611 | 0.5107 | 0.0567 |
| | 10% | 0.4478 | 0.0567 | 0.5154 | 0.0804 | 0.4774 | 0.0613 |
| | 20% | 0.4358 | 0.0701 | 0.4855 | 0.0890 | 0.4588 | 0.0655 |

Table C.18: PlanMiner (NSLV) Results

- **DriverLog.** PlanMiner loses 34 points of precision on noisy data. This loss increases further as the complexity of the experiments increases. If the proportion of noisy elements in the entered plan curves is 20 per cent, PlanMiner's recall drops to 57 per cent. PlanMiner's recall is between 64% and 54%, depending on the complexity of the experiment. Compared to the recall results without noise, PlanMiner has a recall loss of 34% at best. As a result, the F-Score of the domains drops sharply when some noise is added and shows results of 67% (from the original 94.8% in the experiment without noise). These results deteriorate to 55% in the more complex experiments. No range is valid as soon as noise is added.

- **Rovers.** When PlanMiner is confronted with noise, the precision drops to 35% of the original value of 77%. This shows the same problem as in the experiments with outliers. As a result, PlanMiner achieves an precision of 27% in the experiments with more noisy data. The recall metrics improve compared to the precision metrics. However, PlanMiner loses 30 recall points when noise is included. After this sharp drop in recall, PlanMiner worsens the results in a more stable manner and finally shows a recall of 63%. As expected, if noise is included in the input data, the F-Score results drop dramatically from 87% to 46%. In terms of validity, this means that no noise learned domain is valid.

- **Satellite.** Following the same pattern as in the previous experiments, PlanMiner suffers from a severe loss of precision when it is confronted with noise. This loss is reflected in a decrease from 100% to 68% when noise is added. As the experiments proceed, the precision results drop to 61%, 55% and 48% in each individual case. PlanMiner's recall drops to 50 per cent when there is noise in the input data. This downward trend stabilizes in the experiments with higher noise. As expected, F-Score results decrease 56% as the noise is present and gradually decrease to 46% as the complexity of the experiments increases.

- **ZenoTravel.** With 3% noisy input data, PlanMiner loses 32% of its precision. These noise values gradually decrease and stabilize at around 43% precision for the noisiest input data. As with the precision, the recall of PlanMiner is also around 61%, with the initial drop in the very steep score being large and stabilizing in the further course of the experiments. The results of the F-Score show a logical trend with respect to the results of the other metrics, with a large initial drop in the metric.

### C.2.6 PlanMiner-N (NSLV)

- **Depots.** PlanMiner-N loses some precision when noise is included and drops to 93%, but keeps these results until the experiments with input data with 10% noisy elements. As the complexity of the experiments increases, the precision drops to 85%. The recall values are between 96% and 88%. The greatest dips are found in the transition from non-noisy examples to examples with 3% and

| Domain | Noise | $\mu$Precision | $\sigma$Precision | $\mu$Recall | $\sigma$Recall | $\mu$F-Score | $\sigma$F-Score |
|---|---|---|---|---|---|---|---|
| Depots | 0% | 0.9492 | 0.0000 | 1.0000 | 0.0000 | 0.9637 | 0.0000 |
| | 3% | 0.9315 | 0.0742 | 0.9632 | 0.0381 | 0.9467 | 0.0700 |
| | 5% | 0.9315 | 0.0742 | 0.9275 | 0.1064 | 0.9295 | 0.0913 |
| | 10% | 0.8722 | 0.0886 | 0.9009 | 0.1063 | 0.8827 | 0.0919 |
| | 20% | 0.8489 | 0.1060 | 0.8847 | 0.1597 | 0.8668 | 0.1007 |
| DriverLog | 0% | 0.9186 | 0.0000 | 0.9802 | 0.0000 | 0.9483 | 0.0000 |
| | 3% | 0.9108 | 0.0300 | 0.9734 | 0.0150 | 0.9467 | 0.0181 |
| | 5% | 0.9017 | 0.0495 | 0.9534 | 0.0691 | 0.9266 | 0.0415 |
| | 10% | 0.8502 | 0.0454 | 0.9474 | 0.0760 | 0.9035 | 0.0475 |
| | 20% | 0.8139 | 0.0827 | 0.9239 | 0.0994 | 0.8687 | 0.0727 |
| Rovers | 0% | 0.7754 | 0.0000 | 1.0000 | 0.0000 | 0.8730 | 0.0000 |
| | 3% | 0.7575 | 0.0240 | 0.9067 | 0.0318 | 0.8312 | 0.0333 |
| | 5% | 0.7565 | 0.0371 | 0.8609 | 0.0358 | 0.8173 | 0.0406 |
| | 10% | 0.6910 | 0.0400 | 0.8550 | 0.0613 | 0.8022 | 0.0423 |
| | 20% | 0.6712 | 0.0565 | 0.8349 | 0.0616 | 0.7791 | 0.0645 |
| Satellite | 0% | 1.0000 | 0.0000 | 1.0000 | 0.0000 | 1.0000 | 0.0000 |
| | 3% | 0.9725 | 0.0116 | 0.9734 | 0.0168 | 0.9722 | 0.0159 |
| | 5% | 0.9725 | 0.0116 | 0.9734 | 0.0168 | 0.9722 | 0.0159 |
| | 10% | 0.9320 | 0.0212 | 0.9732 | 0.0169 | 0.9528 | 0.0196 |
| | 20% | 0.8581 | 0.0347 | 0.9584 | 0.0215 | 0.8831 | 0.0295 |
| ZenoTravel | 0% | 0.8129 | 0.0000 | 1.0000 | 0.0000 | 0.8962 | 0.0000 |
| | 3% | 0.8089 | 0.0168 | 0.9774 | 0.0278 | 0.8927 | 0.0160 |
| | 5% | 0.7952 | 0.0279 | 0.9152 | 0.0440 | 0.8485 | 0.0286 |
| | 10% | 0.7720 | 0.0440 | 0.8775 | 0.0479 | 0.8193 | 0.0426 |
| | 20% | 0.7524 | 0.0506 | 0.8157 | 0.0755 | 0.7815 | 0.0594 |

Table C.19: PlanMiner-N (NSLV) Results

from 3% to 5%. The F-Score of PlanMiner-N in this area is always above 85%. If PlanMiner-N is faced with some noise, it will lose 1.5 per cent F-Score. In the following experiments, the F-score gradually decreases, resulting in invalid domains.

- **DriverLog.** In the first experiments, the precision decreased by 0.8%. In these more complex experiments, PlanMiner-N loses 5% and 3.5% of precision at 10% and 20% noise, respectively. The recall results follow the same trend as the precision results and lose 0.8% recall with 3% noise. At 5% and 10%, the measured values stabilize at around 95% recall. PlanMiner-N suffers a small additional drop at 20 per cent, which ultimately drops the domain recall to 92 per cent. The F-Score results are between 94% and 87%, which shows little variance in the first few experiments. Although the results are good, the loss of key elements in a certain action of 3% noise degrades the domains.

- **Rovers.** PlanMiner-N lowers the precision of the learned domains to 75 per cent with 3% and 5% noise. With 20 per cent noisy elements, the precision drops to 67 per cent. The recall of PlanMiner-N drops by about 10% when it is confronted with some noise. For the more complex experiments, the recall rate increases by more than 5%, albeit gradually. The end result of the experiment is a recall of 83.4%, 16.6% less than the noiseless data. The F-Score metric loses 5% F-Score when faced with noise. The loss of F-Score is 81% when using input data with 5% noisy elements and invalidates the domains.

- **Satellite.** PlanMiner-N shows itself to be noise-resistant at certain noise levels. The precision drops 3% when noise is added but stays at these values until 10% noisy data occurs. In more complex experiments, the precision of PlanMiner-N is around 85%. PlanMiner-N's recall remains unchanged until the noisiest data is used. In the most complex experiments, PlanMiner-N achieves a recall value of 95.8%. PlanMiner-N achieved an F-Score of 97.2% up to the experiments that use input data with 10% noise. From these noise values, the F-Score of the learned domains drops to 95.2% and to 88.3% for experiments with 10% and 20% noise. The domains are only invalid in the latter experiment.

- **ZenoTravel.** The precision of the results drops by 0.4 percentage points in the first experiments. In more complex experiments, PlanMiner-N loses 4.4% and 6% precision at 10% and 20% noise, respectively. PlanMiner-N keeps the domain recall between 97% and 81%, depending on the complexity of the experiments. The F-Score metric is 89.2% when noise is included. At 5% noise, the F-Score drops by a further 5%, which means that the domains are no longer valid. This decrease continues in the more complex experiments up to 78% F-Score.

# Appendix D

# PlanMiner-C's learned domains

## D.1 Bait domain

```
(define (domain Bait)

    (:requirements
        :equality
        :negative-preconditions
        :typing
    )

    (:types
        cell avatar - object
    )

    (:predicates
        (hasKey ?a - avatar)

        (conn_R ?c1 ?c2 - cell)
        (conn_L ?c1 ?c2 - cell)
        (conn_U ?c1 ?c2 - cell)
        (conn_D ?c1 ?c2 - cell)

        (wallIn ?c - cell)
        (exitIn ?c - cell)
        (keyIn ?c - cell)
        (holeIn ?c - cell)
    )

    (:functions
        (row ?c - cell)
        (column ?c - cell)

        (posX ?a - avatar)
        (posY ?a - avatar)
```

```
)

(:action actUp
    :parameters (?a - avatar ?c1 ?c2 ?c3 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and

            ;Caso 1: Empujar muro
            (when
                (and
                    (conn_U ?c1 ?c2)
                    (conn_U ?c2 ?c3)
                    (wallIn ?c2)
                    (not (wallIn ?c3))
                    (not (holeIn ?c3))
                    (not (keyIn ?c3))
                    (not (exitIn ?c3))
                )
                (and
                    (decrease (posX ?a) 1)
                    (not (wallIn ?c2))
                    (wallIn ?c3)
                )
            )

            ;Caso 2: Tapar agujero con muro
            (when
                (and
                    (conn_U ?c1 ?c2)
                    (conn_U ?c2 ?c3)
                    (wallIn ?c2)
                    (holeIn ?c3)
                )
                (and
                    (decrease (posX ?a) 1)
                    (not (wallIn ?c2))
                    (not (holeIn ?c3))
                )
            )

            ;Caso 3: Coger llave
            (when
                (and
                    (conn_U ?c1 ?c2)
                    (keyIn ?c2)
                    (not (wallIn ?c2))
```

```
                    (not (holeIn ?c2))

                )
                (and
                    (decrease (posX ?a) 1)
                    (not (keyIn ?c2))
                    (hasKey ?a)
                )
            )

            ;Caso 4: Moverse
            (when
                (and
                    (conn_U ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not (holeIn ?c2))
                    (not (keyIn ?c2))
                )
                (and
                    (decrease (posX ?a) 1)
                )
            )
        )
)

 (:action actDown
    :parameters (?a - avatar ?c1 ?c2 ?c3 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            ;Caso 1: Empujar muro
            (when
                (and
                    (conn_D ?c1 ?c2)
                    (conn_D ?c2 ?c3)
                    (wallIn ?c2)
                    (not (wallIn ?c3))
                    (not (holeIn ?c3))
                    (not (keyIn ?c3))
                    (not (exitIn ?c3))
                )
                (and
                    (increase (posX ?a) 1)
                    (not (wallIn ?c2))
                    (wallIn ?c3)
                )
            )
```

```
            ;Caso 2: Tapar agujero con muro
            (when
                (and
                    (conn_D ?c1 ?c2)
                    (conn_D ?c2 ?c3)
                    (wallIn ?c2)
                    (holeIn ?c3)
                )
                (and
                    (increase (posX ?a) 1)
                    (not (wallIn ?c2))
                    (not (holeIn ?c3))
                )
            )

            ;Caso 3: Coger llave
            (when
                (and
                    (conn_D ?c1 ?c2)
                    (keyIn ?c2)
                    (not (wallIn ?c2))
                    (not (holeIn ?c2))

                )
                (and
                    (increase (posX ?a) 1)
                    (not (keyIn ?c2))
                    (hasKey ?a)
                )
            )

            ;Caso 4: Moverse
            (when
                (and
                    (conn_D ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not (holeIn ?c2))
                    (not (keyIn ?c2))
                )
                (and
                    (increase (posX ?a) 1)
                )
            )
        )
)

 (:action actRight
    :parameters (?a - avatar ?c1 ?c2 ?c3 - cell)
    :precondition
        (and
            ( =
                (posX ?a)
                (row ?c1)
            )
            ( =
                (posY ?a)
```

```
                (column ?c1)
            )
        )
    :effect
        (and
            ;Caso 1: Empujar muro
            (when
                (and
                    (conn_R ?c1 ?c2)
                    (conn_R ?c2 ?c3)
                    (wallIn ?c2)
                    (not (wallIn ?c3))
                    (not (holeIn ?c3))
                    (not (keyIn ?c3))
                    (not (exitIn ?c3))
                )
                (and
                    (increase (posY ?a) 1)
                    (not (wallIn ?c2))
                    (wallIn ?c3)
                )
            )

            ;Caso 2: Tapar agujero con muro
            (when
                (and
                    (conn_R ?c1 ?c2)
                    (conn_R ?c2 ?c3)
                    (wallIn ?c2)
                    (holeIn ?c3)
                )
                (and
                    (increase (posY ?a) 1)
                    (not (wallIn ?c2))
                    (not (holeIn ?c3))
                )
            )

            ;Caso 3: Coger llave
            (when
                (and
                    (conn_R ?c1 ?c2)
                    (keyIn ?c2)
                    (not (wallIn ?c2))
                    (not (holeIn ?c2))

                )
                (and
                    (increase (posY ?a) 1)
                    (not (keyIn ?c2))
                    (hasKey ?a)
                )
            )

            ;Caso 4: Moverse
            (when
                (and
```

```
                    (conn_R ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not (holeIn ?c2))
                    (not (keyIn ?c2))
                )
                (and
                    (increase (posY ?a) 1)
                )
            )
        )
)

(:action actLeft
    :parameters (?a - avatar ?c1 ?c2 ?c3 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            ;Caso 1: Empujar muro
            (when
                (and
                    (conn_L ?c1 ?c2)
                    (conn_L ?c2 ?c3)
                    (wallIn ?c2)
                    (not (wallIn ?c3))
                    (not (holeIn ?c3))
                    (not (keyIn ?c3))
                    (not (exitIn ?c3))
                )
                (and
                    (decrease (posY ?a) 1)
                    (not (wallIn ?c2))
                    (wallIn ?c3)
                )
            )

            ;Caso 2: Tapar agujero con muro
            (when
                (and
                    (conn_L ?c1 ?c2)
                    (conn_L ?c2 ?c3)
                    (wallIn ?c2)
                    (holeIn ?c3)
                )
                (and
                    (decrease (posY ?a) 1)
                    (not (wallIn ?c2))
                    (not (holeIn ?c3))
```

```
                    )
                )

                ;Caso 3: Coger llave
                (when
                    (and
                        (conn_L ?c1 ?c2)
                        (keyIn ?c2)
                        (not (wallIn ?c2))
                        (not (holeIn ?c2))

                    )
                    (and
                        (decrease (posY ?a) 1)
                        (not (keyIn ?c2))
                        (hasKey ?a)
                    )
                )

                ;Caso 4: Moverse
                (when
                    (and
                        (conn_L ?c1 ?c2)
                        (not (wallIn ?c2))
                        (not (holeIn ?c2))
                        (not (keyIn ?c2))
                    )
                    (and
                        (decrease (posY ?a) 1)
                    )
                )
            )
    )

    (:action actUse
        :parameters (?a - avatar ?c1 ?c2 - cell)
        :precondition
            (and
                (=
                    (posX ?a)
                    (row ?c1)
                )
                (=
                    (posY ?a)
                    (column ?c1)
                )
            )
        :effect (and)
    )
)
```

Listing D.1: ZenoTravel numeric planning domain

## D.2 Zelda domain

```
(define (domain Zelda)

    (:requirements
        :equality
        :negative-preconditions
        :typing
    )

    (:types
        cell avatar - object
    )

    (:predicates
        (hasKey ?a - avatar)

        (orient_R ?a - avatar)
        (orient_L ?a - avatar)
        (orient_U ?a - avatar)
        (orient_D ?a - avatar)


        (conn_R ?c1 ?c2 - cell)
        (conn_L ?c1 ?c2 - cell)
        (conn_U ?c1 ?c2 - cell)
        (conn_D ?c1 ?c2 - cell)

        (wallIn ?c - cell)
        (exitIn ?c - cell)
        (keyIn ?c - cell)
        (monsterIn ?c - cell)
    )

    (:functions
        (row ?c - cell)
        (column ?c - cell)

        (posX ?a - avatar)
        (posY ?a - avatar)


    )

    (:action actUp
        :parameters (?a - avatar ?c1 ?c2 - cell)
        :precondition
            (and
                (=
                    (posX ?a)
                    (row ?c1)
                )
                (=
                    (posY ?a)
                    (column ?c1)
                )
            )
        :effect
            (and
```

```
(when
    (and
        (=
            (posX ?a)
            (row ?c2)
        )
        (=
            (posY ?a)
            (column ?c2)
        )
        (orient_R ?a)
        (not (orient_U  ?a))
    )
    (and
        (not (orient_R ?a))
        (orient_U ?a)
    )
)

(when
    (and
        (=
            (posX ?a)
            (row ?c2)
        )
        (=
            (posY ?a)
            (column ?c2)
        )
        (orient_L ?a)
        (not (orient_U ?a))
    )
    (and
        (not (orient_L ?a))
        (orient_U ?a)
    )
)

(when
    (and
        (=
            (posX ?a)
            (row ?c2)
        )
        (=
            (posY ?a)
            (column ?c2)
        )
        (orient_D ?a)
        (not (orient_U ?a))
    )
    (and
        (not (orient_D ?a))
        (orient_U ?a)
    )
)
```

```
            (when
                (and
                    (orient_U ?a)
                    (conn_U ?c1 ?c2)
                    (keyIn ?c2)
                    (not (wallIn ?c2))
                    (not  (monsterIn ?c2))
                )
                (and
                    (decrease (posX ?a) 1)
                    (not (keyIn ?c2))
                    (hasKey ?a)

                )
            )

            (when
                (and
                    (orient_U ?a)
                    (conn_U ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not  (monsterIn ?c2))
                    (not  (keyIn ?c2))
                )
                (and
                    (decrease (posX ?a) 1)
                )
            )
        )
)

 (:action actDown
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            (when
                (and
                    (=
                        (posX ?a)
                        (row ?c2)
                    )
                    (=
                        (posY ?a)
                        (column ?c2)
                    )
                    (orient_R ?a)
```

```
            (not (orient_D ?a))
        )
        (and
            (not (orient_R ?a))
            (orient_D ?a)
        )
    )

    (when
        (and
            (=
                (posX ?a)
                (row ?c2)
            )
            (=
                (posY ?a)
                (column ?c2)
            )
            (orient_L ?a)
            (not (orient_D ?a))
        )
        (and
            (not (orient_L ?a))
            (orient_D ?a)
        )
    )

    (when
        (and
            (=
                (posX ?a)
                (row ?c2)
            )
            (=
                (posY ?a)
                (column ?c2)
            )
            (orient_U ?a)
            (not (orient_D ?a))
        )
        (and
            (not (orient_U ?a))
            (orient_D ?a)
        )
    )

    (when
        (and
            (orient_D ?a)
            (conn_D ?c1 ?c2)
            (keyIn ?c2)
            (not (wallIn ?c2))
            (not  (monsterIn ?c2))
        )
        (and
            (increase (posX ?a) 1)
            (not (keyIn ?c2))
```

```
                    (hasKey ?a)

                )
            )

            (when
                (and
                    (orient_D ?a)
                    (conn_D ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not  (monsterIn ?c2))
                    (not  (keyIn ?c2))
                )
                (and
                    (increase (posX ?a) 1)
                )
            )
        )
)

 (:action actRight
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            ( =
                (posX ?a)
                (row ?c1)
            )
            ( =
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            (when
                (and
                    (=
                        (posX ?a)
                        (row ?c2)
                    )
                    (=
                        (posY ?a)
                        (column ?c2)
                    )
                    (orient_L ?a)
                    (not (orient_R ?a))
                )
                (and
                    (not (orient_L ?a))
                    (orient_R ?a)
                )
            )

            (when
                (and
                    (=
```

```
            (posX ?a)
            (row ?c2)
        )
        (=
            (posY ?a)
            (column ?c2)
        )
        (orient_U ?a)
        (not (orient_R ?a))
    )
    (and
        (not (orient_U ?a))
        (orient_R ?a)
    )
)

(when
    (and
        (=
            (posX ?a)
            (row ?c2)
        )
        (=
            (posY ?a)
            (column ?c2)
        )
        (orient_D ?a)
        (not (orient_R ?a))
    )
    (and
        (not (orient_D ?a))
        (orient_R ?a)
    )
)


(when
    (and
        (orient_R ?a)
        (conn_R ?c1 ?c2)
        (keyIn ?c2)
        (not (wallIn ?c2))
        (not  (monsterIn ?c2))
    )
    (and
        (increase (posY ?a) 1)
        (not (keyIn ?c2))
        (hasKey ?a)

    )
)

(when
    (and
        (orient_R ?a)
        (conn_R ?c1 ?c2)
        (not (wallIn ?c2))
```

```
                    (not  (monsterIn ?c2))
                    (not  (keyIn ?c2))
                )
                (and
                    (increase (posY ?a) 1)
                )
            )
        )
)

(:action actLeft
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            (when
                (and
                    (=
                        (posX ?a)
                        (row ?c2)
                    )
                    (=
                        (posY ?a)
                        (column ?c2)
                    )
                    (orient_R ?a)
                    (not (orient_L ?a))
                )
                (and
                    (not (orient_R ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (=
                        (posX ?a)
                        (row ?c2)
                    )
                    (=
                        (posY ?a)
                        (column ?c2)
                    )
                    (orient_D ?a)
                    (not (orient_L ?a))
                )
```

```
                (and
                    (not (orient_D ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (=
                        (posX ?a)
                        (row ?c2)
                    )
                    (=
                        (posY ?a)
                        (column ?c2)
                    )
                    (orient_U ?a)
                    (not (orient_L ?a))
                )
                (and
                    (not (orient_U ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (keyIn ?c2)
                    (not (wallIn ?c2))
                    (not  (monsterIn ?c2))
                )
                (and
                    (decrease (posY ?a) 1)
                    (not (keyIn ?c2))
                    (hasKey ?a)

                )
            )

            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (not (keyIn ?c2))
                    (not (wallIn ?c2))
                    (not  (monsterIn ?c2))
                    (not  (keyIn ?c2))
                )
                (and
                    (decrease (posY ?a) 1)
                )
            )
        )
)
```

```
(:action actUse
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (=
                (posX ?a)
                (row ?c1)
            )
            (=
                (posY ?a)
                (column ?c1)
            )
        )
    :effect
        (and
            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (monsterIn ?c2)
                )
                (and
                    (not (monsterIn ?c2))
                )
            )

            (when
                (and
                    (orient_R ?a)
                    (conn_R ?c1 ?c2)
                    (monsterIn ?c2)
                )
                (and
                    (not (monsterIn ?c2))
                )
            )

            (when
                (and
                    (orient_U ?a)
                    (conn_U ?c1 ?c2)
                    (monsterIn ?c2)
                )
                (and
                    (not (monsterIn ?c2))
                )
            )

            (when
                (and
                    (orient_D ?a)
                    (conn_D ?c1 ?c2)
                    (monsterIn ?c2)
                )
                (and
                    (not (monsterIn ?c2))
                )
```

```
                )
            )
    )
)
```

Listing D.2: ZenoTravel numeric planning domain

## D.3 Boulder Dash

```
(define (domain Boulder)
    (:requirements
        :equality
        :negative-preconditions
        :typing
    )

    (:types
        cell avatar - object
    )

    (:predicates
        (hasGem ?a - avatar)

        (orient_R ?a - avatar)
        (orient_L ?a - avatar)
        (orient_U ?a - avatar)
        (orient_D ?a - avatar)

        (conn_R ?c1 ?c2 - cell)
        (conn_L ?c1 ?c2 - cell)
        (conn_U ?c1 ?c2 - cell)
        (conn_D ?c1 ?c2 - cell)

        (wallIn ?c - cell)
        (exitIn ?c - cell)
        (gemIn ?c - cell)
        (rockIn ?c - cell)
    )

    (:functions
        (hasGem ?a - avatar)

        (row ?c - cell)
        (column ?c - cell)

        (posX ?a - avatar)
        (posY ?a - avatar)
    )

    (:action actUp
        :parameters (?a - avatar ?c1 ?c2 - cell)
        :precondition
            (and
                (= (posX ?a) (row ?c1))
                (= (posY ?a) (column ?c1))
```

```
    )
:effect
    (and
        (when
            (and
                (= (posX ?a) (row ?c2))
                (= (posY ?a) (column ?c2))
                (orient_R ?a)
                (not (orient_U  ?a))
            )
            (and
                (not (orient_R ?a))
                (orient_U ?a)
            )
        )

        (when
            (and
                (= (posX ?a) (row ?c2))
                (= (posY ?a) (column ?c2))
                (orient_L ?a)
                (not (orient_U ?a))
            )
            (and
                (not (orient_L ?a))
                (orient_U ?a)
            )
        )

        (when
            (and
                (= (posX ?a) (row ?c2))
                (= (posY ?a) (column ?c2))
                (orient_D ?a)
                (not (orient_U ?a))
            )
            (and
                (not (orient_D ?a))
                (orient_U ?a)
            )
        )

        (when
            (and
                (orient_U ?a)
                (conn_U ?c1 ?c2)
                (gemIn ?c2)
                (not (wallIn ?c2))
                (not  (rockIn ?c2))
            )
            (and
                (decrease (posX ?a) 1)
                (not (gemIn ?c2))
                (increase (hasGem ?a) 1)

            )
        )
```

```
                (when
                    (and
                        (orient_U ?a)
                        (conn_U ?c1 ?c2)
                        (not (wallIn ?c2))
                        (not (rockIn ?c2))
                        (not (gemIn ?c2))
                    )
                    (and
                        (decrease (posX ?a) 1)
                    )
                )
            )
    )

     (:action actDown
        :parameters (?a - avatar ?c1 ?c2 - cell)
        :precondition
            (and
                (= (posX ?a) (row ?c1))
                (= (posY ?a) (column ?c1))
            )
        :effect
            (and
                (when
                    (and
                        (= (posX ?a) (row ?c2))
                        (= (posY ?a) (column ?c2))
                        (orient_R ?a)
                        (not (orient_D ?a))
                    )
                    (and
                        (not (orient_R ?a))
                        (orient_D ?a)
                    )
                )

                (when
                    (and
                        (= (posX ?a) (row ?c2))
                        (= (posY ?a) (column ?c2))
                        (orient_L ?a)
                        (not (orient_D ?a))
                    )
                    (and
                        (not (orient_L ?a))
                        (orient_D ?a)
                    )
                )

                (when
                    (and
                        (= (posX ?a) (row ?c2))
                        (= (posY ?a) (column ?c2))
                        (orient_U ?a)
                        (not (orient_D ?a))
```

```
                )
                (and
                    (not (orient_U ?a))
                    (orient_D ?a)
                )
            )

            (when
                (and
                    (orient_D ?a)
                    (conn_D ?c1 ?c2)
                    (gemIn ?c2)
                    (not (wallIn ?c2))
                    (not (rockIn ?c2))
                )
                (and
                    (increase (posX ?a) 1)
                    (not (gemIn ?c2))
                    (increase (hasGem ?a) 1)

                )
            )

            (when
                (and
                    (orient_D ?a)
                    (conn_D ?c1 ?c2)
                    (not (wallIn ?c2))
                    (not  (rockIn ?c2))
                    (not  (gemIn ?c2))
                )
                (and
                    (increase (posX ?a) 1)
                )
            )
        )
)

 (:action actRight
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (= (posX ?a) (row ?c1))
            (= posY ?a) (column ?c1))
        )
    :effect
        (and
            (when
                (and
                    (= (posX ?a) (row ?c2))
                    (= (posY ?a) (column ?c2))
                    (orient_L ?a)
                    (not (orient_R ?a))
                )
                (and
                    (not (orient_L ?a))
                    (orient_R ?a)
```

```
    )
)

(when
    (and
        (= (posX ?a) (row ?c2))
        (= (posY ?a) (column ?c2))
        (orient_U ?a)
        (not (orient_R ?a))
    )
    (and
        (not (orient_U ?a))
        (orient_R ?a)
    )
)

(when
    (and
        (= (posX ?a) (row ?c2))
        (= (posY ?a) (column ?c2))
        (orient_D ?a)
        (not (orient_R ?a))
    )
    (and
        (not (orient_D ?a))
        (orient_R ?a)
    )
)

(when
    (and
        (orient_R ?a)
        (conn_R ?c1 ?c2)
        (gemIn ?c2)
        (not (wallIn ?c2))
        (not  (rockIn ?c2))
    )
    (and
        (increase (posY ?a) 1)
        (not (gemIn ?c2))
        (increase (hasGem ?a) 1)

    )
)

(when
    (and
        (orient_R ?a)
        (conn_R ?c1 ?c2)
        (not (wallIn ?c2))
        (not  (rockIn ?c2))
        (not  (gemIn ?c2))
    )
    (and
        (increase (posY ?a) 1)
    )
)
```

```
        )
)

(:action actLeft
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (= (posX ?a) (row ?c1))
            (= (posY ?a) (column ?c1))
        )
    :effect
        (and
            (when
                (and
                    (= (posX ?a) (row ?c2))
                    (= (posY ?a) (column ?c2))
                    (orient_R ?a)
                    (not (orient_L ?a))
                )
                (and
                    (not (orient_R ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (= (posX ?a) (row ?c2))
                    (= (posY ?a) (column ?c2))
                    (orient_D ?a)
                    (not (orient_L ?a))
                )
                (and
                    (not (orient_D ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (= (posX ?a) (row ?c2))
                    (= (posY ?a) (column ?c2))
                    (orient_U ?a)
                    (not (orient_L ?a))
                )
                (and
                    (not (orient_U ?a))
                    (orient_L ?a)
                )
            )

            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (gemIn ?c2)
                    (not (wallIn ?c2))
```

```
                    (not  (rockIn ?c2))
                )
                (and
                    (decrease (posY ?a) 1)
                    (not (gemIn ?c2))
                    (increase (hasGem ?a) 1)

                )
            )

            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (not (gemIn ?c2))
                    (not (wallIn ?c2))
                    (not  (rockIn ?c2))
                    (not  (gemIn ?c2))
                )
                (and
                    (decrease (posY ?a) 1)
                )
            )
        )
)

(:action actUse
    :parameters (?a - avatar ?c1 ?c2 - cell)
    :precondition
        (and
            (= (posX ?a) (row ?c1))
            (= (posY ?a) (column ?c1))
        )
    :effect
        (and
            (when
                (and
                    (orient_L ?a)
                    (conn_L ?c1 ?c2)
                    (rockIn ?c2)
                )
                (and
                    (not (rockIn ?c2))
                )
            )

            (when
                (and
                    (orient_R ?a)
                    (conn_R ?c1 ?c2)
                    (rockIn ?c2)
                )
                (and
                    (not (rockIn ?c2))
                )
            )
```

```
                (when
                    (and
                        (orient_U ?a)
                        (conn_U ?c1 ?c2)
                        (rockIn ?c2)
                    )
                    (and
                        (not (rockIn ?c2))
                    )
                )

                (when
                    (and
                        (orient_D ?a)
                        (conn_D ?c1 ?c2)
                        (rockIn ?c2)
                    )
                    (and
                        (not (rockIn ?c2))
                    )
                )
            )
    )
)
```

Listing D.3: Boulder Dash videogame planning domain

# Bibliography


[ACO19] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia. Learning action models with minimal observability. *Artificial Intelligence*, 275:104 – 137, 2019.

[AFP+18] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 33, 11 2018.

[AJO18] Diego Aineto, Sergio Jiménez, and Eva Onaindia. Learning strips action models with classical planning. In *International Conference on Automated Planning and Scheduling, ICAPS-18*, 2018.

[BCC12] J Benton, Amanda Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[BDR01] Hendrik Blockeel and Luc De Raedt. Top-down induction of logical decision trees. *Artificial Intelligence*, 101:285–297, 12 2001.

[BDRD+20] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[Ben96] Scott Sherwood Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, stanford university PhD thesis, 1996.

[BG01] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence. 2001 Jun; 129 (1-2): 5-33.*, 2001.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[BM08] Roman Bartak and T.L. McCluskey. Workshop on knowledge engineering for planning and scheduling (keps). In *Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), at ICAPS-08*, September 2008.

[BNVB13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[BR07] James G Bellingham and Kanna Rajan. Robotics in remote and hostile environments. *Science*, 318(5853):1098–1102, 2007.

[Byl94] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[CCFL09] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Temporal planning in domains with linear processes. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer, 2009.

[CCFL10] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'10, page 42–49. AAAI Press, 2010.

[CFLM16] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full pddl+ language into smt. In *The 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 01 2016.

[CG11] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *ICAPS 2011 - Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 01 2011.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[CH15] Samprit Chatterjee and Ali S Hadi. *Regression analysis by example*. John Wiley & Sons, 2015.

[CMW13] Stephen N. Cresswell, Thomas L. McCluskey, and Margaret M. West. Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28(2):195–213, 2013.

[Coh95] William W Cohen. Fast effective rule induction. In *Machine learning proceedings 1995*, pages 115–123. Elsevier, 1995.

[CS07] A. I. Coles and A. J. Smith. Marvin: a heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[Dav91] Lawrence Davis. *Handbook of genetic algorithms*. CumInCAD, 1991.

[ELL+13] Marc Ebner, John Levine, Simon M. Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a Video Game Description Language. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[FL03] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[FN71] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, December 1971.

[FOCGPP06] Juan Fdez-Olivares, Luis Castillo, Óscar García-Pérez, and Francisco Palao. Bringing users and planning technology together experiences in siadex. In *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling*, pages 11–20, 2006.

[FW94] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In *Machine Learning Proceedings 1994*, pages 70–77. Elsevier, 1994.

[GCMS+21] Jesús Giráldez-Cru, Pablo Mesejo, José Ángel Segura, Juan Fernández-Olivares, and Antonio González. Herramientas de gamificación para la enseñanza de técnicas de búsqueda heurística en entornos dinámicos. *Actas de las Jenui*, 6:179–186, 2021.

[GGGP15] David García, Juan Carlos Gámez, Antonio González, and Raúl Pérez. An interpretability improvement for fuzzy rule bases obtained by the iterative rule learning approach. *Int. J. Approx. Reasoning*, 67(C):37–58, December 2015.

[GGP14] David García, Antonio González, and Raúl Pérez. Overview of the slave learning algorithm: A review of its evolution and prospects. *International Journal of Computational Intelligence Systems*, 7(6):1194–1221, 2014.

[Gil94] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 87 – 95. Morgan Kaufmann, San Francisco (CA), 1994.

[GL16] Peter Gregory and Alan Lindsay. Domain model acquisition in domains with action costs. In *ICAPS*, pages 149–157, 2016.

[GLH15] Salvador García, Julián Luengo, and Francisco Herrera. *Data preprocessing in data mining*. Springer, 2015.

[GMB00] Ramon Garcia-Martinez and Daniel Borrajo. An integrated approach of learning, planning, and execution. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 29, 02 2000.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann., 2004.

[GP07] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*, volume 2. Springer, 2007.

[GPLL16] Raluca D Gaina, Diego Pérez-Liébana, and Simon M Lucas. General video game for 2 players: Framework and competition. In *2016 8th Computer Science and Electronic Engineering (CEEC)*, pages 186–191. IEEE, 2016.

[GT14] Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014.

[HL03] R. Howey and D. Long. Val's progress The automatic validation tool for pddl2.1 used in the international planning competition. In *Proceedings of the ICAPS 2003 workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*, pages 28–37, Trento, Italy, June 2003.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hof01] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

[Hof03] Jörg Hoffmann. The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables. *Journal of artificial intelligence research*, 20:291–341, 2003.

[HW79] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[JCKV14] Rabia Jilani, Andrew Crampton, Diane E Kitchin, and Mauro Vallati. Automated knowledge engineering tools in planning: state-of-the-art and future challenges. In *International Conference on Automated Planning and Scheduling, ICAPS-14*, 2014.

[JFB08] Sergio Jiménez, Fernando Fernández, and Daniel Borrajo. The pela architecture: integrating planning and learning to improve execution. In *National Conference on Artificial Intelligence (AAAI'2008)*. sn, 2008.

[JRF+12] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, and D. Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012.

[JSAJ19] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34, 01 2019.

[Kee13] C. Maria Keet. *Open World Assumption*, pages 1567–1567. Springer New York, New York, NY, 2013.

[Koz94] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, Jun 1994.

[KPLLT16] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259, 2016.

[KR09] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[LBCE+13] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M. Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, Dagstuhl Follow-Ups, pages 77–84. Dagstuhl Publishing, DEU, November 2013.

[LHTD02] Huan Liu, Farhad Hussain, Chew Lim Tan, and Manoranjan Dash. Discretization: An enabling technique. *Data mining and knowledge discovery*, 6(4):393–423, 2002.

[LJFB07] Jesús Lanchas, Sergio Jiménez, Fernando Fernández, and Daniel Borrajo. Learning action durations from executions. In *Proceedings of the ICAPS*. Citeseer, 2007.

[LT06] Sid Lamrous and Mounira Taileb. Divisive hierarchical k-means. In *2006 International Conference on Computational Inteligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06)*, pages 18–18. IEEE, 2006.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[McD00] Drew M McDermott. The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35, 2000.

[MCRW09] Thomas Leo McCluskey, SN Cresswell, N Elisabeth Richardson, and Margaret M West. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, 2009.

[Mic83] Ryszard S Michalski. A theory and methodology of inductive learning. In *Machine learning*, pages 83–134. Springer, 1983.

[Mit99] Tom M Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11):30–36, 1999.

[MLJ+88] Glenn Miller, Kelly Lindenmayer, Mark Johnston, Shon Vick, and Jeff Sponsler. Knowledge based tools for hubble space telescope planning and scheduling: constraints and strategies. *Telematics and Informatics*, 5(3):197–212, 1988.

[MMC13] R Mitchell, J Michalski, and T Carbonell. *An artificial intelligence approach*. Springer, 2013.

[MMRS56] J McCarthy, M Minsky, N Rochester, and C Shannon. Dartmouth conference. In *Dartmouth Summer Research Conference on Artificial Intelligence*, 1956.

[MNPW98] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial intelligence*, 103(1-2):5–47, 1998.

[Mou14] Kira Mourao. Learning probabilistic planning operators from noisy observations. In *31st Workshop of the UK Planning & Scheduling Special Interest Group (PlanSIG 2013)*, 2014.

[MRS02] Thomas Leo McCluskey, N Elisabeth Richardson, and Ron M Simpson. An interactive method for inducing operator descriptions. In *AIPS*, pages 121–130, 2002.

[MZPS12] K. Mourao, L. S. Zettlemoyer, R. P. A. Petrick, and M. Steedman. Learning STRIPS operators from noisy and incomplete observations. *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 2012.

[N+84] Nils J Nilsson et al. Shakey the robot. *Sri International*, 1984.

[NKD+99] Pandu Nayak, J Kurien, Gregory Dorais, W Millar, K Rajan, R Kanefsky, ED Bernard, BE Gamble Jr, N Rouquette, DB Smith, et al. Validating the ds-1 remote agent experiment. In *Artificial intelligence, robotics and automation in space*, volume 440, page 349, 1999.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[NSS58] Allen Newell, John Calman Shaw, and Herbert A Simon. Elements of a theory of human problem solving. *Psychological review*, 65(3):151, 1958.

[OC96] Tim Oates and Paul R Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *AAAI/IAAI, Vol. 1*, pages 863–868, 1996.

[PFL+16] Wiktor Piotrowski, Maria Fox, Derek Long, Daniele Magazzeni, and Fabio Mercorio. Heuristic planning for pddl+ domains. In *The 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3213–3219. AAAI Press, 2016.

[PLLK+19] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

[PLST+15] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2015.

[PLST+16] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[PS91] Gregory Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. *Knowledge Discovery in Databases*, 1991.

[PY10] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[Pyl99] Dorian Pyle. *Data preparation for data mining*. morgan kaufmann, 1999.

[PZK04] Hanna Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning probabilistic relational planning rules. In *ICAPS*, pages 73–82, 2004.

[PZK07] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Qui14] John Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RN16] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[RNBH08] Jussi Rintanen, Bernhard Nebel, C. Beck, and E. Hansen. Icaps 2008 - proceedings of the 18th international conference on automated planning and scheduling. In *ICAPS*, pages ix–x, 01 2008.

[Rom04] Charles Romesburg. *Cluster analysis for researchers*. Lulu. com, 2004.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Sam59] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[San10] Scott Sanner. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32:27, 2010.

[SBF98] Rudi Studer, V Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data & knowledge engineering*, 25(1-2):161–197, 1998.

[Sch13] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[SHTR16] Enrico Scala, Patrik Haslum, Sylvie Thiebaux, and Miquel Ramirez. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ECAI'16, page 655–663, NLD, 2016. IOS Press.

[SMFOP21] José Á Segura-Muros, Juan Fernández-Olivares, and Raúl Pérez. Learning numerical action models from noisy input data, 2021.

[SMPFO21] José Ángel Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Discovering relational and numerical expressions from plan traces for learning action schemes. *Applied Intelligence*, 2021.

[SS89] Wei-Min Shen and Herbert A Simon. Rule creation and rule learning through environmental exploration. In *IJCAI*, pages 675–680. Citeseer, 1989.

[Ste90] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[Sto74] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[SW03] Hilmar Schuschel and Mathias Weske. Integrated workflow planning and coordination. In *International Conference on Database and Expert Systems Applications*, pages 771–781. Springer, 2003.

[SW10] Claude Sammut and Geoffrey I. Webb, editors. *Deductive Learning*, pages 267–267. Springer US, Boston, MA, 2010.

[Var98] I Var. Multivariate data analysis. *vectors*, 8(2):125–136, 1998.

[Wan95] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *In Proceedings of the 12th International Conference on Machine Learning*, pages 549–557. Morgan Kaufmann, 1995.

[WC13] Tony Worm and Kenneth Chiu. Prioritized grammar enumeration: symbolic regression by dynamic programming. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2013.

[WSF95] Richard Y Wang, Veda C Storey, and Christopher P Firth. A framework for analysis of data quality research. *IEEE transactions on knowledge and data engineering*, 7(4):623–640, 1995.

[YWJ07] Q. Yang, K. Wu, and Y. Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal.*, page 107–143, 2007.

[ZK03] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 24(2):73–73, 2003.

[ZK13] H. H. Zhuo and S. Kambhampati. Action-model acquisition from noisy plan traces. *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence.*, pages 2444–2450, 2013.

[ZNK13] Hankz Hankui Zhuo, Tuan Anh Nguyen, and Subbarao Kambhampati. Refining incomplete planning domain models through plan traces. In *IJCAI*, pages 2451–2458, 2013.

[ZPK05] Luke S Zettlemoyer, Hanna Pasula, and Leslie Pack Kaelbling. Learning planning rules in noisy stochastic worlds. In *AAAI*, pages 911–918, 2005.

[ZPK19] Hankz Hankui Zhuo, Jing Peng, and Subbarao Kambhampati. Learning action models from disordered and noisy plan traces. *arXiv preprint arXiv:1908.09800*, 2019.

[ZY14] Hankz Hankui Zhuo and Qiang Yang. Action-model acquisition for planning via transfer learning. *Artificial intelligence*, 212:80–103, 2014.

[ZYHL10] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540 – 1569, 2010.

# Acronyms and Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| AML | Action Model Learning |
| AP | Automated Planning |
| CWA | Closed World Assumption |
| GVG-AI | General Video Game AI |
| KE | Knowledge Engineering |
| ML | Machine Learning |
| OWA | Open World Assumption |
| PDDL | Planning Definition Domain Language |
| PM | PlanMiner |
| SR | Symbolic Regression |
| STRIPS | Stanford Research Institute Problem Solver |
| XAI | eXplanaible AI |