UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Programa de Doctorado en Tecnologías de la información y la Comunicación

Application of Neuro-Symbolic Artificial Intelligence to Sequential Decision Making

Tesis Doctoral

Carlos Núñez Molina

Directores

Juan Fernández Olivares
Pablo Mesejo Santiago

Granada, diciembre de 2024

Título en Español: Aplicación de Inteligencia Artificial Neuro-Simbólica para Procesos de Decisión Secuencial
Título en Inglés: Application of Neuro-symbolic Artificial Intelligence to Sequential Decision Making
Programa de doctorado: Programa de Doctorado en Tecnologías de la información y la Comunicación
Doctorando: Carlos Núñez Molina
Directores: Juan Fernández Olivares, Pablo Mesejo Santiago

To my father,
the first scientist I ever met.

Acknowledgements
Agradecimientos

I feel strange while writing this, not knowing exactly what to say. This chapter of my life started when Juan, who back then was my AI professor, invited me and other students to visit his lab. After this visit, he would offer me the opportunity to start doing research with him. I must admit that, at first, I was not particularly enthusiastic about my research topic. Juan's main area of expertise was Automated Planning, while I, like most young AI students, wanted nothing to do with *GOFAI* and instead aspired to work on cutting-edge Deep Learning (I am glad LLMs were not a thing back then). Nonetheless, Juan placed his trust in me and granted me the freedom to use Deep Reinforcement Learning instead of more traditional approaches. Thanks to this, I was able to engage in a research project I genuinely enjoyed. Along the way, I would find out that what truly fascinates me is the integration of classical, symbolic AI with modern, Deep Learning techniques, and that this is called "Neuro-symbolic AI". For granting me the opportunity to conduct research, trusting me from the very beginning, and accompanying me throughout this long journey, I want to express my most heartfelt thanks to Juan, my thesis supervisor. Likewise, I wish to express my deepest gratitude to Pablo, my thesis co-supervisor who, alongside Juan, has carefully guided me throughout the four years of this PhD dissertation. I am deeply grateful to have you not only as my supervisors but also as my friends.

As I write this, I find it impossible to forget about Masataro Asai. While Juan and Pablo introduced me to the world of research and supported me throughout this journey, it was you who launched my career to new heights. You granted me the opportunity to carry out research at a top institution, the MIT-IBM Watson AI Lab, an opportunity so extraordinary that it surpassed even my wildest dreams. You closely supervised me, teaching me how to properly conduct meaningful, high-quality research. We worked hand in hand, programming and writing papers together. Over time, I started to consider you not only my mentor and colleague, but also my dear friend. Thank you, Masa, for everything you have done for me.

Por último, quiero dedicar unas palabras de agradecimiento a mi entorno más cercano. A mis padres, por apoyarme y creer siempre en mí, y por inculcarme el valor de la perseverancia y el esfuerzo, tan necesarios en la vida. A mi abuela, que desde que era pequeñito siempre me ha escuchado con gran atención e interés, sin importar de lo que le hablara. A mis amigos, por todos los buenos momentos que hemos pasado juntos y por permitirme desconectar del trabajo. Y especialmente a Aida, mi pareja, por ser mi fan número uno, por escucharme con más atención de la que yo mismo me presto y por apoyarme incondicionalmente. Eres lo mejor que me ha pasado en la vida.

Over the course of this journey, I have come to realize just how many people have been supporting me, each of you fulfilling a different role, all of you encouraging me to pursue my dreams no matter what. For this reason, I believe this thesis is not mine alone, but also yours. Thank you.

Contents

Main Acronyms

Acronym	**Description**
AI	Artificial Intelligence
AP	Automated Planning
CP	Classical Planning
CNN	Convolutional Neural Network
DL	Deep Learning
DNN	Deep Neural Network
DRL	Deep Reinforcement Learning
FOL	First Order Logic
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Squared Error
NLL	Negative Log Likelihood
NLM	Neural Logic Machine
NSP	Subsymbolic/Non-Symbolic Planning
POMDP	Partially Observable Markov Decision Process
PP	Probabilistic Planning
RL	Reinforcement Learning
RRL	Relational Reinforcement Learning
SDM	Sequential Decision Making
SDP	Sequential Decision Process
SP	Symbolic Planning
SSP MDP	Stochastic Shortest-Path Markov Decision Process

Resumen en Español
Thesis Summary in Spanish

1 Introducción al problema

La **Toma de Decisiones Secuenciales** (o **SDM**, por sus siglas en inglés) [145] es un importante subcampo dentro de la **Inteligencia Artificial (IA)** que estudia cómo crear agentes, ya sean físicos o virtuales, capaces de tomar decisiones de manera inteligente con el fin de alcanzar un objetivo o realizar una tarea determinada. La SDM constituye un marco general que ha sido aplicado con éxito a campos tan diversos como la robótica [133], la logística [205], los juegos [217], las finanzas [36] y el procesamiento del lenguaje natural [242], entre otros muchos.

A lo largo de los años del campo, se han propuesto una gran cantidad de métodos que pueden categorizarse en dos enfoques principales: **Planificación Automática (AP)** [84] y **Aprendizaje por Refuerzo (RL)** [224]. Estos dos paradigmas difieren principalmente en la forma de resolver las tareas de SDM y en cómo representan su conocimiento. La AP aprovecha el conocimiento disponible sobre las dinámicas del entorno, codificado en lo que se conoce como un dominio de planificación o modelo de acciones, para llevar a cabo un proceso de búsqueda y razonamiento con el fin de encontrar una política o plan (secuencia de acciones) que resuelva la tarea correspondiente. Este conocimiento a menudo se describe de manera simbólica, por ejemplo, utilizando lógica. Por otro lado, los métodos de RL permiten a los agentes aprender a actuar de manera óptima utilizando, en la mayoría de casos, solo los datos obtenidos al interactuar con su entorno, no requiriendo conocimiento a priori sobre sus dinámicas y sin llevar a cabo ningún proceso de planificación. El conocimiento inferido del entorno a menudo se codifica de manera subsimbólica, por ejemplo, mediante valores numéricos que representan los parámetros o pesos de un modelo de Aprendizaje Automático (ML) [20].

En los últimos años, ha surgido un gran interés por integrar los campos de la AP y el RL, con la esperanza de obtener un método de SDM que exhiba las habilidades de aprendizaje de RL junto con las capacidades de razonamiento de AP. Entre los distintos acercamientos para realizar esta integración se encuentran: el RL basado en modelos [160], los métodos de ML para aprender el conocimiento utilizado en AP (por ejemplo, para aprender dominios de planificación y heurísticas) [117], y aquellos enfoques denominados *neuro-simbólicos* [182], en los cuales se centra esta tesis, que combinan las redes neuronales profundas (DNN) empleadas en los métodos actuales de RL con las representaciones simbólicas comúnmente utilizadas en AP.

2 Desarrollo realizado

El objetivo de esta tesis doctoral es **avanzar el campo de la SDM mediante el estudio y desarrollo de métodos novedosos de IA neuro-simbólica**. El trabajo realizado durante el desarrollo de la tesis puede agruparse en cuatro contribuciones principales.

Como primera contribución [182], se realizó una amplia **revisión (*review*) del campo de la SDM**, abarcando tanto métodos para resolver tareas de SDM como métodos para aprender su estructura, y poniendo especial énfasis en la representación del conocimiento empleada por las distintas técnicas: simbólica, subsimbólica o híbrida. Hasta donde sabemos, ningún otro trabajo en la literatura ofrece una visión tan completa del campo. Como parte de esta revisión, se propusieron una serie de características que un método ideal de SDM debería cumplir y se utilizaron para analizar las ventajas y desventajas de las distintas técnicas de SDM. Como resultado de este análisis, argumentamos que un método ideal de SDM debería integrar los paradigmas del AP y el RL, al mismo tiempo que emplea una representación híbrida (simbólica y subsimbólica) para su conocimiento. Dado que la IA neuro-simbólica es el enfoque que actualmente más se acerca a esta integración, se concluyó que representa un acercamiento muy prometedor para la consecución de un método ideal de SDM. Por lo tanto, la revisión realizada sirve para justificar la importancia y relevancia de la presente tesis doctoral. Esta primera contribución se aborda en el Capítulo III.

Como segunda contribución [179, 183], se desarrolló un **método neuro-simbólico para mejorar la eficiencia de los algoritmos de AP en escenarios de tiempo real mediante la selección de objetivos**. Nuestra propuesta, denominada *Deep Q-Planning* (DQP), integra el algoritmo de RL profundo Deep Q-Learning [158] con el planificador simbólico FastForward [108]. En cada iteración, Deep Q-Learning se utiliza para seleccionar el siguiente subobjetivo a alcanzar, mientras que el planificador FastForward se encarga de encontrar un plan para conseguir el subobjetivo elegido desde el estado actual. Gracias a la combinación de RL profundo para seleccionar objetivos con la AP para alcanzarlos, DQP es capaz de aprovechar la sinergia existente entre la AP y el RL para obtener soluciones de calidad de manera eficiente. Para evaluar la propuesta, se recurrió al juego conocido como *Boulder Dash* que proporciona el entorno *General Video Game AI* (GVGAI) [144]. Los resultados obtenidos muestran que, en comparación con Deep Q-Learning, DQP requiere considerablemente menos datos (como mínimo un orden de magnitud menos) y generaliza mucho mejor a nuevos niveles. En comparación con el planificador FastDownward, DQP reduce drásticamente los tiempos de resolución de tareas a cambio de obtener planes solo un 9% más largos (peores) de media. Esta segunda contribución se aborda en el Capítulo IV.

La tercera contribución [178] fue fruto de una **colaboración con el MIT-IBM Watson AI Lab** y se correspondió con un enfoque neuro-simbólico para mejorar el rendimiento de los algoritmos de AP mediante el aprendizaje de heurísticas. Esta propuesta conllevó el desarrollo de un método para **aprovechar el conocimiento simbólico codificado en las heurísticas admisibles de cara a aprender mejores heurísticas**. Nuestro método modela la heurística aprendida como una distribución Gaussiana Truncada $\mathcal{TN}$ en lugar de una Gaussiana $\mathcal{N}$ (sin truncar). La cota inferior de esta distribución $\mathcal{TN}$ se establece al valor de una heurística admisible, asegurando de esta forma que las predicciones heurísticas siempre sean mayores que dicha heurística admissible. El modelo elegido ($\mathcal{TN}$ en lugar de $\mathcal{N}$) resulta en una nueva función de pérdida a minimizar durante el entrenamiento, diferente al Error Cuadrático Medio (MSE) utilizado normalmente. Llevamos a cabo experimentos donde nuestra función de pérdida se comparó con el MSE de cara a aprender heurísticas en una gran variedad de escenarios, incluyendo cuatro clásicos dominios de planificación: *blocksworld*, *ferry*, *gripper* y *visitall*. Los resultados obtenidos muestran que nuestra función de pérdida basada en $\mathcal{TN}$ provoca que el entrenamiento converja más rápido y, en general, produce heurísticas más precisas que mejoran el rendimiento de los algoritmos de planificación. Más concretamente, al utilizar la arquitectura de DNN conocida como *Neural Logic Machine* (NLM) [53] junto con nuestra configuración propuesta learn/h^{FF} (donde el modelo predice tanto σ como μ y se utiliza aprendizaje residual), nuestra función de pérdida supera al MSE en términos de precisión heurística en 3 de los 4 dominios considerados, y mejora los resultados de planificación en todos ellos. Esto confirma que nuestro método basado en $\mathcal{TN}$ permite extraer de manera efectiva el conocimiento almacenado en las heurísticas admisibles, mejorando de esta forma la calidad de las heurísticas aprendidas. Esta tercera contribución se aborda en el Capítulo V.

Por último, como cuarta contribución [181], se implementó un **método neurosimbólico para generar problemas de planificación válidos (es decir, resolubles y consistentes), diversos y difíciles para cualquier dominio de planificación clásica**. El enfoque propuesto, denominado *NeSIG* (*Neuro-Symbolic Instance Generator*), formula la generación de problemas como un Proceso de Decisión de Markov (MDP) [224]. El estado inicial del problema se genera añadiendo secuencialmente átomos y objetos a un estado vacío. A continuación, el objetivo del problema se obtiene ejecutando una secuencia de acciones en el estado inicial generado. Dos políticas generativas, codificadas como NLMs y entrenadas con RL profundo, son las encargadas de guiar este proceso generativo hacia problemas consistentes, diversos y difíciles. La diversidad se define como la distancia o disimilitud entre problemas, mientras que la dificultad se mide resolviendo los problemas generados con un algoritmo de AP. Por el contrario, la consistencia depende de la semántica del dominio PDDL [96] y de las preferencias humanas, por lo que la información acerca de esta debe ser proporcionada por el diseñador humano. Para reducir el esfuerzo humano lo máximo posible, se implementó un lenguaje semideclarativo que combina Python y lógica de primer orden, lo que permite codificar las reglas de consistencia con facilidad. Se llevaron a cabo experimentos en cinco dominios distintos (*blocksworld*, *logistics*, *sokoban*, *miconic* y *satellite*), comparando el enfoque propuesto con generadores de problemas creados a mano para cada dominio. Los resultados obtenidos muestran que NeSIG aprende a generar problemas válidos y diversos de mucha mayor dificultad, 6.8 veces más de media (geométrica), que los generadores manuales, al mismo tiempo que reduce el esfuerzo humano necesario para generarlos. Además, se evaluó la capacidad de generalización de NeSIG mediante una comparativa con los generadores manuales a lo largo de los cinco dominios, diferentes planificadores (tanto óptimos como no óptimos) y diferentes tamaños de problema (hasta más del doble del tamaño de entrenamiento). De las 20 combinaciones dominio-planificador evaluadas, en 15 de ellas NeSIG supera la dificultad del generador manual para todos los tamaños de problema. Por lo tanto, concluimos que NeSIG también presenta notables capacidades de generalización, siendo capaz de generalizar tanto a distintos tamaños de problema como a distintos planificadores. Esta cuarta contribución se aborda en el Capítulo VI.

Adicionalmente, durante el desarrollo de la tesis se han implementado varios **proyectos Python de código abierto**, los cuales han sido distribuidos mediante el *Python Package Index* (PyPI). Los proyectos desarrollados son: ***lifted-pddl*** [174], un eficiente *parser* para AP que a día de hoy acumula 14000 descargas (según el sitio web *pepy.tech*); ***stable-truncated-gaussian*** [177], una implementación estable y diferenciable de la distribución Gaussiana Truncada $\mathcal{TN}$ utilizando la biblioteca Pytorch (13000 descargas); ***neural-logic-machine*** [176], que implementa la arquitectura NLM también utilizando Pytorch (2000 descargas); y ***pddl-prover*** [175], que permite la evaluación automática de fórmulas lógicas en estados de planificación (2000 descargas).

3 Conclusiones y trabajos futuros

En conclusión, las cuatro aportaciones presentadas en esta tesis doctoral han contribuido al avance del campo de la SDM, tanto desde una perspectiva teórica, gracias a la revisión bibliográfica y análisis de los distintos enfoques existentes en el campo, como desde una perspectiva empírica, mediante el desarrollo de novedosos métodos neuro-simbólicos tanto para resolver MDP como para aprender su estructura. Esperamos que el trabajo desarrollado en la presente tesis haya servido para mostrar el gran potencial que posee la IA neuro-simbólica para mejorar la SDM, especialmente mediante la integración de la AP y el RL, y el campo de la IA en su totalidad.

La aplicación de la IA neuro-simbólica a la SDM es un enfoque prometedor que presenta numerosas oportunidades de investigación. A continuación, se proponen posibles líneas de trabajo futuro para las tres contribuciones empíricas aportadas en esta tesis. En relación a la arquitectura DQP de selección de objetivos, se podría extender su aplicabilidad a entornos estocásticos mediante el uso de Deep Q-Learning para monitorizar la ejecución y reaccionar a situaciones inesperadas. Otra posibilidad sería experimentar con distintos modelos de ML como las Redes Neuronales de Grafos (*Graph Neural Networks*) [204], de cara a mejorar el proceso de selección de objetivos. En relación al método de aprendizaje de heurísticas, se podría explorar cómo extender el enfoque basado en Gaussianas Truncadas $\mathcal{TN}$ a otros escenarios, como aquellos donde se dispone de cotas superiores además de cotas inferiores. También se podría aplicar este enfoque al RL, mejorando el aprendizaje de los valores de estado $V(s)$ y Q-values $Q(s,a)$ mediante el uso de cotas inferiores y/o superiores de estos valores. Por último, en relación al método de generación de problemas, existe la posibilidad tanto de extender y mejorar el acercamiento propuesto como de adaptarlo para una aplicación concreta. Por una parte, se podría extender este método para generar problemas de acuerdo a distintos tipos de preferencias y con variables numéricas, y utilizar Redes Generativas de Flujo (*GFlowNets*) [16] para obtener un mejor balance entre calidad y diversidad. Por otra parte, NeSIG podría ser aplicado al diseño de niveles de videojuegos, a la creación de problemas para optimizar el proceso de aprendizaje de un agente (es decir, para la generación automática de currículos) y a la generación adversaria de problemas, donde el objetivo es obtener problemas difíciles para un algoritmo en específico.

Abstract

Sequential Decision Making (SDM) [145] is an important subfield within Artificial Intelligence (AI) devoted to creating agents, either physical or virtual, capable of making intelligent decisions in order to achieve a goal or fulfill some task. SDM provides a general framework that has been successfully applied to fields as diverse as robotics [133], logistics [205], games [217], finance [36], and natural language processing [242], just to name a few.

Throughout the many years of the field, a large number of different methods have been proposed, which can be categorized into two main approaches: **Automated Planning (AP)** [84] and **Reinforcement Learning (RL)** [224]. These two competing paradigms mainly differ in how they obtain their solution and how they represent their knowledge. AP exploits the prior knowledge about the environment dynamics, encoded in what is known as a planning domain or action model, to carry out a search and reasoning process for a policy or plan (sequence of actions) that achieves the task goal. This knowledge is often described in a symbolic manner, e.g., by using logic. On the other hand, standard RL methods learn to act optimally using only the data obtained by interacting with the environment, requiring no prior knowledge about its dynamics and performing no planning at all. The knowledge learned from the environment is often encoded in a subsymbolic manner, e.g., as real numbers representing the parameters of a Machine Learning (ML) [20] model.

In recent years, there has been growing interest in **bridging the gap between AP and RL**, in the hopes of obtaining an SDM method with the learning abilities of RL and the reasoning capabilities of AP. A few examples of the integration between both fields are: model-based RL [160], ML methods for learning the prior knowledge used in AP (e.g., planning domains and heuristics) [117], and neuro-symbolic approaches [182], which combine the deep neural networks (DNNs) employed in modern RL with the symbolic representations commonly utilized in AP.

This doctoral dissertation focuses on the study, design and implementation of novel **neuro-symbolic methods for SDM**, introducing four main contributions to the field. The first contribution [182] is a broad **review of the state of the art**, covering symbolic, subsymbolic and hybrid (e.g., neuro-symbolic) methods for SDM. Existing SDM techniques are classified into two distinct groups: methods that solve SDM tasks and those that learn their structure. A taxonomy is proposed for each group. Methods for solving SDM tasks are classified into Automated Planning, Reinforcement Learning and a recent group of methods that *learn to plan* [160], thus combining the AP and RL approaches. Methods for learning the structure of SDM tasks are classified into techniques that learn an action model and those that learn a different type of structural knowledge, such as state invariants and landmarks. Furthermore, the advantages and drawbacks of the different approaches for solving SDM tasks are discussed. Based on this discussion, it is concluded that neuro-symbolic AI is the current approach that most closely resembles an ideal integration of AP and RL, thus justifying the relevance and significance of this doctoral dissertation. This first contribution is addressed in Chapter III.

The second and third major contributions of this thesis involve **neuro-symbolic methods for solving SDM tasks**. In the first of these methods [179, 183], Fast-Forward [108], a classical AP algorithm, is combined with Deep Q-Learning [158], a widely-employed RL technique, to solve SDM tasks with real-time restrictions. The goal of the task to solve is decomposed into a series of subgoals (provided as prior knowledge to the system) so that, by sequentially attaining a subset of them, the task goal is achieved. At each decision-making step, Deep Q-Learning selects the next subgoal to achieve and FastForward finds a plan that reaches the selected subgoal from the current state. This neuro-symbolic architecture, named ***Deep Q-Planning*** **(DQP)**, is tested on a video game environment used as a standard test-bed for intelligent system applications. Results show DQP outperforms standalone AP and RL methods when both plan quality (length) and time requirements are considered. On the one hand, DQP is considerably more sample-efficient than Deep Q-Learning and generalizes much better to new game levels. On the other hand, DQP drastically reduces problem-solving times when compared to FastForward, at the expense of obtaining plans with only 9% more actions on average. This second contribution is addressed in Chapter IV.

The second neuro-symbolic method [178] for solving SDM tasks was developed as part of a research collaboration with the MIT-IBM Watson AI Lab. It entails a novel approach for improving **heuristic learning** methods by leveraging the prior knowledge contained in symbolic, domain-independent, admissible heuristics. The decisions made (sometimes unknowingly) in the heuristic learning literature are analyzed from a statistical lens. From this analysis, it is concluded that Mean Squared Error (MSE) is not an appropriate loss to train heuristic learning methods. Therefore, a **novel loss function** is proposed, which models the heuristic to be learned as a *Truncated* Gaussian instead of as a Gaussian distribution, as MSE does. The lower bound of this Truncated Gaussian distribution is set to the value of an admissible heuristic, thereby leveraging the prior information it contains. This novel loss function is compared to MSE in several heuristic learning scenarios, comprising different planning domains and ML models, including neuro-symbolic ones. Results show the proposed loss function improves convergence speed during training and yields more accurate heuristics that result in better planning performance. This third contribution is addressed in Chapter V.

As the fourth major contribution of this thesis [181], a novel **neuro-symbolic method for learning the structure of SDM tasks** is introduced. The method, named ***NeSIG*** **(Neuro-Symbolic Instance Generator)**, learns to automatically generate problems for any particular planning domain so that they are valid, diverse and difficult to solve. Having a large set of planning problems is useful for many applications, e.g., as benchmarks for comparing the performance of AP algorithms and for obtaining data to train ML methods, such as those proposed as the second and third contributions of this thesis. NeSIG follows an incremental approach to problem generation. It first obtains the initial state of the problem by sequentially adding atoms and objects to it. Then, NeSIG obtains the problem goal by executing actions at the generated initial state. This generation process is guided by two generative policies, trained with RL to obtain valid, diverse and difficult problems.

Since information about consistency (a subproperty of validity) must be specified by the user, a novel semi-declarative language is implemented for doing so in an easy manner. NeSIG is evaluated by generating problems for several planning domains. Results show it is able to generate valid and diverse problems of greater difficulty than handcrafted, domain-specific generators, while requiring less human effort. Additionally, it successfully generates larger problems than those seen during training. This fourth contribution is addressed in Chapter VI.

All the code developed in the course of this thesis has been made publicly available. A subset of this code has also been released as several **open-source Python packages** and distributed via the Python Package Index (PyPI). The list of Python packages developed are the following: ***lifted-pddl*** [174], a lightweight parser for AP (currently accumulating 14000 downloads according to the *pepy.tech* website); ***stable-truncated-gaussian*** [177], a stable and differentiable implementation of the Truncated Gaussian distribution using Pytorch (13000 downloads); ***neural-logic-machine*** [176], a simple Pytorch implementation of Neural Logic Machines, a deep neural network suitable for logic data (2000 downloads); and ***pddl-prover*** [175], a prover for automatically evaluating conditions (logic formulas) on planning states (2000 downloads).

In conclusion, this dissertation presents significant advancements on neuro-symbolic AI for SDM, comprising a review that proposes a novel taxonomy of SDM methods and justifies the need for neuro-symbolic approaches, two different neuro-symbolic methods for improving the efficiency of AP algorithms (either by selecting subgoals with RL or learning better planning heuristics) and, lastly, a neuro-symbolic technique for automatically generating planning problems. In addition, several open-source packages have been distributed and have received significant attention from the research community, judging from the large number of downloads.

Chapter I

Introduction

"In preparing for battle I have always found that plans are useless, but planning is indispensable." — Dwight D. Eisenhower

Sequential Decision Making (SDM) [145] is an important subfield within AI devoted to solving **Sequential Decision Processes (SDPs)**. In an SDP, an agent situated in an environment, which can be either physical or virtual, must make a series of decisions in order to complete a task or achieve a goal. These ordered decisions must be selected according to some optimality criteria, such as the maximization of reward or the minimization of cost. SDPs provide a general and powerful framework which has been successfully applied to solve problems in fields as diverse as robotics [133], logistics [205], games [217], finance [36] and natural language processing [242], just to name a few.

Throughout the years, many AI methods have been proposed to solve SDPs, i.e., find the sequence of decisions which optimizes the corresponding metric, such as reward or cost. They can be grouped in two main categories: **Automated Planning (AP)** [84] and **Reinforcement Learning (RL)** [224]. These two paradigms mainly differ in **how they obtain a solution** and **how they represent their knowledge**:

- **AP** techniques exploit the existing prior knowledge about the environment dynamics, encoded in what is known as the action model or planning domain, to carry out a search and reasoning process in order to find a valid plan or policy, i.e., a mapping from states to actions used to achieve a set of goals. They can be grouped up according to the type of knowledge representation employed. Many of them require a symbolic description of the action model in a formal language often based on first-order logic (FOL), such as PDDL [96] (for deterministic tasks) or PPDDL [253] and RDDL [202] (for stochastic tasks). Some AP techniques do not have such requirement and can represent the action model in a subsymbolic manner, often as a *black-box* which outputs the next state resulting from the application of a given action at the current state. We will refer to the first group of techniques as **Symbolic Planning (SP)**, and use the name **Subsymbolic/Non-Symbolic Planning (NSP)** for the latter.

- **RL**, on the other hand, seeks to learn the policy (mapping from states to actions) that maximizes reward, automatically from data with no planning whatsoever. The main precursor of RL is Optimal Control [18, 224], a field primarily concerned with providing optimal solutions to SDPs with complete knowledge of the dynamics using dynamic programming methods. Unlike their predecessor, most RL methods, known as model-free RL, focus on obtaining approximate solutions to SDPs where the action model is unknown. Additionally, the vast majority of RL methods represent their learned knowledge in a subsymbolic manner, although a subset of them, known as relational RL, use a symbolic knowledge representation. Classical RL methods employ a tabular representation, i.e., for every possible state-action pair they store the corresponding policy information, usually the expected future reward. On the other hand, modern RL methods, known as **Deep RL (DRL)** [74], represent their policy as a **deep neural network (DNN)** [142], which allows them to generalize their learned knowledge, not needing to store information about the policy for every single state-action pair.

Although AP and RL share the common goal of solving SDPs, each field has historically followed a separate path: the former proposing an often **symbolic, reasoning-based** paradigm, whereas the latter opts for a predominantly **subsymbolic, learning-based** approach. This division between RL and AP represents a particular instance of a larger discussion which has taken place throughout the history of AI, confronting two approaches: **symbolic** and **subsymbolic** AI. The symbolic approach states that symbol manipulation constitutes an essential part of intelligence [199] and was the predominant view during the majority of the 20th century. In the 21th century however, this claim has been highly contested due to the great success of Machine Learning (ML), specifically **Deep Learning (DL)** [142], in many real-world problems such as natural language processing [28] and computer vision [138], while performing no symbol manipulation whatsoever.

Nevertheless, despite the recent successes of ML and DL, there exists great interest in the reconciliation of both AI paradigms in what is often referred to as ***neuro-symbolic AI***, a hybrid approach that attempts to **combine the DNNs employed in subsymbolic, DL methods with the symbolic knowledge representations and reasoning capabilities of symbolic AI**. Many researchers [47, 78, 139, 153, 215] advocate for hybrid, neuro-symbolic approaches as a way to harness the strengths of ML and DL while addressing their limitations, such as data-inefficiency, poor generalization and lack of interpretability [12, 77, 152]. The need for neuro-symbolic AI is also supported by research from other cognitive sciences. One of the most prominent examples comes from the work of the psychologist Daniel Kahneman. In his famous book *Thinking Fast and Slow* [121], Kahneman explains how the human mind has two different modes of thinking: *System I*, which is fast and intuitive, and *System II*, which is slow and logical. Many AI researchers have drawn parallels between Kahneman's System I and subsymbolic AI, and System II and symbolic AI arguing that, just as humans rely on both types of thinking, integrating the symbolic and subsymbolic paradigms is crucial for developing AI systems that more closely resemble the human mind.

Analogously, many works have pursued this unification for the particular case of SDM. As a result, there have been many proposals which try to **bridge the gap between RL and AP**, with a surge of interest in recent years (see Figure 1). A few notable examples are: model-based RL [160], relational RL [227], ML and DL methods for learning the prior knowledge of AP (e.g., planning heuristics and action models) [117], models which *learn to plan* [160] and, lastly, **neuro-symbolic methods for SDM** [182].

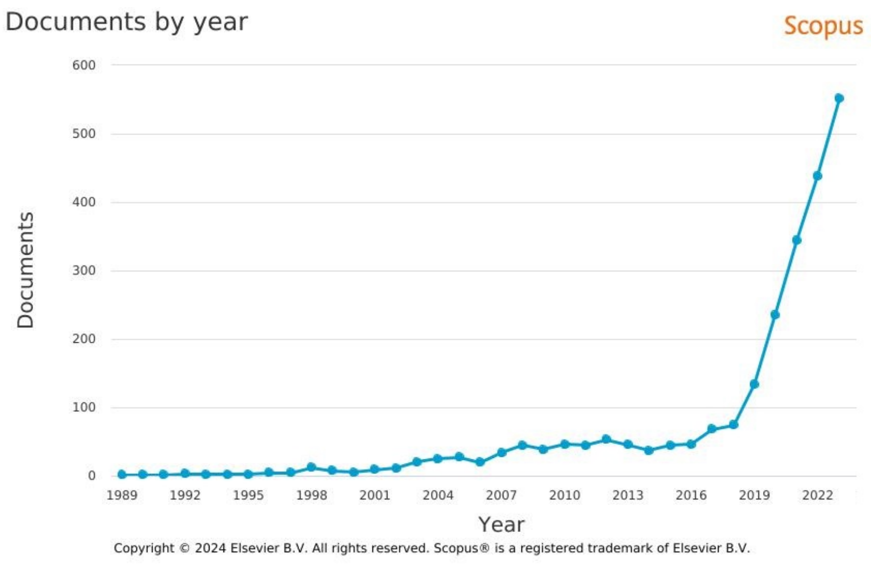


Figure 1: **Number of publications that integrate AP and RL.** This figure was obtained by introducing the following query in Scopus (search performed on February 5 2024): *TITLE-ABS-KEY (("reinforcement learning" AND "automated planning") OR ("model-based reinforcement learning" OR "model-based RL") OR ("relational reinforcement learning" OR "relational RL") OR ("automated planning" AND ("machine learning" OR "deep learning")) OR ("learn to plan" OR "learning to plan") OR (neurosymbolic OR neuro-symbolic OR neuralsymbolic OR neural-symbolic)) AND (LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "MATH") OR LIMIT-TO (SUBJAREA , "ENGI")) AND PUBYEAR > 1979 AND PUBYEAR < 2024 .* © Elsevier B.V.

This dissertation focuses on the study and implementation of neuro-symbolic AI for SDM, making several significant contributions to the field. These include theoretical contributions, such as a comprehensive review of the field of SDM that serves to justify the need for neuro-symbolic methods, and empirical advances, comprising neuro-symbolic methods for both solving SDPs and learning their structure. The following subsections detail the objectives and contributions of this thesis, both in the form of scientific publications and open-source software, and outline the structure of the rest of the document.

I.1 Objectives

The primary objective of this thesis is to **advance the state of the art in neuro-symbolic AI for SDM**, mostly by the integration of AP techniques with DL methods, including DRL. In order to achieve this, four more concrete subgoals are proposed, one of them (**G1**) being **theoretical** in nature (i.e., a review) whereas the others are **empirical**. The three empirical subgoals can be further subdivided in two groups according to the taxonomy proposed in the review of **G1**, which classifies SDM methods into those that solve SDPs and those that learn their structure. The first group, comprising subgoals **G2** and **G3**, corresponds to **neuro-symbolic methods for solving SDPs.** More specifically, it entails methods for **improving the performance of SP** algorithms, either through goal selection (**G2**) or heuristic learning (**G3**). The second group contains a single subgoal, **G4**, corresponding to a **neuro-symbolic method for learning the structure of SDPs**. In particular, **G4** pursues the automated generation of planning problems which, among many other applications, can be used to obtain synthetic data for training DL methods, such as those in **G2** and **G3**. The four previous subgoals are illustrated in Figure 2 and are further detailed below:

- **G1 - Review of SDM**. The first subgoal of this thesis is to write a comprehensive review of SDM, comprising symbolic (e.g., SP), subsymbolic (e.g., DRL) and hybrid (e.g., neuro-symbolic) methods for both solving SDPs and learning their structure (e.g., the action model). The existing SDM methods must be compared across several dimensions, and their advantages and disadvantages must be analyzed. Additionally, this review should also serve to justify the need for neuro-symbolic AI, e.g., as a promising approach towards achieving an ideal method for SDM.

- **G2 - Goal Selection**. The second subgoal is to improve the time efficiency of SP algorithms so that they can be applied to solve tasks with real-time restrictions, without severely degrading the quality (i.e., plan length) of the solutions obtained. In order to achieve this, the proposed method will have access to prior information about *subgoals* so that, by achieving a subset of them in the correct order, the task can be solved optimally. Therefore, the method must be neuro-symbolic in nature and use DRL to learn to select a sequence of subgoals so that, when these are achieved by the SP algorithm, the resulting plan solves the corresponding task in as few actions as possible.

- **G3 - Heuristic Learning**. This subgoal is framed within a research collaboration between the **University of Granada** and the **MIT-IBM Watson AI Lab**. It aims to improve the performance of current SP algorithms by developing better *heuristic learning* methods, i.e., methods that utilize ML or DL to learn planning heuristics from data. To do so, the proposed heuristic learning approach must leverage existing prior knowledge in the form of symbolic, admissible heuristics which, so far, have only been used as training targets in the related literature. Additionally, the different decisions made when designing the method must be well-justified and adhere to established principles in the field of Statistics, such as the *Principle of Maximum Entropy.* Finally, the resulting approach must be general and flexible, being able to improve the quality of learned heuristics in a wide variety of scenarios, such as for different planning domains and ML models.

- **G4 - Problem Generation**. The final subgoal is to design and implement a method that automatically learns to generate problems for any given planning domain, encoded in the PDDL language. Generated problems must satisfy certain properties. Firstly, they must be *valid*, i.e., they must be solvable and their initial state must be consistent, meaning it represents a possible initial situation in the system modeled by the corresponding planning domain. Secondly, they must be *diverse*, i.e., the proposed method must be able to generate many different types of problems. Thirdly, they must be *difficult* to solve by any given state-of-the-art SP algorithm. In addition to generating valid, diverse and difficult problems, the method should require less prior knowledge and human effort than handcrafted, domain-specific problem generators, which need to be tailored to each particular domain.

By achieving these subgoals, this dissertation aims to significantly advance the field of SDM through the study and implementation of neuro-symbolic AI, demonstrating its potential for effectively bridging the gap between AP and RL.

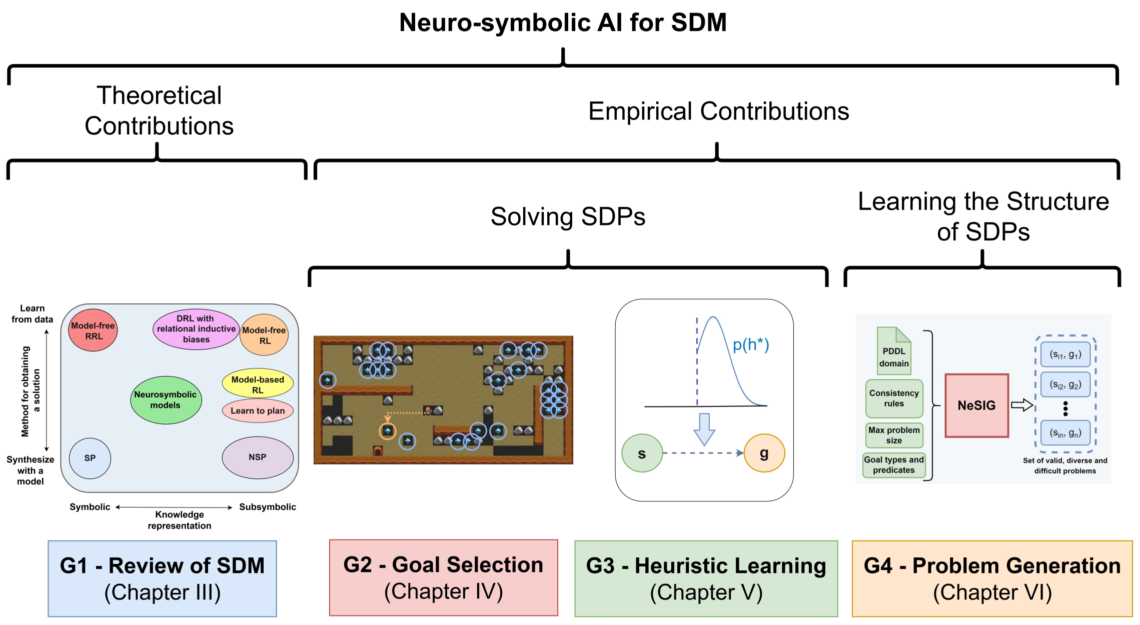

Neuro-symbolic AI for SDM
Theoretical Contributions
Empirical Contributions
Solving SDPs
Learning the Structure of SDPs
Learn from data
Method for obtaining a solution
Synthesize with a model
Model-free RRL
DRL with relational inductive biases
Model-free RL
Neurosymbolic models
Model-based RL
Learn to plan
SP
NSP
Symbolic
Subsymbolic
Knowledge representation
p(h*)
s
g
PDDL domain
Consistency rules
Max problem size
Goal types and predicates
NeSIG
(s_{i1}, g_1)
(s_{i2}, g_2)
(s_{in}, g_n)
Set of valid, diverse and difficult problems
G1 - Review of SDM (Chapter III)
G2 - Goal Selection (Chapter IV)
G3 - Heuristic Learning (Chapter V)
G4 - Problem Generation (Chapter VI)

Figure 2: **Diagram of contributions.** The figure shows the four main contributions of this PhD dissertation: a review of the state of the art in SDM (G1), a neuro-symbolic method for learning to select goals (G2), a neuro-symbolic method for improving heuristic learning (G3) and, lastly, a neuro-symbolic method for generating planning problems (G4). These contributions can be classified as either theoretical or empirical. Additionally, empirical contributions can be subdivided in methods for solving SDPs and those for learning their structure.

I.2 Publications

All the subgoals proposed in Section I.1 have been successfully achieved during the development of this thesis, and their results published in several peer-reviewed journals and conference proceedings. Each of these papers is associated with a subgoal of the thesis, with the exception of [173]. This paper contains the thesis proposal that was presented during the Doctoral Consortium at IJCAI 2022, and was published in the conference proceedings of that year. The complete list of publications with associated subgoals is provided below:

Works published in JCR-indexed journals

G1. Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. (2024c). A review of symbolic, subsymbolic and hybrid methods for sequential decision making. *ACM Comput. Surv.*, 56(11):1–36 — Impact Factor (2023): 23.8 (Q1, D1), 1st/143 in subject category “Computer Science, Theory & Methods”

G2. Núñez-Molina, C., Fernández-Olivares, J., and Pérez, R. (2022). Learning to select goals in automated planning with deep-q learning. *Expert Syst. Appl.*, 202:117265 — Impact Factor (2022): 8.5 (Q1), 22nd/145 in subject category "Computer Science, Artificial Intelligence"

Works published in conferences

G4. Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. (2024b). NeSIG: A neuro-symbolic method for learning to generate planning problems. In *ECAI*, volume 392, pages 4084–4091 — GGS Conference Rating: A- (Class 2)

G3. Núñez-Molina, C., Asai, M., Mesejo, P., and Fernández-Olivares, J. (2024a). On using admissible bounds for learning forward search heuristics. In *IJCAI*, pages 6761–6769 — GGS Conference Rating: A++ (Class 1)

G1-4. Núñez-Molina, C. (2022a). Application of neurosymbolic AI to sequential decision making. In *IJCAI 2022 Doctoral Consortium*, pages 5863–5864 — GGS Conference Rating: A++ (Class 1)

G2. Núñez-Molina, C., Vellido, I., Nikolov-Vasilev, V., Pérez, R., and Fdez-Olivares, J. (2021). A proposal to integrate deep q-learning with automated planning to improve the performance of a planning-based agent. In *CAEPIA*, pages 23–32 — GGS Conference Rating: not ranked

Research Works under Review

G4. Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. Automated planning instance generation with neuro-symbolic AI. *Artif. Intell.* Submitted on October 3, 2024 — Impact Factor (2023): 5.1 (Q1), 44th/197 in subject category "Computer Science, Artificial Intelligence"

I.3 Software contributions

All the code developed during the course of this thesis has been made publicly available, ensuring full reproducibility of the results presented in the publications of Section I.2. A subset of this code is provided in the form of standalone, open-source Python packages, distributed using the Python Package Index (PyPI), in order to facilitate its use in other projects. The complete list of released packages can be found below, along with their total number of downloads (according to the *pepy.tech* website) by December 2, 2024:

- **lifted-pddl** [174] (13,599 downloads). This project implements a lightweight framework for the efficient parsing and manipulation of PDDL. It achieves this by working on the PDDL description in its *lifted* form, i.e., without grounding it first. This framework provides functionality for parsing PDDL domain and problem files and inspecting their elements, obtaining the list of applicable functions at a given state, and obtaining the next state resulting from applying a particular action to the current state (successor function).

- **stable-truncated-gaussian** [177] (12,865 downloads). This package was developed as part of [178]. It contains a differentiable implementation of the Truncated Gaussian (Normal) distribution using Pytorch, which is numerically stable even when the μ parameter of the distribution lies outside the interval $[a, b]$ given by its bounds. In this situation, a naive evaluation of the mean, variance and log-probability of the distribution could result in *catastrophic cancellation* and other numerical issues. At the moment, this package provides numerically-stable methods for calculating the mean, variance, log-probability, KL-divergence and sampling from the distribution.

- **pddl-prover** [175] (1,924 downloads). This project contains the implementation of the novel semi-declarative language used for encoding *consistency constraints* in [181, Núñez-Molina et al.]. More generally, it implements a Python prover for evaluating conditions on PDDL states. It provides a declarative syntax for constructing arbitrary FOL formulas (including counting quantifiers) in Python, along with an algorithm for evaluating the truth value of these formulas on a knowledge base. This knowledge base must be encoded in the same manner as PDDL states, as a set of true FOL atoms.

- **simple-nlm** [176] (2,026 downloads). This package provides a simple-to-use, Pytorch implementation of Neural Logic Machines (NLMs) [53], which has been utilized in several of the works of this thesis [181, 178, Núñez-Molina et al.]. It contains two different NLM implementations: one is time-efficient and specifically optimized for GPU usage, whereas the other sacrifices speed in order to reduce memory consumption.

I.4 Structure of the dissertation

This dissertation is organized into five main parts in addition to the Introduction: *Fundamentals*, *Related Work*, *Proposals*, *Final Remarks* and *Appendix*.

Part I, *Fundamentals*, covers the theoretical background required in order to understand this dissertation. It first provides a general formulation of SDM tasks in terms of Stochastic Shortest-Path Markov Decision Processes (SSP MDPs). Then, it introduces the field of AP, explaining how AP tasks can be represented and the concept of heuristics. Next, it briefly overviews the field of ML, describing two of the DNN architectures used in this thesis: Convolutional Neural Networks (CNNs) and Neural Logic Machines (NLMs). Finally, it outlines the field of RL, focusing on the Q-Learning, Deep Q-Learning and Proximal Policy Optimization (PPO) algorithms.

Part II, *Related Work*, provides a comprehensive review of existing work in the SDM literature, thus aligning with the first subgoal (G1) of this dissertation (see Section I.1) and framing the role of the contributions presented in the overall landscape of SDM. This review covers both methods for solving SDM tasks (e.g., AP and RL) and methods for learning their structure (e.g., the action model). Additionally, it analyzes the advantages and disadvantages of existing approaches and theorizes about the characteristics of an ideal method for SDM. Based on this analysis, it is concluded that neuro-symbolic AI poses a promising approach for achieving an ideal method for SDM via the integration of AP and RL, thus justifying the relevance and significance of this doctoral dissertation.

Part III, *Proposals*, presents the empirical contributions of this thesis, corresponding to subgoals G2, G3 and G4. Chapter IV describes the Deep Q-Planning (DQP) algorithm, a neuro-symbolic method for improving the performance of AP in real-time scenarios via goal selection, thus aligning with subgoal G2. Chapter V describes a statistically-motivated, neuro-symbolic approach for leveraging the prior knowledge contained in symbolic, admissible heuristics in order to improve heuristic learning and, therefore, the performance of AP algorithms, which aligns with subgoal G3. Finally, Chapter VI describes NeSIG (Neuro-Symbolic Instance Generator), a neuro-symbolic technique for generating valid, diverse and difficult problems for any Classical Planning domain with low human effort, thus aligning with subgoal G4.

Part IV, *Final Remarks*, summarizes the conclusions of this doctoral dissertation, suggests possible avenues for future work to extend the contributions presented, provides the acknowledgements, and lists the complete bibliography.

Finally, **Part V, *Appendix***, provides appendixes for Chapters V and VI.

Part I

Fundamentals

Chapter II

Theoretical Background

II.1 Sequential Decision Processes

The field of SDM aims to solve tasks where an agent, situated in either a virtual or physical environment, must make a series of decisions or execute a sequence of actions in order to achieve a particular goal. These tasks receive the name of **Sequential Decision Processes (SDPs)**. This thesis focuses on SDPs with a finite number of states and actions (decisions) and where time is discrete, i.e., after the execution of an action the environment immediately transitions from time instant t to $t + 1$. For totally observable environments where the agent has access to full information about the current state, this type of SDPs are commonly described as a finite Markov Decision Process (MDP) [224]. However, there exist different alternative formulations for MDPs. In this dissertation, we will use the one given by finite **Stochastic Shortest-Path MDPs (SSP MDPs)** [167], as they provide a general MDP formulation which suits both AP and RL. An SSP MDP, which is depicted in Figure 3, is constituted by the following elements:

- **State space** S, the finite set of states of the system. In some SSP MDPs, at the beginning of the task the agent always starts from a state s randomly sampled from a set of initial states $S_i \subset S$. In other SSP MDPs, the agent may start from any state $s \in S$, i.e., $S_i = S$.

- **Action space** A, the finite set of actions the agent can execute. In some SSP MDPs, only a subset of applicable actions $App(s) \subset A$ are available to the agent at a given state s. In other SSP MDPs, the agent can execute every action at every state, i.e., $App(s) = A \ \forall s \in S$.

- **Transition function** $T : S \times A \times S \to [0, 1]$. It describes the dynamics of the environment, by specifying the probability $T(s, a, s') = P(s'|s, a)$ of the environment (SSP MDP) transitioning into state s' after the agent executes an (applicable) action $a \in App(s)$ at the current state s. If given some state $s \in S$ and action $a \in App(s)$ the environment always transitions into the same state s' (i.e., $P(s'|s, a) = 1$ and $P(s''|s, a) = 0 \ \forall s'' \neq s'$), the MDP is said to be deterministic. Otherwise, it is stochastic.

- **Cost function** $C : S \times A \times S \to [0, \infty)$. It gives a finite strictly positive cost $c(s, a, s') > 0$ when the agent goes from state s to s' by executing the (applicable) action a. Transitions from a goal state are the only ones with a cost of zero.

- **Goal set** $G \subseteq S$. It contains a finite set of goal states $s_g \in S$, one of which must be reached by the agent. For every goal state $s_g \in G$, action $a \in A$ and non-goal state $s \notin G$, the following conditions are met: $T(s_g, a, s_g) = 1$, $T(s_g, a, s) = 0$, $c(s_g, a, s_g) = 0$. Intuitively, these conditions mean that goal states are terminal, since once reached the agent cannot leave them and no longer incurs in additional costs.

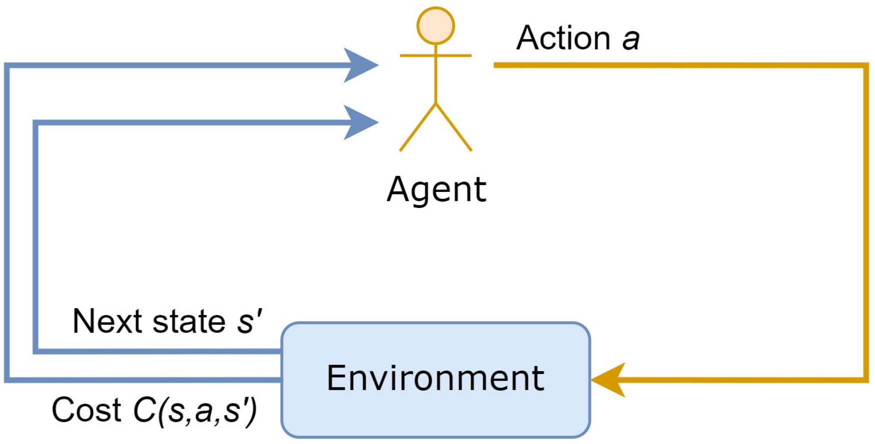


Figure 3: **SDM task, as formulated by SSP MDPs.** At each step, the agent must execute an action $a \in App(s)$ that is applicable at the current state s of the SSP MDP. Then, the agent receives from the environment the next state s' and associated cost $C(s, a, s')$. This process repeats until the agent eventually reaches a goal $g \in G$.

A **policy** $\pi : S \times A \to [0, 1]$ is a (possibly partial) function that maps states $s \in S$ to probability distributions over applicable actions $a \in App(s)$. The **state value** (or simply *value*) $V^\pi(s)$ of a state s under some policy π is equal to the total cost that we expect to obtain if, starting from s, we follow policy π until G is reached. Similarly, a **Q-value** $Q^\pi(s, a)$ represents the expected cost obtained by first executing action a at s and, then, executing actions according to π until G is reached. State values and Q-values are related by the following equation: $V^\pi(s) = \sum_{a \in App(s)} \big(\pi(a|s) \cdot Q^\pi(s, a)\big)$. A solution of an SSP MDP is an optimal policy π^*, i.e., a policy that minimizes the expected cost needed to reach G. If π^* is optimal, then its value function V^* satisfies the following property: $V^*(s) \leq V^\pi(s)\ \forall s, \pi$. A policy is said to be *complete* if it is defined for every MDP state $s \in S$. However, for MDPs with a single initial state s_i it is often more efficient to only compute a policy π for some subset of states $S' \subset S$, e.g., those reachable from s_i by following π. Such a policy is deemed *partial*. Finally, if both a policy π and MDP are deterministic and the MDP contains a single initial state s_i, then π can be simply represented as a plan, i.e., as a sequence of actions $a_0, a_1, ..., a_n$ that reach G when executed from s_i. This can be done because, in the deterministic setting, the execution of π from some state s_0 always results in the same sequence (trajectory) of states and actions $s_0, a_0, s_1, a_1, ..., s_{n-1}, a_{n-1}, s_n$. Figure 4 shows a graphical comparison between the three different types of policies.

We previously stated that SSP MDPs are suitable for both AP and RL. However, in most RL tasks the goal is to find a policy π^* which maximizes the total sum of

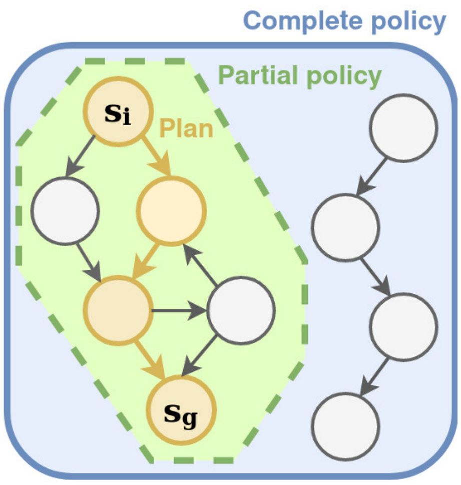


Figure 4: **Comparison between complete policies, partial policies and plans.** Nodes in the image represent MDP states, and arrows show the possible transitions between them. A complete policy is defined for the entire MDP state space S (blue square in the picture). A partial policy is defined for a subset $S' \subset S$ of states (dashed green area in the picture). Finally, a plan only stores the action to execute for the states $S'' \subset S'$ of a single trajectory from s_i to some s_g (gold-coloured nodes with bold emphasis in the picture).

rewards, known as the **return**, instead of minimizing the cost needed to reach a goal state $s_g \in G$. In **finite-horizon (FH)** MDPs, the return must be maximized over a finite number of time steps, known as an **episode**. In **infinite-horizon (IFH)** MDPs on the other hand, the goal is to maximize the discounted return, where each individual reward is scaled down by a factor γ, over an infinite number of time steps. It can be proven that both types of reward-based MDPs, FH and IFH MDPs, can be expressed as an equivalent SSP MDP, in terms of goals and costs instead of rewards [167]. Therefore, the reward-based MDPs usually employed in RL are merely subclasses of the more general SSP MDPs. For this reason, in this dissertation we will interchangeably employ the reward-based formulation of RL and the goal-based formulation of AP, knowing that both of them can be expressed as SSP MDPs.

Finally, all types of MDPs exhibit a very important feature, known as the *Markov property*. This property states that, in any MDP, the cost (or reward) and transition function only depend on the current state of the MDP and the action executed by the agent at that state. This property allows agents to select actions by only considering the information about their current state and not their past history, i.e., past states and actions. If the environment is partially observable, i.e., the agent lacks information about the current state, the SDP must be described as a **Partially Observable Markov Decision Process (POMDP)** [148]. POMDPs share the same formulation as (totally observable) MDPs and also follow the Markov property. However, the current state of POMDPs is hidden and the agent only receives partial information in the form of observations about the state. These observations must then be used by the agent to infer the actual state of the environment and solve the POMDP.

II.2 Automated Planning

Automated Planning (AP) [84] proposes a deliberative paradigm for solving SDPs. In order to be applied, AP methods require an **action model** (also known as a world model or planning domain) containing information about the environment dynamics and how the agent can affect them, i.e., the available actions for the agent and how these affect the state of the world. This prior knowledge is then leveraged by AP techniques to conduct a reasoning process, often involving search, in order to find a policy or plan that achieves the MDP goals from the initial state set S_i.

Throughout the years, many different AP algorithms have been proposed, which can be classified according to several criteria. Firstly, based on their knowledge representation, AP methods can be categorized in **Symbolic Planning (SP)** and **Subsymbolic/Non-Symbolic Planning (NSP)** [182]. The first group of methods requires a symbolic specification of the action model, often in a formal logic-based language such as PDDL or RDDL (see Section II.2.1), describing the environment dynamics in full detail. Conversely, the second group of methods does not require any particular representation (e.g., in PDDL) for the action model. Generally, a *black-box* action model suffices, i.e., a model that returns the next state (and cost/reward, if needed) resulting from the application of a particular action at a given state, without providing any further information about the environment dynamics. Additionally, there exist novel AP methods that combine the symbolic and subsymbolic knowledge representations. For instance, the approach proposed in [73] only requires a symbolic description for states and goals, so actions and their effects can be encoded as a black-box procedure.

AP techniques can also be split according to the type of MDPs they can be applied to solve. **Probabilistic Planning (PP)** [167] methods solve the general class of (stochastic or deterministic) MDPs, and are used to find a policy that minimizes the expected cost needed to reach a goal $g \in G$. Some PP algorithms compute complete policies, i.e., they find the optimal action to take for every MDP state $s \in S$. For efficiency purposes, other PP algorithms compute partial policies, i.e., they only find the optimal action to take for a subset of the states in S, such as those reachable from the initial state set S_i. Contrary to PP methods, **Classical Planning (CP)** [84] focuses on solving deterministic MDPs with single initial state s_i. Solving this particular subclass of MDPs entails finding the cost-optimal plan that reaches a goal $g \in G$ from s_i. An example of a CP task, represented symbolically using PDDL, is depicted in Figure 5. Finally, Chapter III provides specific examples for the different categories of AP methods described in this section.

II.2.1 Planning Task Representation

As explained in the previous section, many AP algorithms, known as SP, require a symbolic description of the action model. In these cases, AP tasks are usually represented using a declarative, FOL-based language such as **PDDL** [96], which stands for *Planning Domain Definition Language*. In PDDL, tasks are split into two different items: a planning **domain** and a planning **problem**. The PDDL domain represents the *general* aspects of the task, corresponding to the state encoding and environment dynamics. More specifically, it describes the existing types of objects, predicates (or relations) and the actions available to the agent. For each action,

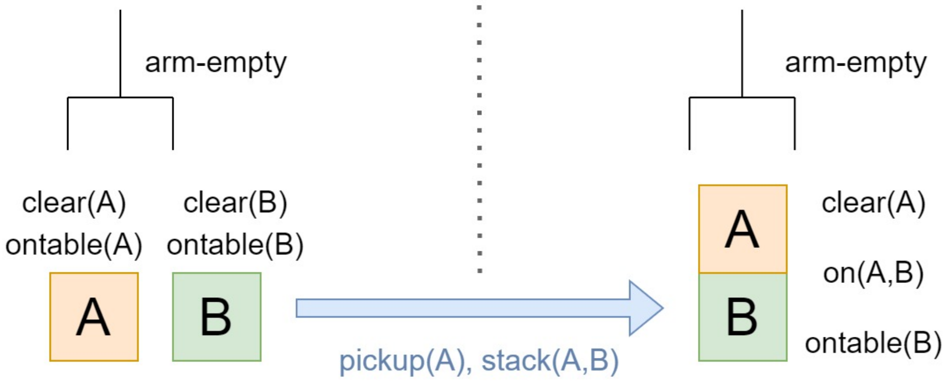
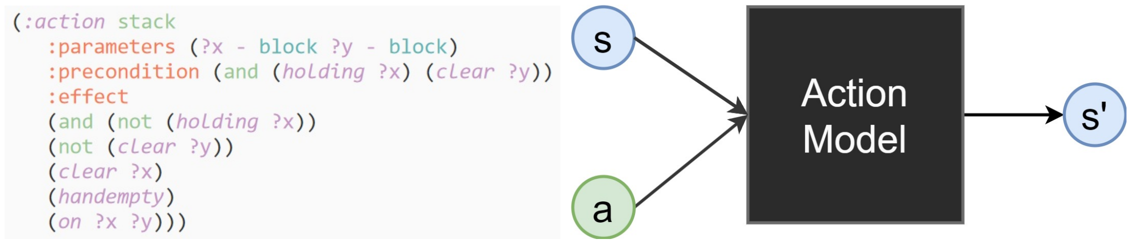


Figure 5: **CP task, encoded using PDDL.** The task belongs to the PDDL domain known as *blocksworld*, consisting of blocks that can be stacked one upon another with a gripper arm. The blue arrow represents a plan that achieves a goal state (right) starting from the initial state of the task (left).

it details which conditions need to be true at the current state for it to be applied (known as *preconditions*), and the atoms (facts) that will be made true (*positive/add effects*) and false (*negative/delete effects*) once it is executed. Figure 6 shows an extract from the *blocksworld* PDDL domain. In contrast, the PDDL problem encodes the *specific* aspects of the task, corresponding to the particular objects it contains, the initial state s_i (described as the set of atoms that are initially true), and the goal G to achieve (often described as a conjunction of atoms that must be made true). The elements in the PDDL domain are represented in *lifted* form, i.e., in terms of FOL variables. These variables will then be instantiated on the objects of a particular problem to obtain the *ground* representation of the planning task, required by the vast majority of SP algorithms.



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

Figure 6: **Symbolic and subsymbolic action models. Left:** Extract from a symbolic action model, corresponding to the PDDL description of action *stack* in the *blocksworld* domain. Given the ground, symbolic description of a state s, as a set of true atoms, and an applicable action a, it makes it possible to obtain the resulting next state s' by applying the effects of a to s. **Right:** Subsymbolic action model, corresponding to a black-box. It receives a subsymbolic representation of s (e.g., as an image or as a set of state variables) and a as inputs, and outputs s'. The symbolic action model is amenable to interpretation, whereas the subsymbolic one is usually not.

Representing planning tasks as a domain-problem tuple offers great reusability, as the same PDDL domain can be utilized by all the tasks that model the same system/environment and which only differ in their initial state and goal to achieve. For instance, all *blocksworld* tasks model the same system: a table filled with blocks that can be stacked on top of each other with a gripper. Therefore, we can use a single planning domain to encode these common elements, and a different planning problem for each task, describing the particular initial and goal configurations of the blocks.

One limitation of PDDL is that it is only useful for encoding deterministic tasks, such as those in CP. PDDL is extended to the general, PP case by the **PPDDL** [253] language, which adds support for stochastic action effects that occur with a given probability. **RDDL** [202] is an alternative language for modeling PP tasks. In RDDL, everything (including actions) is represented as a parameterized variable of a particular type. This modeling approach is better suited than PPDDL for domains where actions have many uncorrelated effects, as it is often the case in systems with many objects that mostly evolve independently from each other.

II.2.2 Planning Heuristics

Planning is computationally expensive. Solving the general class of (stochastic) SSP MDPs (as in PP) is EXPTIME-complete, whereas solving deterministic SSP MDPs with a single initial state (as in CP) is PSPACE-complete [167]. For this reason, if we hope to apply AP methods to real-world problems, we need to harness *control knowledge* to direct the search, which often comes in the form of **planning heuristics** (see Figure 7).

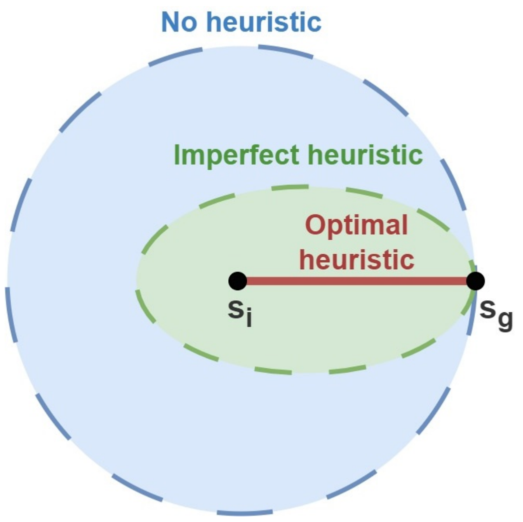


Figure 7: **Planning with a heuristic.** The figure illustrates how heuristics help reduce planning effort. For simplicity, we depict the case where the MDP is deterministic and search is carried out from the initial state s_i to a goal state s_g. When no heuristic is employed, the planning algorithm needs to explore the state space in all directions until s_g is finally found (see blue circle in the image). A heuristic can prevent this by providing guidance and reducing the number of states that are explored (green ellipse in the image). Finally, if this heuristic is optimal/perfect (i.e., it predicts the optimal cost for every state $s \in S$), only those states on the optimal plan(s) from s_i to s_g need to be explored. Analogously, for stochastic MDPs, the only states that need to be explored are those reachable from s_i by following the optimal policy π^*.

A heuristic $h(s, P)$ is a function of a planning problem P and a state s of such problem which, for deterministic MDPs, estimates the cost of the optimal plan from s to the goal G of P. This optimal cost is sometimes referred to as the **perfect heuristic** h^*. In the case of stochastic MDPs, heuristics provides an estimate of the expected cost from s to G under the optimal policy π^*, i.e., they estimate $V^*(s)$. Heuristics are said to be **admissible** if, for every state $s \in S$, their value $h(s, P)$ is no larger than the optimal cost $h^*(s, P)$. In other words, admissible heuristics provide an *optimistic* estimate of the cost to reach the goal. When heuristics do not meet this property, they are called inadmissible. Additionally, heuristics can be classified into domain-specific, i.e., those that are tailored to problems from a specific domain, and domain-independent, i.e., those that are applicable to many different planning domains, such as the entire set of CP domains. Finally, in situations where heuristics are unavailable, they can be learned from data using ML, an approach that receives the name of **heuristic learning**. Examples of both heuristics and heuristic learning methods can be found in Chapter III.

II.3 Machine Learning

Machine Learning (ML) [20] is an important AI subfield comprising methods for *learning from data*. ML methods are often classified according to the type of data they learn from in Supervised Learning, Unsupervised Learning and Reinforcement Learning. In a **Supervised Learning** task, the goal is to predict the value of a target variable y associated with a particular input x. Supervised Learning techniques learn the relationship between x and y from examples of input-output pairs (x, y). On the other hand, in an **Unsupervised Learning** task data is unlabelled, i.e., the target variable y is absent and only x is available. Unsupervised Learning methods make possible to learn the structure of unlabelled data and extract patterns from it. Examples of methods in this category include: clustering [203] (i.e., organizing data in groups sharing similar characteristics), dimensionality reduction [116] (i.e., representing data with a smaller number of features), and generative modelling [23] (i.e., learning the data distribution in order to generate new samples). Lastly, **Reinforcement Learning (RL)** comprises methods that are able to learn from a reward signal measuring the quality of a sequence of agent's decisions or actions. Whereas in Supervised Learning we know the correct output y (i.e., answer) for each input x in the training dataset, in RL we only receive feedback about how *good* a particular sequence of outputs $y_1, y_2, ..., y_n$ (actions) is.

Throughout the years, many different learning models have been proposed, of which a few notable examples are: logistic regression [45], support vector machines [44], Bayesian networks [189], decision trees [193], and neural networks [198]. Among these models, neural networks have enjoyed great popularity due to many reasons, such as their biological inspiration in the human brain and their universal function approximation capabilities. For many years, the training of neural networks with many layers, i.e., **deep neural networks (DNNs)**, proved to be a difficult endeavour, mainly due to the presence of unstable (either *exploding* or *vanishing*) gradients in the training process [17]. Nonetheless, important advances in the last decades, such as novel activation functions [166], better optimization algorithms [131] and the use of residual connections [98], along with ample computational resources, have made it possible to train DNNs of increasingly larger sizes. This trend has given rise to the promising field of **Deep Learning (DL)** [142], which has been responsible for many of the AI advancements in the 21st century, encompassing fields as diverse as natural language processing [28], computer vision [138], and SDM [74]. Despite the relative youth of the field, a large number of different DNN architectures have been proposed, in what is sometimes referred to as the *neural network zoo*. Examples include: feed-forward DNNs [85], Convolutional Neural Networks (CNNs)

[138], Long-Short Term Memory (LSTM) networks [106], Graph Neural Networks (GNNs) [204], Transformers [238], and Neural Logic Machines (NLMs) [53]. In the following subsections, we explain in more detail the CNN and NLM architectures, due to their importance in this dissertation.

II.3.1 Convolutional Neural Networks

A **Convolutional Neural Network (CNN)** [138] is a type of neural network specialized to the structure of images, which is inspired by the animal visual cortex. One of the most significant breakthroughs of CNNs was achieved in the ImageNet LSVRC-2012 competition [49], where the goal was to classify high-resolution images into 1000 classes. The winning model in this competition was a deep CNN named AlexNet [138] (see Figure 8), which managed to outperform the competing approaches by a great margin. From this moment onwards, CNNs became the staple method in computer vision, although they have recently found competition in alternative DNN architectures such as the Vision Transformer [54]. Due to their great success, CNNs have also been applied to non-image data types such as time series [256] and audio [2].

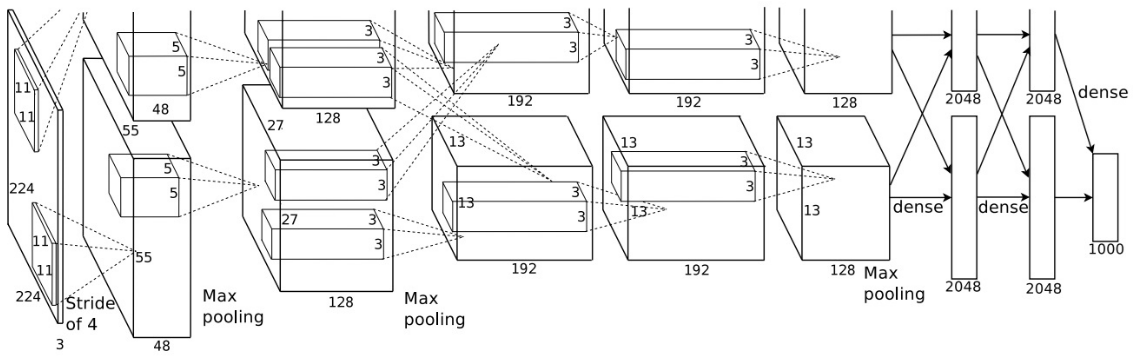


Figure 8: **AlexNet architecture.** This CNN receives an image of size 224x224 pixels with 3 colour channels. Then, it successively applies convolutions and pooling operations to extract high-level features from the image. These features are then used by a feed-forward neural network (labelled as *dense* in the image) to predict the probability that the input image belongs to each of 1000 different classes. Additionally, the computation of the CNN operations is split across two different GPUs, as depicted in the image. Figure extracted from [138].

CNNs are based on the convolution operation. Given an NxM matrix of real numbers, which receives the name of kernel, a convolution traverses the input image obtaining, for each region of adjacent pixels, a new value as the weighted sum of the kernel with the values of the pixels in that region. The *stride* dictates how many pixels the kernel moves between consecutive convolutions, whereas the *padding* determines the values added to the image borders, in case they are needed. Convolutions make it possible to extract local features from images. The first layers in a CNN often learn to extract simple, low-level features such as lines and shapes. Then, these simple features are hierarchically composed in the following CNN layers by the application of more convolution operations, in order to extract high-level features such as entire objects (e.g., the wheel of a car). These high-level features are then utilized to solve the task at hand, e.g., to classify the input image into one of several classes. In addition to convolutions, CNNs sometimes also employ other techniques such as pooling operations, used to aggregate features from adjacent regions in the image.

II.3.2 Neural Logic Machines

A **Neural Logic Machine (NLM)** [53] is a deep neural network capable of learning from FOL data and performing logic reasoning. An NLM receives as input a set of predicates grounded on a set of objects. Then, it sequentially applies first-order rules to obtain a different set of output predicates instantiated on the same objects. Input predicates are represented as binary tensors containing the truth value for each grounding of the predicate on the set of objects. Given some input predicate p, if $p(o_i, o_j, ..., o_k)$ is true (where i, j, k represent object indexes), then its associated tensor will contain a value of 1 at the $(i, j, ..., k)$ position, and 0 otherwise. Output predicates and those inferred internally by the NLM are also represented as tensors, but they contain real values between 0 and 1. The NLM operates with these tensors by using neural modules that approximate boolean operators (*and*, *or*, *not*) and quantifications ($\forall$ and $\exists$), being expressive enough to learn any set of definite Horn clauses in FOL without cyclic references among predicates, and perform inference with them. Therefore, NLMs are more expressive than alternative architectures such as Graph Neural Networks [11], which is why they are used in this dissertation.

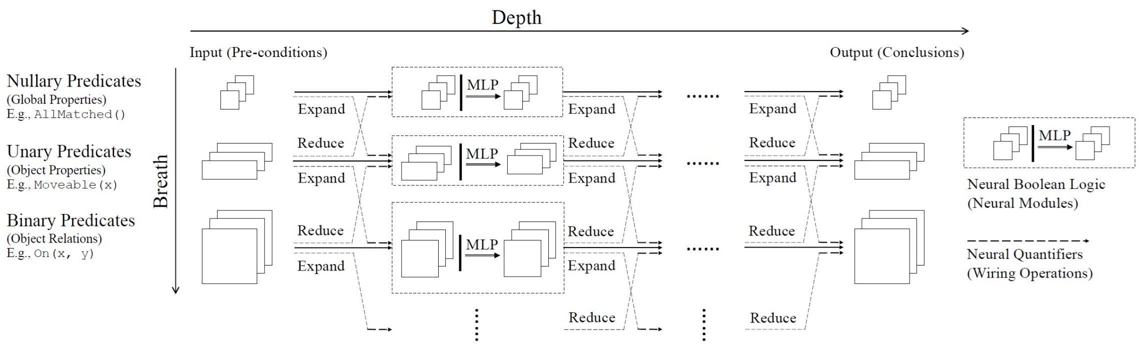


Figure 9: **Architecture of a Neural Logic Machine.** An NLM is characterized by its *breadth* (maximum predicate arity) and *depth* (number of layers). Each NLM layer receives a group of predicates instantiated on a set of objects, and outputs a new group of predicates instantiated on the same objects. The *expand* and *reduce* operations implement the FOL quantifiers $\forall$ and $\exists$, connecting predicates of different arity. The boolean operators *and*, *or* and *not* are implemented by feed-forward neural networks (MLPs). Image extracted from [53].

II.4 Reinforcement Learning

Reinforcement Learning (RL) [224] is a subfield of ML that provides an alternative approach to AP for solving MDPs. Instead of employing an action model to synthesize a solution of the MDP, RL techniques use the data gathered from the environment to learn the optimal policy that maximizes reward. Most RL methods represent their learned knowledge subsymbolically, although a subset of them known as **Relational Reinforcement Learning (RRL)** [227] employ a symbolic knowledge representation. In order to learn the optimal policy, RL algorithms must balance the exploration of the environment, i.e., the process of trying out new actions and observing their outcomes, with the exploitation of the learned knowledge, i.e., selecting the best action found so far (which might not be the optimal one). This is known as the **exploration-exploitation tradeoff**. Additionally, many environments exhibit sparse rewards, meaning that the agent only receives a reward after the execution of a long sequence of actions. This hinders the learning process, as the agent must determine the responsibility of each action in the final outcome (represented by the obtained reward), in what is known as the **credit assignment problem**.

Most RL methods, known as **model-free RL**, do not require a model of the environment (action model) in order to learn the optimal policy. Nonetheless, there exist other techniques, known as **model-based RL**, which leverage the prior knowledge contained in the action model to facilitate the learning process. RL algorithms can also be classified in **value-based** and **policy-based** RL. The first group of methods learn the optimal state values $V^*(s)$ or Q-values $Q^*(s,a)$, which are then used to obtain the optimal policy π^*. For instance, given $Q^*(s,a)$, the optimal action a^* at state s is the one with maximal Q-value: $a^* = \arg\max_{a \in App(s)} Q^*(s,a)$. Conversely, policy-based RL explicitly learns π^* without first needing to estimate $V^*(s)$ or $Q^*(s,a)$. There also exist a third group of methods known as **actor-critic RL** that combine the two previous approaches, learning both a policy π and its corresponding value (either $V^\pi(s)$ or $Q^\pi(s,a)$) at the same time. Lastly, approaches pertaining to **classical RL** employ a tabular representation, i.e., for every possible state or state-action pair they store the corresponding policy information, usually $V^\pi(s)$ or $Q^\pi(s,a)$. On the other hand, modern RL methods, known as **Deep Reinforcement Learning (DRL)**, represent their policy as a DNN, which allows them to generalize their learned knowledge, not needing to store policy information for every single state. The following subsections describe two DRL algorithms that have been employed in this thesis: Deep Q-Learning [158], based on the classical Q-Learning algorithm [244], and Proximal Policy Optimization (PPO) [209]. Chapter III provides additional examples for the different RL categories described in this section.

II.4.1 Q-Learning and Deep Q-Learning

The **Bellman Optimality Equation** [15] plays a central role in RL. It provides a recursive formulation of the optimal Q-value of a state s in terms of the optimal Q-values of other states s' and MDP rewards r:

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}_{s',r}\Big(r + \gamma \max_{a' \in App(s')} Q^*(s',a')\Big) \\ &= \sum_{s',r} P(s',r|s,a)\Big(r + \gamma \max_{a' \in App(s')} Q^*(s',a')\Big) \end{aligned} \tag{II.1}$$

where γ is the MDP discount factor and $\mathbb{E}_{s',r}$ represents the expectation over next states and rewards. This expectation is required since, in stochastic MDPs, the execution of the same action a at the same state s will not always result in the same next state s' and reward r. It can be calculated by using $P(s',r|s,a)$, which represents the probability of the MDP transitioning into s' and outputting reward r when a is executed at s.

Q-Learning [244] is a widely-known classical, value-based RL algorithm that provides a simple method, based on the Bellman Optimality Equation, for learning the Q-values $Q^*(s,a)$ of the optimal policy π^*. Due to its model-free nature, Q-Learning cannot compute the expectation $\mathbb{E}_{s',r}$ over next states and rewards in Equation II.1. The reason is that, after taking an action at the current state, the environment transitions into another state, so the agent cannot simply *go back* to the previous state to execute the same action again, looking for a different outcome. Q-Learning solves this problem by slowly updating (according to the learning rate $\alpha \in [0,1]$) the current Q-value estimates $Q(s,a)$ with the rewards r and next states s' observed from the environment, which are sampled from $P(s',r|s,a)$, until $Q(s,a)$ eventually converges to $Q^*(s,a)$. The exact update rule is shown below:

$$Q^{new}(s,a) \leftarrow (1-\alpha)\cdot Q^{old}(s,a) + \alpha \cdot \Big(r + \gamma \max_{a' \in App(s')} Q^{old}(s',a')\Big) \tag{II.2}$$

One limitation of Q-Learning is that it needs to store $Q(s,a)$ for every (s,a) pair, which results infeasible for MDPs with large state spaces as in most real-world applications. **Deep Q-Learning** [158] addresses this issue by using a DNN to predict the Q-values (see Figure 10), which allows it to generalize to unseen states and removes the need to store information for every (s,a) pair. The loss function L employed by this algorithm can be regarded as an adaptation of the Q-Learning update rule (see Equation II.2) and is detailed below:

$$L = \Big(Q(s,a) - Q^{target}(s,a)\Big)^2 = \Big(Q(s,a) - \big(r + \gamma \max_{a' \in App(s')} Q(s',a')\big)\Big)^2 \tag{II.3}$$

where Q(s,a) represents the Q-value predicted by the DNN for the (s,a) pair and $Q^{target}(s,a)$ is the value the DNN is trained to predict, which receives the name of TD-target. When calculating the gradient of this loss L, the TD-target is considered a constant term so that gradients flow though $Q(s,a)$ but not $Q(s',a')$.

II.4.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [209] is an actor-critic DRL algorithm that has been successfully applied to solve a wide variety of tasks. PPO is composed of two main parts, an actor and a critic, which can be implemented either as separate DNNs or as a single one. The goal of the actor is to learn the optimal policy π^*, i.e., select actions that maximize the expected return. On the other hand, the critic is tasked with evaluating the quality of the actions taken by the actor, in order to guide the learning of π^*. Given a state s and action a', the advantage $A(s,a')$ measures the relative quality of a' when compared to the rest of actions $a \in App(s)$, i.e., it describes how better (or worse) a' is when compared to the *average* action at s. The following formula provides a simple way of calculating the advantage:

$$A(s,a') = Q(s,a') - V(s) \tag{II.4}$$

where $Q(s,a')$ is the Q-value representing the expected return obtained by following the current policy π after executing a' at s, and $V(s)$ is the state value measuring the expected returned obtained by following π from s. $Q(s,a')$ can be calculated as

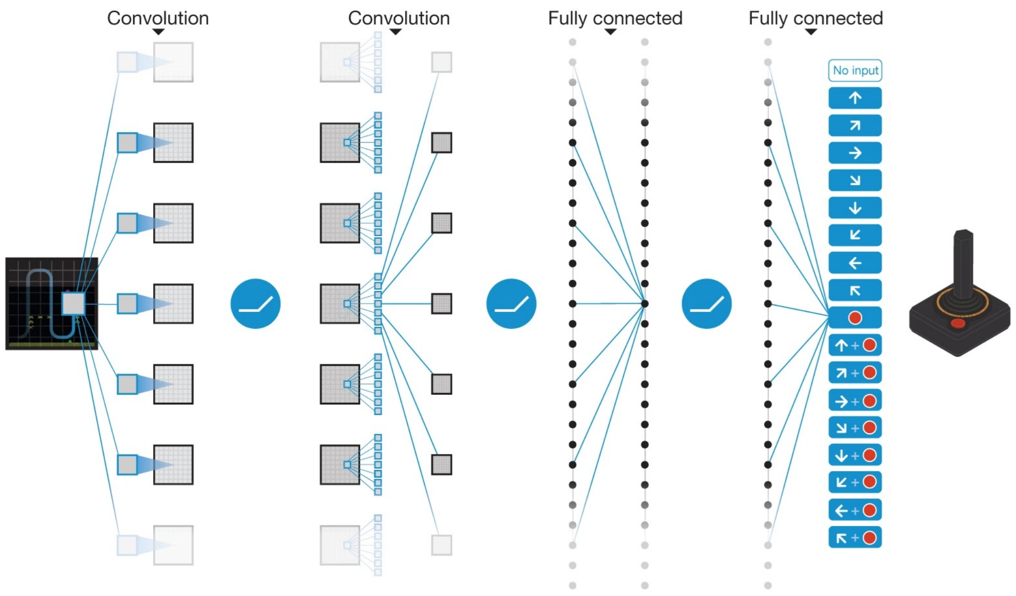


Figure 10: **Deep Q-Learning for Atari.** The figure shows a CNN trained with the Deep Q-Learning algorithm for playing an Atari game. The CNN receives as input the current MDP state s, represented as the concatenation of the last four game frames. Then, it predicts in parallel the Q-value $Q(s, a)$ associated with each action $a \in App(s)$, i.e., each valid combination of controller inputs. The feedback obtained from the environment (in the form of rewards and new states) when playing the game is used to compute the loss in Equation II.3. By minimizing it, the agent learns to improve its gameplay, achieving superhuman performance in some games. Image extracted from [159].

the (possibly discounted) sum of rewards obtained during an episode, whereas $V(s)$ is learned by the DNN of the critic.

Policy gradient methods are a group of RL techniques that estimate the gradient of the expected return R with respect to the policy parameters θ (e.g., the weights of a DNN). By *ascending* this gradient, the agent learns to favour actions that resulted in high return in the past, thus approaching the optimal policy. A common formula for estimating the policy gradient is shown below:

$$\nabla_\theta R = \mathbb{E}_{s,a\sim\tau}\Big[\nabla_\theta \log \pi(a|s) A(s,a)\Big] \tag{II.5}$$

where τ represents a trajectory, i.e., a sequence of states, actions and rewards, and $\pi(a|s)$ represents the probability of taking action a at state s according to the (stochastic) policy π.

Trust Region RL methods improve policy gradient techniques by preventing large policy updates during training. This helps reduce training instability and improves data efficiency, as we can now safely train for several epochs on the same data, i.e., (s, a, r) samples. PPO offers a simple, yet effective, implementation of this approach with its clipped surrogate objective. Let $\rho(s, a)$ denote the ratio $\pi^{new}(a|s)/\pi^{old}(a|s)$ between the probability of executing a at s output by the current policy π^{new} and the old policy π^{old} (i.e., the one that was used to collect the training data). Then, PPO *maximizes* the following objective L^{CLIP} for training the actor:

$$L^{CLIP} = \mathbb{E}_{s,a\sim\tau}\Big[\min\Big(\rho(s,a)A(s,a),\ \text{clip}\big(\rho(s,a), 1-\epsilon, 1+\epsilon\big)A(s,a)\Big)\Big] \tag{II.6}$$

where clip(x, a, b) is a function that clips x to the interval $[a, b]$, and ϵ is a hyperparameter that controls the maximum allowed magnitude of policy updates. Figure 11 shows the relationship between L^{CLIP}, the policy probability ratio ρ and the advantage A.

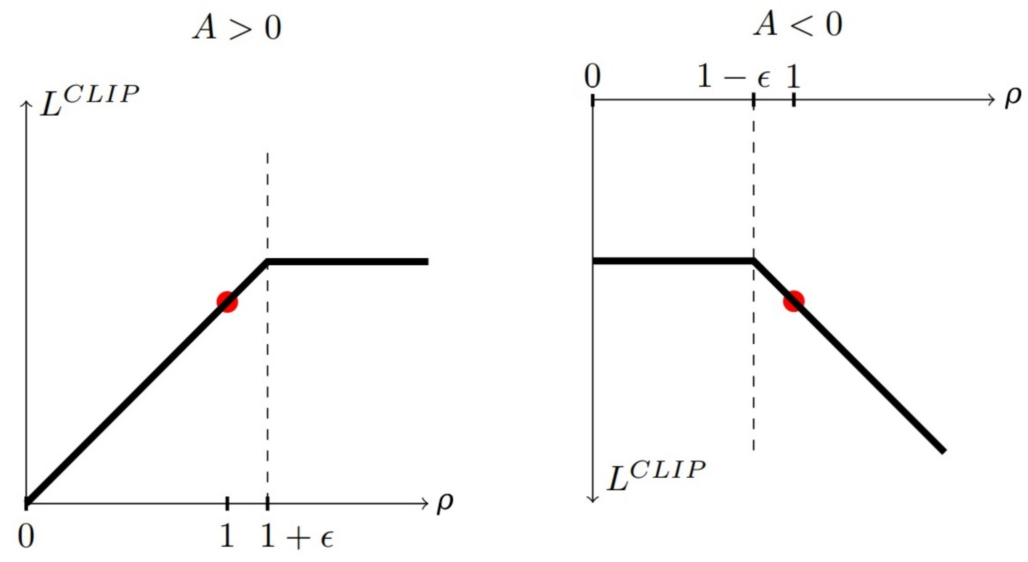


Figure 11: **PPO loss diagram.** The figure shows how L^{CLIP} (see Equation II.6) changes with respect to ρ when the advantage A is positive (left) and when it is negative (right). When the action executed at a particular state is good, i.e., $A > 0$, L^{CLIP} encourages the increase in action probability (ρ) up to a certain threshold $(1 + \epsilon)$, whereas it discourages any decrease in ρ. When the executed action is bad, i.e., $A < 0$, L^{CLIP} encourages the decrease in action probability (ρ) up to a certain threshold $(1 - \epsilon)$, whereas it discourages any increase in ρ. Therefore, L^{CLIP} prevents large policy updates which could make training unstable. Image adapted from [209].

Finally, PPO maximizes the following objective $L^{CLIP+VF+S}$ for simultaneously training the actor and the critic:

$$L^{CLIP+VF+S} = \mathbb{E}_{s,a\sim\tau}\left[L^{CLIP} - c_1 L^{VF} + c_2 S[\pi](s)\right] \tag{II.7}$$

where L^{VF} represents the critic loss, measuring the errors made by the critic when predicting the value function $V^{\pi}(s)$, S is an entropy bonus that encourages the actor to learn highly-stochastic policies π, and c_1, c_2 are coefficients controlling the weight of the different terms in the $L^{CLIP+VF+S}$ loss.

Part II

Related Work

Chapter III

Related Work

This chapter presents a **broad overview of existing work in the field of SDM**, thus aligning with the first subgoal (**G1**) of this dissertation, as described in Section I.1. It comprises methods with symbolic, subsymbolic and hybrid (e.g., neuro-symbolic) knowledge representations, for both solving MDPs and learning their structure (e.g., the action model). Due to its broad scope, this chapter does not cover SDPs with continuous state and/or action spaces. Existing techniques are classified into a novel taxonomy and compared with each other across several dimensions. Additionally, Section III.3 theorizes what properties an **ideal method for SDM** should exhibit. Based on these properties, the advantages and disadvantages of the different approaches for solving MDPs are analyzed. As a result, it is argued that an ideal method for SDM should integrate the AP and RL paradigms for solving an MDP, while combining the symbolic and subsymbolic knowledge representations. Since neuro-symbolic AI is the current approach most closely resembling this ideal integration, it is concluded that it poses a very promising avenue of research, thus justifying the relevance and significance of this doctoral dissertation.

This chapter is organized as follows. Sections III.1 and III.2 describe the taxonomy of methods for SDM, the first focusing on methods for solving MDPs whereas the latter focuses on methods for learning the structure of MDPs. Next, Section III.3 discusses the characteristics of an ideal method for SDM, as previously explained. Lastly, Section III.4 proposes several directions to further advance the field of SDM via the integration of symbolic and subsymbolic AI.

III.1 Methods to solve MDPs

In this section we present an overview of the main methods to solve MDPs (see Figure 12). Historically, there have been two competing paradigms. AP [84] proposes a synthesis-based approach, in which prior information about the MDP is used to carry out a search and reasoning process in order to find its solution. RL [224], on the other hand, presents a learning-based approach inspired by ML, where the agent does not synthesize an MDP solution but rather learns it from data. In this chapter, we also discuss a novel family of methods that *learn to plan* [160], thus combining the AP and RL approaches. In the same manner as AP, these methods carry out a planning process to find a solution of the MDP. However, unlike them, they learn how to actually perform the planning computations automatically from data, similarly to how RL techniques learn the optimal policy also from data. Table

Table 1: **Comparison among methods to solve MDPs**. The first column groups the methods in Section III.1 according to their knowledge representation: symbolic, subsymbolic or a combination (*hybrid*). **Solution Representation (Sol.)**: complete policy (C), partial policy (P) or plan. **Action Model (Model)**: whether the method uses an action model (✓), learns it from data (◯) or does not need it (×). **Stochasticity (Stoch.)**: can the method tackle stochastic MDPs? **Heuristic (Heur.)**: whether the method leverages prior control knowledge in the form of a heuristic or not. **Generalizability (Gen.)**: can the policy obtained by the method generalize to novel states and/or goals? In principle, methods that utilize DNNs learn a policy applicable to any state $s \in S$ (i.e., a complete policy) and can generalize to new states and/or goals not seen during training, although the generalization capability will depend on the particular algorithm. Among those which can generalize, methods with a symbolic or hybrid knowledge representation often do it better.

	Methods	**Sol.**	**Model**	**Stoch.**	**Heur.**	**Gen.**
Subsymbolic	PI, VI [224]	C	✓	✓	×	×
	LAO* [94], LRTDP [26]	P	✓	✓	✓	×
	MCTS [29]	P	✓	✓	×	×
	A* [95]	plan	✓	×	✓	×
	DFS [229], BFS [30]	plan	✓	×	×	×
	Q-Learning [244]	C	×	✓	×	×
	Deep Q-Learning [158], REINFORCE [246], A2C [157]	C	×	✓	×	✓
	Dyna [223]	C	◯	✓	×	×
	AlphaZero [217]	C	✓	✓	×	✓
	MuZero [207]	C	◯	✓	×	✓
	VIN [228]	C	×	✓	×	✓
	UPN [221]	C	×	×	×	✓
	MCTSnet [90]	C	✓	×	×	✓
	IBP [187], TreeQN [62], DRC [89]	C	◯	✓	×	✓
Symbolic	SPUDD [107]	C	✓	✓	×	×
	Symbolic LAO* [64], SSiPP [234]	P	✓	✓	✓	×
	FF-Replan [251]	plan	✓	✓	✓	×
	FF [108], FD [100]	plan	✓	×	✓	×
	Relational Q-Learning [57], Deep Symbolic Policy [140]	C	×	✓	×	✓
Hybrid	SDRL [149]	C	✓	✓	×	✓
	SORL [120]	C	◯	✓	×	✓
	DRL with Relational Inductive Biases [255]	C	×	✓	×	✓
	ASNet [233]	C	✓	✓	✓	✓

1 shows an overview of the methods discussed in this section, classified according to

their properties.

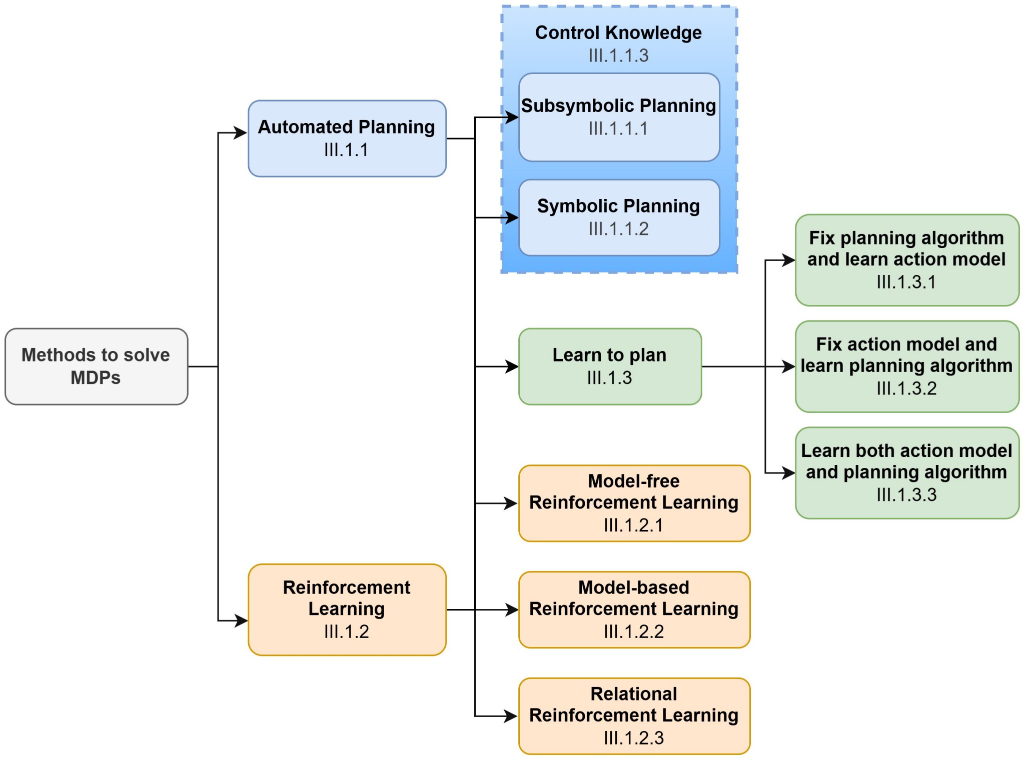

Control Knowledge
III.1.1.3
Subsymbolic Planning
III.1.1.1
Symbolic Planning
III.1.1.2
Automated Planning
III.1.1
Methods to solve MDPs
Learn to plan
III.1.3
Fix planning algorithm and learn action model
III.1.3.1
Fix action model and learn planning algorithm
III.1.3.2
Learn both action model and planning algorithm
III.1.3.3
Model-free Reinforcement Learning
III.1.2.1
Reinforcement Learning
III.1.2
Model-based Reinforcement Learning
III.1.2.2
Relational Reinforcement Learning
III.1.2.3

Figure 12: **Proposed taxonomy of methods to solve MDPs**, with section numbers indicated. Model-free and model-based RL methods with a symbolic or hybrid knowledge representation are placed in the *Relational Reinforcement Learning* category.

III.1.1 Automated Planning

AP [84] comprises methods that harness the information about the environment dynamics, encoded in the planning domain, to synthesize a solution of the MDP. Some AP techniques, known as Probabilistic Planning (PP) [167], solve the general class of stochastic MDPs and are used to find a policy that minimizes the expected cost needed to reach G from s_i. Conversely, Classical Planning (CP) [84] methods tackle the specific class of deterministic MDPs with a single initial state s_i and their goal is to find a minimal-cost plan from s_i to G. We group AP methods according to the type of knowledge representation employed in Subsymbolic/Non-Symbolic Planning (NSP) and Symbolic Planning (SP). Additionally, in Section III.1.1.3, we explain how both NSP and SP techniques can exploit available knowledge to solve the MDP more efficiently.

III.1.1.1 Subsymbolic Planning

We use the name NSP to refer to those AP methods which do not require a symbolic (e.g., FOL-based) description of the MDP. NSP methods only need as action model a transition function that maps a state-action (s, a) pair into its corresponding next state s' and do not care about how such a function is implemented. For this reason, these techniques are very versatile and can be applied to a wide range of situations where the environment dynamics are known but are not given in a formal, logic-based description.

Subsymbolic PP algorithms can be grouped according to whether they obtain complete or partial policies. Two foundational algorithms that find complete policies are Policy Iteration (PI) and Value Iteration (VI) [224]. PI is a dynamic programming algorithm that involves two steps: policy evaluation, where the value $V^{\pi}(s)$ of the current policy π is computed for every state $s \in S$, and policy improvement, where π is updated by selecting, for each s, the action a that optimizes $Q^{\pi}(s,a)$. These two steps are repeated until π converges to the optimal policy π^*. VI is another dynamic programming algorithm. It directly estimates the value $V^*(s)$ of π^* by iteratively updating the current estimate of $V^*(s)$ using the Bellman Optimality Equation (see Equation II.1 in previous chapter). Once $V^*(s)$ is computed, π^* can be obtained by simply selecting, for each state s, the action a with best $Q^*(s,a)$. Some PP methods that obtain partial policies leverage a heuristic (see Section III.1.1.3) to direct the search. LAO* [94] gradually expands states reachable from s_i, initializes their value with a heuristic and backs up this information by running VI or PI on those states whose value could have changed. LRTDP [26] repeatedly samples trajectories from s_i using the current policy π, initializes the value of new states with a heuristic and updates π and the values of states in the trajectories using the Bellman Optimality Equation. Unlike these methods, MCTS [29] does not require a heuristic. This algorithm gradually builds a search tree from s_i, estimating state values from the results of sampled trajectories. MCTS was originally developed for deterministic environments, but it has been extended to manage stochasticity. Finally, subsymbolic CP methods can also be grouped according to the use of heuristics. A widely-known CP algorithm that leverages heuristics is A* [95], which can in fact be viewed as a specialization of LAO* for deterministic MDPs. Other CP methods like depth-first search (DFS) [229] and breadth-first search (BFS) [30] explore S without heuristics. DFS does so by exploring as far as possible along the current trajectory before moving to the next, whereas BFS expands all states with depth n before exploring those with depth $n+1$.

III.1.1.2 Symbolic Planning

We use the name SP to refer to those methods which, unlike NSP, require a symbolic description of the MDP in a formal language, such as (P)PDDL or RDDL. Symbolic MDP descriptions provide several advantages over subsymbolic ones. Firstly, languages such as (P)PDDL and RDDL make it possible to specify a wide variety of MDPs with ease. Secondly, all the planning tasks within a domain can reuse the same PDDL domain specification, thereby reducing design effort. Thirdly, symbolic MDP descriptions can be exploited by SP algorithms to speed up planning.

Many symbolic PP algorithms leverage MDP descriptions by using *algebraic decision diagrams* (ADDs). ADDs represent MDP elements (e.g., values, transitions and costs) as functions of boolean variables (e.g., FOL atoms), using a special type of decision tree. They encode the inner MDP structure and group similar states together (like those that share the same value), which allows for more efficient implementations of the NSP algorithms seen in the previous section. For example, SPUDD [107] integrates VI with ADDs, whereas Symbolic LAO* [64] does so for LAO*. Other PP techniques use symbolic descriptions to compute heuristics. FF-Replan [251] solves the all-outcomes determinization (see Section III.1.1.3) of the

MDP with the classical planner FastForward (FF) [108]. Then, it executes the plan on the original, stochastic MDP until it fails, at which point FF is called again. SSiPP [234] works by successively decomposing the MDP into subproblems and solving them. For each subproblem, it computes a policy that can be executed for at least t time steps before a new policy is needed. Finally, much effort has been devoted to developing planners for the symbolic CP case, such as FF and FastDownward (FD) [100], which mainly differ in the search algorithms and heuristics employed. New classical and probabilistic planners are periodically compared with each other in the International Planning Competitions [236], which evaluate planners on a set of benchmark (P)PDDL/RDDL domains.

III.1.1.3 Control knowledge

The term *control knowledge* refers to all the information used to guide the search and reasoning process of AP algorithms, in order to solve planning tasks as efficiently as possible. This knowledge often comes in the form of planning heuristics, used to estimate the expected cost from a state s to the goal G.

Most SP techniques exploit symbolic MDP descriptions to compute powerful domain-independent heuristics. A popular technique for doing so is called *delete-relaxation* [128], in which the heuristic estimate is computed in a simplified version of the planning task where the *delete* effects of actions are ignored. There exist a wide variety of symbolic, domain-independent heuristics for CP, both admissible and inadmissible. Notable admissible heuristics include h^{LMcut} [102], $h^{\max}$ [25] and h^{+} [19], whereas h^{FF} [108], h^{add} [25] and h^{GC} [68] are prominent examples of inadmissible heuristics. Probabilistic planners often leverage heuristics by obtaining a deterministic version (*determinization*) of the MDP being solved and computing a CP heuristic on it, which is then used as a heuristic for the original, stochastic MDP. The most widely employed one is known as the *all-outcome determinization* [251]. Given a stochastic MDP M, it obtains a deterministic version M' of M which contains a separate deterministic action for each probabilistic effect of every action in M. Alternatively, some methods utilize domain-specific heuristics to solve problems from a particular planning domain in a very efficient manner. For instance, the L1 norm or *Manhattan distance* provides a simple heuristic which has been successfully applied to many pathfinding problems [33].

In some situations, domain-independent heuristics may be unfeasible to compute (e.g., when a symbolic MDP description is not provided) and we may also lack the prior knowledge needed to derive a domain-specific heuristic. In order to address these cases, heuristics can be learned from example trajectories thanks to the use of ML. Example methods include linear regression [250], regression trees and support vector machines [235]. Neuro-symbolic approaches have also been applied to heuristic learning. [212] utilizes Graph Neural Networks [12] to predict the heuristic value from the delete-relaxation representation of the problem, whereas [79] leverages domain-independent heuristics to efficiently learn domain-specific heuristics with a Neural Logic Machine [53] trained using DRL. A more comprehensive analysis of methods for learning control knowledge can be found in [117].

III.1.2 Reinforcement Learning

RL [224] is a subfield of ML that provides an alternative approach to AP for solving MDPs. Instead of employing an action model to synthesize a solution of the MDP, RL techniques use the data gathered from the environment to learn the optimal policy that maximizes the expected return. We group RL techniques in model-free methods, i.e., those which do not require a prior action model in order to learn the optimal policy, and model-based methods, i.e., those that leverage an existing action model to facilitate the learning process. The vast majority of RL methods, both model-free and model-based, represent their knowledge in a subsymbolic manner (e.g., as the weights of a DNN). Therefore, we distinguish a third group of methods, regardless of whether they use an action model or not, which utilize a symbolic knowledge representation, known as Relational RL (RRL).

III.1.2.1 Model-free RL

Model-free RL provides methods to learn the optimal policy when the model of the world, i.e., the action model, is unknown. These methods can be further grouped in (model-free) value-based and policy-based RL. Value-based techniques learn the optimal state values $V^*(s)$ or Q-values $Q^*(s,a)$, which are then used to obtain the optimal policy π^*. A classical algorithm for value-based RL is Q-Learning [244], which can be seen as an adaptation of the Value Iteration algorithm to the model-free setting where the environment dynamics are unknown. It leverages the Bellman Optimality Equation to estimate the $Q^*(s,a)$ values from the rewards and state transitions observed when executing actions in the environment. These Q-values are stored in a table for every possible combination of states and actions, which results infeasible for MDPs with large state spaces. Deep Q-Learning [158] solves this problem by using a DNN to approximate the Q-values. Since this DNN is capable of generalizing to new states, it does not need to memorize the Q(s, a) value for every (s, a) pair, thus making it possible to apply this algorithm to real-world problems with large state spaces. A more exhaustive description of both Q-Learning and Deep Q-Learning can be found in Section II.4.1.

In contrast to value-based methods, policy-based RL explicitly learns π^* without first needing to estimate $V^*(s)$ or $Q^*(s,a)$. One of the most well-known algorithms in this category is REINFORCE [246]. REINFORCE is a policy gradient algorithm that utilizes a DNN to approximate the policy, i.e., given an input state s it returns a probability distribution $\pi(a|s)$ over the actions $a \in App(s)$. This DNN is then trained using gradient-based methods to maximize the probability of selecting actions with a large $Q^\pi(s,a)$ associated, where $Q^\pi(s,a)$ is often estimated as the (possibly discounted) sum of rewards obtained by first executing a at s and then following π until the end of the episode. One main issue of REINFORCE is that $Q^\pi(s,a)$ is not always a good measure of action optimality, since this measure does not only depend on the action a but, also, on the state s it is executed. Advantage Actor Critic (A2C) [157] addresses this issue by substituting $Q^\pi(s,a)$ for the advantage $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$, which measures how good a is when compared to the average action at s. As an actor-critic method, A2C trains a separate DNN for the actor and the critic. The critic learns to predict the value $V^\pi(s)$ of a given state, which is then used to calculate $A^\pi(s,a)$. The actor learns π^* by using the same method as REINFORCE. However, it utilizes the advantage $A^\pi(s,a)$ obtained from the critic to measure action optimality, instead of $Q^{\pi}(s, a)$ (see Equation II.5). Thus, A2C entails a hybrid approach that integrates both value-based and policy-based RL. In addition to A2C, several improvements over REINFORCE have been proposed over the years. One notable example is Proximal Policy Optimization (PPO) [209], described in Section II.4.2. Finally, a deeper view into classical RL and DRL methods can be found in [224, 143, 211].

III.1.2.2 Model-based RL

Model-based RL tries to combine the fields of RL and AP. It enhances standard, model-free RL algorithms with a model of the world, which can be employed in two main ways. The first alternative is to use the action model as a simulator to obtain data for training the policy, thus minimizing the required amount of interaction with the real world. For this reason, model-based RL techniques are more sample-efficient, i.e., need less data to learn the optimal policy, than model-free methods [122]. A second alternative is to integrate a deliberative process into the decision-making cycle of RL. Instead of simply selecting the best action according to the policy, we can use the action model to carry out a planning process, guided by the learned policy/value function, in order to select the next action to execute. This approach blurs the line between model-based RL and AP with a learned heuristic. Finally, we can differentiate between model-based RL methods which require the action model to be given a priori and those which do not. This second category of methods use the data collected from the environment to learn the action model in addition to a policy, employing some of the techniques commented in Section III.2.1.

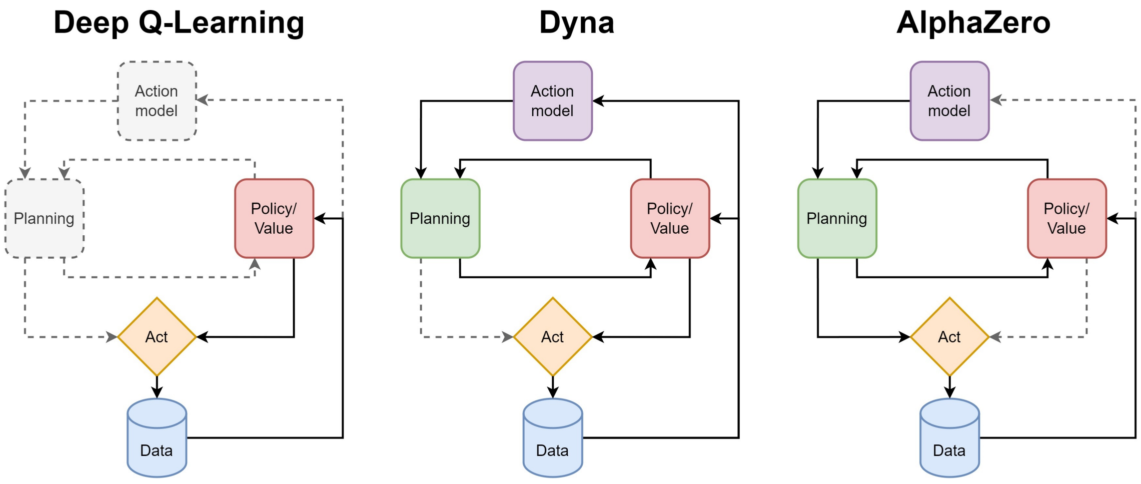


Figure 13: **Model-free vs model-based RL.** Adapted from [160] with permission by the authors. The figure compares the architectures of a model-free RL algorithm, Deep Q-Learning, and two model-based RL methods, Dyna and AlphaZero. Thick lines and colored elements are used by the algorithm, whereas grayed out elements and dotted lines are not, and are displayed just for comparative purposes. Deep Q-Learning learns a (state) value function from data, which is then used to select the action to execute with no planning whatsoever. On the other hand, Dyna utilizes data to learn both a value function and action model. This model is used to obtain extra data to train the value function, which ultimately decides the action to execute. AlphaZero utilizes data to train both a value function and a policy. These are used to guide a planning process over an action model provided in advance, in order to decide the next action to execute.

We will use Dyna [223] and AlphaZero [217] as illustrative examples of the different existing model-based RL algorithms. One of the oldest examples of model-based

RL can be found in Dyna. This method learns a subsymbolic, black-box model of the world. Then, it trains the Q-Learning algorithm on both experience obtained from the real world and data sampled using the learned action model. Dyna performs reactive execution, i.e., the action to execute is selected according to the value function trained with Q-Learning, with no planning involved whatsoever. AlphaZero is a novel model-based RL method which has been successfully applied to play the games of chess, shogi and Go at superhuman level. Unlike Dyna, it requires a prior model of the world, although a newer version of this method known as MuZero [207] overcomes this limitation. AlphaZero trains both a value function and a policy, implemented as a single CNN (see Section II.3.1), which together guide the planning process performed by the MCTS algorithm. This planning process outputs a probability distribution from which the action to execute is sampled. The value function is trained to predict the game winner from the current state whereas the policy is trained to match the probabilities obtained by MCTS. Thus, in AlphaZero there exists a clear synergy between RL and AP: RL is used as a heuristic to guide the planning process, which in return makes it possible to obtain data to train the RL policy, acting as a *policy improvement operator*. A comparison between Deep Q-Learning, Dyna and AlphaZero is shown in Figure 13. Finally, a more comprehensive review of model-based RL is provided in [160, 191].

III.1.2.3 Relational RL

Relational Reinforcement Learning (RRL) [227] can be considered the intersection of RL and Relational Learning [83]. It is comprised of methods that combine RL with the symbolic representations employed in SP. These symbolic representations are well suited for MDPs that can be naturally described in terms of objects and their interactions, known as object-oriented MDPs [52]. RRL methods often leverage lifted representations, i.e., symbolic representations with variables, to abstract from concrete objects and, thus, naturally generalize to tasks with varying number of objects. In addition, the knowledge learned by RRL techniques is more amenable to interpretation than the one typically learned in DRL. One of the best known RRL algorithms is relational Q-Learning [57]. This method utilizes a symbolic knowledge representation for the goal-conditioned Q-values $Q^*(s, a, g)$, which generalize Q-values $Q^*(s, a)$ to different goals g. This function is learned with a relational regression tree [137] which, given the current state s, goal g to achieve and action a to execute, predicts the associated Q-value $Q^*(s, a, g)$ (see Figure 14).

In recent years, there has been a renewed interest in RRL. [149] proposes a model-based, symbolic DRL framework which integrates SP with hierarchical DRL. It leverages a symbolic action model containing a high-level description of the environment dynamics to compute plans composed of subtasks to achieve. Then, DRL is used to learn a low-level policy for each subtask. In [120], the previous neuro-symbolic framework is augmented with the ability to learn the symbolic action model from trajectories. To do so, a function that maps subsymbolic, low-level states to symbolic, high-level states is required. [140] utilizes a recurrent neural network to generate symbolic policies represented as concise mathematical expressions. Generated policies are then evaluated on the environment, and the obtained rewards are used to train the policy generator with RL itself. On the other hand, [255] proposes a Deep RRL algorithm that departs from the classical RRL definition, since it does not actually use a symbolic representation. Instead, the scene objects are detected

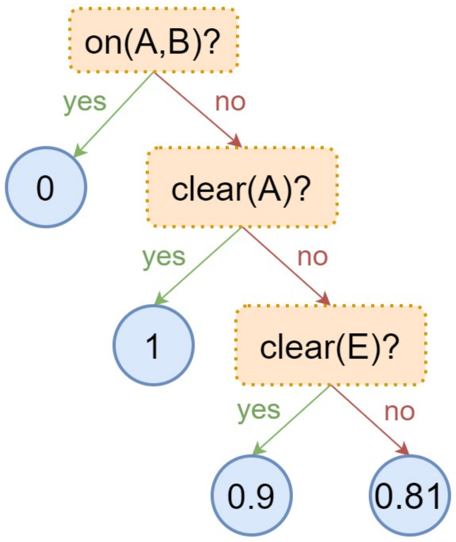


Figure 14: **Regression tree learned with relational Q-Learning** (adapted from [57]). The example corresponds to the *blocksworld* domain (see Figure 5). The tree encodes the $Q^*(s, a, g)$ value for the goal $g = on(A, B)$ and action $a = stack(D, E)$. Inner nodes (orange) check the truth value of a grounded predicate of the state s, whereas leaf nodes (blue) return the corresponding Q-value.

with a CNN and a deep attention-based model [238] is used to reason about their interactions. The authors show their approach improves the efficiency, generalization and interpretability of conventional DRL. More information about RRL techniques can be found in [227, 254, 3].

III.1.3 Learn to plan

As we have previously explained, many MDP-solving techniques use an action model, which can be either known or learned from data. This model of the world can be used in several ways. The first option is to plan over it. We can use a symbolic planner, e.g., FF, or an NSP procedure, e.g., MCTS, depending on the type of knowledge representation used by the model, in order to simulate different courses of action and find the best one. The second option is to leverage the action model to obtain data for training model-based RL methods. In addition to these two options, there exists a third alternative which has not been discussed yet: to *learn to plan* [160]. This idea combines the RL and AP approaches and it is inspired by the novel area of algorithmic reasoning [31], which studies how to teach DNNs to compute algorithms. In the case of SDM, instead of considering the planning procedure as an external process, we can integrate it into the computational graph of our DNN, i.e., the graph containing the sequence of operations performed by the model to transform the inputs into outputs. There exist three main ways to embed the planning procedure into a DNN. Firstly, we can learn an action model that is compatible with a planning algorithm chosen a priori. Secondly, given an action model, we can learn how to perform the actual planning computations on it. Finally, we can jointly learn the action model and planning algorithm at the same time. These different approaches are compared in Figure 15.

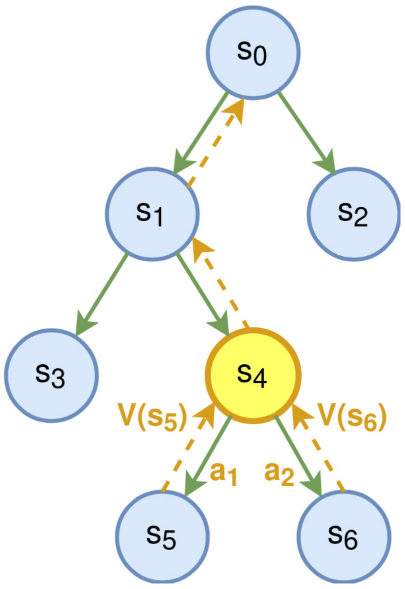


Figure 15: **Learning to plan approaches.** Green arrows represent the action model which determines the transitions between states (we assume determinism for simplicity purposes). Yellow items represent the different aspects of a planning algorithm, which mainly comprises 1) expanding some state (s_4 in the image example), 2) executing one or more actions (a_1, a_2 in the example), 3) evaluating the resulting states (s_5, s_6) and 4) propagating this new information to the other states (yellow dashed lines). Some methods only learn the action model (green items) for a given planning algorithm [228, 221], other techniques only learn the planning computations (yellow items) [90, 187, 233] and, finally, some learn both [62, 89].

III.1.3.1 Fix the planning algorithm and learn the action model

Some methods choose a planning algorithm a priori and embed it into the computational graph of the decision-making model. If the planning procedure is differentiable, meaning the gradient of the learned policy/value function can be backpropagated through the planning operations, it is possible to learn an action model compatible with it by using gradient-based optimization techniques. The action model is trained to output the optimal policy or value function as a result of the iterative computations performed by the chosen planning algorithm. Therefore, the action model learns a representation of the dynamics tailored to the task at hand and planning algorithm employed, as opposed to most action models which represent the environment dynamics in a general, task-independent way. This special type of action models belong to the family of value-equivalent action models (see Section III.2.1.2).

Value Iteration Networks (VINs) [228] are a good example of this. The authors of this work propose a differentiable implementation of the classical Value Iteration algorithm using recurrent CNNs. This planning procedure is then embedded into a DNN architecture which can be trained end-to-end to predict the optimal policy by using standard RL and Supervised Learning, i.e., ML methods which are trained on labeled samples corresponding to *(input, output)* pairs. The obtained model generalizes better than reactive policies, i.e., those that do not perform planning, when applied to new problems of the same domain. Universal Planning Networks (UPNs) [221] follow a similar approach. This work proposes a DNN architecture which learns a policy for a continuous task. The architecture integrates a differentiable planning module, which performs planning by gradient descent, and is trained end-to-end with Supervised Learning to predict the optimal policy. The resulting action model exhibits a state representation suitable for gradient descent planning, which can be adapted to other tasks.

These methods are similar to those discussed in Section III.2.1.2 like MuZero, since they also learn a value-equivalent action model that is suitable for planning. The difference between both types of methods is how the planning process is implemented. In MuZero, planning is performed explicitly, outside the computational graph. It learns a state representation useful for predicting the rewards, value function and policy for any given state. In VINs and UPNs, however, planning is performed implicitly, as a differentiable process embedded into the computational graph to predict the optimal policy. Thus, they may learn a slightly different state representation to that employed by MuZero and other models where planning is performed as a separate process.

III.1.3.2 Fix the action model and learn the planning algorithm

Other methods follow the opposite approach. Given an action model, which can be either known a priori or learned as a first step, they learn how to plan over it in order to solve the corresponding SDM task. The planning algorithm obtained will be able to harness the information contained in the action model to predict the optimal policy or value function. The existing methods in the literature provide different amounts of freedom to the planning algorithm, by controlling which parts of it are fixed a priori and which ones must be learned.

[90] proposes MCTSnet, a DNN architecture that embeds the MCTS algorithm into its computational graph. This model learns how to perform the different MCTS operations (selection, expansion, simulation and backpropagation) in order to play the *Sokoban* game. Other works give even more freedom to the planning process. [187] proposes Imagination-Based Planner (IBP), a DNN architecture capable of constructing, evaluating and executing plans. It learns when to plan, which states to expand and when to stop planning. Both MCTSnet and IBP utilize a subsymbolic representation for their learned knowledge. Instead, [233] proposes Action Schema Networks (ASNets), a neuro-symbolic model for learning to plan. ASNets correspond to DNNs specialized to the structure of planning problems. They are composed of alternating action and proposition layers, with the specific network topology given by the associated planning domain and problem to solve, and a final layer which outputs the policy. Policies learned by ASNets are shown to generalize to different problems of the same domain.

III.1.3.3 Learn both the action model and planning algorithm

Lastly, some methods combine the two previous approaches. They embed a differentiable action model and differentiable planning procedure into the same computational graph, and then jointly optimize both parts to solve a particular SDM task. Although this idea represents the most end-to-end approach for learning to plan, the resulting model is hard to optimize, due to its great complexity and the interdependence between the action model and planning algorithm, as the quality of one depends on the quality of the other and vice versa.

In [62], RL is used to jointly learn an action model and how to plan over it. The proposed method, called TreeQN, employs the learned model to try all possible action sequences up to a predefined depth, learning to predict the state values and rewards along the simulated trajectories. These values are then backed up the search tree to estimate the Q-values at the current state. [89] proposes the Deep Repeated ConvLSTM (DRC) model, a powerful DNN architecture capable of learning to plan even though it incorporates no inductive bias for that purpose beyond its iterative nature. The model is composed of stacked ConvLSTM [216] blocks which are repeatedly unrolled to predict the policy and value function. The authors show that DRC exhibits characteristics of planning, such as an increase in performance when given additional thinking time.

III.2 Methods to learn the structure of MDPs

Every MDP has a different underlying structure. Its most important features are encoded in the action model, which describes the environment dynamics and how the agent can affect them, being required by many of the MDP-solving methods seen in the previous section, such as AP and model-based RL. Additionally, there exist specific aspects of the MDP structure, e.g., landmarks [109] and state invariants [71], which if known can facilitate its resolution and provide insight into the properties of the MDP. Moreover, some learning methods [212, 10, 110] require training data in the form of problem instances and example trajectories, which often need to be provided by domain experts. In this section, we discuss the main methods (see Figure 16) for learning these different aspects of the MDP structure.

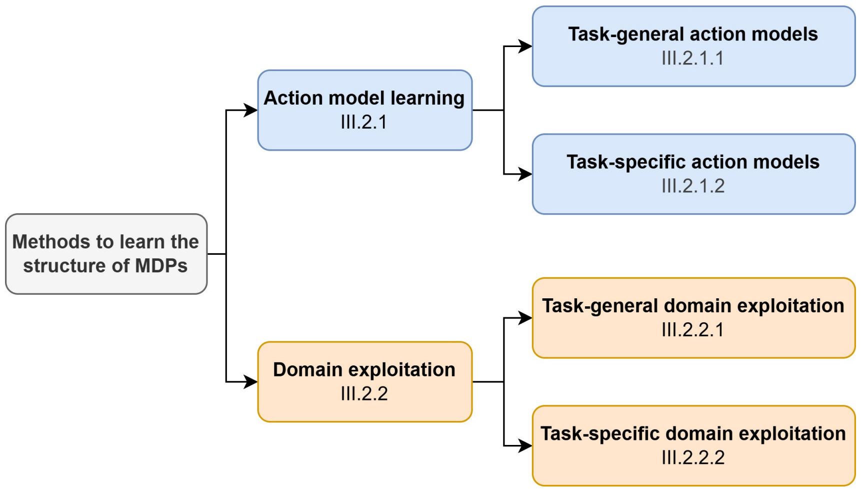

Methods to learn the structure of MDPs
Action model learning
III.2.1
Task-general action models
III.2.1.1
Task-specific action models
III.2.1.2
Domain exploitation
III.2.2
Task-general domain exploitation
III.2.2.1
Task-specific domain exploitation
III.2.2.2

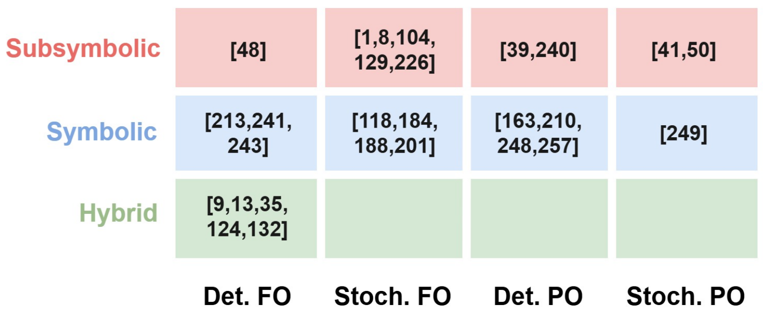
Figure 16: **Proposed taxonomy of methods to learn the structure of MDPs**, with section numbers indicated.

III.2.1 Action model learning

The action model, which also receives the names of planning domain and world model, represents the most important aspects of the MDP structure. It encodes the dynamics of the environment and how the agent can affect them, i.e., the available actions for the agent and the effect each action has on the world state. The action model is essential for AP techniques, which need it to carry out their deliberative process. It is also advantageous for RL as it has been shown that those techniques which use an action model, i.e., model-based RL, are more sample-efficient than those which do not, i.e., model-free RL [122]. Here, we discuss methods for automatically learning the action model from data. These techniques can be categorized according to the scope of the learned model in methods which learn task-general action models and those which learn task-specific action models.

III.2.1.1 Task-general action models

These methods try to learn a general model that represents the dynamics of the environment as accurately as possible and which can, in theory, be applied to solve any task of the corresponding domain. These techniques can be further grouped according to the type of knowledge representation employed by the learned model (see Figure 17).



	Det. FO	Stoch. FO	Det. PO	Stoch. PO
Subsymbolic	[48]	[1,8,104, 129,226]	[39,240]	[41,50]
Symbolic	[213,241, 243]	[118,184, 188,201]	[163,210, 248,257]	[249]
Hybrid	[9,13,35, 124,132]			

Figure 17: **Comparison between methods to learn task-general action models.** Methods are grouped according to their knowledge representation (Y axis) and the type of MDPs they can be applied to (X axis): deterministic MDPs (*Det. FO*), stochastic MDPs (*Stoch. FO*), deterministic POMDPs (*Det. PO*) and stochastic POMDPs (*Stoch. PO*). Methods for learning (stochastic) POMDPs are the most general, as they can be applied to every other MDP category. Methods that use a symbolic or hybrid representation for their learned action model tend to generalize better to states not seen during training. Finally, we note that all the hybrid methods presented in this section tackle the deterministic fully observable case. These works focus on improving the properties (e.g., data-efficiency, interpretability and generalizability) of subsymbolic methods via their proposed hybrid representation, leaving more complex settings (e.g., stochastic and PO) for future work.

Task-general models with subsymbolic knowledge representation. These models represent the environment dynamics in a subsymbolic way, often as a black-box which receives as inputs a state s of the world and an action a to execute, and outputs the next state s'. They are usually trained with ML and DL techniques, e.g., linear regression [226], random forests [104] and DNNs [240], in a supervised manner on samples of the form (s, a, s') collected from the environment.

For an effective learning of subsymbolic action models, some aspects which contribute to the uncertainty of the model must be considered. Firstly, we need to take into account the estimation errors which occur when the model is applied to regions of the state-space not seen during training. Most works, such as PILCO [48], address this problem by estimating the uncertainty in the model predictions. Secondly, most environments are non-deterministic, so the learned action model must reflect this stochasticity in some way. Two possible solutions are to approximate the entire next state distribution [129] and to learn a generative model from which we can draw samples [50]. Thirdly, some environments exhibit partial observability, i.e., they are POMDPs. Action models for POMDPs need to incorporate information about previous states using methods such as belief states [41] or recurrent neural networks [39]. Finally, in order to perform a multi-step look-ahead, the predicted next state s' must be repeatedly fed into the model as input. To prevent prediction errors from accumulating, some works use multi-step prediction losses for training [1] whereas others learn different models for each n-step prediction [8]. A deeper view into subsymbolic action models can be found in [160, 191].

Task-general models with symbolic knowledge representation. These models encode the environment dynamics in a symbolic manner, in terms of a set of objects and their relations, using an interpretable, formal language often based on FOL. In CP, the most popular language is PDDL, whereas PPDDL and RDDL are the languages used for describing stochastic MDPs. Further details on symbolic action model representations can be found in Section II.2.1.

Methods for learning symbolic action models usually receive as input a set of trajectories $(s_0, a_0, s_1, ..., a_{n-1}, s_n)$, obtained by solving planning problems of the corresponding domain. Then, they try to find the planning domain, i.e., the preconditions and effects of each action in the domain, which best fits the given trajectories. The simplest scenario corresponds to learning planning domains for totally-observable, deterministic environments. This is a well-studied problem which has been solved in numerous ways [213, 243, 241]. Action preconditions are inferred by analysing the predicates which appear at the states preceding an action whereas action effects are learned by comparing the predicates of the states before and after applying the action, in what is known as the *delta-state*. Other works learn planning domains for environments with uncertainty, which can come in the form of non-deterministic actions or partial observability of states (POMDPs). [184, 188, 118] tackle the case of non-determinism whereas [248, 257, 163] do the same for partially observable environments. [210] also tackles partially observable environments but is able to learn more expressive domains than the previous methods, with numerical variables and relations. The hardest case corresponds to learning action models for environments which present both non-determinism and partial observability. This problem has been poorly studied, with just one preliminary work trying to address it [249]. Finally, we can classify the existing methods according to the type of algorithm used to learn the planning domain, e.g., RL [201], Supervised Learning [163], inductive rule learning [210], MAX-SAT [248] and transfer learning [257]. A more thorough analysis of symbolic action models can be found in [117, 6].

Task-general models with hybrid knowledge representation. Several works use a hybrid knowledge representation for the action model, one that sits between the black-box representation usually employed by subsymbolic methods and the logic-based representation of symbolic methods. They encode the action model in terms of objects and their relations, which results in better interpretability and generalization to novel situations (e.g., different number of objects) than purely subsymbolic models. [13] learns a physics simulator which receives as input a graph encoding a set of objects and interactions to consider, and applies a DNN to predict the new states of the objects. [35] also learns a physics simulator but implements it as an encoder-decoder architecture, where the encoder summarizes each pair-wise interaction of an object with its neighbours and the decoder predicts the future state of the object. [124] learns a set of abstract *schemas* which encode local cause-effect relationships between objects; these schemas are instantiated with the objects at the scene to form the schema network, a probabilistic model used to predict the reward obtained by executing a given sequence of actions. [132] uses a CNN to extract objects from images, obtains an embedding for each object with an encoder and predicts the interactions between objects with a Graph Neural Network (see Figure 18). Lastly, [9] proposes Latplan, a neuro-symbolic method which uses Variational Autoencoders [130] to learn a planning domain from image pairs describing environment transitions. The learned domain is represented as *grounded* PDDL, i.e., as a PDDL model with only constants and no variables, thus corresponding to propositional logic instead of FOL.

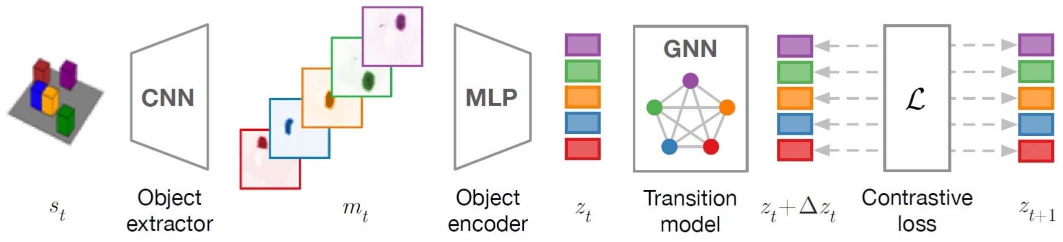


Figure 18: **Action model with a hybrid knowledge representation**. Reproduced with permission by the authors of [132]. Firstly, the *object extractor* (a CNN) receives an image representation of the current state s_t and outputs a set of object masks m_t, used to extract the objects in s_t (each one associated with a different color in the image). Secondly, the *object encoder* (a multilayer perceptron or MLP) receives m_t and returns a set of abstract object states z_t. Thirdly, the *transition model* (a Graph Neural Network) receives z_t and the action to apply to each object as inputs, and predicts the resulting abstract state $z_t + \Delta z_t$. This prediction is then compared with the ground truth z_{t+1}, in order to train the model. © Thomas Kipf

III.2.1.2 Task-specific action models

Unlike the techniques previously discussed, these methods learn a model specifically tailored for a task or set of tasks, i.e., a task-specific action model. This model represents the dynamics of the environment and, additionally, encodes task-related knowledge which is useful for solving the corresponding tasks. Techniques for learning task-specific action models can be further categorized according to how they integrate this extra knowledge inside the action model (see Figure 19).

Value-equivalent action models. The dynamics of complex environments are very difficult to model accurately (as task-general action models try to do), since states are composed of a large number of interrelated elements. However, in most cases only a subset of these state features are actually relevant for the task at hand. Value-equivalent models [87] are a type of subsymbolic action models which, instead of being trained to predict the next state as accurately as possible, learn a state representation useful for predicting the value, i.e., reward, of future states. Thanks to this, they learn to only focus on task-relevant state features and abstract away those aspects of the environment not useful to solve the task. One of the most successful implementations of this idea can be found in MuZero [207]. This work extends the AlphaZero [217] algorithm to the scenario where no action model is provided a priori. MuZero learns a value-equivalent action model which is used by the MCTS planning procedure to predict the value of future states and select the best action. MuZero achieved state-of-the-art results on the Atari video game environment and matched the performance of AlphaZero on Go, chess and shogi.

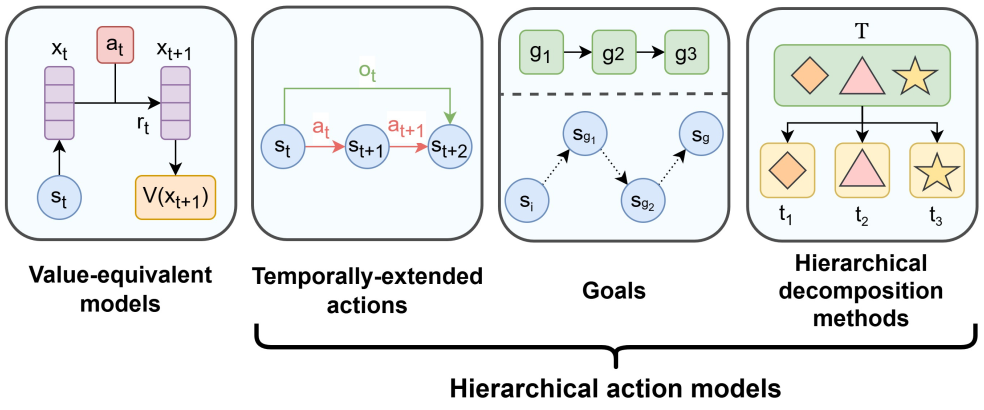


Figure 19: **Comparison among methods to learn task-specific action models** . Methods are grouped according to how they integrate task-specific knowledge into the action model. **Value-equivalent models:** in the image example, the current state s_t is encoded into a task-specific latent representation x_t, which is then unrolled with action a_t to predict the next latent state x_{t+1}, reward r_t and value $V(x_{t+1})$. **Temporally-extended actions:** in the image example, applying the temporally-extended action o_t to s_t is equivalent to executing the sequence of primitive actions a_t, a_{t+1}. **Goals:** in the image example, the sequence of goals g_1, g_2, g_3 guides the low-level policy (first from s_i to s_{g_1}, then from s_{g_1} to s_{g_2} and, finally, to $s_{g_3} = s_g$). Dotted arrows abstract several state transitions. **Hierarchical decomposition methods:** in the image example, the high-level task T is decomposed into three subtasks (each represented by a different shape): t_1, t_2 and t_3.

Value Prediction Networks [185] follow a similar approach, training an action model to predict future rewards and values, which is then used by a search algorithm. In a similar fashion, the Predictron [218] learns an action model which can be repeatedly rolled forward to predict the value of the state received as input.

Hierarchical action models. These models distribute knowledge across multiple levels of hierarchy or abstraction. The bottom level represents the environment dynamics in a general, task-independent way akin to task-general action models. Then, one or more higher levels encode specific, task-dependent knowledge which facilitates the resolution of the corresponding task(s). The main methods for representing hierarchical knowledge are temporally-extended actions [225], goals [4] and hierarchical decomposition methods [81].

Temporally-extended actions [225], also known as macroactions [136] and options [225], are a type of abstract, high-level actions whose execution extends across several time steps, as opposed to primitive, low-level actions which are applied in just one step. A set of options defined over an MDP constitutes a semi-Markov Decision Process [225], a special type of MDP where actions take variable amounts of time to finish. Options consist of three components: a policy, an initiation set and a termination condition. In order to start executing an option, the current state must belong to the initiation set. Once started, the agent selects actions according to the option policy until the termination condition is met. Options make it possible to group actions which are often executed together. In AP, this translates into a reduction in the depth of the search tree [117] whereas, in RL, options facilitate exploration and value propagation, which in turn speeds up learning [156]. Nevertheless, options also increase the number of alternatives to choose from at each state. For this reason, it is important to carefully consider how many options to use. Options have been successfully learned and applied to both the field of AP

[68, 27, 43] and RL [222, 150, 134].

Goal Reasoning [4] provides a design philosophy for agents whose behaviour revolves around goals. It makes it possible to design agents which not only learn how to obtain a particular goal, but also reason about what should be the goal to achieve in the first place. This is especially important for dynamic environments where unexpected events (discrepancies) may require a change in the goals to pursue. Goal-Driven Autonomy [162] provides a general framework for goal-reasoning agents that detect discrepancies, generate possible explanations for them, formulate new goals and manage (prioritize) the pending goals. [245] proposes an agent which learns to formulate goals using expert demonstrations for the StarCraft game. [192] learns to predict future goals based on current and past states and performs an anticipatory planning process which considers both current and future goals. In [179], we propose a neuro-symbolic method which combines DRL with SP to select and achieve goals in order to reduce planning times. Finally, in RL goals have also been utilized as a method to improve exploration [70, 141]. These methods learn a goal space from which goals are sampled in order to direct exploration towards interesting regions of the state space.

Hierarchical decomposition methods make it possible to split a complex problem into a series of subproblems which are simpler to solve, in order to reduce computational effort. The most common way to perform this decomposition is provided by Hierarchical Task Networks (HTN) [81]. HTN domains contain a set of tasks, which can be grouped in either primitive or compound. Primitive tasks correspond to actions that can be directly executed in the environment, in a similar fashion to PDDL actions. Conversely, compound tasks cannot be directly executed and need to be decomposed into a series of primitive or compound tasks. Every compound task has associated one or more decomposition methods, representing different ways to achieve the same task. Each decomposition method has some preconditions which must be true in order to be applied and defines a sequence of subtasks the original task can be decomposed into. HTN planners receive as inputs an HTN domain and a compound task to achieve, known as the goal task. Then, they find a valid decomposition of the goal task as a sequence of primitive tasks by repeatedly applying the decomposition methods available. There exist a wide variety of HTN planners such as NOAH [200], Nonlin [230], SHOP2 [168] and SIADEX [32]. One issue of HTN planning is the substantial time investment needed to encode HTN domains. To alleviate this burden, several methods have been developed to automatically learn HTN domains from data [110, 170, 37]. This learning data is comprised of a set of trajectories obtained from experts and, often, some additional information such as annotated tasks or partial method definitions.

III.2.2 Domain exploitation

The structure of an MDP is completely determined by a planning domain and problem tuple. Some aspects of this structure are explicitly encoded in these elements, e.g., the environment dynamics and available actions are described in the planning domain. However, other properties of the MDP, such as state invariants and landmarks, do not appear in these descriptions. Throughout the years, many techniques have tried to learn this additional structural information with different purposes, such as facilitating the resolution of the MDP or generating data for training some of the methods discussed in Section III.1. Unlike methods for learning action models, this field lacks a proper structure. Thus, we propose to group these techniques under the name *domain exploitation*. In addition, we further categorize them in methods that learn information about the entire domain and those which only learn information about a specific problem or task.

III.2.2.1 Task-general domain exploitation

These methods learn information about the entire domain and do not focus on any particular task. We discuss four different approaches: state invariants, state space clustering, planning problem generation and scenario planning (see Figure 20).

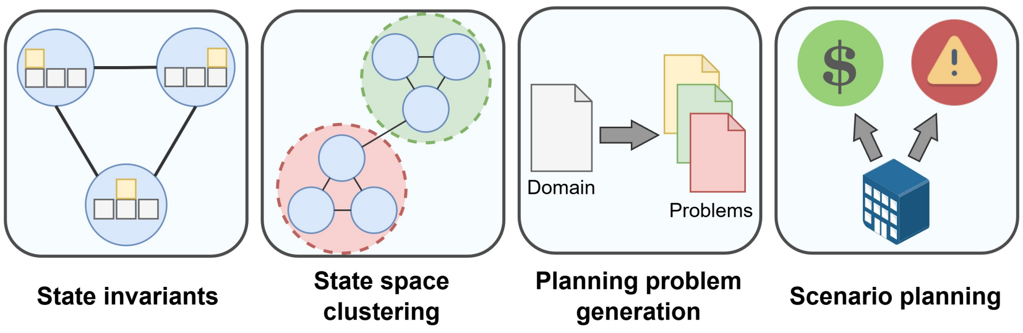


Figure 20: **Comparison among task-general domain exploitation methods. State invariants:** the figure shows an example *exactly-1* invariant where, at each state (blue node), the yellow block is on top of exactly one other block. **State space clustering:** the figure shows an example state clustering method which groups together states that are densely connected (red and green circles in the image). **Planning problem generation:** these methods generate a set of planning problems pertaining to some particular domain. **Scenario planning:** in the image example, a company (represented by a blue building) wants to foresee possible future scenarios, both good (green circle) and bad (red circle), to select the best course of action.

State invariants. These are properties which hold true for every valid, i.e., reachable, state of the MDP. The most widely-known invariant type corresponds to *mutex constraints* [21], which declare that several state properties are mutually exclusive, i.e., cannot be true at the same time. For example, in the *blocksworld* domain, a block X_1 can never be on top of more than one other block at the same time, i.e., the set of atoms $\{on(X_1, X_2), ..., on(X_1, X_n)\}$ are pair-wise mutually exclusive for all n blocks at the state. Another invariant is given by the *exactly-n* constraint which, given a set $P = \{p_1, p_2, ..., p_m\}$ of properties, states that exactly n properties from P must be true in every MDP state. For example, at each *blocksworld* state s, the arm must either be holding one block X_i (i.e., $holding(X_i) \in s$) or be empty (i.e., $handempty() \in s$). This corresponds to an *exactly-1* invariant where $P = \{holding(X_1), ..., holding(X_n), handempty()\}$. There exist many methods for automatically extracting state invariants given a symbolic (e.g., PDDL) domain description. Most methods [101, 82, 196] extract invariants via inductive reasoning: they prove that, if a given invariant is true at some state s, it will remain true at all successor states s' of s (i.e., those obtained by executing an applicable action at s). Some methods follow a different approach. For instance, TIM [71] models the behavior of objects in the domain using finite state machines. It forms classes of objects with shared behavior, from which state invariants are inferred. Finally, state invariants have many applications in SP. For example, the FD planner [100] needs to extract mutex constraints in order to translate the planning task from PDDL to a different encoding in terms of multi-valued variables, whereas the STAN [146] planner leverages the state invariants extracted by TIM in order to enhance system performance.

State space clustering. Some methods apply clustering techniques to the state space of the MDP. They group states together according to a notion of similarity or distance between states, which must be properly defined in order to obtain clusters with the desired qualities. This definition can incorporate information about the task at hand or be completely task-agnostic. Thus, some state space clustering techniques are task-general and obtain a clustering for the entire domain while others are task-specific and focus on obtaining a clustering suitable for a concrete task. Additionally, several methods require a symbolic description of the MDP whereas others do not impose this restriction. [219] proposes a RL algorithm that groups together states s with similar value $V^*(s)$ using a form of soft aggregation, where a state belongs to each cluster with a certain probability. The RL agent learns a value function at the cluster level, and calculates the value of a given state as the weighted sum of the values of the clusters it belongs to. [67] partitions the MDP state space into a smaller number of abstract states or clusters and uses the resulting abstract MDP to compute the optimal policy in a more efficient manner. The used clustering algorithm groups together states that are connected and have similar value. In SP, the most popular approach for state clustering selects a subset P of state variables (e.g., atoms), called a *pattern*, and assigns any two MDP states s_i, s_j to the same abstract state (i.e., cluster) if they share the same value for all variables in P. The clustering obtained is then often leveraged to calculate a planning heuristic, known as a *pattern database heuristic*, based on distances computed over the abstract state space induced by the pattern P [58, 59, 197].

Planning problem generation. Given a planning domain, we may be interested in obtaining a set of planning problems pertaining to that particular domain. Among other applications, they can be used as training data for methods that apply ML to SP (e.g., [212, 10]) and as benchmarks to compare planning performance, as done in the International Planning Competitions. In most situations, these problems need to be created by hand or produced by hard-coded, domain-specific problem generators, which requires great effort from the human designers. Nonetheless, there exist several methods for automatically generating planning problems. [65] generates problems through random walks. It randomly generates an initial state s_i and executes n random actions to arrive at another state s_n. Then, it selects a subset of the atoms of s_n, which constitutes the goal g, and returns the corresponding planning problem (s_i, g). Problems generated with this method are always solvable but they may be inconsistent, i.e., the initial state s_i generated may correspond to an impossible situation of the world (e.g., in *blocksworld*, a state where a block is simultaneously on top of two blocks). [75] also employs a random walk approach but is able to generate problems that are valid, i.e., both solvable and consistent. To achieve this, it receives as inputs the domain description along with some additional information that determines the characteristics of the problems generated, in order to preserve consistency. [232] proposes Autoscale, a method that leverages domain-specific generators in order to obtain problems that are valid, diverse and of graded difficulty, for their use in planning competitions. In [181], we propose NeSIG, a

neuro-symbolic method that uses DRL to learn to generate problems for a given PDDL domain, so that they are valid, diverse and difficult to solve. Lastly, [126] generates complete planning tasks (i.e., domain-problem pairs) that are difficult, diverse and of a particular structure specified by the user.
Scenario planning. This is a decision support technique where the goal is to generate a variety of possible future scenarios to help organizations foresee the future and adapt to it. [127] proposes a semi-automatic, neuro-symbolic method for performing scenario planning in the real world. It uses DNNs to extract forces and their causal relations from a set of documents encoded in natural language. These forces are the elements of study in scenario planning, e.g., *pandemic*, *lockdown* and *loss of benefits*. The causal relations between forces determine *what causes what*, e.g., *pandemic* may cause *lockdown* which in turn may cause *loss of benefits*. Once this information has been extracted, it is translated into a PDDL planning domain and problem, which can then be solved with a symbolic planner. The solutions (plans) found by the planner correspond to possible future scenarios, which can be provided to human experts for their analysis.

III.2.2.2 Task-specific domain exploitation

We now present methods that, instead of extracting information about the entire domain, focus on a specific problem/task and its solution. We discuss three different approaches: goal recognition or inverse RL, landmark detection, and policy validation or safe RL (see Figure 21).

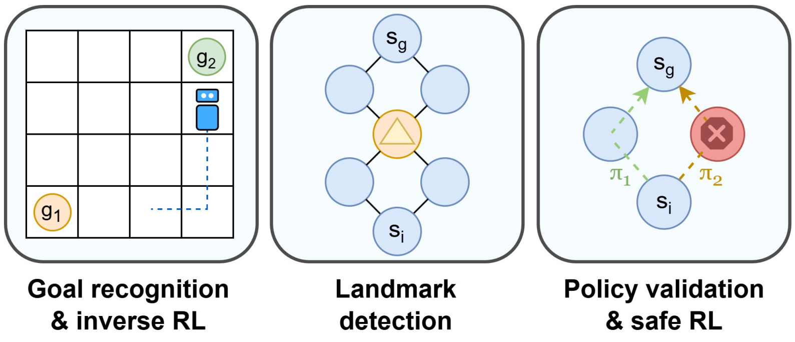


Figure 21: **Comparison among task-specific domain exploitation methods. Goal recognition & inverse RL:** the figure shows an example goal recognition task, where we want to determine whether the agent (represented by a robot) is pursuing goal g_1 or g_2. From the agent's actions (blue dotted line in the image), it results evident that it is pursuing g_2. **Landmark detection:** the figure shows an example state landmark (yellow node with a triangle), as every trajectory from s_i to s_g must necessarily traverse this state. **Policy validation & safe RL:** the image depicts an example safe RL task. Policy π_2 is unsafe, as it traverses an unsafe state (red crossed node), whereas π_1 is safe since it does not traverse that state.

Goal recognition & inverse RL. Goal recognition, also referred to as plan recognition, is the problem of finding the goals that best explain the observed behaviour of an agent. [194] follows a *plan recognition as planning* approach. Instead of using a library of plans, the proposed method only needs to know the planning domain, the possible set G of goals and a partially observed plan p representing the behaviour of the agent. Then, the authors use standard SP techniques to find those goals $g \in G$ for which the optimal plan that achieves them is compatible with the observations in p. Several works have extended this approach, such as [220], which provides a relaxation of the problem formulation that allows it to consider noisy and missing observations.

In the context of RL, goal recognition is known by the name of inverse RL. Here, a different formulation is employed, in terms of rewards instead of goals. The aim of inverse RL is to infer the reward function being optimized by an agent, given its policy or some observations about its behaviour. [172] provides a foundational method to address this problem. It takes an expert's policy as input and utilizes linear programming to obtain the reward function that maximally differentiates the input policy from others, in terms of their optimality. [40] follows a different approach. It uses Bayesian inference to estimate the posterior probability of the reward functions, given some prior distribution over them and the observed behaviour data. A comprehensive survey of inverse RL is provided in [7].

Landmark detection. In SP, landmarks are properties (or actions) that must be true (or executed) at some point for every plan that solves a particular planning problem. Landmarks have been successfully applied to a wide range of problems, such as computing planning heuristics [125] and performing goal recognition [190]. One of the most widely used methods for automatically obtaining planning landmarks can be found in [109]. Given the description of a CP task, this work is able to extract as landmarks those facts (propositions) which must necessarily be made true during the execution of any solution plan. The proposed method also approximates the order in which these landmarks must be achieved, and encodes this information as a directed graph where the nodes correspond to landmarks and the edges to order restrictions between them. Then, this graph is used to decompose the planning task into smaller sub-tasks. An iterative algorithm is used where, at each step, the leaf nodes of the graph (corresponding to those landmarks that can be directly achieved) are handed to the planner as goals and deleted from the graph. This process is repeated until all the landmarks have been achieved. The experiments carried out by the authors show this method can significantly reduce planning times when used in conjunction with state-of-the-art planners. [111] presents a method to automatically extract landmarks for HTN planning. The proposed method represents the HTN task with an AND/OR graph which is used to obtain different types of landmarks corresponding to facts, actions and methods. The authors show this technique is able to extract more than twice the number of landmarks obtained by other methods, which can then be employed to improve the time performance of HTN planners.

Policy validation & safe RL. Given an MDP and a policy (or plan), we may be interested in testing whether the policy actually corresponds to a valid solution of the MDP or not. This problem receives the name of policy (or plan) validation. In SP, one of the most important plan validation systems is VAL [112], which was initially developed to automatically validate the plans produced as part of the 3rd International Planning Competition. It can check whether a given plan, represented in PDDL, is executable and achieves the corresponding goals. In case the plan is flawed, VAL gives advice on how the user can fix it. Additionally, VAL provides different visualization options. Another form of plan validation can be found in the plan monitoring process carried out by online planning architectures. These systems, which interleave planning and execution, must be able to detect *discrepancies*, i.e., unexpected events that require a modification in the behaviour of the agent. For example, the goal reasoning framework proposed in [162] allows agents to detect discrepancies, infer their causes and generate new goals to pursue, which may require a new plan.

Policy safety is a very important aspect of policy validation. Given a policy, we may be interested in determining if there exists some state in the MDP for which our policy performs very poorly. Safe RL tries to address this issue. It comprises RL techniques which, in addition to maximizing reward, satisfy certain criteria regarding the performance (safety) of the system during the learning and/or deployment processes. This is especially important for real-world environments with critical safety requirements, such as helicopter flight [135] and gas turbine control [93]. One possible approach to safe RL is to transform the reward function so that it includes some notion of risk. For example, [99] proposes a pessimistic version of the classical Q-Learning algorithm. Instead of predicting the expected total reward $Q^*(s, a)$ associated with a state s and action a, it estimates the minimum possible total reward (under the optimal policy) for (s, a). This way, it learns the policy that maximizes reward for the worst-case scenario. Other works focus on adapting the exploration process followed by the agent in order to learn the policy. In [231], a human user is allowed to guide an RL agent. The human teacher can provide feedback to the agent in the form of a reward function and, additionally, restrict the set of actions the agent can take at any given moment. Thanks to this guidance, the RL agent is able to learn the optimal policy more efficiently while exploring the state space in a safer way. A comprehensive survey of safe RL can be found in [76].

III.3 Towards an ideal method for SDM

In this section, we try to provide further insight into the existing methods for solving MDPs. Firstly, we propose to categorize these methods along two main dimensions: how they solve the MDP and how they represent their knowledge. Secondly, we discuss what properties an ideal method for SDM should exhibit. Based on these properties, we then analyse the advantages and disadvantages of the different approaches for solving MDPs.

We note that the existing methods for solving MDPs differ from one another in two main aspects. Firstly, they differ in the approach employed to obtain a solution of the MDP. Some methods (e.g., AP algorithms) use an action model to synthesize their solution, often by performing a search or reasoning process over it. Conversely, other methods (e.g., model-free RL algorithms) do not require a model of the MDP and, instead, learn their solution using the data obtained by interacting with the environment. Secondly, methods for solving MDPs differ in the type of knowledge representation employed. Whereas some methods represent their knowledge symbolically using a formal, logic-based language (e.g., PDDL), other methods encode their knowledge subsymbolically, usually into the weights of a DNN.

Figure 22 shows a diagram with different methods for solving MDPs, organized according to how they solve the MDP (Y axis) and how they represent their knowledge (X axis). The bottom of the diagram contains methods that employ an action model to synthesize a solution of the MDP, i.e., AP methods. AP methods that employ a symbolic knowledge representation, i.e., SP methods, are placed on the bottom left corner of the diagram, whereas those with a subsymbolic representation, i.e., NSP methods, are placed on the bottom right corner. The top of the diagram

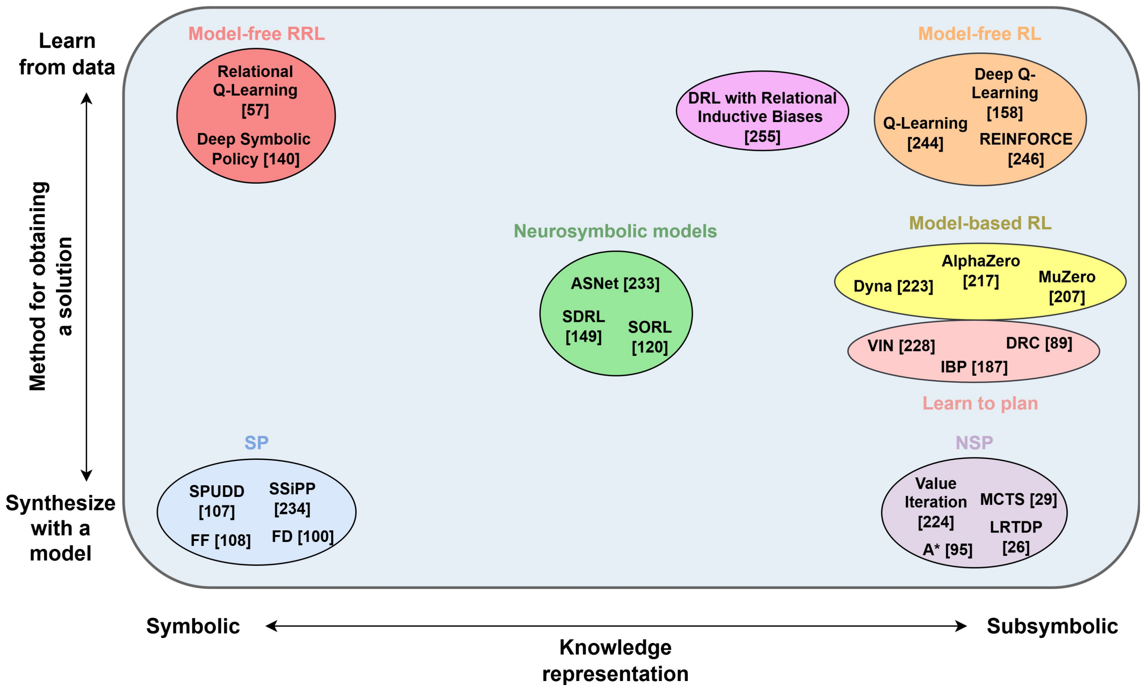


Figure 22: **Diagram of methods according to their knowledge representation and method for obtaining their solution.** Methods placed in the middle of an axis represent hybrid approaches that combine the two extremes of the corresponding axis, i.e., methods that both learn from data and synthesize their solution with a model (Y axis) and methods that integrate symbolic and subsymbolic knowledge representations (X axis). Colored bubbles are used to group similar methods together.

corresponds to methods that do not require an action model and instead learn the MDP solution from the data obtained by interacting with the environment, i.e., model-free RL methods. Most model-free RL methods, from both tabular RL and DRL, represent their knowledge subsymbolically and, thus, are placed on the top right corner of the diagram. Some of them, known as model-free RRL, do employ a symbolic knowledge representation and, hence, are placed on the top left corner. Some RL methods, known as model-based RL, leverage an action model in order to learn their solution of the MDP. Since they combine the two alternative methods for obtaining the MDP solution, they are placed in the middle of the Y axis on the diagram. Since all the model-based RL methods discussed in the paper employ a subsymbolic knowledge representation, they are placed on the right of the diagram. The middle right part of the diagram also corresponds to those methods that *learn to plan*, as they represent their knowledge subsymbolically and both employ an action model and learn their solution from data. The Deep RRL method proposed in [255] corresponds to a model-free RL algorithm (thus placed on the top of the diagram) that employs a DNN with a relational inductive bias in the form of an attention mechanism. Therefore, this work uses a subsymbolic knowledge representation that shares some properties with symbolic ones and, hence, is placed between the middle and right part of the X axis. Finally, neuro-symbolic models combine both methods for obtaining a solution, the *learn from data* of RL and *synthesize with a model* of AP, and additionally integrate the symbolic and subsymbolic knowledge representations. This is why they are placed in the middle of our diagram.

In light of so many different approaches for solving MDPs, we may wonder what are the advantages and disadvantages of the existing methods. In order to answer this question, we first discuss what properties an ideal AI for SDM should exhibit. We frame this question from the perspective of a user who *simply* wants to solve a particular SDM task in the best way possible. Firstly, an ideal SDM method should be **applicable** to any task posed by the user. Secondly, the method should be **easy to use**, requiring as little human effort as possible. Furthermore, it should solve the task **efficiently**. In addition, it should be **interpretable** by the user. Finally, the solution obtained by the method should **generalize** to other tasks different from the one it was obtained for. These requirements give shape to five different properties that result desirable for MDP-solving methods:

- **Applicability.** An ideal method for SDM should be capable of solving all different sorts of MDPs. Nonetheless, there exist many characteristics that make MDPs more difficult to solve and, thus, limit the applicability of SDM methods. In this work, we have focused on two of them: stochasticity and partial observability (i.e., POMDPs). A few other MDP features that limit applicability are: continuity (i.e., MDPs with continuous state spaces S and/or action spaces A), multiple agents (e.g., games like chess where an opponent must be beaten), noisy state observations (e.g., a robot with faulty sensors) and/or noisy actions (e.g., a robot with faulty actuators), the size of the state space, and the complexity of the MDP dynamics.

- **Ease of use.** This property refers to the amount of human effort needed to adapt an SDM method to a particular task, so the larger the effort, the harder it is to use. An ideal method for SDM should require as little human effort as possible. The amount of effort is directly proportional to the quantity of prior knowledge (about the MDP to solve) required by the method and how difficult it is to encode this knowledge in a suitable representation for it.

- **Efficiency.** An ideal method for SDM should obtain a solution of the MDP as efficiently as possible. In Computer Science, efficiency is usually measured in terms of the time and space (i.e., memory) an algorithm needs to solve a problem. Analogously, efficiency in AP is usually measured as the time a method spends to find the MDP solution or, alternatively, as the number of tasks from a set that it is able to solve given some time and memory limits, what is known as *coverage*. On the other hand, efficiency in RL is often measured as the amount of data a method needs to achieve a certain performance, what is known as *data-efficiency*.

- **Interpretability.** In recent years, there has been great interest in developing interpretable/explainable AI (XAI) [91] systems for many different applications, including SDM [34, 105]. An ideal SDM method should be fully interpretable, i.e., humans should be able to understand how it works, the characteristics of the solution obtained by the method and the reasons behind the actions it takes. In addition to understanding and verifying the system, interpretability allows us to modify its behaviour, thus opening the door to building collaborative systems where humans and SDM methods cooperate to solve problems.

- **Generalizability.** In some cases, we may be interested in solving not one but a set of different (although similar) MDPs. Solving each MDP separately may be computationally intractable if the number of MDPs in the set is very large (or even infinite). For this reason, an ideal method for SDM should be able to generalize, i.e., the solution obtained by the method for a particular MDP should also be applicable to solve other similar MDPs.

We now leverage these five desirable properties to compare the different existing methods for solving MDPs, assessing their advantages and disadvantages.

Regarding applicability, most subsymbolic methods manage many different types of MDPs either off the shelf or with some minor modifications. For example, Deep Q-Learning naturally manages non-determinism and has been extended to also manage partial observability and noise [97], and continuous state and action spaces [88]. On the other hand, the symbolic community has historically focused on solving the CP case so, even though there exist symbolic methods capable of managing aspects such as non-determinism (e.g., probabilistic planners) and partial observability [56], their support is currently more limited than the one provided by subsymbolic methods. We believe the reason behind this is that, in order to manage these extensions to the CP case, symbolic techniques require more complex methods for representing knowledge (e.g., PPDDL instead of PDDL for non-determinism) and reasoning with it, which must be designed by humans, as opposed to subsymbolic techniques where a DNN usually *takes care of everything* and naturally manages all these aspects. Therefore, subsymbolic methods exhibit better applicability, in general, than symbolic ones. Additionally, it may be the case that we do not have access to an action model (e.g., the MDP dynamics are unknown) and we cannot learn it with the methods described in Section III.2.1 because either we lack the necessary data or the dynamics are too complex to learn. In these scenarios, methods that synthesize an MDP solution with a model, such as AP, cannot be employed. Therefore, these methods exhibit worse applicability than those that learn their solution from data without a model, such as model-free RL.

Regarding ease of use, SDM methods that require an action model (e.g., AP) are harder to use than those which do not (e.g., model-free RL). This is due to the fact that, as previously commented, in some situations the action model cannot be learned and needs to be designed by a human expert, thus requiring additional human effort. Additionally, symbolic methods generally require more human effort, and thus provide worse ease of use, than subsymbolic ones. Symbolic methods require knowledge to be represented symbolically, in a logic-based language such as PDDL. Nonetheless, some MDPs may be hard (or even impossible) to describe using FOL or some other logic. On the other hand, subsymbolic methods often impose no restrictions on how knowledge must be represented, so the user can choose whichever representation scheme it likes (including symbolic ones). For example, subsymbolic model-based RL methods such as AlphaZero and Dyna are indifferent to how the action model is represented, as long as it allows them to obtain the next state s' and reward r associated with a given state-action (s, a) pair.

The efficiency of SDM methods is directly proportional to the amount of task knowledge that they leverage. On the one hand, methods that employ an action model are usually more efficient than those which do not. For example, it has been shown that model-based RL achieves higher data-efficiency than model-free RL [122]. On the other hand, symbolic representations often provide more knowledge than their subsymbolic counterparts and, hence, result in higher efficiency. For instance, SP techniques make use of powerful domain-independent heuristics to speed up search which, in order to be computed, require a symbolic description of the action model, so they cannot be exploited by AP methods with purely subsymbolic representations.

Regarding interpretability, SDM methods that synthesize their solution with a model (e.g., AP) tend to be more interpretable than those which do not (e.g., model-free RL). This is mainly due to the fact that methods in the first category solve the MDP via an iterative process, carrying out a series of steps which can then be analyzed in order to provide insight that is out of reach for model-free methods. For instance, [151] provides a visualization tool of the search tree of CP algorithms, showing the heuristic value for different stages of the search. This is useful for understanding how different planners solve the task, step by step. Additionally, symbolic methods are more interpretable than subsymbolic ones, as they represent their knowledge in a logic-based language that is comprehensible to humans (or, at least, human experts). For example, PDDL action models explicitly state what are the effects of each action, whereas subsymbolic models do not. Another example is the solution obtained by symbolic methods. For instance, the solution obtained by a CP algorithm is encoded as a sequence of grounded actions and, hence, is more interpretable than the one obtained by a DRL method (which is encoded in the weights of a DNN).

Finally, generalizability depends on the specific scope of each SDM method. Some methods (e.g., CP algorithms) focus on solving a single MDP, whereas others (e.g., Generalized Planning [119] and many DRL algorithms) aim to solve a set of different MDPs. Given the same scope, policies that carry out an iterative process over an action model in order to select the action to execute, i.e., *deliberative* policies, often generalize better than those which do not, i.e., *reactive* policies. For instance, [228] shows that deliberative policies learned by VINs generalize better than reactive policies when applied to new problems of the same domain. Besides, representing knowledge symbolically also helps achieve good generalization. Two examples are RRL, where FOL-based representations are leveraged in order to generalize to tasks with different number of objects, and DNNs with relational inductive biases [255], which generalize better than purely subsymbolic DNNs.

To summarize, SDM methods that synthesize their MDP solution with a model exhibit better efficiency, interpretability and generalizability, whereas methods that learn their solution from data, without a model, are more widely applicable and easier to use. Analogously, symbolic methods present better efficiency, interpretability and generalizability, whereas subsymbolic methods display better applicability and ease of use. As a result of this analysis, we conclude that the two competing paradigms for obtaining the MDP solution (synthesizing it with a model versus learning it from data) and representing knowledge (symbolically versus subsymbolically) are complementary, as the shortcomings of each one correspond to the strengths of the other. Therefore, we argue that an ideal method for SDM should integrate these different approaches into the same architecture: it should both learn and plan in order to obtain its solution, and combine the symbolic and subsymbolic representations for its knowledge. As shown in Figure 22, neuro-symbolic methods perform this integration, which is why they are situated in the center of the diagram. As a result, we believe neuro-symbolic AI poses a very promising approach towards achieving an ideal method for SDM, one exhibiting the five desirable properties presented in this section. This conclusion serves as motivation for the neuro-symbolic

SDM methods developed during the course of this thesis, which are described in Part III.

III.4 Future directions for Sequential Decision Making

The study of related work conducted in this chapter has helped us to identify promising opportunities to further advance the field of SDM via the integration of symbolic and subsymbolic AI. We outline some of these approaches below:

- **Interpretability.** The interpretability of AI systems has become a big concern in recent years due to the ever-increasing ubiquity of such systems, especially in safety-critical applications. Despite advancements in the fields of DL and DRL to design more interpretable architectures such as Graph Neural Networks, the symbolic approach still has the upper hand when it comes to interpretability. Neuro-symbolic methods make it possible to integrate the capabilities of DL and DRL to extract complex patterns from data with the interpretability of classical symbolic representations such as PDDL. They pose a promising approach towards building systems that not only solve tasks in an effective and efficient manner, but which are also able to explain their decisions (actions) to a human supervisor. Additionally, it would be interesting to explore representations not only interpretable by human experts, such as PDDL, but also by any human user, such as natural language.

- **Human-machine collaboration.** The current paradigm for problem solving entails building autonomous systems based on DL and DRL which try to rely on the user as little as possible. However, in most real-world scenarios, humans have access to crucial domain-specific knowledge which can greatly facilitate the learning process of the agent. The main difficulty in providing this information to the system comes from the fact that the knowledge representation commonly employed in modern AI, based on DNNs, is very different to that used by humans. Thus, in order to achieve effective human-machine collaboration, it is essential to reconcile these two representations. One possibility is provided once again by neuro-symbolic methods, since these techniques utilize symbolic representations that are understandable by humans (experts). An alternative approach is to implement a communication interface capable of translating between the subsymbolic representation of the AI system and one understandable by humans (e.g., natural language). Regardless of the chosen method, the integration of both types of representation will make it possible to build AI systems that effectively communicate and cooperate with humans. For example, users will be able to easily provide prior knowledge to the agent and guide its behaviour during the task-solving process. Additionally, it would also be useful that the agent itself could query the human for assistance when needed, e.g., in order to escape a dead-end situation.

- **Goal Reasoning.** Goal Reasoning enables the creation of AI systems that are capable of reasoning about their own goals. This is similar to the way we humans think, as our actions are influenced by a set of goals, intentions, desires, etc. which are not fixed but rather vary with time. Thus, Goal Reasoning provides an ideal framework to design AI agents which collaborate with humans. Ideally, such a system should be able to function autonomously, being capable of formulating, selecting and pursuing the necessary goals to achieve the corresponding task. At the same time, the human user should be able to understand the behaviour and intentions of the system and, at any given moment, modify the goals the agent is pursuing. To build such a system, the neuro-symbolic approach is a good fit. A neuro-symbolic goal reasoning method could use DNNs to build a latent representation of the domain which allowed it to infer interesting goals, encoded in a symbolic representation. Then, a symbolic reasoning method, e.g., an automated planner, could reason about which goals should be pursued at each moment and the specific method for achieving them. In Chapter IV, we propose a neuro-symbolic goal-reasoning approach where an agent receives prior, symbolic knowledge about the environment and goals, a DRL method is used for learning to select goals and, finally, a symbolic AP technique is used to achieve the selected goals.

- **Symbolic value-equivalent action models.** Value-equivalent action models only encode those aspects of the environment dynamics which result useful for the task at hand. This is an essential requirement to learn action models of complex environments, such as those often encountered in real-world tasks, as it is infeasible to accurately depict every aspect of their dynamics. However, so far all value-equivalent action models employ a subsymbolic knowledge representation. We believe this value-equivalence principle would prove even more useful in the case of symbolic action models. Subsymbolic action models employ DNNs to encode their knowledge and, since neural networks are universal approximators, they can accurately represent any aspect of the environment regardless of how complex it is (although learning such a complex representation would require huge amounts of data and would have the risk of overfitting, thus the need for value-equivalent models). On the other hand, some complex environments may be hard (or even impossible) to accurately represent using a symbolic action model, in terms of a set of distinct objects and their relations/interactions. Therefore, when confronted with such kind of environments, we should instead try to obtain a symbolic description of only those aspects of the dynamics that are needed to solve the task at hand, i.e., a symbolic value-equivalent model.

- **Neuro-symbolic action models.** The main idea behind symbolic value-equivalent action models is to find a symbolic description of the environment that is *good enough* for the task at hand, even if it completely ignores some aspects about the dynamics. However, we might wonder if such a description exists in the first place. For some environments, a symbolic description of their dynamics may prove highly inaccurate and leave out aspects that result crucial to solve the task. For example, it would be infeasible to obtain a high-quality description of the dynamics of a complex physics simulator in PDDL and, thus, the resulting symbolic action model would not be of great use to solve any task in this domain. In these cases, those crucial aspects for which a good symbolic description cannot be obtained should be encoded subsymbolically. This neuro-symbolic action model would employ a symbolic representation for those elements of the environment that can be properly encoded in such a manner, and a subsymbolic representation for the rest. This approach could, in theory, combine the best of both worlds: the abstraction and interpretability of a symbolic representation with the accuracy and wide applicability of a subsymbolic one.

Part III

Proposals

Chapter IV

Goal Selection with Deep Q-Learning

IV.1 Introduction

This chapter presents a **neuro-symbolic approach for improving the efficiency of SP algorithms in real-time scenarios**. This is achieved by **learning to select subgoals with Deep Q-Learning**. Therefore, the method described in this chapter fulfills the second subgoal (**G2**) of this dissertation.

The proposed method, called Deep Q-Planning (DQP), integrates a symbolic planner and the DRL algorithm Deep Q-Learning into a hybrid, planning and acting architecture in order to solve tasks where decisions must be made in real-time. DQP receives as prior knowledge the task description in PDDL, along with a set of subgoals that are useful for solving the task. At each step, it selects a subgoal from this set using Deep Q-Learning. Then, an SP algorithm receives the chosen subgoal and finds a plan to achieve it from the current state. Once the agent has finished executing this plan, a new subgoal must be selected, thus repeating the cycle until the task is solved.

This approach is tested on the General Video Game AI (GVGAI) environment [144], used as a standard test-bed for intelligent system applications. Results show DQP outperforms both the SP algorithm and Deep Q-Learning on their own, when both plan quality (i.e., plan length) and time requirements are considered. On the one hand, DQP is considerably more sample-efficient (by at least one order of magnitude) than Deep Q-Learning, and generalizes much better to new game levels. On the other hand, DQP drastically reduces problem-solving times when compared to the planner on its own, at the expense of obtaining plans with only 9% more actions on average. Therefore, DQP is able to exploit the synergy between AP and RL in order to balance plan quality and time efficiency.

IV.2 Related works

The use of DNNs in AP has been a topic of great interest in recent years. Some works have applied Deep Q-Learning to solve planning and scheduling problems as a substitute for online search algorithms. [214] uses Deep Q-Learning to solve the *ship stowage planning problem*, i.e., in which slot to place a set of containers so that the slot scheme satisfies a series of constraints, and optimizes several objective functions at the same time. [164] also employs Deep Q-Learning, but this time to solve the *lane changing problem*. In this problem, autonomous vehicles must automatically change lanes in order to avoid the traffic, and get to the exit as quickly as possible. Here, Deep Q-Learning is only used to learn the long-term strategy, while relying on a low-level module to change between adjacent lanes without collisions. The method proposed in this chapter also employs Deep Q-Learning but, instead of being used as a substitute for Classical Planning, it is integrated along with planning into a planning and acting architecture.

There are other works which use neural networks to solve planning problems but, instead of relying on RL techniques such as Deep Q-Learning, train a DNN so that it learns to perform an *explicit planning process*. [233] proposes a novel neural architecture known as *Action Schema Networks* (ASNet) which, as they explain in their work, *are specialised to the structure of planning problems much as CNNs are specialised to the structure of images*. [228] uses a CNN that performs the computations of the Value Iteration (VI) planning algorithm, thus making the planning process differentiable. Therefore, both works use DNN architectures that *learn to plan* (see Section III.1.3).

These DNNs are trained on a set of training problems and evaluated on different problems of the same planning domain, showing better generalization abilities than most RL algorithms. [228] argues that this happens because, in order to generalize well, DNNs need to learn an *explicit planning process*, which most RL techniques do not. Although our proposed architecture does not learn to plan, it does incorporate an off-the-shelf planner which performs explicit planning. We believe this is why it displays good generalization abilities.

Neural networks have also been applied to other aspects of planning. For instance, [51] trains a DNN that learns a planning domain just from visual observations, assuming that actions have local preconditions and effects. The learned domain is generalizable across different problems of the same domain and, thus, can be used by a planner to solve these problems.

There exist several techniques which facilitate the application of AP to real-time scenarios, such as Goal Reasoning [161], Anytime Planning [195], Hierarchical Planning (e.g., HTN [81]) and domain-specific heuristics learned using ML [252]. [92] presents PELEA, a domain-independent, online execution architecture which performs planning at two different levels, *high* and *low*, being able to learn domain models, low-level policies and planning heuristics. [155] proposes T-REX, an online execution system used to control autonomous underwater vehicles. This system partitions deliberation across a set of concurrent *reactors*. Each reactor solves a different part of the planning problem and cooperates with the others, interchanging goals and state observations.

Some works incorporate Goal Selection into planning and acting architectures. [114] proposes a Goal Reasoning architecture which combines Case-Based Reasoning with Q-Learning. In our proposed method, the focus is on learning to select subgoals, using Deep Q-Learning instead of traditional Q-Learning, in order to give our architecture the ability to generalize to new states. [22] makes use of a CNN which learns to select subgoals from images. Unlike our method, the CNN is trained by a hard-coded expert procedure in a supervised fashion, and the set of eligible subgoals is always the same, regardless of the state of the game.

Finally, it is worth to mention previous disruptive work on Deep RL [159] that addresses how to learn models to control the behavior of reactive agents in ATARI games. Contrary to this work, we are interested in addressing how deliberative behaviour (as planning is) can be improved by mainstream techniques in ML. This is one of the main reasons we chose the GVGAI video game framework, since it provides an important repertory of video games where deliberative behaviour is mandatory to achieve a high-level performance.

IV.3 Materials

IV.3.1 GVGAI and the Boulder Dash game

The method described in this chapter has been tested on the GVGAI environment [144], which comprises a wide variety of tile-based games. For example, it comprises purely reactive games, such as *Space Invaders*, but also games that require deliberative, long-term planning in order to be solved successfully, such as *Sokoban*. Additionally, GVGAI game levels are described using plain text files (known as level description files), as shown in Listing 1, which allows us to create as many levels as we need to train and test our approach.

We have chosen to use a deterministic version of the GVGAI game known as *Boulder Dash* (see Figure 23). We use this game to extract the trajectories/episodes our planning and acting architecture is trained on. In our version of this game, there are no enemies and boulders do not fall. The goal of the player is to collect nine gems and then get to the exit, while minimizing the number of actions used.

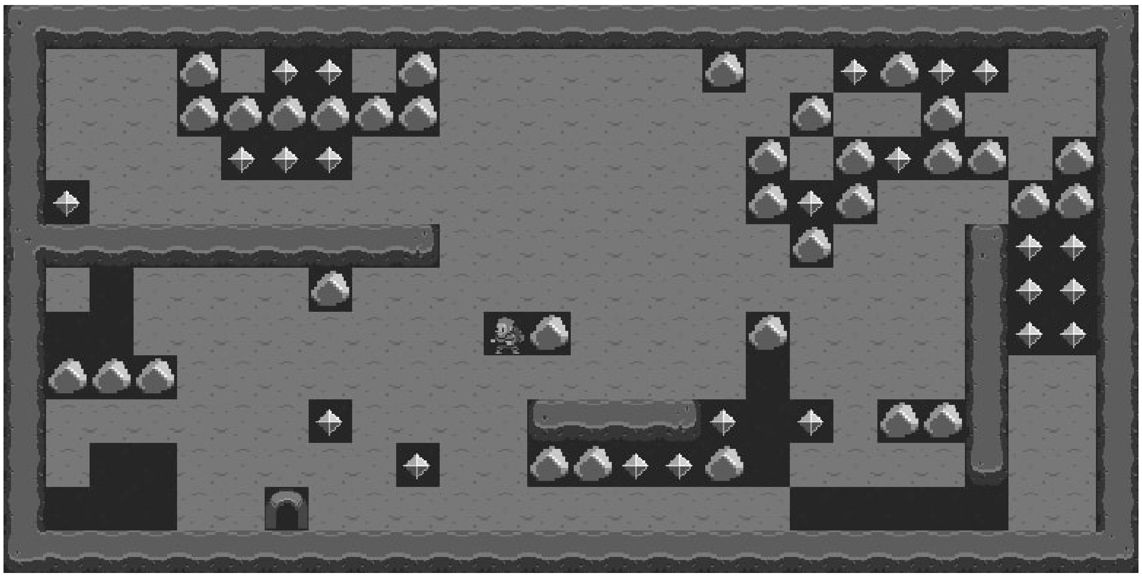


Figure 23: **A level of the Boulder Dash game.**

The player has five different actions at their disposal: *UP*, *DOWN*, *LEFT*, *RIGHT*, and *USE*. The four first actions let the player traverse the level, one tile at a time. The last action, *USE*, is used by the player to break a boulder with its pickaxe before passing through. The player is always pointing in one of four different directions: *NORTH*, *SOUTH*, *EAST* or *WEST*. When the player uses a movement action, they turn towards the corresponding direction or, if they were already facing

```
wwwwwwwwwwwwwwwwwwwwwwwwww
w...o.xx.o......o..xoxx..w
w...oooooo........o..o...w
w....xxx.........o.oxoo.ow
wx...............oxo...oow
wwwwwwwwww........o...wxxw
w.-....o..............wxxw
w--........Ao....o....wxxw
wooo.............-....w..w
w......x....wwwwx-x.oow..w
w.--.....x..ooxxo-....w..w
w---..e...........-----..w
wwwwwwwwwwwwwwwwwwwwwwwwww
```

Listing 1: **Level description file for the Boulder Dash level shown in Figure 23.** Each letter represents a different type of object: "w" for walls, "o" for boulders, "x" for gems, "A" for the player, "e" for the exit and "." for dirt. The character "-" represents empty tiles.

that way, they move one tile. For instance, if the player executes action *UP* and was facing *SOUTH*, they will now face *NORTH*. But, if the player was already facing *NORTH*, they will move up one tile.

We have utilized a static version of Boulder Dash because we need a controllable environment to conduct the experimentation of our proposed method. This way, we can test and validate our goal selection method in an isolated manner, without having to deal with dynamism or uncertainty. For instance, in the original version of Boulder Dash boulders may fall and kill the agent. If this were also the case for our version of the game, we could not assume that a valid plan is always successful, i.e., that it always takes the agent from the current state of the game to a state where the corresponding subgoal has been achieved. Thus, our architecture would additionally need to detect and manage risks associated with the execution of plans in environments with uncertainty, which is left for future work.

Despite this, our deterministic version of Boulder Dash still represents a great challenge for RL and AP techniques, as the results of Section IV.5.4 show. Each level contains 23 gems, but the agent only needs to obtain 9 of them. If we assume the agent always obtains 9 gems and then gets to the exit, then the number of total possible trajectories (plans) is given by the following expression[1]: $\binom{23}{9} * 9! = 296.541.907.200$. Therefore, there exist more than 200 billion different trajectories for a single Boulder Dash level, meaning that Boulder Dash is very hard to solve even without stochasticity.

IV.4 Methods

IV.4.1 The planning and acting architecture

The proposed planning and acting architecture is depicted in Figure 24. The **Execution Monitoring** Module communicates with the GVGAI environment, receiving the current state s of the game. It also supervises the state of the current plan. If it is not empty, it returns the next action a. If it is empty, the architecture

[1]The set of 9 subgoals can be achieved in 9! different ways by the agent.

needs to find a new plan. The **Goal Formulation** Module receives s and generates the compound subgoal G_s, which is the set of the eligible subgoals $g_1, g_2, ..., g_n$ that can be selected at state s. The final goal g_f is also included in G_s. The **Subgoal Pattern** contains the prior information about the domain needed to automatically generate G_s given s. In the Boulder Dash game (see Section IV.3.1), each subgoal g corresponds to getting one of the available gems in s, and the final goal g_f corresponds to getting to the exit. Since all GVGAI games are tile-based, we have associated each subgoal (and also the final goal) with getting to its corresponding tile (cell). The **Goal Selection** Module receives the compound subgoal G_s, and selects the best subgoal $\hat{g} \in G_s$ given s. The **PDDL Parser** encodes $\hat{g}$ as a PDDL single goal, i.e., (*got gem13*), and s as a PDDL initial state, which together make up the PDDL problem. The **Planner** Module receives the PDDL problem along with the PDDL domain, which is provided by a human expert, and generates a plan $p(s, \hat{g})$ which achieves $\hat{g}$ starting from s. In case the Goal Selection Module has selected as $\hat{g}$ a subgoal which cannot be reached from the state s, the Planner Module will not be able to find a valid plan $p(s, \hat{g})$, and will return an empty one. We will refer to this as a *goal selection error*. In Boulder Dash, a goal selection error occurs when the Goal Selection Module chooses the final goal, i.e., $\hat{g} = g_f$, but the agent has not obtained nine gems yet. When this happens, the Goal Selection Module must select a new subgoal $\hat{g}$. Once a valid plan $p(s, \hat{g})$ has been found by the planner, the Execution Monitoring Module receives it and the cycle completes.

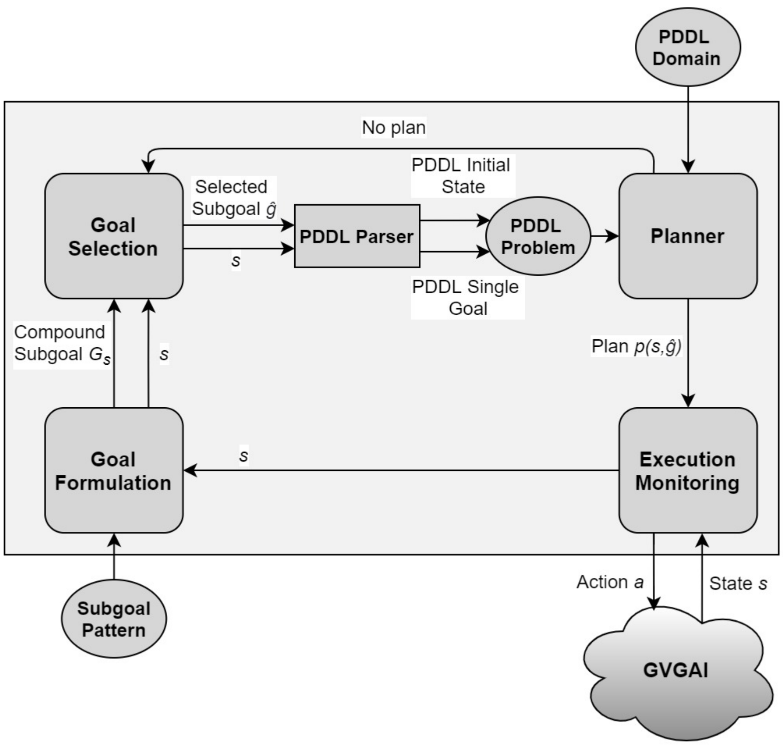


Figure 24: **Overview of the planning and acting architecture.** It integrates an SP algorithm (*planner* submodule in the figure) with the DRL algorithm Deep Q-Learning (*goal selection* submodule) in order to control the behaviour of an agent in a real-time scenario (*GVGAI* in the figure).

IV.4.2 Goal selection learning

This section first provides a formulation of goal selection as a deterministic MDP. Then, it describes the architecture of the CNN used by the Goal Selection Module and how it is trained.

IV.4.2.1 Goal selection as a deterministic MDP

The goal of our planning and acting architecture is to achieve the final goal g_f, i.e., complete a game level, with as few actions as possible. To achieve this, it must select a series of subgoals $g_1, g_2, ..., g_n, g_f$ in the optimal order so as to minimize the length of the total plan that solves the level. In order to select the best subgoal $\hat{g}$ for the current game state s, the Goal Selection Module iterates over every eligible subgoal $g \in G_s$, and predicts the length $l_{P(s,g)}$ of the *total plan* associated with each one. This value $l_{P(s,g)}$ corresponds to the length of the plan $P(s,g)$ which, starting from s, achieves g and, once obtained it, then achieves the final goal g_f (after obtaining the rest of the required subgoals in an optimal order). The best subgoal $\hat{g}$ is the one for which has been predicted the minimum length $l_{P(s,\hat{g})}$. This is the subgoal output by the Goal Selection Module.

In order to predict $l_{P(s,g)}$ for a given (s,g) pair, the Goal Selection Module employs a CNN. This DNN receives as input the current state s of the game and a subgoal $g \in G_s$, both encoded as a three-dimensional tensor, which will be referred to as the *one-hot tensor*, and outputs the number of actions of the associated total plan $P(s,g)$. The first two dimensions of this one-hot tensor are associated with the (x,y) position of a game tile, where the third one is used to encode the object present at that tile. The information about objects is encoded as a one-hot vector. In our version of Boulder Dash, there are six different types of objects: *player*, *exit*, *boulder*, *gem*, *wall* and *dirt*. Each type is associated with a distinct number $i \in \{1..6\}$, representing the *i-th* position of the one-hot vector. If an object of type i is present at the (x,y) tile of the level, then the position (x,y,i) of the one-hot tensor will contain a value of 1. In order to represent subgoals g, we simply treat them as an additional type of object whose associated number is $i = 7$. This way, if the subgoal g of a given (s,g) pair is at the (x,y) tile, then the associated one-hot tensor will contain a value of 1 in its $(x,y,7)$ position.

Unlike most RL problems where the action space is fixed, i.e., where the set of applicable actions $App(s)$ is the same for every MDP state $s \in S$, in our problem the set of eligible subgoals $g \in G_s$ (i.e., applicable actions) depends on the current state s of the game. In addition, each level will contain a different initial set of subgoals, i.e., the subgoals will be in different positions. For this reason, the CNN needs to receive both s and g (encoded as the one-hot tensor (s,g)) as inputs, and must be able to generalize to both new states and subgoals.

We can associate two different deterministic MDPs to our version of Boulder Dash: $M = (S, A, C, T)$ and $M^g = (S^g, G_s, C^g, T^g)$. M corresponds to the standard RL description, whereas M^g is an abstract MDP that describes the game from the perspective of selecting subgoals $\hat{g} \in G_s$ instead of executing actions $\hat{a} \in A$.[2] This

[2] M^g can also be formulated as a semi-MDP [225], where each eligible subgoal $g \in G_s$ would be associated with a different macroaction or option. The policy associated with each option $g \in G_s$ would be encoded by the plan $p(s,g)$ obtained by the planner, its initiation set would be the set of states from which g can be selected, and the termination condition would be the attainment of

alternative formulation M^g is the one our planning and acting architecture is built upon. The correspondence between M and M^g is detailed below:

- S^g is the state space of M^g, which only contains a subset of the states present in the state space S of M, i.e., $S^g \subset S$. S^g contains the initial state of S and also the final states s' of the plans $p(s, g)$ which achieve the subgoals, where $s \in S^g$ and $g \in G_s$.

- G_s is the compound subgoal, i.e., the set of eligible subgoals the agent can choose from at state s. It always contains the final goal g_f. $G_s^{\circ} \subset G_s$ is the set of attainable subgoals for state s, containing the subgoals and/or final goal g for which there exist a valid plan $p(s, g)$ that achieves g starting from state s. If the subgoal selected by the agent is not attainable, the Planner Module will not be able to find a valid plan and a goal selection error will be produced (*see Section IV.4.1*).

- $C^g : S \times G_s \to [0, \infty)$ is the cost function, which depends on the state s and the selected subgoal $\hat{g} \in G_s$. If the subgoal is attainable, i.e., $\hat{g} \in G_s^{\circ}$, then the cost is equal to the length $l_{p(s,\hat{g})}$ of the plan $p(s, \hat{g})$ which achieves $\hat{g}$ starting from s. If $\hat{g}$ is not attainable, then the cost is equal to some large value λ which serves as a penalization for the agent.

- $T^g : S \times G^s \to S$ is the transition function, which determines the next state s' of the environment when the agent selects an eligible subgoal $\hat{g} \in G_s$ at state s. If the selected subgoal is attainable, i.e., $\hat{g} \in G_s^{\circ}$, then the next state s' corresponds to the final state of the plan $p(s, \hat{g})$ that achieves $\hat{g}$ starting from s. Since the planner used to obtain $p(s, \hat{g})$ is deterministic, i.e., for a given $(s, \hat{g})$ it always obtains the same plan, the transition function T^g is also deterministic. If the chosen subgoal is not attainable, i.e., $\hat{g} \notin G_s^{\circ}$, then $s' = s$.

After defining all the elements of M^g we can now adapt the notions of cumulative/total cost C and deterministic policy π to this special type of MDP. A deterministic policy $\pi^g : S^g \to G_s$ maps each state $s \in S^g$ to an eligible subgoal $g \in G_s$. The cumulative cost C^g is the sum of the (immediate) costs $c^g(s, g)$ obtained by the agent when it selects at each state s the subgoal $\pi^g(s) = \hat{g}$, given by its policy, until the end of the episode. The optimal policy π^{g^*} is the one which minimizes C^g, as our goal is to solve each game level using the minimum possible number of actions. π^{g^*} represents the optimal sequence of subgoal selections, where the last subgoal is always the final goal g_f. Finally, it is worth mentioning that one of the main advantages of using M^g to formulate and solve an RL problem, instead of the standard MDP description M, is that the state space is reduced (since $S^g \subset S$) and, thus, the problem is simplified. The implications of this will be explored in Section IV.5.4.

Using the formulation provided by M^g, we can adapt the Deep Q-Learning algorithm to this new type of MDP. In this case, Deep Q-Learning predicts a Q-value $Q(s, g)$ for each (s, g) pair, where $s \in S^g$ and $g \in G_s$, and selects the subgoal $\hat{g}$ with the lowest Q-value. This Q-value $Q(s, g)$ represents the immediate cost $c^g(s, g)$ plus the rest of the cumulative cost C^g, obtained by following the optimal policy

g (i.e., the completion of $p(s, g)$).

π^{g^*}, from the next state s' until the end of the episode. If the subgoal is attainable ($g \in G_s^\circ$), then $Q^*(s,g) = l_{P(s,g)}$. That is to say, the correct Q-value, i.e., the Q-target $Q^*(s,g)$, is equal to the length of the total plan $P(s,g)$ associated with the (s,g) pair. If g is not attainable, then $Q^*(s,g) = c^g(s,g) = \lambda$, i.e., the Q-target is equal to the penalization λ. As in standard Deep Q-Learning, the value of the Q-target is unknown, so it needs to be recursively estimated with the Bellman Optimality Equation (see Equation II.1). Thus, the loss function L^g minimized by Deep Q-Learning for the MDP M^g is as follows:

$$L^g = \Big(Q(s,g) - Q^*(s,g)\Big)^2 = \begin{cases} \Big(Q(s,g) - l_{P(s,g)}\Big)^2, & \text{if } g \in G_s^\circ. \\ \\ \Big(Q(s,g) - \lambda\Big)^2, & \text{if } g \notin G_s^\circ. \end{cases} \tag{IV.1}$$

where $\Big(Q(s,g) - l_{P(s,g)}\Big)^2 = \Big(Q(s,g) - \big(l_{p(s,g)} + \gamma \min_{g' \in G_{s'}} Q(s',g')\big)\Big)^2$, s is the current state, $g \in G_s$ is an eligible subgoal at state s, s' is the next state, and $\gamma \in [0,1]$ is the discount factor for the costs.

IV.4.2.2 CNN architecture and training

The loss function L^g of Equation IV.1 is used to train the CNN of the Goal Selection Module. The architecture of this network has been heavily inspired by the one used in the original Deep Q-Learning paper [158], and is shown in Figure 25. Initially, the size of the one-hot tensor is $(13, 26, 7)$, so we increase its first two dimensions by adding zeros, i.e., we apply zero-padding, until both have the same size of 30. As a result, the CNN receives a square one-hot tensor of size $(30, 30, 7)$ as input. Then, the CNN applies three convolutional layers. The first layer contains 32 filters and the other two 64 filters each, the same as in [158]. The first two convolutional layers apply kernels of size 4×4 with a stride of size 2. The third layer uses kernels of size 3×3 with a stride of size 1. After the convolutional layers, a fully-connected layer with 128 units is applied. Finally, the output layer contains a single unit which outputs the Q-value. We apply batch normalization [113] before each layer of the network except for the output layer. Regarding the Deep Q-Learning algorithm, we have tested different discount factors, and found the best value to be $\gamma = 0.7$. In addition, we have employed several auxiliary techniques to improve the performance of Deep Q-Learning: Fixed Q-targets [159] with $\tau = 10000$, Double Q-learning [237] and Prioritized Experience Replay [206].

This CNN is trained in an offline fashion on datasets extracted from 200 training levels, different from the levels used for testing. These datasets are collected by an agent which performs *random exploration* on these levels, i.e., which plays the levels by selecting subgoals completely at random. The process is the following. The agent starts at the initial state s of the corresponding level. It selects a random eligible subgoal $\hat{g} \in G_s$ and tries to find a plan $p(s,\hat{g})$ to it. If $\hat{g}$ is attainable, the agent executes the obtained plan until it achieves $\hat{g}$ and arrives at state s'. Then, a new sample of the form $(s, \hat{g}, l_{p(s,g)}, s')$ is created and added to the dataset of the level. If $\hat{g} = g_f$, then there is no next state, i.e., $s' = \text{Null}$. If the subgoal $\hat{g}$ is not attainable, no valid plan will be found, so a sample of the form $(s, \hat{g}, \lambda, \text{Null})$ is

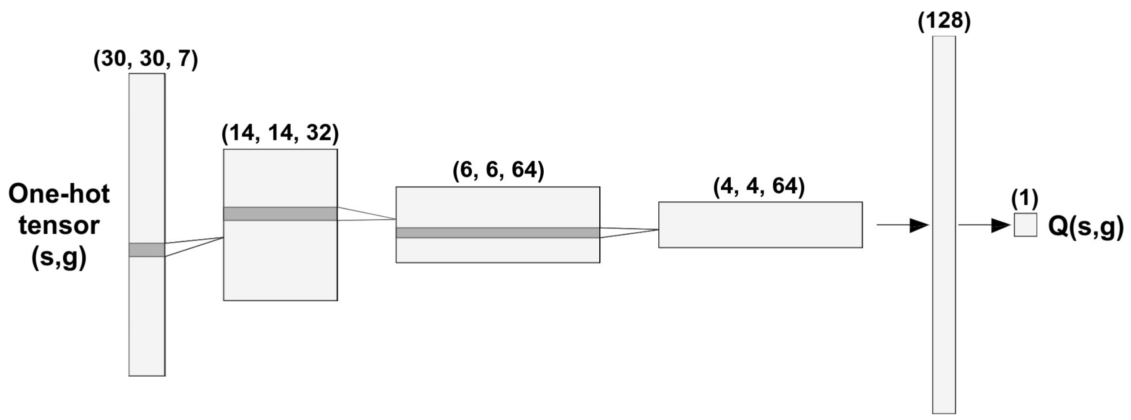


Figure 25: **CNN architecture of the Goal Selection Module.** This diagram shows how the size of the one-hot tensor changes as it passes through the network layers. The CNN receives an input of size $(30, 30, 7)$, corresponding to the one-hot tensor of a given (s, g) pair, and outputs a single prediction representing the Q-value $Q(s, g)$.

created and added to the dataset. This entire process is repeated until the final goal g_f is selected and achieved, so every sample is part of a trajectory which successfully solves the level. After achieving $\hat{g}$, this process starts again from the initial state of the level. Once 500 unique samples have been gathered, the dataset is saved. This algorithm is used to extract the training dataset for each level, for a total of 100000 unique samples, which are then used to train the CNN.

IV.5 Experiments and analysis of results

This section describes the experimental setup and analyzes the results obtained. The goal of our experimentation is three-fold. Firstly, we assess the quality of the plans obtained with our approach depending on the amount of training data used. Secondly, we compare our model with a state-of-the-art planner. Thirdly, we compare it with the standard Deep Q-Learning algorithm. This way, we are able to evaluate the performance of our approach, named Deep Q-Planning (DQP), with respect to both plan quality and time efficiency, when compared to alternative methods.

IV.5.1 Experiment 1: Performance of Deep Q-Planning with respect to dataset size

We have used the FastForward (FF) Planning System [108] for our Planner Module because it is a state-of-the-art classical planner, and one of the most referenced in the planning literature. Furthermore, it is compatible with PDDL2.1 features such as conditional effects and PDDL functions, which are expressive enough to represent domains such as those of video games. Among the different search strategies available for FF, we have decided to use best-first search (BFS). Initially, we tried to obtain plans of optimal length, but this proved too computationally expensive for some levels. Thus, we resorted to the evaluation function $f = g + 5 * h$ for BFS, where g is the current plan length, and h is a plan length estimate (i.e., heuristic value). As a result, the Planner Module is able to obtain plans of almost optimal length in real-time.

We conducted a first experiment designed to evaluate the performance (measured as plan length) of our Deep Q-Planning approach as more training data is made available. We trained the DQP model on the datasets extracted from 10, 25, 50 and 100 training levels, randomly selected among the 200 training levels, and finally on the whole training dataset corresponding to all 200 levels. The DQP model was trained for 1.2 million iterations for every dataset size, using the Adam [131] optimizer with learning rate $\alpha = 1e - 05$ and batch size equal to 32. This translated into 3 hours of training time on a machine with a Ryzen 5 3600X CPU and a RTX 2060 GPU when training 5 model instances in parallel. We set a penalization value of $\lambda = 200$. (*see Section IV.4.2*).

For each dataset size, we trained 10 instances of the DQP model and evaluated each one on 11 test levels, measuring the number of actions needed to solve them. These test levels are different from the training ones, in order to measure the generalization ability of the DQP model when applied to levels not seen during training. In addition, they can be grouped into *easy* and *hard* levels. The easy levels correspond to the 5 Boulder Dash levels provided in the GVGAI environment by default. Additionally, we created 6 more levels to test the DQP model on. These levels were purposely designed to be as hard to solve by the FF planner as possible, but without increasing the level size. For example, we found that FF had trouble solving levels which contained a large amount of boulders.

To put the length of the plans obtained by the DQP model into perspective, we tried to compare them with the optimal plan for each test level. However, as mentioned earlier, obtaining the optimal plans proved to be an intractable problem. Thus, we compared the DQP model with a naive, baseline model, which we call the *Random Model* (RM). This model works the same way as the DQP model except for the fact that, instead of selecting the subgoal with lowest Q-value, it selects a $\hat{g} \in G_s$ completely at random. Therefore, for each trained DQP instance, we divided the length of the plans obtained for every test level by the length of those obtained using the Random Model. Then, we calculated the geometric average of this quotient across all 11 test levels. This way, we obtained for each trained DQP model a metric, called the *action coefficient*, representing the quality of the plans obtained by the model on the test levels. For instance, an action coefficient of 0.6 means that the DQP model uses, on (geometric) average, only 60% of the actions that the Random Model would need to solve the same 11 test levels. Figure 26 shows the average action coefficient (across the 10 repetitions) of the DQP model, and its standard deviation as the dataset size (i.e., number of training levels) increases.

IV.5.2 Experiment 2: Comparison of Deep Q-Planning with a state-of-the-art planner

In addition to this first experiment, we conducted a second set of experiments to compare the performance, measured as both plan quality and time efficiency, of the DQP model with alternative, state-of-the-art approaches. The results are shown in Table 2 and discussed in Section IV.5.4. The DQP model was trained on the entire dataset, corresponding to all 200 training levels. For each test level, we obtained the length of the total plan used to solve it and the time required to obtain this plan. For the DQP model, this time is equal to the sum of the goal selection time

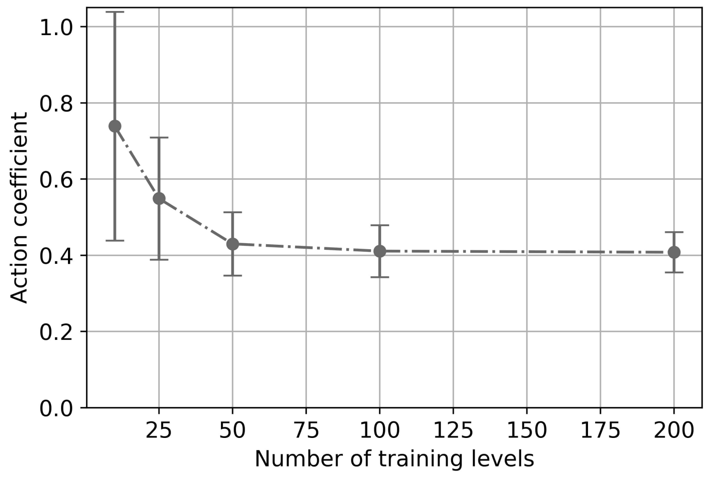


Figure 26: **Plan quality of the DQP model for different dataset sizes.** This plot shows the average action coefficient (lower is better) of the DQP model as the number of training levels is increased. Each error bar represents an interval of ± 1 standard deviation.

and the planning time. The goal selection time is the total time used by the Goal Selection Module to select each subgoal $\hat{g} \in G_s$. The planning time is the total time used by the Planner Module to obtain each plan $p(s, \hat{g})$. We also consider all the time wasted due to goal selection errors (i.e., when $\hat{g} \notin G_s^{\circ}$).

We have decided to use a classical, symbolic planner which performs no goal selection whatsoever as one of the approaches to compare the DQP model against. Specifically, we have chosen FF since it is the same planner the DQP model uses. Therefore, we have applied FF to solve every test level without selecting subgoals, measuring both plan lengths and planning times. We have tested three different search strategies for FF: best-first search with $f = g + h$, so every plan obtained is optimal in length (OPT model), best-first search with exactly the same evaluation function $f = g + 5 * h$ the DQP model uses in order to obtain close-to-optimal plans (BFS model), and enforced hill-climbing with no optimization options (EHC model). Since FF is deterministic, we have performed a single execution per test level. In addition, we have set a maximum of one hour of planning time per level, after which we assume the corresponding model could not solve the level, so a timeout is produced.

IV.5.3 Experiment 3: Comparison of Deep Q-Planning with Deep Q-Learning

We have also compared the DQP model with the standard Deep Q-Learning algorithm, which we will refer to as the DQL model. We have trained the DQL model in an offline fashion, on datasets extracted from the same 200 training levels as DQP, and evaluated it on all 11 test levels, measuring both plan lengths and action selection times. Just as with DQP, we have repeated each execution 10 times. We have also established a maximum of 2000 actions per each test level. If the DQL model does not complete a level under 2000 actions, we assume it could not solve that level.

In order to extract the dataset for each training level, we have not used the ϵ-

Models	Plan Length (Number of Actions)										
	Easy Levels					Hard Levels					
	0	1	2	3	4	5	6	7	8	9	10
DQP	85 ±7	52 ±5	72 ±6	84 ±12	67 ±16	97 ±18	137 ±15	167 ±27	108 ±11	89 ±12	106 ±13
DQL	–	1165 ±0	–	–	–	–	–	–	–	–	–
RM	207 ±47	188 ±59	144 ±32	177 ±45	190 ±46	214 ±67	302 ±90	456 ±101	235 ±91	239 ±55	262 ±74
BFS	81	67	55	77	43	–	–	–	–	–	116
EHC	–	49	44	86	54	117	–	140	86	–	–
OPT	–	31	42	–	38	–	–	–	–	–	–

Models	Time (seconds)										
	Easy Levels					Hard Levels					
	0	1	2	3	4	5	6	7	8	9	10
DQP	1.39 ±0.12	0.54 ±0.05	1.41 ±0.13	0.57 ±0.07	1.35 ±0.14	0.74 ±0.08	1.49 ±0.1	1.09 ±0.19	1.58 ±0.14	0.6 ±0.06	1.43 ±0.1
DQL	–	1.88 ±0	–	–	–	–	–	–	–	–	–
RM	0.47 ±0.1	0.43 ±0.12	0.32 ±0.06	0.39 ±0.1	0.45 ±0.1	0.55 ±0.12	0.61 ±0.16	0.97 ±0.2	0.58 ±0.2	0.53 ±0.11	0.52 ±0.12
BFS	192.52	0.44	0.13	509.35	0.06	–	–	–	–	–	95.17
EHC	–	0.07	0.04	0.32	0.07	813.41	–	102.21	284.06	–	–
OPT	–	4.75	8.62	–	305.92	–	–	–	–	–	–

Models	Success Rate (%)										
	Easy Levels					Hard Levels					
	0	1	2	3	4	5	6	7	8	9	10
DQL	0	10	0	0	0	0	0	0	0	0	0

Table 2: **Comparison of DQP with alternative models.** The table shows mean and std values for plan length (uppermost table) and time requirements (middle table), when solving easy and hard game levels, for all the models considered. A value of – means the corresponding model could not solve that level. In case of BFS, EHC and OPT, this means a 1h timeout was produced. In case of Deep Q-Learning (DQL), this means the model could not complete the level under 2000 actions. Since DQL is stochastic, we also show values (lower table) for its success rate, by measuring how many times it is able to solve each level out of 10 executions. We have performed 10 repetitions for the DQP, DQL and RM models. Since FF is deterministic, we have performed a single repetition for the BFS, EHC and OPT models.

greedy exploration-exploitation strategy commonly employed in RL. This is because, unlike most RL settings, we separate training (exploration) and test (exploitation) in two distinct phases, using different levels for each phase. Thus, we have designed an algorithm inspired by ϵ-greedy but adapted to our problem. The agent starts at the initial state of the level, selects a random subgoal $\hat{g} \in G_s$ (i.e., gem), obtains the plan $p(s, \hat{g})$, and executes it. Once $\hat{g}$ has been achieved, the agent executes a number n of random actions, where n has been uniformly sampled from 1 to 10. After executing the random actions, it obtains and executes the plan to another random subgoal. These two phases (plan execution and random walk) interleave until the agent achieves g_f and solves the level, at which point this process starts again from the initial state. Each time the agent executes an action, a sample is collected and, once 5000 unique samples have been gathered, the dataset is saved.

This algorithm is used to extract the training dataset of the DQL model for each of the 200 levels, for a total of one million unique samples, ten times more samples than for the DQP model. The plan execution phase guarantees each trajectory always solves the level (the agent ultimately achieves g_f), whereas the random walk phase helps the agent explore all the state space. This way, we are able to obtain samples with both good diversity and quality.

These samples are of the form (s, a, r, s') and correspond to the MDP M, which describes the game from a standard RL perspective (*see Section IV.4.2.1*). Each sample is interpreted as follows: the agent is at a state of the game s, it then executes an action a among the set of possible actions $A = \{UP, DOWN, LEFT, RIGHT, USE\}$, arriving at the next state s', and obtaining the immediate cost $c = 1$. It is important to note that, unlike the DQP model, planning is only used here to collect samples. The DQL model itself does not learn to select subgoals, and does not perform any type of planning when solving the test levels.

The DQP and DQL models use the same architecture and hyperparameters, with the following exceptions:

- The CNN of the DQL model receives as inputs both the current state s and a possible action $a \in A$, and returns the predicted Q-value $Q(s, a)$. The state s is encoded as a one-hot tensor, whereas the action a is encoded as a separate one-hot vector, and does not form part of the one-hot tensor encoding. The DQL model also needs to know the orientation of the agent (*NORTH*, *EAST*, *SOUTH* or *WEST*), which is encoded as an additional one-hot vector. These two one-hot vectors, corresponding to the action and the orientation, are concatenated to the flattened output of the third convolutional layer, and together are provided as input to the first fully-connected layer of the CNN.

- Since the DQL model selects actions instead of subgoals like DQP, game trajectories are effectively longer and, thus, we need to apply a smaller discount to costs (i.e., use a larger value for γ). The discount factor of DQL is set to $\gamma = 0.99$, the same value used in [158]. In addition, we experimented with different learning rates and number of training steps (up to 10 million). We found that $\alpha = 5e-06$ was the optimal learning rate, and 6.5 million the best number of training iterations.

- Finally, we have implemented a mechanism for detecting and avoiding loops. A loop occurs when the DQL model arrives at a state s' which has already been visited. Since the agent is deterministic, it will execute the exact same sequence of actions from that point onwards, and will be forever trapped in a loop. In order to avoid this, our algorithm saves a record of the states visited by the agent and how many times they have been visited. If the agent arrives at an unvisited state s', the action with the highest Q-value is selected. If s' has already been visited, then the agent selects the action with the highest Q-value which has not been tried yet. If every action has been executed, then the agent simply takes a random action. This way, the agent can escape loops. However, due to this loop detection mechanism, the behaviour of the DQL model becomes stochastic. For this reason, apart from action selection times and plan lengths, we have also measured its success rate for each test level, i.e., how many times it is able to solve each level out of 10 total executions, while using fewer than 2000 actions.

IV.5.4 Discussion

As Figure 26 shows, the action coefficient of the DQP model decreases as the number of training levels increases. This means that the model performs better, i.e., obtains shorter plans, as it is trained on more data, which was to be expected. First, the performance of the model improves rapidly from 10 to 50 training levels. Then, it slows down from 50 to 100 levels, and there is only a slight improvement when using 200 training levels instead of 100. Thus, the DQP model is able to obtain a close-to-optimal performance when trained on 50000 samples split across 100 levels, reaching an action coefficient of 0.41. This means that, on average, it obtains plans with only 41% of the number of actions used by a model which selects subgoals at random.

As it will be later argued, these are quite remarkable results, since they show that the DQP model is able to properly generalize what has learned in the training levels to the previously unseen test levels, while needing only a fraction of the training data which standard RL algorithms use. Finally, we want to mention that, as more training data is used, the standard deviation of the action coefficient is also reduced. For a dataset of 100 training levels, the standard deviation is equal to 0.068 and, for 200 levels, it is equal to 0.053. This means that, when trained on enough data, the DQP model is able to obtain consistent, stable results across all executions.

Table 2 compares the results obtained by the different models, in terms of plan lengths, time requirements and success rate (the latter metric is only used for the DQL model). Regarding time requirements, we observe how the DQP model is able to solve every test level in under two seconds of total time, which is equal to the sum of goal selection and planning times. Besides, there is no significant difference between the times obtained for easy and hard levels. The Random Model also obtains similar times across all levels, although these times are smaller than those of the DQP model. This happens because the RM model does not spend time selecting subgoals, as they are selected at random.

The behaviour of the models based on Classical Planning (BFS, EHC and OPT) is very different. Planning times for these models vary greatly depending on the particular level, not being able to solve many of them under one hour. The BFS model, which uses the exact same search strategy as the Planner Module of the DQP model, can only solve three easy levels in reasonable time, approximately needs 1.5, 3 and 8 minutes to solve levels 10, 0 and 3, respectively, and cannot solve five out of the six hard levels. The EHC model can only solve four easy levels in a reasonable period of time, can find plans for three hard levels but at the cost of spending much more planning time and, finally, cannot solve one easy level and three hard levels. Lastly, the OPT model can only find the optimal plans for three easy levels, while requiring more than five minutes for one of them.

In addition, it has a success rate of 10% for that level, meaning that it is only able to do it once out of the ten executions. As these results surprised us, we performed additional experimentation for the DQL model in order to validate the model. We designed a simple level consisting of a small confined area with only 23 gems, the agent and the exit. We collected 30000 samples from this level and trained the

DQL model only on the dataset extracted from this level. The DQL model was able to consistently solve this simple level, although not in an optimal way. Therefore, the bad performance of the DQL model shows that the standard Deep Q-Learning algorithm needs to be trained on more than just one million samples (and possibly also for more training iterations) to be able to successfully solve the Boulder Dash game, especially when trained and tested on different sets of levels.

Unlike DQL, the DQP model is able to solve each test level and obtains plans of good quality. It attains an action coefficient of 0.4 meaning that, by learning to select subgoals with Deep Q-Learning, we are able to obtain plans on average 60% shorter than those obtained by selecting subgoals at random. If we divide the length of the plans obtained by the DQP model by those obtained using the BFS model, not considering those levels BFS cannot solve, and calculate the geometric average (as we did to obtain the action coefficient), we obtain a value of 1.09. If we repeat the same calculation for the EHC model, we now obtain a value of 1.15. These values mean that, on average, the DQP model obtains plans with 9% and 15% more actions than the BFS and EHC models, respectively.

To sum up, the results of our experiments show how the DQP model performs better than standard Deep Q-Learning and Classical Planning, when both plan quality and time requirements are considered. On the one hand, we are able to drastically increase the performance of Deep Q-Learning by applying it to select subgoals instead of actions, and using a planner to achieve the selected subgoals. Results show how the DQP model greatly outperforms DQL while being trained on a dataset ten times smaller, meaning our DQP approach is at least one order of magnitude more sample-efficient than standard DQL. Moreover, DQP also generalizes better when applied to levels not seen during training, as DQL only solves 1 out of the 11 test levels considered, and only does so in one out of the ten executions. We attribute this gap in performance to the formulation of the task using goal selection (i.e., as M^g) instead of the standard RL approach (i.e., as M), which reduces the state space (since $S^g \subset S$) and simplifies the learning problem.

On the other hand, the comparison between DQP and the models based on Classical Planning has shown how our goal selection approach is able to substantially reduce the time requirements of classical planners for most levels. In addition, it achieves consistent, stable times across all 11 test levels, whereas FF exhibits drastic variation in time performance, depending on the test level and the search strategy applied. This comes at the expense of a small decrease in plan quality, as our approach obtains plans with 9% more actions than the BFS model, which applies the exact same search strategy as the Planner Module of the DQP model.

IV.6 Conclusion

In this chapter, we proposed a neuro-symbolic, planning and acting architecture that combines DRL with a symbolic AP algorithm. Our method learns to select subgoals using Deep Q-Learning, which are then achieved with the help of a classical, PDDL-based planner. It was trained on a deterministic version of the game known as Boulder Dash, using different levels for training and testing in order to measure its generalization abilities. We conducted experiments to measure how the quality (i.e., length) of the plans obtained with our approach improves as more training data is made available. Additionally, we compared the performance of our model, in terms of both plan quality and time requirements, with that of standard Deep Q-Learning and Classical Planning methods.

The results obtained show our DQP model is able to find plans of good quality while meeting real-time requirements. Thanks to goal selection, it is able to exploit the existing synergy between AP and RL and obtain better results than any of these techniques on their own. On the one hand, we adapted the MDP formulation to our goal selection and planning approach, training the Deep Q-Learning algorithm to select subgoals instead of actions. This way, we were able to improve sample-efficiency and generalization across levels, with DQP obtaining plans of better quality than standard Deep Q-Learning despite the latter being trained on ten times more data. On the other hand, our DQP model substantially reduces the time requirements of AP techniques, at the expense of obtaining plans with only 9% more actions on average. Thanks to goal selection, DQP is able to solve every game level under 2 seconds of total time. Conversely, when no goal selection is performed, the same planner DQP uses can only solve 3 out of 11 levels under one minute.

Chapter V

Heuristic Learning with Admissible Bounds

V.1 Introduction

This chapter presents a **neuro-symbolic method for improving the performance of SP algorithms by using heuristic learning**, which was developed as part of a collaboration between the **University of Granada** and the **MIT-IBM Watson AI Lab**. More specifically, it describes a **statistically-motivated approach for leveraging the prior knowledge contained in symbolic, admissible heuristics in order to learn better heuristics with ML**. Therefore, the method detailed in this chapter fulfills the third subgoal (**G3**) of this dissertation.

This chapter first provides a statistical formulation of heuristic learning as a particular type of supervised learning. Then, it analyzes from a statistical lens the different decisions taken in the heuristic learning literature, such as the choice of training targets and loss to optimize, putting particular focus on how admissible heuristics are utilized during training. From the Principle of Maximum Entropy, it is argued that the heuristic to be learned should be modelled as a Truncated Gaussian (or Truncated Normal) $\mathcal{TN}$ distribution (see Figure 27), where an admissible heuristic acts as its lower bound, thus contraining heuristic predictions to be larger than this admissible heuristic. This modelling choice results in a novel training loss, different from the standard Mean Squared Error (MSE) loss that models the learned heuristic as a Gaussian (or Normal) $\mathcal{N}$ distribution.

The proposed loss function is compared to MSE for learning heuristics from optimal plan costs in a variety of planning domains and learning settings. Experiment results show that modelling the learned heuristic as a $\mathcal{TN}$ instead of a $\mathcal{N}$, i.e., training with the proposed loss function instead of standard MSE, results in faster learning and overall yields more accurate heuristics that improve planning performance.

V.2 Related works

Using admissible heuristics as lower bounds of a $\mathcal{TN}$ distribution may appear trivial in the hindsight. Existing works use $\mathcal{TN}$ for ML most often in the context

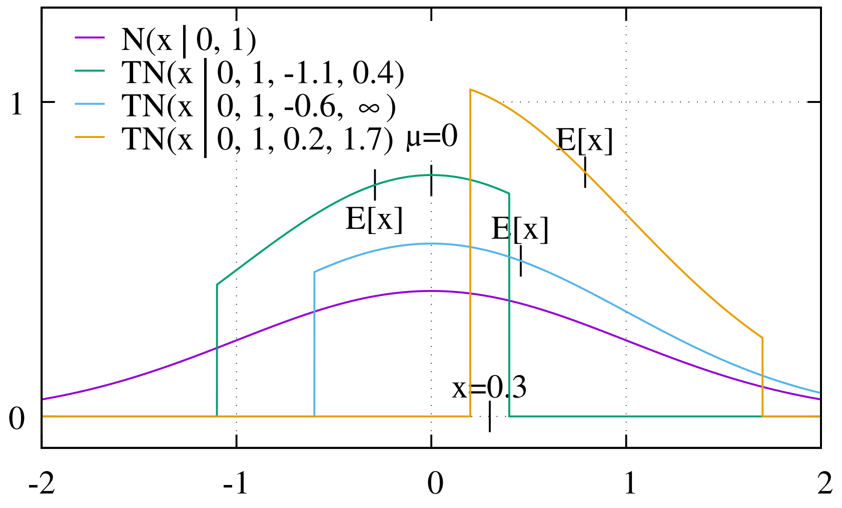


Figure 27: **The Truncated Gaussian distribution.** The figure shows the probability density function (PDF) of several Truncated Gaussian $\mathcal{TN}(\mu, \sigma, l, u)$ distributions. All distributions share the same $\mu = 0$ and $\sigma = 1$ parameters, whereas the lower bound l and upper bound u is different for each one. A $\mathcal{TN}(\mu, \sigma, l, u)$ distribution assigns a probability $p(x)$ equal to zero for any data point x outside the interval (l, u), therefore *truncating* the PDF of a Gaussian distribution $\mathcal{N}(\mu, \sigma)$ to this range. In the method described in this chapter, learned heuristics are modelled as a $\mathcal{TN}(\mu, \sigma, l = h, u = +\infty)$, where the lower bound $l = h$ is given by some admissible heuristic such as $h = h^{\text{LMcut}}$. This serves to encode the prior knowledge about the optimal heuristic/cost-to-go h^* always being equal or larger than an admissible heuristic h. The mean of a $\mathcal{TN}(\mu, \sigma, l, u)$ (denoted as $\mathbb{E}$[x] in the picture) will always lie in the (l, u) range, so the heuristic predictions obtained at planning time (equal to $\mathbb{E}$[x]) will never be lower than the admissible heuristic h used as the lower bound l.

of safety-aware planning, where the upper/lower bounds are *arbitrary* constraints imposed by the environment or by a domain expert. For example, [165] uses $\mathcal{TN}$ to model a Simple Temporal Network with Uncertainty (STNU) [239], which can model a distribution of time within a specific start time or deadline. [60] uses $\mathcal{TN}$ to optimize wireless device allocations, where the truncation encodes the range of signal power. In robotics, $\mathcal{TN}$ is often used to limit the measurement uncertainty [123].

In contrast, the admissible heuristics used as lower bounds in our proposed method are *formal bounds automatically proved by* symbolic algorithms. For example, h^{LMcut} [102] is computed by deriving a so-called landmark graph, and then reducing the costs on the edges that constitute a cut of the graph. To the best of our knowledge, the method detailed in this chapter is the first ever to combine such formally derived bounds with a $\mathcal{TN}$ distribution.

For instance, in applications of ML to Operations Research problems, such as the vehicle routing problem, existing work often tries to learn to solve these tasks without the help of heuristics [169]. Although [247] uses the optimal solution obtained by a traditional optimal method (e.g. Concorde solver) for training and combines it with existing admissible heuristics (i.e., the LKH heuristic) during testing, it does not utilize the heuristic for training.

In the context of heuristic learning for AP, off-the-shelf heuristics have only been used as a training target [212] or as a basis for residual learning [252]. In RL, it is common practice to accelerate training through reward shaping, which is theoretically equivalent to residual learning [171]. There exists an extension of reward shaping [38] that uses hand-crafted heuristics, and [80] employed h^{FF} [108] to shape rewards for CP. However, to the best of our knowledge, none has leveraged admissible heuristics as lower or upper bounds. [38] also discussed the *pessimistic* and *admissible* heuristics as desirable properties of RL and planning heuristics, but their method does not explicitly use the upper/lower bound property for training.

V.3 Methods

V.3.1 Formulation of heuristic learning as a Supervised Learning task

Let $p^*(\mathrm{x})$ be the unknown ground-truth probability distribution of an observable random variable x. In Supervised Learning, our goal is to learn a model (i.e., distribution) $p(\mathrm{x})$ that estimates $p^*(\mathrm{x})$ as closely as possible, using a training dataset $\mathcal{X} = \left\{x^{(1)}, \ldots, x^{(N)}\right\}$ composed of N data points (i.e., examples) sampled from $p^*(\mathrm{x})$. Given a dataset $\mathcal{X}$, we denote its empirical data distribution as $q(\mathrm{x})$, which draws samples from $\mathcal{X}$ uniformly. This distribution is different from both $p(\mathrm{x})$ and $p^*(\mathrm{x})$ because it corresponds to a distribution over a finite set of points, i.e., a uniform mixture of dirac's delta δ distributions (see Equation V.1). The *Maximum Likelihood Estimation* (MLE) framework assumes that the ground-truth distribution $p^*(\mathrm{x})$ is equal to the estimate or model $p(\mathrm{x})$ that maximizes the probability $p(x)$ of observing each training example $x \sim q(\mathrm{x})$. Therefore, under the MLE framework, we maximize the expectation of $p(\mathrm{x})$ over $q(\mathrm{x})$, as shown in Equation V.2:

$$q(\mathrm{x}) = \sum_{i=1}^{N} q(\mathrm{x}|i)q(i) = \sum_{i=1}^{N} \delta(\mathrm{x} = x^{(i)}) \cdot \frac{1}{N} \tag{V.1}$$

$$p^*(\mathrm{x}) = \arg\max_p \mathbb{E}_{q(\mathrm{x})} p(\mathrm{x}) = \arg\max_p \sum_{x \in \mathcal{X}} q(x) \cdot p(x) = \arg\min_p \mathbb{E}_{q(\mathrm{x})} - \log p(\mathrm{x}) \tag{V.2}$$

Typically, we assume $p^*(\mathrm{x})$ and $p(\mathrm{x})$ are of the same family of functions parameterized by θ, such as a set of DNN weights or the trees in random forests, i.e., $p^*(\mathrm{x}) = p_{\theta^*}(\mathrm{x})$, $p(\mathrm{x}) = p_\theta(\mathrm{x})$. This makes MLE a problem of finding the θ maximizing $\mathbb{E}_{q(\mathrm{x})} p_\theta(\mathrm{x})$. This is often achieved by minimizing a *loss* such as the *negative log likelihood* (NLL) $-\log p(\mathrm{x})$, since the logarithm is monotonic and preserves the optima θ^* (see right-most side of Equation V.2). Furthermore, $\mathbb{E}_{q(\mathrm{x})} \ldots$ is often estimated by Monte-Carlo sampling, i.e., $\mathbb{E}_{q(\mathrm{x})} - \log p(\mathrm{x}) \approx \frac{1}{N} \sum_{i=1}^{N} - \log p(x_i)$, where each x_i is sampled from $q(\mathrm{x})$.

We further assume $p(\mathrm{x})$ to follow a specific distribution such as the Gaussian (or Normal) distribution $\mathcal{N}(\mu, \sigma)$:

$$p(\mathrm{x}) = \mathcal{N}(\mathrm{x} \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\mathrm{x}-\mu)^2}{2\sigma^2}} \tag{V.3}$$

We emphasize that **the choice of distribution determines the loss**. When the model designer assumes $p(\mathrm{x}) = \mathcal{N}(\mu, \sigma)$, then the NLL is a shifted and scaled squared error:

$$-\log p(\mathrm{x}) = \tfrac{(\mathrm{x}-\mu)^2}{2\sigma^2} + \log\sqrt{2\pi\sigma^2} \tag{V.4}$$

Likewise, a Laplace distribution $L(\mathrm{x}|\mu, b) = \frac{1}{2b}e^{-\frac{|\mathrm{x}-\mu|}{b}}$ represents the absolute error because its NLL is $\frac{|\mathrm{x}-\mu|}{b} + \log 2b$.

For this reason, the NLL loss is more fundamental and theoretically grounded than losses such as the Mean Squared Error (MSE), although it is "more complex" due to the division by $2\sigma^2$ and the second term in Equation V.4. A reader unfamiliar with statistics may rightfully question why such complications are necessary or why σ is not commonly used by the existing literature. It is because many applications happen to require only a single prediction for a single input (i.e., a *point estimate*). When we model the output distribution as a Gaussian $\mathcal{N}(\mu, \sigma)$, we often predict μ, which is simultaneously the mean and the mode of the distribution and does not depend on σ.

Moreover, the MSE is a special case of the NLL loss that can be derived from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$. To derive the MSE, we first simplify the loss into the squared error $(x-\mu)^2$ by setting σ to an arbitrary constant, such as $\sigma = \frac{1}{\sqrt{2}}$, because the variance/spread of the prediction does not matter in a point estimation of μ. As a result, we can also ignore the second term of Equation V.4 which is now a constant. We then compute the expectation $\mathbb{E}_{q(\mathrm{x})}(\mathrm{x}-\mu)^2$ with a Monte-Carlo estimate that samples N data points $x_1, \dots x_N \sim q(\mathrm{x})$, predict $\mu = \mu_\theta$ using an ML model with parameters θ, and compute the average $\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_\theta)^2$, thus obtaining the formula of the MSE loss. In other words, **the MSE loss is nothing more than the Monte-Carlo estimate of the NLL loss of a Gaussian with a fixed** $\sigma = \frac{1}{\sqrt{2}}$. In contrast, *distributional estimates* represent the entire $p(\mathrm{x})$. For instance, if $p(\mathrm{x}) = \mathcal{N}(\mu, \sigma)$, then the model predicts both μ and σ.

The MLE framework can be applied to the supervised heuristic learning setting as follows. Let $q(\mathrm{s}, \mathrm{x})$ be the empirical data distribution, where s is a random variable representing a state-goal pair (from now on, we will implicitly assume that states s also contain goal information) and x is a random variable representing the cost-to-go (regardless of whether it corresponds to a heuristic estimate, optimal or suboptimal cost). Then, the goal is to learn $p^*(\mathrm{x} \mid \mathrm{s})$ where:

$$p^*(\mathrm{x} \mid \mathrm{s}) = \arg\max_\theta \mathbb{E}_{q(\mathrm{s},\mathrm{x})} p_\theta(\mathrm{x}|\mathrm{s}) \tag{V.5}$$

$$p_\theta(\mathrm{x} \mid \mathrm{s}) = \mathcal{N}(\mathrm{x} \mid \mu = \mu_\theta(\mathrm{s}), \sigma = \tfrac{1}{\sqrt{2}}) \tag{V.6}$$

and $\mu_\theta(\mathrm{s})$ is the main body of the learned model, such as a DNN parameterized by weights θ. Supervised heuristic learning with distributional estimates is formalized in a similar manner, where the only difference is that an additional model (e.g. another DNN) with parameters θ_2 predicts σ:

$$p_{(\theta_1,\theta_2)}(\mathrm{x} \mid \mathrm{s}) = \mathcal{N}(\mathrm{x} \mid \mu = \mu_{\theta_1}(\mathrm{s}), \sigma = \sigma_{\theta_2}(\mathrm{s})) \tag{V.7}$$

V.3.1.1 The Principle of Maximum Entropy

The discussion above models $p(\mathrm{x})$ as a Gaussian distribution. While the assumption of normality (i.e., following a Gaussian) is ubiquitous, one must be able to justify such an assumption. The *Principle of Maximum Entropy* [115] states that $p(\mathrm{x})$ should be modeled as the maximum entropy (*max-ent*) distribution among all those that satisfy our constraints or assumptions, where the entropy is defined as $\mathbb{E}_{p(x)}\langle -\log p(x)\rangle$. A set of constraints defines its corresponding max-ent distribution which, being the *most random* among those that satisfy those constraints, minimizes assumptions other than those associated with the given constraints. Conversely, a non max-ent distribution implicitly encodes additional or different assumptions that can result in an accidental, potentially harmful bias. For example, if we believe that our random variable x has a finite mean, a finite variance and a support (also called domain or range) equal to $\mathbb{R}$, then it *must* be modeled as a Gaussian distribution according to this principle because it is the max-ent distribution among all those that satisfy these three constraints.

In other words, a person designing a loss function for training an ML model must devise a reasonable set of constraints on the target variable x to be learned. Then, it must identify the max-ent distribution $p(\mathrm{x})$ associated with these constraints, which will *automatically* determine the *correct* NLL loss for training the model. In this chapter, we try to follow this principle as faithfully as possible.

V.3.2 Utilizing bounds for learning heuristics

In the previous section, we provided some statistical background on heuristic learning. We now leverage this background to analyze many of the decisions taken in the existing literature, sometimes unknowingly, putting particular focus on how admissible heuristics are used during training. Based on this analysis, we argue that the proper way of utilizing the information provided by admissible heuristics is using them as the lower bound of a Truncated Gaussian $\mathcal{TN}$ distribution representing the learned heuristic.

We previously explained that the heuristic to be learned is modeled as a probability distribution (e.g., a Gaussian $\mathcal{N}$), instead of as a single value. The main reason for this is that the ML model is unsure about the true heuristic value h^* associated with a state s. When it predicts μ, it believes not only that μ is the most likely value for h^*, but also that other values are still possible. The uncertainty of this prediction is given by σ: the larger this parameter is, the more unsure the model is about its prediction. The commonly used MSE loss is derived from the ad-hoc assumption that σ is fixed, i.e., independent from s, which means that the model is equally certain (or uncertain) about h^* for every state s. This is unrealistic in most scenarios: it is generally more difficult to accurately predict h^* for states that are further from the goal, for which the uncertainty should be larger. Therefore, the model should predict σ in addition to μ, i.e., it should output a distributional estimate of h^* instead of a point estimate.

Another crucial decision involves selecting *what* to learn, i.e., the target or ground truth to use for training. It is easy to see that training a model on a dataset containing a practical (i.e., computable in polynomial time) heuristic, admissible or otherwise, such as h^{LMcut} or h^{FF}, does not provide any practical benefits because, even if the training is successful, all we get is a noisy, lossy and computationally-expensive copy of a heuristic that is already efficient to compute. Worse still, trained models always lose the admissibility property even if the training target is admissible. To outperform existing poly-time heuristics, i.e., achieve a *super-symbolic benefit* from learning, it is imperative to train the model on data of better quality, such as h^+ as proposed in [212] or optimal solution costs h^*. Although obtaining these datasets may prove computationally expensive in practice, e.g., h^+ is NP-hard to compute, we can aspire to learn a heuristic that outperforms the poly-time heuristics by training on these targets.

If poly-time admissible heuristics are not ideal training targets, does this mean they are completely useless for learning a heuristic? Intuitively, this should not be the case, given the huge success of heuristic search where they provide strong search guidance towards the goal. Our main question is then *how* we should exploit the information they contain. To answer this question, we must revise the assumption we previously made by using squared errors, i.e., that $\mathrm{x} = h^*$ follows a Gaussian distribution $\mathcal{N}(\mu, \sigma)$. The issue with this assumption is that $\mathcal{N}(\mu, \sigma)$ assigns a non-zero probability $p(x)$ to every $x \in \mathbb{R}$, but we actually know that h^* cannot take some values, e.g., negative values. Additionally, given an admissible heuristic such as h^{LMcut}, we know that $h^{\mathrm{LMcut}} \leq h^*$ holds for every state; therefore $p(x) = 0$ when $x < h^{\mathrm{LMcut}}$. Analogously, if for some state s we know the cost h^{sat} of a satisficing (non-optimal) plan from s to the goal, then h^{sat} acts as an *upper bound* of h^*.

According to the Principle of Maximum Entropy, which serves as our *why*, if we have a lower l and upper u bound for h^*, then we should model h^* using the max-ent distribution with finite mean, finite variance, and a support equal to (l, u), which is the *Truncated Gaussian* distribution $\mathcal{TN}(\mathrm{x}|\mu, \sigma, l, u)$, as proven by [55]. Equation V.8 shows the formula for this distribution:

$$\mathcal{TN}(\mathrm{x}|\mu, \sigma, l, u) = \begin{cases} \frac{1}{\sigma} \frac{\phi(\frac{\mathrm{x}-\mu}{\sigma})}{\Phi(\frac{u-\mu}{\sigma}) - \Phi(\frac{l-\mu}{\sigma})} & l \leq \mathrm{x} \leq u \\ 0 & \text{otherwise,} \end{cases} \tag{V.8}$$

$$\text{where } \phi(\mathrm{x}) = \frac{1}{\sqrt{2\pi}} \exp \frac{\mathrm{x}^2}{2},\ \Phi(\mathrm{x}) = \frac{1}{2}(1 + erf(\mathrm{x})),$$

l is the lower bound, u is the upper bound, μ is the pre-truncation mean, σ is the pre-truncation standard deviation, and erf is the error function. $\mathcal{TN}$ has the following NLL loss:

$$-\log \mathcal{TN}(\mathrm{x}|\mu, \sigma, l, u) = \frac{(\mathrm{x} - \mu)^2}{2\sigma^2} + \log \sqrt{2\pi\sigma^2} + \log \left(\Phi\left(\tfrac{u-\mu}{\sigma}\right) - \Phi\left(\tfrac{l-\mu}{\sigma}\right)\right) \tag{V.9}$$

Modeling h^* as a $\mathcal{TN}$ instead of $\mathcal{N}$ presents several advantages. Firstly, $\mathcal{TN}$ constrains heuristic predictions to lie in the range (l, u) given by the bounds of the distribution. Secondly, $\mathcal{TN}$ generalizes $\mathcal{N}$ as $\mathcal{TN}(\mu, \sigma, -\infty, \infty) = \mathcal{N}(\mu, \sigma)$ when no bounds are provided. Finally, $\mathcal{TN}$ opens the possibility for a variety of training scenarios for heuristic learning, with a sensible interpretation of each type of data, including admissible heuristics and satisficing solution costs.

In this chapter, we focus on the scenario where an admissible heuristic h is provided along with the optimal solution cost h^* for each state, leaving other settings (e.g., satisficing costs h^{sat} in addition to admissible heuristics) for future work. In this case, h acts as the lower bound l of h^*, which is modeled as a $\mathcal{TN}(\mathrm{x} = h^*|\mu, \sigma, h, \infty)$, where μ and σ are predicted by an ML model. Note that we cannot use h^* as $\mathcal{TN}(h^*|\mu, \sigma, h^*, h^*)$ since, during evaluation/test time, we do not have access to the optimal cost h^*. Also, this modeling decision is feasible even when no admissible heuristic is available (e.g., when the PDDL description of the environment is not known, as in Atari games [14]) since we can always resort to the blind heuristic $h^{\text{blind}}(s)$ or simply do $l = 0$, which still results in a tighter bound than the one provided by an *untruncated* Gaussian $\mathcal{N}(\mu, \sigma) = \mathcal{TN}(\mu, \sigma, -\infty, \infty)$.

Finally, our setting is orthogonal and compatible with *residual learning* [252], where the ML model does not directly predict μ but rather a *residual* or offset $\Delta\mu$ over a heuristic h, where $\mu = h + \Delta\mu$. Residual learning can be seen as initializing the model output μ around h which, when h is a good *unbiased estimator* of h^*, facilitates learning. This technique can be used regardless of whether h^* is modeled as a $\mathcal{TN}$ or $\mathcal{N}$ because it merely corresponds to a particular implementation of $\mu = \mu_\theta(s)$, which is used by both distributions. Residual learning is analogous to the data normalization commonly applied in standard regression tasks, where features are rescaled and shifted to have a mean equal to 0 and variance equal to 1. However, residual learning is superior in the heuristic learning scenario because targets (e.g., h^*) are skewed above 0 and because the heuristic used as the basis for the residual can handle out-of-distribution data due to its symbolic nature.

V.3.2.1 Planning with a Truncated Gaussian heuristic

At planning time, we must obtain a point estimate of the output distribution, which will be used as a heuristic to determine the ordering between search nodes. As a point estimate, we can use any statistic of central tendency, thus we choose the mean. It is important to note that the μ parameter of $\mathcal{TN}(\mu, \sigma, l, u)$ is *not* the mean of this distribution since μ corresponds to the mean of $\mathcal{N}(\mu, \sigma)$ (i.e., the mean of the distribution *before truncation*) and does not necessarily lie in the interval (l, u). The mean of a Truncated Gaussian is obtained according to Equation V.10. It is important to note that a naive implementation of this formula results in rounding errors when calculating the mean. In the Appendix, we explain how it can be implemented in a numerically stable manner.

$$\mathbb{E}[\mathrm{x}] = \mu + \sigma \frac{\phi(\frac{l-\mu}{\sigma}) - \phi(\frac{u-\mu}{\sigma})}{\Phi(\frac{u-\mu}{\sigma}) - \Phi(\frac{l-\mu}{\sigma})} \tag{V.10}$$

Equation V.10 satisfies the constraint $l \leq \mathbb{E}[\mathrm{x}] \leq u$. This means that, when a lower bound l is provided (e.g., by an admissible heuristic), the heuristic prediction (equal to $\mathbb{E}[\mathrm{x}]$) returned by the model will never be smaller than l. Analogously, when an upper bound u is also provided (e.g., by a satisficing solution cost), the model will never predict a heuristic value larger than u. Due to this, our hypothesis was that the use of a $\mathcal{TN}$ during planning would help the model make predictions closer to h^* than the bounds l, u themselves, potentially helping it achieve a super-symbolic improvement over admissible heuristics. The results of our experiments (see Section V.4) confirm this hypothesis.

In contrast, the mode $\arg\max_x p(x)$ of the $\mathcal{TN}$ distribution is uninteresting. While we could use it as an alternative point estimate, it is the same as the un-truncated mean μ when the predicted μ is within the bounds, and equal to either the upper or lower bound otherwise (see Figure 27). However, this inspires a naive alternative that is applicable even to $\mathcal{N}$, which is to clip the heuristic prediction $\mathbb{E}[\mathrm{x}]$ (equal to μ for $\mathcal{N}$) to the interval $[l, u]$. This trick should only provide marginal gains as it only improves *really bad* predictions, i.e., those which would lie outside $[l, u]$ otherwise, and does not affect predictions that correctly lie inside $[l, u]$. In our experiments (see Section V.4), we show this approach is inferior to our first method.

We emphasize again that despite the use of admissible heuristics during training the learned heuristic is itself inadmissible, the same as heuristics learned using any other method from existing literature. In case a distributional estimate is employed, i.e., if the ML model also learns to predict σ, we could then discuss *likely-admissibility* [61, 154]. However, this extension is left for future work.

V.4 Experiments and analysis of results

This section describes the experimental setup and analyzes the results obtained. Experiments are designed to compare the accuracy and planning performance of the heuristics learned with our proposed loss function versus standard MSE under the domain-specific generalization setting, where learned heuristics are required to generalize across different problems of a single domain. In this section, we focus on the most important subset of experiments, leaving the rest of the experimentation for the Appendix, which also provides an in-depth description of the experiment configurations.

V.4.1 Experimental setup

Data Generation. We trained our system on four CP domains: *blocksworld*, *ferry*, *gripper*, and *visitall*. For each domain, we generated three sets of problem instances (train, validation, test) with parameterized generators used in the International Planning Competitions [236]. We provided between 456 and 1536 instances for training (the variation is due to the difference in the number of generator parameters in each domain), between 132 and 384 instances for validation and testing (as separate sets), and 100 instances sampled from the test set for planning. The Appendix describes the domains and generator parameters. Notably, the test instances are generated with larger parameters in order to assess the generalization capability. To generate the dataset from these instances, we optimally solved each instance with A* [95] and h^{LMcut} in FastDownward [100] under 5min runtime / 8GB memory (train,val) and 30min runtime / 8GB memory (test). Whenever it failed to solve an instance within the limits, we retried generation with a different random seed up to 20 times until success, thus ensuring a specified number of instances were generated. We also discarded trivial instances where the goal was already satisfied at the initial state. For each state s in the optimal plan, we archived h^* and the values of several heuristics (e.g., h^{LMcut} and h^{FF}). Therefore, each instance was used to obtain several data points.

Model Configurations. We evaluated three different ML methods to show that our proposed approach is implementation-agnostic. Neural Logic Machine (NLM) [53] is an architecture designed for inductive learning and reasoning over symbolic data (see Section II.3.2 for more information). STRIPS-HGN [212] (HGN for short) is another architecture based on the notion of *hypergraphs*. Lastly, we used linear regression with the hand-crafted features proposed by [86], which comprise the values of the goal-count h^{GC} [69] and h^{FF} [108] heuristics, along with the total and mean number of effects ignored by the relaxed plan of the FastForward planner.

We analyze our learning & planning system from several orthogonal axes. **Gaussian vs. Truncated:** Using $\mu(s)$ as the parameter of a Gaussian $\mathcal{N}(\mu(s), \sigma(s))$ or Truncated Gaussian $\mathcal{TN}(\mu(s), \sigma(s), l, \infty)$ distribution. **Learned vs. fixed sigma:**

Predicting $\sigma(s)$ or using a constant value $\sigma(s) = \frac{1}{\sqrt{2}}$, as it is done for the MSE loss. **Lower bounds:** Computing the lower bound l with the h^{LMcut} heuristic. When we use a Gaussian distribution, l is used to clip the heuristic prediction $\mathbb{E}[\mathrm{x}] = \mu(s)$ to the interval $[l, \infty)$. Ablation studies with $l = h^{\max}(s)$ [25] and $l = h^{\text{blind}}(s)$ are included in the Appendix. **Residual learning:** Either using the model to directly predict $\mu(s)$ or to predict an offset $\Delta\mu(s)$ over a heuristic $h(s)$, so that $\mu(s) = \Delta\mu(s) + h(s)$. We use $h = h^{\text{FF}}$ as our unbiased estimator of h^*, as proposed in [252]. In the Appendix, we conduct experiments with h^{LMcut} as the basis of the residual.

Training. We trained each configuration with 5 different random seeds on a training dataset that consists of 400 problem instances subsampled from the entire training problem set (456-1536 instances, depending on the domain). Due to the nature of the dataset, these 400 problem instances can result in a different number of data points depending on the length of the optimal plan of each instance. We performed 4×10^4 weight updates (training steps) using AdamW [147] with batch size 256, weight decay 10^{-2} to avoid overfitting, gradient clip 0.1, learning rate of 10^{-2} for the linear regression and NLM, and 10^{-3} for HGN. All models use the NLL loss for training, motivated by the theory, but note that the NLL of $\mathcal{N}(\mu, \sigma = 1/\sqrt{2})$ matches the MSE up to a constant, as previously noted. For each model, we saved the weights that resulted in the best validation MSE metric during training. On a single NVIDIA Tesla V100, each NLM training took ≈ 0.5 hours except in *visitall* (≈ 2 hours). HGN was much slower (≈ 3 hours except ≈ 15 hours in *blocksworld*). Linear models trained much faster (12-20 minutes).

Evaluation Scheme. We first report two different metrics on the test set: "MSE" and "MSE+clip". Here, MSE is the mean squared error between the ground truth $h^*(s_i)$ and the predicted value $h(s_i) = \mathbb{E}[\mathrm{x}]$, i.e., $\frac{1}{N}\sum_{i=1}^{N}(h(s_i) - h^*(s_i))^2$, for the i-th state s_i of N states in the test dataset. $\mathbb{E}[\mathrm{x}]$ of $\mathcal{TN}$ is given by Equation V.10, while $\mathbb{E}[\mathrm{x}]$ of $\mathcal{N}$ is simply μ. "+clip" variants are exclusive to $\mathcal{N}$ and they clip μ to l, i.e., use $\max(\mu, l)$ in place of μ to compute the MSE. We also obtained the MSE for $h = h^{\text{FF}}$ and $h = h^{\text{LMcut}}$.

We then evaluate the planning performance using the point estimate provided by each model as a heuristic function to guide a search algorithm. Since the learned heuristic is inadmissible, we evaluate our heuristics in an agile search setting, where Greedy Best-First Search (GBFS) [24] is the standard algorithm. We do not use A* because it does not guarantee finding the optimal (shortest) plan [199] with inadmissible heuristics and it is slower than GBFS for agile search, since it needs to explore all nodes below the current best $f = g + h$ value, which is unnecessary for finding a satisficing solution. In our experiments, we evaluate search performance as the combination of the number of solved instances and the number of heuristic evaluations required to solve each instance, with a limit of 10000 evaluations per problem. We do not use runtime as our metric so that results are independent of the hardware and software configuration. Additionally, we evaluated GBFS with the off-the-shelf h^{FF} heuristic as a baseline. The planning component is based on Pyperplan [5].

				learn/h^{FF}		learn/none		fixed/h^{FF}		fixed/none	
domain	metric	h^{FF}	h^{LMcut}	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$
blocks	MSE	22.8	25.06	.76±.1	**.65±.1**	3.26±.6	**2.71±.4**	.83±.1	**.66±.1**	2.97±.9	**2.44±.3**
	+clip			.76±.2		2.91±.4		.83±.2		2.74±.6	
ferry	MSE	9.77	11.10	3.73±.7	**3.45±.8**	141.05±29.4	**8.63±2.7**	**2.98±1.4**	3.85±.9	118.59±10.4	**9.58±1.5**
	+clip			3.72±.6		10.44±28.4		**2.98±1.1**		10.50±9.6	
gripper	MSE	9.93	15.82	3.65±.9	3.70±.9	68.12±16.0	**5.65±1.3**	3.69±.9	3.72±.9	68.22±16.1	**11.97±2.2**
	+clip			3.65±.7		13.37±15.2		3.69±.8		13.38±14.5	
visitall	MSE	13.9	36.4	7.67±.4	**5.30±.6**	25.31±7.9	**9.70±1.6**	6.49±.6	6.62±.9	21.71±2.6	**14.11±1.0**
	+clip			7.60±.4		18.79±7.3		**6.35±.6**		16.38±2.3	

Table 3: **Test accuracy of NLM heuristics.** Each number represents the mean±std of 5 random seeds. For each configuration, we performed 10^4 training steps, saving the checkpoints with the best validation MSE metric. We tested several orthogonal configurations: 1) Learning σ (*learn*) or fixing it to $\frac{1}{\sqrt{2}}$ (*fixed*) and 2) Using residual learning (h^{FF}) or not (*none*). For each configuration, we compare the test MSE metric (smaller is better) of the Gaussian ($\mathcal{N}$) and Truncated Gaussian ($\mathcal{TN}$) models. Rows labeled as *+clip* denote a $\mathcal{N}$ model where μ is clipped above h^{LMcut}. For each configuration, the best average MSE among $\mathcal{N}$, $\mathcal{N}$+clip, and $\mathcal{TN}$ is highlighted in **bold**, if the value gap to the second-best is larger than 0.1. Results for linear regression and STRIPS-HGN models are provided in the Appendix.

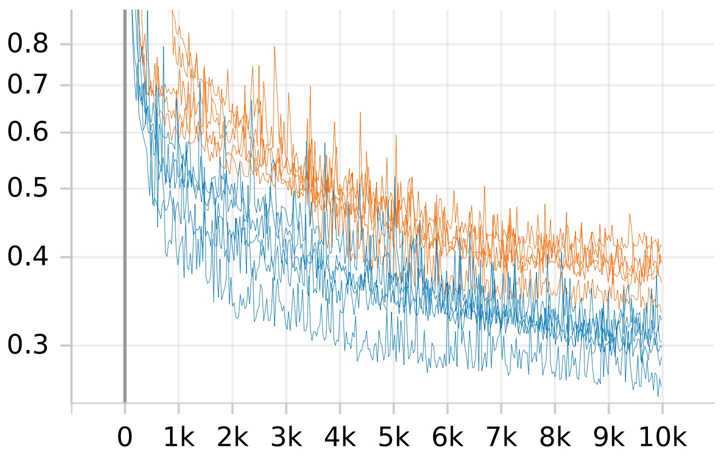


Figure 28: **Convergence speed of $\mathcal{TN}$ vs $\mathcal{N}$ models.** The figure shows a comparison of the training curves (x-axis: training step) for the validation MSE loss (y-axis, logarithmic) between Gaussian $\mathcal{N}$ (orange) and Truncated Gaussian $\mathcal{TN}$ (blue) models on the *logistics* CP domain. For each model, we recorded results from independent runs with five different random seeds. The training loss converges faster for the $\mathcal{TN}$ models due to the additional information provided by the admissible lower bound $l = h^{\mathrm{LMcut}}$.

V.4.2 Heuristic accuracy analysis

We focus on the results obtained by the NLM models, as our conclusions from the Linear and HGN models were not substantially different (see Appendix). Table 3 shows the MSE metric of the heuristics obtained by different configurations evaluated on the test instances (which are significantly larger than the training instances). Compared to the models trained with the NLL loss of $\mathcal{N}$, those trained with our proposed $\mathcal{TN}$ loss often result in significantly more accurate heuristics. For example, in *ferry* and *gripper*, some $\mathcal{N}$ models completely fail to learn a useful heuristic, as shown by the large heuristic errors (e.g., the base $\mathcal{N}$/fixed/none model on *ferry* obtains an MSE of 118.59). In these situations, the clipping trick often reduces errors significantly (e.g., the $\mathcal{N}$+*clip*/fixed/none model on the same domain obtains an MSE of 10.50). However, this simply indicates that the $\mathcal{N}$ models are falling back to the h^{LMcut} heuristic for those (many) predictions which are smaller than h^{LMcut}. This is why, even with clipping, $\mathcal{N}$ models fail to match the accuracy of $\mathcal{TN}$ models in many cases. For example, the MSE of $\mathcal{N}$+clip/learn/none on *gripper* is 7.7 points larger than the one of $\mathcal{TN}$/learn/none. This confirms our hypothesis that admissible heuristics such as h^{LMcut} should be used as the lower bound of $\mathcal{TN}$, instead of simply to perform post-hoc clipping of heuristic predictions.

Additional observations are detailed below. **First,** $\mathcal{TN}$ tends to converge faster during training, as shown in Figure 28. **Second,** residual learning often improves accuracy considerably, thus proving to be an effective way of utilizing inadmissible heuristics. **Third,** we observe that trained heuristics, including those that use residual learning from h^{FF}, tend to be more accurate than h^{FF}. This rejects the hypothesis that residual learning is simply copying the values predicted by h^{FF}. **Fourth,** learning σ helps $\mathcal{TN}$ exclusively. For every $\mathcal{N}$ and $\mathcal{TN}$ model, Table 3 contains 2 comparisons related to σ (learn/none vs. fixed/none and learn/h^{FF} vs. fixed/h^{FF}) across 4 domains, resulting in a total of 8 comparisons. Out of 8 cases, learning σ degrades the MSE of $\mathcal{N}$ in 5 cases, while it improves the MSE of $\mathcal{TN}$ in 7 cases. This happens because σ affects the mean $\mathbb{E}[x]$ of $\mathcal{TN}$ used as the heuristic prediction but it does not for $\mathcal{N}$. In other words, $\mathcal{TN}$ models require both μ and σ in order to achieve good heuristic accuracy. This explains why $\mathcal{TN}$/fixed/h^{FF} is not as competitive as $\mathcal{N}$/fixed/h^{FF}: fixed/h^{FF} is an ill-defined configuration for $\mathcal{TN}$.

V.4.3 Planning performance analysis

		learn/h^{FF} (proposed)			fixed/none (baseline)		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)							
blocks	.13	.84±.19	.85±.19	**.88±.14**	**.79±.29**	.50±.35	.55±.33
ferry	.82	.91±.19	.91±.19	**.98±.05**	.01±.01	.57±.10	**.58±.13**
gripper	.96	**1**	**1**	**1**	0	.92±.12	**1**
visitall	.86	.97±.07	**.98±.06**	**.98±.05**	.82±.33	**1**	**1**
Average node evaluations (smaller is better)							
blocks	9309	2690±2128	2681±2121	**2060±1607**	**4118±2663**	6268±2675	5903±2685
ferry	5152	3216±1964	3117±1967	**2477±1093**	9933±92	6675±582	**6475±725**
gripper	3918	1642±139	1643±141	**1637±492**	10000±0	2941±1513	**1709±658**
visitall	3321	2156±1451	2148±1511	**1683±1290**	3384±3448	**591±216**	612±363

Table 4: **Planning performance of NLM heuristics.** For each model, the table shows the average±stdev of the ratio of solved instances under 10^4 node evaluations and number of evaluated nodes. We use a value of 10^4 node evaluations for instances the planner failed to solve. Results are obtained using the NLM weights that resulted in the best validation MSE during training. We also show results for the off-the-shelf h^{FF} heuristic. For each configuration (learn/h^{FF} or fixed/none), we highlight the best values in bold.

We compared the search performance of GBFS using heuristic functions obtained by the different models as well as the off-the-shelf h^{FF} heuristic. We included our proposed *learn/*h^{FF} configuration and the baseline *fixed/none* configuration. Results for *learn/none* and *fixed/*h^{FF} can be found in the Appendix. Table 4 shows the average±stdev of the ratio of problem instances solved (i.e., coverage), where a value of 1 means all instances are solved, and the average number of node evaluations per problem over 5 seeds. The second metric is introduced to differentiate between methods that solve most or all of the instances.

We observed that, with our proposed *learn/*h^{FF} configuration, the learned heuristics significantly outperform the off-the-shelf h^{FF} heuristic. Additionally, $\mathcal{TN}$ outperforms $\mathcal{N}$ and $\mathcal{N}$+clip in every domain when both the ratio of solved instances and number of node evaluations are considered (the second metric is used to break ties in the first one).

Conversely, with the traditional but less ideal *fixed/none* configuration, several learned heuristics are surpassed by h^{FF} and, also, $\mathcal{TN}$ is outperformed by $\mathcal{N}$ or $\mathcal{N}$+clip in some cases. These results align with those shown in Table 3. Firstly, $\mathcal{N}$ models which do not use clipping sometimes learn dismal heuristics (e.g., in *gripper*, $\mathcal{N}$/fixed/none fails to solve any instance). Secondly, $\mathcal{TN}$ models need to predict σ (in addition to μ) in order to learn heuristics of good quality.

V.5 Conclusion

In this chapter, we studied the problem of supervised heuristic learning under a statistical lens, focusing on how to effectively utilize the information provided by admissible heuristics. Firstly, we provided some statistical background on heuristic learning which was later leveraged to analyze the decisions made (sometimes unknowingly) in the literature. We explained how the commonly used MSE loss implicitly models the heuristic to be learned as a Gaussian distribution. Then, we argued that this heuristic should instead be modeled as a Truncated Gaussian, where admissible heuristics are used as the lower bound of the distribution. We conducted extensive experimentation, comparing the heuristics learned with our truncated-based statistical model versus those learned by minimizing squared errors. Results show that our proposed method improves convergence speed during training and yields more accurate heuristics that result in better planning performance, thus confirming that it is the correct approach for utilizing admissible bounds in heuristic learning.

Our findings serve to answer three important questions within heuristic learning: **What should the ML model learn?** To achieve super-symbolic benefits, we should use expensive targets such as h^*, not poly-time heuristics or sub-optimal plan costs; **how should we train the model?** We should maximize the likelihood of the observed targets h^* assuming a Truncated Gaussian distribution lower-bounded by an admissible heuristic. **Why so?** Due to the Principle of Maximum Entropy. In this setting, the Truncated Gaussian is the distribution that encodes our prior knowledge about h^* being lower-bounded by an admissible heuristic, without any extra assumptions that could result in harmful bias.

Chapter VI

Problem Generation with Neuro-Symbolic AI

VI.1 Introduction

This chapter presents a **neuro-symbolic method for the automated generation of planning problems for any CP domain**[1]**, so that they are valid, diverse and difficult to solve**. Therefore, the method described in this chapter fulfills the fourth subgoal (**G4**) of this dissertation. To the best of our knowledge, no other method in the scientific literature manages to achieve this.

The proposed approach, named NeSIG (Neuro-Symbolic Instance Generator), receives as inputs a PDDL domain description, a set of consistency constraints generated problems must satisfy, the maximum allowed problem size and a list with the predicates and object types that can appear in problem goals. Then, it leverages this information to learn to generate valid, diverse and difficult problems for the planning domain provided as input. Problem generation is formulated as an MDP, in which problems are created step-by-step. First, the problem initial state is obtained by sequentially adding atoms and objects to some starting state, often the empty state. Next, a sequence of actions are executed at the generated initial state to arrive at another (goal) state, from where the problem goal will be obtained according to the goal types and predicates provided by the user. To guide this generation process, two generative policies, which are encoded by NLMs, are trained with DRL to generate valid, diverse and difficult problems. Problem diversity is measured as the distance/dissimilarity between generated problems, whereas difficulty is calculated as the planning effort required to solve them. In order to assess whether a problem is valid or not, we utilize the consistency constraints provided by the user, which may encode rules such as "*an object cannot be at two different places at the same time*". We provide a novel, semi-declarative language [175] that combines Python and FOL for encoding such rules with ease. Therefore, NeSIG reduces human effort when compared to handcrafted domain-specific generators.

We test our method on five CP domains, comparing the problems generated by NeSIG with those obtained by domain-specific generators and several ablations. Results show NeSIG is able to automatically generate valid and diverse problems of much greater difficulty (6.8 times more on geometric average) than domain-specific

[1]Due to the limitations of the parser [174] employed, we have restricted our scope to typed-STRIPS domains with existential and negative preconditions.

generators. Additionally, NeSIG exhibits remarkable generalization capabilities, being able to generalize to problems more than twice the size of those encountered during training.

VI.2 Related works

Several works have proposed domain-independent methods for planning problem generation but, to the best of our knowledge, none of them have been able to generate problems that are simultaneously valid, of good quality and diverse. [65] proposes a random-walk approach to generate planning problems. It randomly creates an initial state s_i and executes n actions at random to arrive at state s_g. Then, it selects a subset of the atoms of s_g, which constitutes the goal g, and returns the planning problem (s_i, g). Although the problems obtained are always solvable, they may not exhibit the other properties (consistency, quality and diversity), as they are generated at random. [75] also employs a random-walk approach but, unlike the previous work, it uses semantics-related information provided by the user to guarantee the consistency of the problems obtained. Thus, this method always generates valid problems but provides no guarantees about their diversity or quality, since they are also generated at random. [154] follows a different approach. It starts from a predefined goal state and performs a backward search for the initial state. The problems obtained are used to learn a planning heuristic. The proposed method estimates its uncertainty and uses this value to search for problems with the right difficulty for training the heuristic. Hence, this method is able to obtain valid problems of good quality. However, it only works for domains where all problems share the same single, predefined goal and for which there exists an *inverse transition model*, i.e., for every action a that transitions from state s to s' an inverse action a' that goes from s' to s must exist, which needs to be provided to the method.

Finally, it is worth to mention several works that address a similar problem to the one tackled in this chapter. [126] proposes a method for obtaining diverse and difficult planning tasks with different causal graphs. This work generates complete tasks (i.e., domain-problem pairs) whereas NeSIG generates planning problems for the particular domain provided by the user. [232] proposes Autoscale, a method for obtaining valid and diverse problems with graded difficulty for their use in planning competitions. However, unlike our proposed method, Autoscale does not generate problems on its own. Instead, it relies on domain-specific instance generators, selecting a set of problems with graded difficulty among the ones they generate. Therefore, Autoscale can be considered complementary to our approach, as it could be used to select problems among those NeSIG generates.

VI.3 Methods

In this section we describe our method, shown in Figure 29. NeSIG takes as inputs a PDDL planning domain, a set of consistency rules generated problems must satisfy and some extra information, corresponding to the maximum size of the problems to generate, and a list with the predicates and object types which

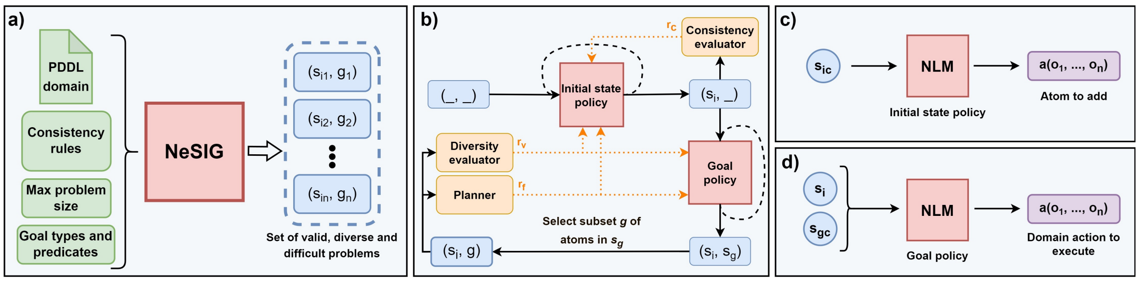


Figure 29: **NeSIG. a) Architecture overview.** NeSIG receives as inputs a PDDL domain, several consistency rules and some extra information (maximum problem size and goal types and predicates). It then trains two generative policies with Deep RL (see subfigure b) so that they learn to generate valid, diverse and difficult problems for the domain provided as input. **b) Policy training with Deep RL.** Dashed lines represent the application of several MDP actions, corresponding to adding an atom to the initial state in the case of the initial state policy (see subfigure c), or executing a domain action in the goal state in the case of the goal policy (see subfigure d). Dotted lines indicate the reward signal, accounting for the consistency r_c, diversity r_v and difficulty r_f of the problems generated. **c) Initial state policy.** It receives an MDP state $(s_{ic}, \_)$ corresponding to a partially-generated initial state and selects the next atom to add to s_{ic}. **d) Goal policy.** It receives an MDP state (s_i, s_{gc}) representing a complete initial state but a partially-generated goal state and selects the next domain action to execute in s_{gc}.

can appear in the problem goals.[2] It then learns to generate problems for that particular domain so that they are valid, diverse and difficult to solve (see Figure 29a). Problems are created via an iterative process that first generates the problem initial state by sequentially adding objects and atoms to some predefined (often empty) state and, then, executes domain actions from the initial state generated to arrive at the goal state, where the problem goal is obtained according to the goal predicates and types specified by the user. We now discuss how validity, diversity and difficulty are defined and measured, present our novel MDP formulation of problem generation and explain how we leverage Deep RL to learn to generate problems with the desired properties.

VI.3.1 Problem properties

VI.3.1.1 Validity

This property can be decomposed into two sub-properties: **solvability** and **consistency**. A problem is considered solvable if there exists at least one valid plan that achieves the problem goal starting from its initial state, i.e., which *solves* the problem. By design, every problem generated by NeSIG is solvable, since the goal of a problem is generated by executing applicable domain actions from its initial state. In other words, the trajectory followed to generate the problem goal from its initial state is a valid plan that solves the problem. A problem is considered *consistent* if its initial state represents a possible initial situation (state) within the system modeled by the planning domain, in other words, if it *makes sense*. An example consistency rule would be "*an object cannot be at two places at the same time.*" Consistency constraints arise from the semantics of the domain and, since

[2]Additionally, NeSIG may also take as input the list of object types that can be added to the problem initial state during generation. Nonetheless, this is completely optional and is only used for improving NLM efficiency.

they are not encoded in its PDDL description, they need to be provided separately. Additionally, they depend on human interpretation and preferences. Going back to our previous example, some user could consider a state where one object is at two different places ($at(o, p_1)$, $at(o, p_2)$) at the same time to be consistent, and that choice would be completely valid as there is nothing in the PDDL domain that forbids it. Since NeSIG learns to avoid inconsistent problems, consistency rules can then be used to control the distribution of problems generated (e.g., forbid problems with more than N objects of type t in their initial state). Thus, consistency rules act as *hard constraints* imposed on the initial state of problems.

Due to the sequential nature of our proposed method, in which problem initial states are generated by incrementally adding objects and atoms to an initially empty state or some other state provided by the user, we distinguish between **continuous** and **eventual** consistency. A continuous consistency rule is one which must be *continuously* satisfied throughout the entire initial state generation process. In order to make a continuous-inconsistent state consistent again we would need to remove some atom(s) and/or object(s) from the state, which is forbidden in our method. For this reason, NeSIG never adds objects or atoms which result in continuous-inconsistent states. An example continuous consistency rule would be "*an object o cannot be at two places p_1, p_2 at the same time*", i.e., $at(o, p_1), at(o, p_2)$ is forbidden. If this constraint is not met, we would need to remove either $at(o, p_1)$ or $at(o, p_2)$ from the initial state which, as previously stated, is forbidden. On the other hand, eventual consistency rules are those which must be *eventually* satisfied once the initial state has been completely generated, but do not need to be met at each step of the generation process. An eventual-inconsistent state can be made consistent if some particular combination of object(s) and/or atom(s) are added to it. Therefore, eventual consistency is only checked at the end of the initial state generation process. An example eventual consistency rule would be "*the initial state must contain at least one object of type t*". If this constraint is not met, we can simply add an object of type t to the state to make it eventual-consistent. We note that, whenever some constraint can be encoded with both continuous and eventual consistency rules, we should always choose the former. The reason for this is that continuous consistency rules serve to prune partially-generated initial states from which no eventual-consistent state can be generated, thus facilitating learning by removing dead-end states. Intuitively, the distinction between continuous and eventual consistency is analogous to that of dense versus sparse rewards in RL.

Consistency rules are encapsulated in a consistency evaluator that provides two methods. The first one returns whether the state resulting from adding some atom (and optionally some objects) to the current state is continuous-consistent or not. The second method receives a completely-generated initial state and checks whether it is eventual-consistent or not. Although consistency rules must be provided by a human designer on a per-domain basis, doing so is often much simpler than devising a procedure for generating a diverse set of consistent problems, i.e., programming an instance generator. To reduce human effort even further, we have designed a novel, semi-declarative language for describing consistency rules. It allows the construction of first-order logic (FOL) formulas (with counting quantifiers) expressing conditions about state objects and atoms. For example, the consistency rule "*the initial state must contain at least 3 objects of type city*" can be concisely expressed as $TE(x, type(x, city)) >= 3$, where TE stands for *There Exists* and x is a FOL variable. These formulas are then automatically evaluated, and their truth value is stored in a Python boolean variable. Therefore, we can encode consistency rules using either standard Python, FOL or a combination of them. This choice is transparent to NeSIG and does not impact training. We detail the consistency rules for each domain in the Appendix, showing how our semi-declarative language makes possible to represent consistency constraints in an interpretable manner with just a few lines of code. Finally, we have released our semi-declarative language as a standalone Python package on GitHub [175], making it available for use in other projects.

VI.3.1.2 Diversity

This property measures how different generated problems are from each other. In order to measure diversity, we automatically extract a set of features for each problem. We say that two objects are *connected* if they are instantiated on the same atom, regardless of position. Based on this idea, we define the sets of *connection features* c_μ and c_σ. $c_\mu[t_i][p][t_j]$ encodes how many objects of type t_j, on average, each object of type t_i is connected to through atoms of predicate type p. Analogously, $c_\sigma[t_i][p][t_j]$ contains the standard deviation instead of the mean number of connections. For example, a value $c_\mu[city][in][location] = 3$ means that each city contains (atom *in*) an average of three locations, whereas $c_\sigma[city][in][location] = 2$ means that the standard deviation between the number of locations in each city is 2 (i.e., not every city contains the same number of locations). In total, we extract 7 groups of features, corresponding to the number of objects of each type in the problem and, separately for the initial state and goal, the number of atoms of each predicate type, c_μ and c_σ. They are divided by their sum so that, for each problem, features in each group add up to one. Then, we calculate the pairwise problem distance as the absolute difference between their feature vectors, dividing distances by $7 * 2 = 14$ to normalize them to the $[0, 1]$ range. Finally, the diversity of a problem is equal to its average distance to all the problems in the set (excluding itself). We have chosen this group of features for measuring problem diversity because they are easily interpretable, can be efficiently extracted by a domain-independent method, and consider all the constituents of a problem (objects and atoms) and how they relate among them (c_μ and c_σ).

VI.3.1.3 Difficulty

In this work, we measure the quality of a problem by its difficulty. In other words, our goal is to generate problems which are as hard to solve by a planner as possible (in addition to being consistent and diverse). We have chosen difficulty as our quality measure because it plays a central role in AP, where great effort has been devoted to studying problem difficulty [42] and developing efficient algorithms for solving difficult problems [25]. We measure difficulty as either the planning time or number of expanded nodes of a particular planner in order to solve the problem. Since this measure depends on the planner employed, during training we calculate problem difficulty with a satisficing planner and then, at test time, we utilize a different set of satisficing and optimal planners in order to assess whether NeSIG is capable of generating problems that are challenging for different planners.

VI.3.2 Formulation of problem generation as an MDP

We propose to generate problems of the form (s_i, g), where s_i is the problem initial state and g is the goal, via an iterative process which first generates s_i and then g. The initial state generation phase starts either from an empty state (with no objects or atoms) or from some predefined state provided by the user. Then, at each step, a new atom is added to the initial state and, optionally, one or more new objects. Once s_i has been completely generated, the goal generation phase begins if the state meets the eventual consistency constraints. Otherwise, the problem is discarded. Starting from s_i, the goal generation phase successively executes the actions available in the domain to arrive at another state, known as the goal state s_g. Finally, the goal g is obtained by selecting a subset of the atoms in s_g, according to the goal predicates and object types specified by the user. For instance, in the *blocksworld* domain, problem goals only contain atoms of the form *on(block,block)* by design. This entire process is depicted in Figure 29b and a handcrafted example is provided in the Appendix. It can be formulated as an undiscounted, reward-based, finite-horizon MDP (S, A, app, t, r):

- S is the state space of the MDP. In our case, states correspond to (incomplete or fully-generated) planning problems, represented by a two-element tuple $s = (s_{ic}, s_{gc})$, where the first element is the initial state and the second element is the goal state. We use the subindex c (*current*) to denote when the initial state s_{ic} and goal state s_{gc} may not be completely generated yet. During the initial state generation phase, we assume s_{gc} is the empty state, represented by the "_" symbol.

- A is the action space, while $App : S \times A \to \{0, 1\}$ is the applicability function that determines if an action can be executed at a state or not. The set of applicable actions $App(s)$ of a state s is different for the initial state and goal generation phases. In the initial state generation phase, $App(s)$ corresponds to adding a new atom to the initial state s_{ic} which preserves the continuous consistency constraints (see Section VI.3.1.1). The objects this new atom is instantiated on can already be present in s_{ic} or not. If they are not, we refer to them as *virtual* objects, and are added to s_{ic} alongside their corresponding atom. For example, if the applicable action *add* $ontable(b1)$ is selected and the object $b1$ does not exist in s_{ic}, then both the atom $ontable(b1)$ and the object $b1$ will be added to s_{ic}. Thus, instantiating atoms on virtual objects is the mechanism we use to add new objects to the problem. In the goal generation phase, $App(s)$ is the subset of actions in the planning domain for which their preconditions are met at the current goal state s_{gc}. Additionally, we add a *termination action end* to $App(s)$. When *end* is applied during the initial state generation phase, $s_i = s_{ic}$ is fixed and, if s_i is eventual-consistent, the goal generation phase starts from $s_{gc} = s_i$. Otherwise, the MDP episode concludes. When *end* is applied during the goal generation phase, $s_g = s_{gc}$ is fixed and the goal g is obtained from s_g, so the problem (s_i, g) is returned and the episode concludes. In order to control problem size, we set a maximum number of actions for each generation phase so, if this number is reached, *end* is executed and the corresponding phase concludes.

- $t : S \times A \to S$ is the transition function. In our setting, t is deterministic and returns the next MDP state (i.e., problem) resulting from executing an applicable action at the current state. At the initial state generation phase, executing an action $a \in App(s)$ (different to *end*) at the current MDP state $s = (s_{ic}, \_)$ results in the state $s' = (s'_{ic}, \_)$, where s'_{ic} contains all the objects and atoms of s_{ic} in addition to the atom associated with a and the virtual objects instantiated on it (if any). At the goal generation phase, applying an action $a \in App(s)$ (different to *end*) at the current MDP state $s = (s_i, s_{gc})$ results in the state $s' = (s_i, s'_{gc})$, where s'_{gc} is obtained by applying the (positive and negative) effects of a to s_{gc}.

- $r : S \times A \rightarrow \mathbb{R}$ is the reward function. In our setting, there are three different reward sub-types accounting for problem consistency, difficulty and diversity. At the end of the initial state generation phase, a consistency reward $r_c = -1$ is given if s_i is eventual-inconsistent[3], as a form of penalization. At the end of the goal generation phase, problems receive a difficulty reward r_f equal to the logarithm of their difficulty, and a diversity reward r_v equal to their diversity. In every other situation, r_c, r_f and r_v are all 0. Finally, the (aggregate) reward r is calculated as follows:

$$r = r_c + min\Big(\frac{r_v}{\theta}, 1\Big) \cdot r_f \tag{VI.1}$$

where $\theta \in [0, 1]$ is a hyperparameter known as the *diversity threshold*. We now explain the rationale behind Equation VI.1. MDP trajectories resulting in eventual-inconsistent problems will receive a reward $r = -1$ in their last sample, since r_v and r_f will both be 0. For trajectories resulting in consistent problems, the reward (for the last sample) will be equal to r_f scaled down by a factor $min(r_v/\theta, 1)$, which depends on diversity: if $r_v \geq \theta$, then $r = r_f$ whereas, if $r_v < \theta$, r_f will be scaled down up to a minimum of $r = 0$, in case $r_v = 0$. This reward function balances problem consistency, diversity and difficulty. By maximizing it, we hope NeSIG will learn to generate consistent problems with a diversity close to θ (since diversity values r_v larger than θ do not increase r and values lower than θ reduce r considerably) and as difficult to solve as possible.

VI.3.3 Learning to generate problems with RL

We use two different policies for guiding problem generation. One policy generates the initial state s_i of each problem, whereas the other generates its goal g. Each policy is encoded by a separate NLM.

At each step, the corresponding NLM receives information about the current MDP state. In the case of the initial state policy, it receives a tensor representation of the atoms and objects in the current initial state s_{ic}. This set of objects contains both the actual objects in s_{ic} and the new, virtual objects that can be added to the state alongside the next atom. The set of virtual objects is automatically inferred from the predicate information encoded in the PDDL domain. In the case of the goal policy, the NLM receives as input a concatenation of the tensor representations of the initial state s_i and current goal state s_{gc}. Since no new objects can be added during

[3]The reward function does not need to consider continuous consistency since actions resulting in continuous-inconsistent states are never executed.

the goal generation phase, no virtual objects are used. Additionally, both NLMs receive as extra information the percentage of actions executed in the corresponding phase (relative to the maximum number of actions allowed), for each object its type and whether it is virtual or not, the total number of objects of each type, and the total number of atoms of each predicate type in the initial state and, for the goal policy NLM, also in the goal state.

The output of the NLM is represented as a new set of atoms, where each atom is associated with a different MDP action $a \in A$, corresponding to either a new atom to add to s_{ic} (for the initial state policy) or a domain action to apply to s_{gc} (for the goal policy), in addition to the termination action *end*. The NLM outputs a real value for each atom (action) in this set. Then, we mask out inapplicable actions $a \notin App(s)$, corresponding to either atoms that violate the continuous consistency constraints (for the initial state policy) or domain actions whose preconditions are not met at s_{gc} (for the goal policy). Finally, we apply the softmax function to obtain a probability distribution over applicable actions $a \in App(s)$, from which we sample the action to execute at the current MDP state s.

In order to train the initial state and goal policies, we resort to the DRL algorithm Proximal Policy Optimization (PPO) [209], which is described in Section II.4.2. Since PPO is an actor-critic algorithm, we need to employ an additional critic NLM for each policy, whose sole purpose is to evaluate the current MDP state s, i.e., predict $V(s)$. The two policies are trained simultaneously in an end-to-end fashion. The initial state policy receives rewards accounting for problem consistency, diversity and difficulty. On the other hand, the reward signal the goal policy receives accounts for diversity and difficulty but not consistency, since the consistency of a problem is independent of its goal g and, thus, of the goal policy. In order to calculate the PPO advantages, we use the Generalized Advantage Estimation (GAE) [208] method. However, we found the best λ value to be equal to 1, which is equivalent to simply calculating advantages using the return R obtained in the trajectory (i.e., not using GAE). Moreover, we use a policy entropy bonus as proposed in [209] to encourage sufficient exploration, in addition to the diversity reward.

VI.4 Experiments and analysis of results

In this section, we detail our experimental setup and analyze the results of our experiments, which compare the problems generated by NeSIG with those obtained by handcrafted, domain-specific generators and several ablations. We compare the consistency, diversity, difficulty and generation times of the different approaches, and evaluate whether NeSIG can generalize to larger problems than those encountered during training.

VI.4.1 Experimental setup

We perform experiments on a set of diverse and well-known CP domains: *blocksworld*, *logistics*, *sokoban*, *miconic* and *satellite*. In *blocksworld*, a set of stackable blocks needs to be re-assembled with a gripper. *Logistics* represents a transportation task where a set of packages needs to be delivered across locations and cities using airplanes and trucks. *Sokoban* is a puzzle where boxes must be pushed to designated goal locations. This game is known for its great difficulty, being

PSPACE-complete to solve [46]. In *miconic*, an elevator is used to move passengers between the different floors of a building. Lastly, *satellite* is inspired from NASA's real observatory missions. In this domain, a series of instrument-equipped satellites must be controlled in order to collect various measurements of space phenomena. In *sokoban*, the initial state generation phase starts from a state s_i encoding an empty NxM map with no robots, walls or boxes, which will be added during the generation process. For the rest of domains, the initial state generation state starts from an empty state s_i with no objects or atoms. The PDDL description for each domain can be found in the Appendix.

We train NeSIG separately on each domain, performing 5000 training steps on *blocksworld*, *logistics* and *sokoban* and 10000 steps on *miconic* and *satellite*, as these two domains require longer training. We utilize Adam [131] as our optimizer, with a learning rate of 10^{-3}. Each experiment is run on 25 threads of an AMD EPYC 7742 CPU and one Nvidia A100 GPU, although our method can be trained on consumer-grade GPUs since only a maximum of 8 GBs of VRAM are required. In each training step, we generate a set of problems by executing up to 15 initial state actions (i.e., adding a maximum of 15 atoms to s_i) and up to 60 goal actions in *blocksworld* and *logistics*. For *sokoban*, *miconic* and *satellite*, we execute up to 75 goal actions, as these domains are more challenging than the previous two. Additionally, a map of size 5x5 is used by *sokoban* during training. Every 250 training steps, we perform one validation epoch, where 100 problems are generated and the reward r of each problem is obtained using Equation VI.1. We calculate the *validation score* of the model as the average problem reward and, once training concludes, we load the model checkpoint with the best validation score for testing. The remaining hyperparameter values are provided in the Appendix. We use very similar values for each domain so as to show our method needs little hyperparameter tuning.

Problem difficulty is calculated as the planning effort (i.e., time or number of expanded nodes) of a particular planner to solve the problem, thus being a planner-dependent measure. We employ the planners provided by the FastDownward (FD) suite [100]. During training, problem difficulty is calculated as the number of nodes LAMA-first [195], a fast satisficing planner, needed to expand in order to solve the problem. For each problem, it can use up to 500 MB of memory and 5 minutes of planning time. We assign a difficulty value of 10^6 for problems that could not be solved under those limits, which we will refer to as the *terminated difficulty.* At test time, we use a different set of (satisficing and optimal) planners to assess whether problems generated by NeSIG are challenging for several planners. The set of test planners considered are: LAMA-first, the same planner used during training; the satisficing algorithm lazy greedy-best-first-search (GBFS) with the FF heuristic [108]; A* search with the LM-cut heuristic [102], resulting in an optimal algorithm; and the planning portfolio FastDownward Stone Soup (FDSS) [103], in its optimal version. For every planner except FDSS, we measure difficulty as the number of expanded nodes, as this is a hardware-independent measure. However, as a portfolio, FDSS utilizes several planners internally, assigning a different computational budget to each of them. Therefore, we decided to measure difficulty for FDSS as the total planning time (in seconds) required to find an optimal solution. Every test planner except FDSS uses a time limit of 30 minutes, a memory limit of 8 GB and a terminated difficulty of 10^8. FDSS uses the same memory limit, a time limit of

10 minutes (due to its higher efficiency when compared to A*) and a terminated difficulty that is always equal to its time limit plus one second. Nonetheless, solving blocksworld problems in an optimal manner proved to be very difficult. For this reason, in this domain only, we increase the time limit of A* to one hour and of FDSS to 30 minutes, keeping the other parameters unchanged. Finally, for efficiency purposes, we generate small problems during training and then evaluate the generalization abilities of NeSIG by generating larger problems at test time (see Figure 30).

Several methods are compared to NeSIG in our experiments. First, we employ ablations where either s_i (*random-init* models), s_g (*random-goal* models) or both (*random-both* models) are generated by executing random actions $a \in App(s)$. We note our *random-both* model is equivalent to the method proposed in [75], which also generates s_i and s_g at random. We do not compare with Autoscale [232] since it leverages domain-specific generators to obtain problems of graded difficulty, often by gradually incrementing their size, whereas our goal is instead to maximize problem difficulty given a limit on their size. For this reason, we directly utilize the ad hoc, domain-specific generators (*ad hoc* models) used in the International Planning Competitions (IPCs) [236], choosing their parameter values to maximize problem diversity (the exact values can be found in the Appendix). Nonetheless, the *sokoban* generator allowed for little flexibility (e.g., problems of size 5x5 could not have more than two boxes), so we have implemented our own based on a trial and error strategy which obtains s_i by placing objects at random on the grid, randomly moves boxes to obtain g, makes sure g can be achieved from s_i and, otherwise, discards the problem and starts again. We note that the previous, IPC *sokoban* generator also followed a random procedure for placing objects on the grid. Our *sokoban* generator simply attains more flexibility at the expense of greater generation times (due to its trial and error strategy). Finally, in order to perform a fair comparison with NeSIG, we discard *blocksworld*, *logistics*, *miconic* and *satellite* problems with a size smaller than $D - 2$, where D is the maximum problem size, measured as the maximum number of atoms allowed in the initial state of the problems generated by NeSIG. We do so because small problems tend to be easier to solve. In *sokoban*, we instead discard problems where less than 25% of cells are empty (i.e., without walls or boxes) since, otherwise, the problem is very likely to be unsolvable. We note that the time spent discarding problems of incorrect size (or incorrect number of empty cells in *sokoban*) is not considered when measuring generation times.

VI.4.2 Results and discussion

Table 5 compares the problems generated by NeSIG, its ablations and the domain-specific generators (*ad hoc* models) using the same problem size for training and testing. Results show that NeSIG is able to generate consistent, diverse and difficult problems for all the domains considered. Firstly, NeSIG successfully learns to generate problems according to the user-defined consistency rules, thus achieving almost perfect consistency in every domain: 98.6% in *blocksworld*, 99.8% in *logistics*, 100% in *sokoban*, 99.6% in *miconic* and 99% in *satellite*. Regarding difficulty, NeSIG is the model with the highest difficulty value for each domain and planner combination. It generates problems that are significantly more difficult, on geometric average across the four planners, than those from domain-specific generators: 9.28 times in

Property	Blocksworld				
	NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.986±.015	.14±.022	.986±.012	.14±.022	1.0±0
Diversity	.025±.002	.026±.014	.033±.006	.026±.014	.024±0
Time	26±4	11±2	22±1	9±0	4±0
LAMA	394±46	101±33	35±4	30±2	80±0
GBFS	301±49	87±24	34±5	27±3	81±0
A*	$5.5e6$±$3.4e6$	7634±7583	328±94	114±111	$8.2e4$±0
FDSS	127±80	.528±.242	.299±.017	.277±.015	21±0

Property	Logistics				
	NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.998±.004	.254±.047	.994±.008	.254±.047	1.0±0
Diversity	.196±.011	.222±.009	.167±.003	.25±.007	.264±0
Time	28±3	14±1	26±1	13±1	4±0
LAMA	81±4	15±3	12±1	5±1	17±0
GBFS	70±5	13±2	12±1	5±1	16±0
A*	317±121	53±26	11±2	5±1	104±0
FDSS	1±.121	.218±.022	.212±.004	.164±.013	.232±0

Property	Sokoban				
	NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	1.0±.0	.994±.005	.998±.004	.994±.005	1.0±0
Diversity	.016±.001	.007±.0	.013±.001	.007±.0	.016±0
Time	221±17	327±8	255±36	333±4	1019±0
LAMA	$3.2e5$±$1.2e5$	5±1	$2.5e4$±$2.9e4$	6±1	1483±0
GBFS	$2.7e5$±$9.1e4$	5±0	$2.1e4$±$1.9e4$	6±1	1305±0
A*	1105±900	6±1	787±460	6±1	683±0
FDSS	.303±.013	.239±.006	.292±.026	.245±.006	.296±0

Table 5: **Same test-size experiment results.** The table compares the problems generated by NeSIG, several ablations (*random-init*, *random-goal* and *random-both* models) and the domain-specific generator (*ad hoc* model) in the *blocksworld*, *logistics*, *sokoban*, *miconic* and *satellite* domains. For each domain and model, we generate 100 test problems with the same maximum number of initial state and goal actions used for training, i.e., we generate problems of the same size as during training. In *sokoban*, we also use a map size of 5x5. We evaluate the consistency, difficulty, diversity and generation time of the test problems generated, showing for each property its mean value and standard deviation (±) across 5 random seeds. Since the ad hoc models do not require training, we use a single initial random seed (which will be used to deterministically obtain the seed to generate each problem), which is why their std values are always 0. Consistency is measured as the percentage of problems that meet the eventual consistency rules. Diversity is measured as the average pairwise distance among problems, according to the method detailed in Section VI.3.1.2. We break down the difficulty values for several planners. For the LAMA-first (*LAMA*), lazy greedy-best-first-search with the FF heuristic (*GBFS*) and A* with the LM-cut heuristic (*A**), difficulty is measured as the number of expanded nodes. For the FastDownward Stone Soup (*FDSS*) planner, it is measured as the total planning time in seconds. Time refers to the total generation time (in seconds) needed to generate the whole set of 100 test problems. Finally, when calculating the mean diversity and difficulty for each planner, we do not consider inconsistent problems.

blocksworld, 4.07 times in *logistics*, 16.49 times in *sokoban*, 4.01 times in *miconic*, and 5.68 times in *satellite*, for an overall (geometric) average of 6.8 times more difficulty across all domains and planners. These are remarkable results, considering the fact that two of the planners tested (A* and FDSS) are optimal, whereas NeSIG has only been trained to maximize the difficulty of a satisficing planner (LAMA-first). Therefore, NeSIG is able to generate problems that are challenging not only for the planner utilized during training, but for a range of different planners, thus success-

Property	Miconic				
	NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.996±.009	.028±.015	.995±.007	.022±.018	1.0±0
Diversity	.211±.014	.13±.03	.186±.009	.051±.044	.202±0
Time	26±3	6±0	25±1	7±0	6±0
LAMA	124±20	19±6	29±1	28±17	66±0
GBFS	184±78	19±8	28±2	27±16	64±0
A*	$4.4e5$±$3.3e5$	51±64	222±120	886±1845	$4.6e4$±0
FDSS	5±5	.194±.008	.211±.002	.239±.2	1±0

Property	Satellite				
	NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.99±.01	.003±.005	1.0±.0	.008±.008	1.0±0
Diversity	.11±.007	.0±.0	.112±.005	.041±.093	.179±0
Time	29±3	6±0	29±0	7±0	8±0
LAMA	120±12	.25±.5	22±2	5±4	38±0
GBFS	95±8	.25±.5	20±2	4±4	37±0
A*	4687±1891	.25±.5	32±13	5±4	303±0
FDSS	5±9	0±0	.177±.001	.107±.098	.604±0

Table 5: **Continuation of Table 5.**

fully achieving *planner-wise generalization*. At the same time, NeSIG attains almost the same diversity as the domain-specific generators, with only 13% less diversity on geometric average across domains. This means that NeSIG does not need to sacrifice (much) diversity in order to generate hard problems, e.g., by learning to only generate a particular type of problem, thus effectively balancing difficulty and diversity, which is an essential requirement of our method. Moreover, by leveraging parallel GPU computation, we can obtain 100 problems with NeSIG in under half a minute in *blocksworld*, *logistics*, *miconic* and *satellite*, and in less than four minutes in *sokoban*.

We now turn our attention to the ablation models. It can be observed that using a random policy for initial state generation (*random-init* and *random-both* models) severely degrades consistency in all domains but *sokoban*, due to the simplicity of the eventual consistency rules of this domain. An extreme instance of this effect can be observed in *satellite*, where NeSIG obtains a consistency percentage of 99% but the *random-init* and *random-both* models obtain values of 0.3% and 0.8%, respectively. These results show how unlikely it is for random generation to achieve consistency, so it is necessary to train an initial state policy in order to reliably generate consistent problems. Additionally, ablations also significantly impair problem difficulty, although the effect of each policy ablation depends on the particular domain considered. In *blocksworld*, it is more important to train the goal policy than the initial state policy, since the *random-init* model achieves better difficulty than the *random-goal* one. In *sokoban*, *miconic* and *satellite*, the opposite case happens, as *random-goal* achieves better difficulty than *random-init*. Finally, in *logistics* the two policies seem to be equally important, as both ablations attain similar difficulty. Regardless of the relative importance of each policy, NeSIG obtains much higher difficulty than all ablations in every domain, showing that the two policies always play a role in the generation of difficult problems.

The *random-both* model represents the full ablation where no policy is trained, thus obtaining the worst results among all models. However, an important advantage of this model over NeSIG and the other ablations is that it does not require

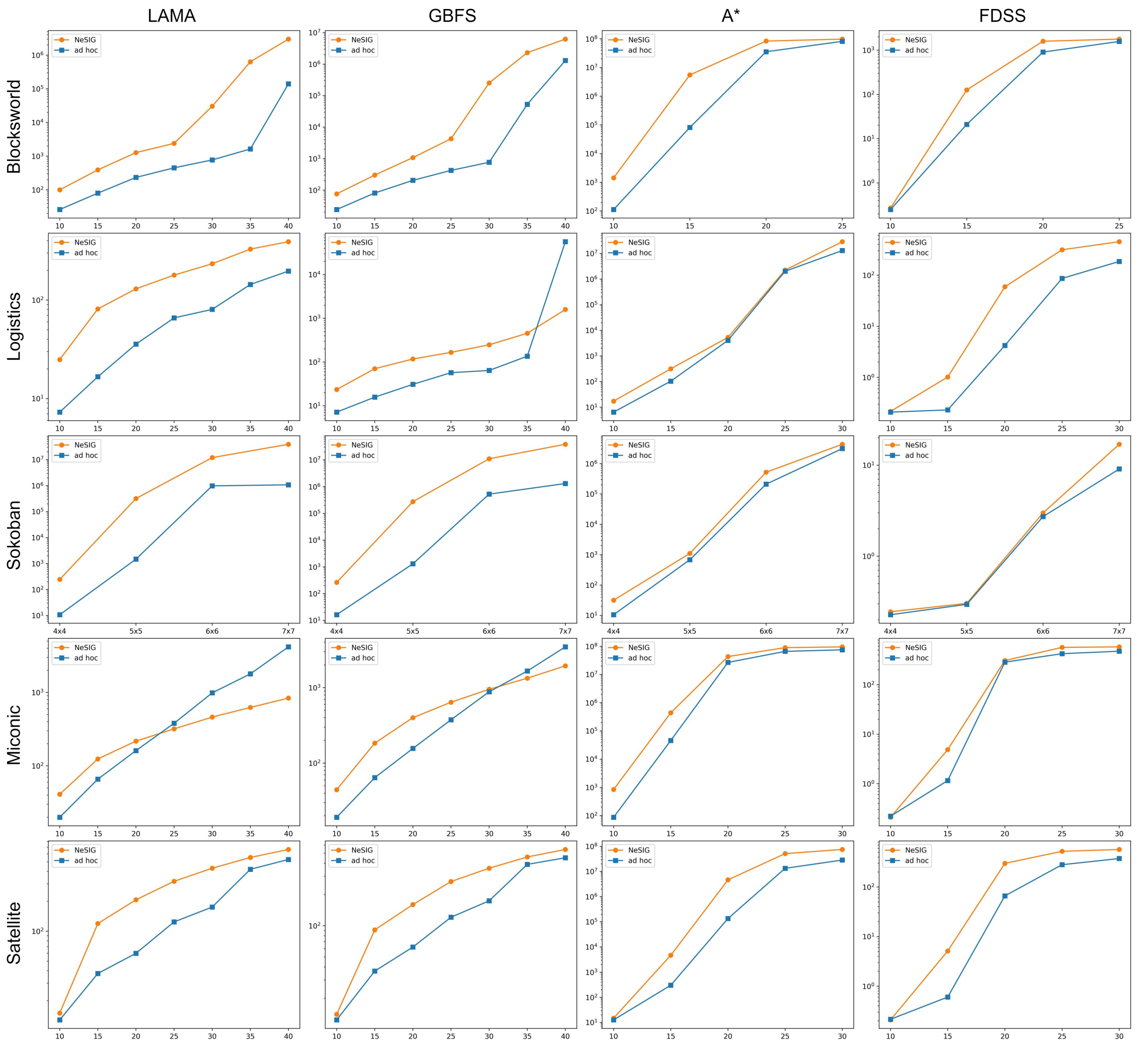


Figure 30: **Problem size generalization results.** The plots show the mean difficulty obtained by NeSIG across five different seeds, when tested on larger (and smaller) problems than those seen during training. We also plot the difficulty of the domain-specific generators (*ad hoc* models) for comparison purposes. Each row in the figure is associated with a different domain, whereas each column is associated with a different planner. The Y axis of each plot measures problem difficulty (in log scale), corresponding to the number of expanded nodes for LAMA-first, GBFS and A*, and to the total planning time (in seconds) for FDSS. The X axis of each plot measures problem size, corresponding to the map size for *sokoban* and to the maximum number of initial state actions (atoms) for the rest of domains. Due to their optimal nature, we could only test a subset of problem sizes for A* and FDSS in *blocksworld*, *logistics*, *miconic* and *satellite*. For example, note how mean difficulty for A* in *blocksworld* saturates to the maximum value of 10^8 for large sizes, as most problems could not be solved under the allotted time and memory. The Appendix details the maximum number of initial state and goal actions used by NeSIG for each problem size, along with the parameters of the *ad hoc* models.

any type of training so, as long as consistency rules are provided (which can be done easily using our proposed consistency language), it can be quickly applied to generate problems for any (typed-STRIPS) planning domain. Although problems generated with this approach are easier to solve than those from the *ad hoc* models, problem difficulty can often be easily raised by incrementing problem size, just as Autoscale and *ad hoc* models do. Therefore, for cases where increasing problem size is acceptable or problem difficulty is not a concern, the *random-both* model offers a general and low-effort alternative to domain-specific generators, serving as a side contribution of our proposal.

Figure 30 shows the difficulty obtained by NeSIG when tested on problems of different size than those used during training. We also plot the difficulty of domain-specific generators for comparison purposes. Out of the 20 domain-planner pairs (each one corresponding to a different plot in the figure), in 15 of them NeSIG exceeds the difficulty of the domain-specific generator for every problem size tested. In *blocksworld*, NeSIG considerably outperforms the *ad hoc* model in terms of difficulty for all four planners and most problem sizes. We note that, due to the logarithmic Y axis employed in the plots, difficulty gaps are actually much larger than they appear in the figure. For the optimal planners (A* and FDSS), difficulty already saturates (i.e., approximates its maximum value) for problems of size 20, as most problems could not be solved under the resource limits. In *logistics*, NeSIG beats the *ad hoc* model by a great margin for LAMA-first and FDSS, by a smaller margin for A* and, for GBFS, it greatly outperforms the domain-specific generator for all sizes except the largest one, corresponding to 40 atoms. In *sokoban*, NeSIG beats *ad hoc* by a large gap for the satisficing planners (LAMA and GBFS) and by a smaller gap for the optimal ones (A* and FDSS). In *miconic*, NeSIG outperforms *ad hoc* for the optimal planners but it is only able to do so for problems up to size 20 and 30 for LAMA-first and GBFS, respectively. Finally, in *satellite*, NeSIG far exceeds the difficulty of *ad hoc* for all planners and almost every size. In light of these impressive results, we conclude that NeSIG successfully achieves *size-wise generalization*, being able to generalize to problems more than twice the size of those used during training.

To summarize, NeSIG is able to reliably generate consistent problems that exhibit great difficulty and diversity. Additionally, it can generalize to both problem sizes and planners different from those used during training. These are remarkable results, especially taking into consideration that our method is domain-independent, whereas *ad hoc* models have been tailored to each particular domain and leverage extensive domain knowledge. For example, the *blocksworld* generator uses an ad hoc formula to make sure that every consistent state has the same probability of being generated. As another example, the *logistics* generator obtains the goal by randomly shuffling the packages in the initial state, knowing in advance that such a goal will always be achievable. When compared to ad hoc generators, NeSIG requires little prior knowledge, as it only receives as inputs the maximum problem size, the types and predicates that can appear in goals, and the set of properties (consistency constraints) initial states must satisfy. Moreover, with our proposed semi-declarative language, these consistency constraints can be easily and intuitively encoded (see the Appendix for concrete examples), thus reducing human effort even further.

VI.5 Conclusion

In this chapter we introduced NeSIG, to the best of our knowledge the first domain-independent method for the automated generation of planning problems that are simultaneously valid, diverse and difficult to solve. We formulated problem generation as an MDP, training two policies with Deep RL to generate problems with the desired properties. Both policies were encoded by NLMs, a neuro-symbolic deep neural network architecture capable of working with FOL data.

A remarkable feature of our method is that it does not require a training dataset of example problems. Instead, it only receives as inputs the PDDL domain description and a set of consistency constraints generated problems must satisfy, along with some extra information (maximum problem size and the types and predicates that are allowed in goals). Therefore, NeSIG requires less prior knowledge than hand-crafted, domain-specific generators such as those often used in the IPCs. Moreover, we proposed a semi-declarative language for encoding consistency constraints in an intuitive and interpretable manner, thus reducing human effort even further.

We tested NeSIG on five classical domains, comparing our approach against domain-specific generators and several ablations. Results show NeSIG successfully generates valid problems which are almost as diverse as those from domain-specific generators but considerably more difficult (6.8 times more on geometric average). Additionally, it showcases impressive generalization abilities both size-wise and planner-wise since, out of the 20 domain-planner combinations tested, in 15 of them NeSIG exceeds the difficulty of the domain-specific generator for all problem sizes considered, including problems more than twice the size of those seen during training. In light of the results obtained, we believe our method establishes a new state of the art in planning problem generation and hope it will prove useful to the Automated Planning community.

Part IV

Final Remarks

Chapter VII

Final Remarks

"Donde una puerta se cierra, otra se abre" — Miguel de Cervantes, Don Quijote de la Mancha

VII.1 Conclusions

This PhD dissertation has presented four significant contributions to the field of SDM via the study and implementation of neuro-symbolic AI methods. One of these contributions is theoretical in nature, in the form of a comprehensive review of SDM that serves to justify the need for neuro-symbolic approaches. The remaining contributions are empirical, and cover the two main categories of the taxonomy proposed in this review: methods for solving MDPs and methods for learning the MDP structure. More specifically, the empirical contributions of this thesis comprise neuro-symbolic methods for improving the performance of SP algorithms (either through goal selection or heuristic learning) and automatically generating SP problems which, among many other applications, can be used as training data for the two previous MDP-solving methods.

Firstly, as our theoretical contribution, we provided a broad review of SDM covering methods for both solving MDPs and learning their structure, emphasizing the knowledge representation of each approach: symbolic, subsymbolic or hybrid. To the best of our knowledge, no other work in the literature offers such a comprehensive overview of the field. Additionally, we also discussed what properties an ideal method for SDM should exhibit, and used these properties to analyze the advantages and disadvantages of the different MDP-solving approaches covered in our review. As a result of our analysis, we argued that an ideal method for SDM should integrate the AP paradigm, in which an action model is used to synthesize a solution of the MDP, with the RL paradigm, in which MDP solutions are learned from data. Furthermore, such ideal method should utilize a hybrid knowledge representation, combining the symbolic and subsymbolic paradigms. Since neuro-symbolic AI is the current approach that most closely performs this integration, i.e., the integration of the AP and RL paradigms with the symbolic and subsymbolic knowledge representations, we concluded that it poses a very promising line of work towards achieving an ideal method for SDM. Therefore, our review serves to justify the relevance and significance of this doctoral dissertation.

Secondly, as our first MDP-solving method, we proposed a neuro-symbolic approach for improving the efficiency of SP algorithms in real-time scenarios through goal selection. Our proposal, called Deep Q-Planning (DQP), integrates the DRL algorithm Deep Q-Learning with the symbolic, classical planner FastForward. At each step, Deep Q-Learning is used to select the next subgoal to achieve, whereas the FastForward planner is in charge of finding a plan that attains the chosen subgoal from the current state. By interleaving this high-level, DRL-based, goal selection strategy with the low-level, SP-based, goal attainment procedure, DQP is able to exploit the existing synergy between AP and RL in order to balance solution quality and time efficiency. We tested our approach on the Boulder Dash game included in the General Video Game AI (GVGAI) environment. When compared to standard Deep Q-Learning, DQP is considerably more sample-efficient (by at least one order of magnitude) and generalizes much better to new game levels. When compared to the standalone FastForward algorithm, DQP drastically reduces problem-solving times at the expense of obtaining plans with only 9% more actions on average.

Thirdly, as our second MDP-solving method, we proposed a neuro-symbolic approach for improving the performance of SP algorithms with heuristic learning. Our proposal entails a statistically-motivated method for leveraging the prior knowledge encoded in symbolic, admissible heuristics in order to learn better heuristics. Our method models the heuristic to be learned as a Truncated Gaussian $\mathcal{TN}$ distribution instead of an (untruncated) Gaussian $\mathcal{N}$. The lower bound of this $\mathcal{TN}$ distribution is set to some admissible heuristic, thus contraining heuristic predictions to be larger than its value. This modelling choice ($\mathcal{TN}$ instead of $\mathcal{N}$) results in a novel loss to minimize during training, different from standard Mean Squared Error (MSE). We compared our proposed loss function to the MSE loss for learning heuristics from optimal costs in a variety of learning scenarios, including four different CP domains: *blocksworld*, *ferry*, *gripper* and *visitall*. Experiment results show our $\mathcal{TN}$-based loss makes training converge faster and overall yields more accurate heuristics that improve planning performance. Specifically, when using the NLM model with our proposed learning configuration learn/h^{FF} (where the model predicts σ alongside μ and employs residual learning), our proposed loss beats MSE in terms of heuristic accuracy in 3 out of 4 domains and improves planning performance in all of them. These results confirm that our $\mathcal{TN}$-based method entails an effective approach for extracting the prior knowledge encoded in admissible heuristics in order to improve heuristic learning.

Fourthly, as our method for learning the MDP structure, we proposed a neuro-symbolic approach for generating valid (i.e., solvable and consistent), diverse and difficult problems for any CP domain. Our proposed method, called NeSIG (Neuro-Symbolic Instance Generator), formulates problem generation as an MDP. The initial state of the problem is generated by sequentially adding atoms and objects to some starting state and, then, the problem goal is obtained by executing a sequence of actions at the generated initial state. Two generative policies, encoded as NLMs, are trained with DRL in order to guide this generative process towards consistent, diverse and difficult problems. Problem diversity is defined as the distance/dissimilarity between problems, whereas difficulty is measured by solving generated problems with an SP algorithm. Conversely, consistency depends on the semantics of the PDDL domain and human preferences, so consistency information must be provided by humans. In order to reduce human effort as much as possible, we implemented a semi-declarative language combining Python and FOL that allows the encoding of consistency constraints in an easy and interpretable manner. We compared our proposed approach to handcrafted, domain-specific generators and several ablations for generating problems in five different CP domains: *blocksworld*, *logistics*, *sokoban*, *miconic* and *satellite*. Experiment results show NeSIG successfully learns to generate valid and diverse problems of much greater difficulty (6.8 times more on geometric average) than domain-specific generators, while reducing human effort when compared to them. We also evaluated the generalization ability of NeSIG by performing a comparison with domain-specific generators across all five domains, different planners (including satisficing and optimal ones) and different problem sizes (up to more than twice the training size). Out of the 20 domain-planner combinations tested, in 15 of them NeSIG exceeds the difficulty of the domain-specific generator for every problem size. Therefore, we conclude that NeSIG also exhibits remarkable generalization capabilities, being able to generalize to both different problem sizes and planners.

In conclusion, the four contributions presented in this thesis have helped advance the field of SDM, both from a theoretical perspective, by providing a broad overview of the different approaches in the field, and an empirical perspective, through the development of neuro-symbolic methods for solving MDPs and learning aspects of their structure. We hope this thesis also serves to highlight the great potential of neuro-symbolic AI to enhance SDM, especially through the integration of AP and RL, and the field of AI altogether.

VII.2 Future works

Neuro-symbolic AI is a fertile approach that has proven immensely useful for solving a wide variety of tasks including SDM, as explained in this dissertation. Therefore, there exist ample future work opportunities to explore the application of neuro-symbolic AI to SDM. In this section, we focus on how the three empirical contributions presented in this thesis could be extended in the future:

- **Goal Selection with Deep Q-Learning.** In future work, we propose to extend the applicability of our DQP architecture, so that it can be used to solve a wider range of tasks. An important extension would be the management of stochasticity, in order to solve non-deterministic tasks. In order to achieve this, we would no longer assume that the abstract, high-level MDP M^g is deterministic, which would need to be considered when estimating the Q-value $Q(s, g)$ associated with a subgoal g. For instance, it could now happen that, while executing the plan to achieve g, an obstacle appears and the plan fails. In such a situation, replanning will be needed unless the Goal Selection module selects a different subgoal to achieve. These situations could be anticipated and avoided by periodically predicting the Q-value $Q(s, g)$ of the subgoal g currently being pursued so that, when $Q(s, g)$ increases too much (e.g., the Goal Selection module believes an obstacle will appear in the future), g can be substituted for a different subgoal. In other words, we propose to use Deep Q-Learning to monitor plan execution and react to discrepancies (i.e., unexpected situations), thus approaching Goal Driven Autonomy [162, 114, 186, 245]. Finally, we could also experiment with alternative DNN architectures in addition to CNNs. For instance, Graph Neural Networks should be well-suited for relational/symbolic tasks, with message-passing computations enabling a high-level, abstract planning procedure for selecting subgoals.

- **Heuristic Learning with Admissible Bounds.** In future work, we propose to extend our Truncated Gaussian $\mathcal{TN}$ approach to other learning settings. An interesting scenario is given by iterative SP algorithms like LAMA, where the cost of the best plan found so far acts as an upper bound of the optimal cost-to-go. Therefore, in this case, we could leverage both lower bounds l (as admissible heuristics) and upper bounds u (as satisficing plan costs) to constrain the support (l, u) of the $\mathcal{TN}$ modelling the learned heuristic and, hopefully, improve heuristic predictions even further. Additionally, we could also apply our method to the RL setting, where state-value $V(s)$ or Q-value $Q(s, a)$ functions are learned instead of heuristics. In order to do so, we would require a method for obtaining a lower or upper bound on the value ($V(s)$ or $Q(s, a)$) being learned, which would be modelled as a $\mathcal{TN}$ distribution, in the same way as a heuristic.

- **Problem Generation with Neuro-Symbolic AI.** Finally, there are many possible avenues for future work on NeSIG, covering both enhancements or extensions to our approach and concrete applications. Regarding the first group, we propose to adapt NeSIG to generate problems according to different user preferences (e.g., maximizing plan length instead of planning difficulty), encoding these preferences in either the reward function or consistency constraints; enhance its expressivity, e.g., by generating PDDL2.1 [72] problems with numeric information (i.e., fluents); and train NeSIG using Generative Flow Nets [16] instead of PPO to better balance problem quality and diversity. Additionally, we could leverage NeSIG for specific use cases. These include: automated curriculum generation, i.e., generating tasks adapted to a particular agent in order to streamline its learning process; adversarial problem generation, i.e., generating problems that are challenging for a particular algorithm, which can provide insight into its weaknesses; and video game level generation.

VII.3 Acknowledgements

This doctoral thesis has been supported by the Andalusian Regional predoctoral grant no. 21-111-PREDOC-0039. Additionally, the research conducted in this dissertation has also received funding from the following sources: Spanish MINECO R&D Project [RTI2018-098460-B-I00]; EU FEDER Funds; Grant PID2022-142976OB-I00, funded by MICIU/AEI/ 10.13039/501100011033, by "ERDF/EU" and by "ESF Investing in your future".

I would like to express my deepest gratitude to Dr. Masataro Asai, for granting me the opportunity of conducting a PhD stay at the MIT-IBM Watson AI Lab, and for his invaluable guidance during the course of this thesis. Additionally, I want to sincerely thank Vladislav Nikolov-Vasilev and Ignacio Vellido-Expósito, for their help in the implementation of the DQP architecture.

Chapter VIII

Bibliography

[1] Abbeel, P. and Ng, A. Y. (2004). Learning first-order markov models for control. In *NeurIPS*, pages 1–8.

[2] Abdel-Hamid, O., Mohamed, A., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE ACM Trans. Audio Speech Lang. Process.*, 22(10):1533–1545.

[3] Acharya, K., Raza, W., Dourado, C., Velasquez, A., and Song, H. H. (2023). Neurosymbolic reinforcement learning and planning: A survey. *IEEE Trans. Artif. Intell.*, 5(5):1939–1953.

[4] Aha, D. W. (2018). Goal reasoning: Foundations, emerging applications, and prospects. *AI Mag.*, 39(2):3–24.

[5] Alkhazraji, Y., Frorath, M., Grützner, M., Helmert, M., Liebetraut, T., Mattmüller, R., Ortlieb, M., Seipp, J., Springenberg, T., Stahl, P., and Wülfing, J. (2020). Pyperplan. Zenodo.

[6] Arora, A., Fiorino, H., Pellier, D., Métivier, M., and Pesty, S. (2018). A review of learning planning action models. *Knowl. Eng. Rev.*, 33:e20.

[7] Arora, S. and Doshi, P. (2021). A survey of inverse reinforcement learning: Challenges, methods and progress. *Artif. Intell.*, 297:103500.

[8] Asadi, K., Cater, E., Misra, D., and Littman, M. L. (2018). Towards a simple approach to multi-step model-based reinforcement learning. *arXiv preprint arXiv:1811.00128*.

[9] Asai, M., Kajino, H., Fukunaga, A., and Muise, C. (2022). Classical planning in deep latent space. *J. Artif. Intell. Res.*, 74:1599–1686.

[10] Balduccini, M. (2011). Learning and using domain-specific heuristics in asp solvers. *AI Commun.*, 24(2):147–164.

[11] Barceló, P., Kostylev, E. V., Monet, M., Pérez, J., Reutter, J. L., and Silva, J. P. (2020). The logical expressiveness of graph neural networks. In *ICLR*.

[12] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R.,

Gülçehre, Ç., Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M. M., Vinyals, O., Li, Y., and Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261.*

[13] Battaglia, P. W., Pascanu, R., Lai, M., Rezende, D. J., and Kavukcuoglu, K. (2016). Interaction networks for learning about objects, relations and physics. In *NeurIPS*, pages 4502–4510.

[14] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, 47:253–279.

[15] Bellman, R. (1957). *Dynamic Programming.* Princeton Univ. Pr.

[16] Bengio, Y., Lahlou, S., Deleu, T., Hu, E. J., Tiwari, M., and Bengio, E. (2023). Gflownet foundations. *J. Mach. Learn. Res.*, 24(1):10006–10060.

[17] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.*, 5(2):157–166.

[18] Bertsekas, D. (2019). *Reinforcement learning and optimal control.* Athena Scientific.

[19] Betz, C. and Helmert, M. (2009). Planning with h^+ in theory and practice. In *KI 2009: Advances in Artificial Intelligence*, volume 5803, pages 9–16.

[20] Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning.* Information science and statistics. Springer.

[21] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artif. Intell.*, 90:281–300.

[22] Bonanno, D., Roberts, M., Smith, L., and Aha, D. W. (2016). Selecting subgoals using deep learning in minecraft: A preliminary report. In *IJCAI Workshop on Deep Learning for Artificial Intelligence.*

[23] Bond-Taylor, S., Leach, A., Long, Y., and Willcocks, C. G. (2021). Deep generative modelling: A comparative review of vaes, gans, normalizing flows, energy-based and autoregressive models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(11):7327–7347.

[24] Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *ECP*, pages 360–372.

[25] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33.

[26] Bonet, B. and Geffner, H. (2003). Labeled rtdp: Improving the convergence of real-time dynamic programming. In *ICAPS*, volume 3, pages 12–21.

[27] Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res.*, 24:581–621.

[28] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In *NeurIPS*, volume 33, pages 1877–1901.

[29] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43.

[30] Bundy, A. and Wallen, L. (1984). Breadth-first search. *Cat. Artif. Intell. Tools*, pages 13–13.

[31] Cappart, Q., Chételat, D., Khalil, E. B., Lodi, A., Morris, C., and Velickovic, P. (2021). Combinatorial optimization and reasoning with graph neural networks. In *IJCAI*, pages 4348–4355.

[32] Castillo, L. A., Fernández-Olivares, J., Garcia-Perez, O., and Palao, F. (2006). Efficiently handling temporal knowledge in an htn planner. In *ICAPS*, pages 63–72.

[33] Cazenave, T. (2006). Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *2006 IEEE Symp. Comp. Intell. Games*, pages 27–33.

[34] Chakraborti, T., Sreedharan, S., and Kambhampati, S. (2020). The emerging landscape of explainable automated planning & decision making. In *IJCAI*, pages 4803–4811.

[35] Chang, M., Ullman, T. D., Torralba, A., and Tenenbaum, J. B. (2017). A compositional object-based approach to learning physical dynamics. In *ICLR*.

[36] Charpentier, A., Elie, R., and Remlinger, C. (2021). Reinforcement learning in economics and finance. *Comput. Econ.*, pages 1–38.

[37] Chen, K., Srikanth, N. S., Kent, D., Ravichandar, H., and Chernova, S. (2021). Learning hierarchical task networks with preferences from unannotated demonstrations. In *CoRL*, pages 1572–1581.

[38] Cheng, C., Kolobov, A., and Swaminathan, A. (2021). Heuristic-guided reinforcement learning. In *NeurIPS*, pages 13550–13563.

[39] Chiappa, S., Racanière, S., Wierstra, D., and Mohamed, S. (2017). Recurrent environment simulators. In *ICLR*.

[40] Choi, J. and Kim, K. (2011). MAP inference for bayesian inverse reinforcement learning. In *NeurIPS*, pages 1989–1997.

[41] Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, pages 183–188.

[42] Cohen, E. and Beck, J. C. (2017). Problem difficulty and the phase transition in heuristic search. In *AAAI*, pages 780–786.

[43] Coles, A. I. and Smith, A. J. (2007). Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res.*, 28:119–156.

[44] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, 20(3):273–297.

[45] Cox, D. R. (1958). The regression analysis of binary sequences. *J. R. Stat. Soc. Ser. B Methodol.*, 20(2):215–232.

[46] Culberson, J. C. (1997). Sokoban is PSPACE-complete. Technical Report TR 97-02, Department of Computing Science, University of Alberta.

[47] d'Avila Garcez, A. and Lamb, L. C. (2023). Neurosymbolic AI: the 3rd wave. *Artif. Intell. Rev.*, 56(11):12387–12406.

[48] Deisenroth, M. and Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. In *ICML*, pages 465–472.

[49] Deng, J., Berg, A., Satheesh, S., Su, H., Khosla, A., and Li, F. (2012). Large scale visual recognition challenge 2012. In *ILSVRC 2012 workshop*.

[50] Depeweg, S., Hernández-Lobato, J., Doshi-Velez, F., and Udluft, S. (2017). Learning and policy search in stochastic dynamical systems with bayesian neural networks. In *ICLR*.

[51] Dittadi, A., Bolander, T., and Winther, O. (2018). Learning to plan from raw data in grid-based games. In *GCAI*, volume 55, pages 54–67.

[52] Diuk, C., Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *ICML*, pages 240–247.

[53] Dong, H., Mao, J., Lin, T., Wang, C., Li, L., and Zhou, D. (2019). Neural logic machines. In *ICLR*.

[54] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*.

[55] Dowson, D. and Wragg, A. (1973). Maximum-entropy distributions having prescribed first and second moments (corresp.). *IEEE Trans. Inform. Theory*, 19(5):689–693.

[56] Draper, D., Hanks, S., and Weld, D. S. (1994). Probabilistic planning with information gathering and contingent execution. In *AIPS*, pages 31–36.

[57] Džeroski, S., De Raedt, L., and Driessens, K. (2001). Relational reinforcement learning. *Mach. Learn.*, 43:7–52.

[58] Edelkamp, S. (2001). Planning with pattern databases. In *Proc. ECP*, volume 1, pages 13–24.

[59] Edelkamp, S. (2007). Automated creation of pattern database search heuristics. *Lect. Notes Comput. Sci.*, 4428:35.

[60] Eisen, M., Zhang, C., Chamon, L. F., Lee, D. D., and Ribeiro, A. (2019). Learning optimal resource allocations in wireless systems. *IEEE Trans. Signal Process.*, 67(10):2775–2790.

[61] Ernandes, M. and Gori, M. (2004). Likely-admissible and sub-symbolic heuristics. In *ECAI*, volume 16, pages 613–617.

[62] Farquhar, G., Rockt aschel, T., Igl, M., and Whiteson, S. (2018). Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. In *ICLR*.

[63] Fawcett, C., Helmert, M., Hoos, H., Karpas, E., Röger, G., and Seipp, J. (2011). FD-Autotune: Domain-specific configuration using fast downward. In *ICAPS Workshop on Planning and Learning*, pages 13–17.

[64] Feng, Z. and Hansen, E. A. (2002). Symbolic heuristic search for factored Markov decision processes. In *AAAI/IAAI*, pages 455–460.

[65] Fern, A., Yoon, S. W., and Givan, R. (2004). Learning domain-specific control knowledge from random walks. In *ICAPS*, pages 191–199.

[66] Fernandez-de Cossio-Diaz, J. (2018). Moments of the univariate truncated normal distribution. `https://github.com/cossio/TruncatedNormal.jl/blob/23bfc7d0189ca6857e2e498006bbbed2a8b58be7/notes/normal.pdf`.

[67] Feyzabadi, S. and Carpin, S. (2017). Planning using hierarchical constrained Markov decision processes. *Auton. Robots*, 41:1589–1607.

[68] Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972a). Learning and executing generalized robot plans. *Artif. Intell.*, 3:251–288.

[69] Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972b). Learning and executing generalized robot plans. *Artif. Intell.*, 3:251–288.

[70] Forestier, S., Portelas, R., Mollard, Y., and Oudeyer, P.-Y. (2022). Intrinsically motivated goal exploration processes with automatic curriculum learning. *JMLR*, 23(1):6818–6858.

[71] Fox, M. and Long, D. (1998). The automatic inference of state invariants in TIM. *J. Artif. Intell. Res.*, 9:367–421.

[72] Fox, M. and Long, D. (2003). Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124.

[73] Frances, G., Ramírez Jávega, M., Lipovetzky, N., and Geffner, H. (2017). Purely declarative action descriptions are overrated: Classical planning with simulators. In *IJCAI*, pages 4294–4301.

[74] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., and Pineau, J. (2018). An introduction to deep reinforcement learning. *Found. Trends Mach. Learn.*, 11(3-4):219–354.

[75] Fuentetaja, R. and De la Rosa, T. (2012). A planning-based approach for generating planning problems. In *Workshops at AAAI*.

[76] Garcıa, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *J. Mach. Learn. Res.*, 16:1437–1480.

[77] Garnelo, M., Arulkumaran, K., and Shanahan, M. (2016). Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*.

[78] Garnelo, M. and Shanahan, M. (2019). Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Curr. Opin. Behav. Sci.*, 29:17–23.

[79] Gehring, C., Asai, M., Chitnis, R., Silver, T., Kaelbling, L., Sohrabi, S., and Katz, M. (2022a). Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *ICAPS*, volume 32, pages 588–596.

[80] Gehring, C., Asai, M., Chitnis, R., Silver, T., Kaelbling, L., Sohrabi, S., and Katz, M. (2022b). Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *ICAPS*, volume 32, pages 588–596.

[81] Georgievski, I. and Aiello, M. (2015). Htn planning: Overview, comparison, and beyond. *Artif. Intell.*, 222:124–156.

[82] Gerevini, A. and Schubert, L. (1998). Inferring state constraints for domain-independent planning. In *AAAI*, pages 905–912.

[83] Getoor, L. and Taskar, B. (2007). *Introduction to Statistical Relational Learning*. MIT Press.

[84] Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated planning and acting*. Cambridge University Press.

[85] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9, pages 249–256.

[86] Gomoluch, P., Alrajeh, D., Russo, A., and Bucchiarone, A. (2017). Towards learning domain-independent planning heuristics. *arXiv preprint arXiv:1707.06895*.

[87] Grimm, C., Barreto, A., Singh, S., and Silver, D. (2020). The value equivalence principle for model-based reinforcement learning. In *NeurIPS*, volume 33, pages 5541–5552.

[88] Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *ICML*, pages 2829–2838.

[89] Guez, A., Mirza, M., Gregor, K., Kabra, R., Racanière, S., Weber, T., Raposo, D., Santoro, A., Orseau, L., Eccles, T., Wayne, G., Silver, D., and Lillicrap, T. P. (2019). An investigation of model-free planning. In *ICML*, pages 2464–2473.

[90] Guez, A., Weber, T., Antonoglou, I., Simonyan, K., Vinyals, O., Wierstra, D., Munos, R., and Silver, D. (2018). Learning to search with MCTSnets. In *ICML*, pages 1822–1831.

[91] Gunning, D. and Aha, D. (2019). Darpa's explainable artificial intelligence (xai) program. *AI Mag.*, 40:44–58.

[92] Guzmán, C., Alcázar, V., Prior, D., Onaindia, E., Borrajo, D., Fdez-Olivares, J., and Quintero, E. (2012). PELEA: a domain-independent architecture for planning, execution and learning. In *ICAPS*, volume 12, pages 38–45.

[93] Hans, A., Schneegaß, D., Schäfer, A. M., and Udluft, S. (2008). Safe exploration for reinforcement learning. In *ESANN*, pages 143–148.

[94] Hansen, E. A. and Zilberstein, S. (2001). Lao*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, 129(1-2):35–62.

[95] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107.

[96] Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). An introduction to the planning domain definition language. *Synth. Lect. Artif. Intell. Mach. Learn.*, 13:1–187.

[97] Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *AAAI*, pages 29–37.

[98] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CVPR*, pages 770–778.

[99] Heger, M. (1994). Consideration of risk in reinforcement learning. In *Mach. Learn.*, pages 105–111. Elsevier.

[100] Helmert, M. (2006). The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246.

[101] Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artif. Intell.*, 173:503–535.

[102] Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: what's the difference anyway? In *ICAPS*, volume 19, pages 162–169.

[103] Helmert, M., Röger, G., and Karpas, E. (2011). Fast downward stone soup: a baseline for building planner portfolios. In *ICAPS Workshop on Planning and Learning*, volume 2835, page 8.

[104] Hester, T. and Stone, P. (2013). Texplore: real-time sample-efficient reinforcement learning for robots. *Mach. Learn.*, 90:385–429.

[105] Hickling, T., Zenati, A., Aouf, N., and Spencer, P. (2023). Explainability in deep reinforcement learning: A review into current methods and applications. *ACM Comput. Surv.*, 56(5):1–35.

[106] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[107] Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD: stochastic planning using decision diagrams. In *UAI*, pages 279–288.

[108] Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302.

[109] Hoffmann, J., Porteous, J., and Sebastia, L. (2004). Ordered landmarks in planning. *J. Artif. Intell. Res.*, 22:215–278.

[110] Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956.

[111] Höller, D. and Bercher, P. (2021). Landmark generation in htn planning. In *AAAI*, volume 35, pages 11826–11834.

[112] Howey, R., Long, D., and Fox, M. (2004). Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *IEEE Int. Conf. Tools Artif. Intell.*, pages 294–301.

[113] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, volume 37, pages 448–456.

[114] Jaidee, U., Munoz-Avila, H., and Aha, D. W. (2012). Learning and reusing goal-specific policies for goal-driven autonomy. In *ICCBR*, pages 182–195.

[115] Jaynes, E. T. (1957). Information theory and statistical mechanics. *Phys. Rev.*, 106(4):620–630.

[116] Jia, W., Sun, M., Lian, J., and Hou, S. (2022). Feature dimensionality reduction: a review. *Complex Intell. Syst.*, 8(3):2663–2693.

[117] Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., and Borrajo, D. (2012). A review of machine learning for automated planning. *Knowl. Eng. Rev.*, 27(4):433–467.

[118] Jiménez, S., Fernández, F., and Borrajo, D. (2008). The PELA architecture: integrating planning and learning to improve execution. In *AAAI*. AAAI Press.

[119] Jiménez, S., Segovia-Aguas, J., and Jonsson, A. (2019). A review of generalized planning. *Knowl. Eng. Rev.*, 34:e5.

[120] Jin, M., Ma, Z., Jin, K., Zhuo, H. H., Chen, C., and Yu, C. (2022). Creativity of ai: Automatic symbolic option discovery for facilitating deep reinforcement learning. In *AAAI*, volume 36, pages 7042–7050.

[121] Kahneman, D. (2011). *Thinking, fast and slow.* Farrar, Straus and Giroux.

[122] Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Mohiuddin, A., Sepassi, R., Tucker, G., and Michalewski, H. (2020). Model based reinforcement learning for atari. In *ICLR*.

[123] Kamran, D., Engelgeh, T., Busch, M., Fischer, J., and Stiller, C. (2021). Minimizing safety interference for safe and comfortable automated driving with distributional reinforcement learning. In *IROS*, pages 1236–1243.

[124] Kansky, K., Silver, T., Mély, D. A., Eldawy, M., Lázaro-Gredilla, M., Lou, X., Dorfman, N., Sidor, S., Phoenix, S., and George, D. (2017). Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In *ICML*, pages 1809–1818.

[125] Karpas, E. and Domshlak, C. (2009). Cost-optimal planning with landmarks. In *IJCAI*, pages 1728–1733.

[126] Katz, M. and Sohrabi, S. (2020). Generating data in planning: SAS planning tasks of a given causal structure. *ICAPS 2020 Workshop on Heuristics and Search for Domain-independent Planning*, page 41.

[127] Katz, M., Srinivas, K., Sohrabi, S., Feblowitz, M., Udrea, O., and Hassanzadeh, O. (2021). Scenario planning in the wild: A neuro-symbolic approach. pages 15–23.

[128] Keyder, E., Hoffmann, J., and Haslum, P. (2014). Improving delete relaxation heuristics through explicitly represented conjunctions. *J. Artif. Intell. Res.*, 50:487–533.

[129] Khansari-Zadeh, S. M. and Billard, A. (2011). Learning stable nonlinear dynamical systems with gaussian mixture models. *IEEE Trans. Robot.*, 27(5):943–957.

[130] Kingma, D. and Welling, M. (2014). Auto-encoding variational bayes international. In *ICLR*.

[131] Kingma, D. P. and Ba, J. (2015). Adam: a method for stochastic optimization. In *ICLR*.

[132] Kipf, T. N., van der Pol, E., and Welling, M. (2020). Contrastive learning of structured world models. In *ICLR*, pages 1–21.

[133] Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274.

[134] Konidaris, G. D. and Barto, A. G. (2007). Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900.

[135] Koppejan, R. and Whiteson, S. (2011). Neuroevolutionary reinforcement learning for generalized control of simulated helicopters. *Evol. Intell.*, 4:219–241.

[136] Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artif. Intell.*, 26:35–77.

[137] Kramer, S. (1996). Structural regression trees. In *AAAI*, pages 812–819.

[138] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *NeurIPS*, pages 1106–1114.

[139] Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). Building machines that learn and think like people. *Behav. Brain Sci.*, 40:e253.

[140] Landajuela, M., Petersen, B. K., Kim, S., Santiago, C. P., Glatt, R., Mundhenk, N., Pettit, J. F., and Faissol, D. (2021). Discovering symbolic policies with deep reinforcement learning. In *ICML*, pages 5979–5989.

[141] Laversanne-Finot, A., Pere, A., and Oudeyer, P.-Y. (2018). Curiosity driven exploration of learned disentangled goal spaces. In *CoRL*, pages 487–504.

[142] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nat.*, 521:436–444.

[143] Li, Y. (2018). Deep reinforcement learning. *arXiv preprint arXiv:1810.06339.*

[144] Liebana, D. P., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couëtoux, A., Lee, J., Lim, C., and Thompson, T. (2016). The 2014 general video game playing competition. *IEEE Trans. Comput. Intell. AI Games*, 8(3):229–243.

[145] Littman, M. L. (1996). *Algorithms for sequential decision-making.* Brown University.

[146] Long, D. and Fox, M. (1999). Efficient implementation of the plan graph in stan. *J. Artif. Intell. Res.*, 10:87–115.

[147] Loshchilov, I. and Hutter, F. (2017). Fixing weight decay regularization in adam. *arXiv preprint arXiv:1711.05101.*

[148] Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Ann. Oper. Res.*, 28(1):47–65.

[149] Lyu, D., Yang, F., Liu, B., and Gustafson, S. (2019). SDRL: interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *AAAI*, volume 33, pages 2970–2977.

[150] Machado, M. C., Bellemare, M. G., and Bowling, M. (2017). A laplacian framework for option discovery in reinforcement learning. In *ICML*, pages 2295–2304.

[151] Magnaguagno, M. C., Fraga Pereira, R., Móre, M. D., and Meneguzzi, F. R. (2017). Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In *ICAPS UISP Workshop.*

[152] Marcus, G. (2018). Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631.*

[153] Marcus, G. (2020). The next decade in AI: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177.*

[154] Marom, O. and Rosman, B. (2020). Utilising uncertainty for efficient learning of likely-admissible heuristics. In *ICAPS*, volume 30, pages 560–568.

[155] McGann, C., Py, F., Rajan, K., Thomas, H., Henthorn, R., and McEwen, R. S. (2008). A deliberative architecture for AUV control. In *ICRA*, pages 1049–1054.

[156] McGovern, A. and Sutton, R. S. (1998). Macro-actions in reinforcement learning: An empirical analysis. *Computer Science Department Faculty Publication Series*, page 15.

[157] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *ICML*, pages 1928–1937.

[158] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[159] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533.

[160] Moerland, T. M., Broekens, J., Plaat, A., and Jonker, C. M. (2023). Model-based reinforcement learning: A survey. *Found. Trends Mach. Learn.*, 16(1):1–118.

[161] Molineaux, M., Floyd, M. W., Dannenhauer, D., and Aha, D. W. (2018). Human-agent teaming as a common problem for goal reasoning. In *AAAI Spring Symposia*.

[162] Molineaux, M., Klenk, M., and Aha, D. (2010). Goal-driven autonomy in a navy strategy simulation. In *AAAI*, volume 24, pages 1548–1554.

[163] Mourao, K., Petrick, R. P., and Steedman, M. (2008). Using kernel perceptrons to learn action effects for planning. In *CogSys*, pages 45–50.

[164] Mukadam, M., Cosgun, A., Nakhaei, A., and Fujimura, K. (2017). Tactical decision making for lane changing with deep reinforcement learning. In *NIPS Workshop on Machine Learning for Intelligent Transportation Systems*.

[165] Murray, A., Arulselvan, A., Cashmore, M., Roper, M., and Frank, J. (2023). A column generation approach to correlated simple temporal networks. In *ICAPS*, volume 33, pages 295–303.

[166] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *ICML*, pages 807–814.

[167] Natarajan, M. and Kolobov, A. (2022). *Planning with Markov decision processes: An AI perspective*. Synth. Lect. Artif. Intell. Mach. Learn. Springer Nature.

[168] Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *J. Artif. Intell. Res.*, 20:379–404.

[169] Nazari, M., Oroojlooy, A., Snyder, L. V., and Takác, M. (2018). Reinforcement learning for solving the vehicle routing problem. In *NeurIPS*, pages 9861–9871.

[170] Nejati, N., Langley, P., and Konik, T. (2006). Learning hierarchical task networks by observation. In *ICML*, pages 665–672.

[171] Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287.

[172] Ng, A. Y. and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *ICML*, volume 1, pages 663–670.

[173] Núñez-Molina, C. (2022a). Application of neurosymbolic AI to sequential decision making. In *IJCAI 2022 Doctoral Consortium*, pages 5863–5864.

[174] Núñez-Molina, C. (2022b). Lifted PDDL. GitHub. https://github.com/AI-Planning/lifted-pddl.

[175] Núñez-Molina, C. (2023). PDDL Prover. GitHub. https://github.com/TheAeryan/PDDL-Prover.

[176] Núñez-Molina, C. and Asai, M. (2023a). Simple NLM. GitHub. https://github.com/TheAeryan/simple-NLM.

[177] Núñez-Molina, C. and Asai, M. (2023b). Stable Truncated Gaussian. GitHub. https://github.com/TheAeryan/stable-truncated-gaussian.

[178] Núñez-Molina, C., Asai, M., Mesejo, P., and Fernandez-Olivares, J. (2024a). On using admissible bounds for learning forward search heuristics. In *IJCAI*, pages 6761–6769.

[179] Núñez-Molina, C., Fernández-Olivares, J., and Pérez, R. (2022). Learning to select goals in automated planning with deep-q learning. *Expert Syst. Appl.*, 202:117265.

[Núñez-Molina et al.] Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. Automated planning instance generation with neuro-symbolic AI. *Artif. Intell.* Submitted on October 3, 2024.

[181] Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. (2024b). NeSIG: A neuro-symbolic method for learning to generate planning problems. In *ECAI*, volume 392, pages 4084–4091.

[182] Núñez-Molina, C., Mesejo, P., and Fernández-Olivares, J. (2024c). A review of symbolic, subsymbolic and hybrid methods for sequential decision making. *ACM Comput. Surv.*, 56(11):1–36.

[183] Núñez-Molina, C., Vellido, I., Nikolov-Vasilev, V., Pérez, R., and Fdez-Olivares, J. (2021). A proposal to integrate deep q-learning with automated planning to improve the performance of a planning-based agent. In *CAEPIA*, pages 23–32.

[184] Oates, T. and Cohen, P. R. (1996). Searching for planning operators with context-dependent and probabilistic effects. In *AAAI*, pages 863–868.

[185] Oh, J., Singh, S., and Lee, H. (2017). Value prediction network. In *NeurIPS*, pages 6118–6128.

[186] Paisner, M., Cox, M., Maynord, M., and Perlis, D. (2014). Goal-driven autonomy for cognitive systems. In *CogSci*, volume 36, pages 2085–2090.

[187] Pascanu, R., Li, Y., Vinyals, O., Heess, N., Buesing, L., Racanière, S., Reichert, D., Weber, T., Wierstra, D., and Battaglia, P. (2017). Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*.

[188] Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *J. Artif. Intell. Res.*, 29:309–352.

[189] Pearl, J. (2014). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.

[190] Pereira, R. F., Oren, N., and Meneguzzi, F. (2020). Landmark-based approaches for goal recognition as planning. *Artif. Intell.*, 279:103217.

[191] Plaat, A., Kosters, W., and Preuss, M. (2023). High-accuracy model-based reinforcement learning, a survey. *Artif. Intell. Rev.*, pages 1–33.

[192] Pozanco, A., Fernández, S., and Borrajo, D. (2018). Learning-driven goal generation. *AI Commun.*, 31(2):137–150.

[193] Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.

[194] Ramırez, M. and Geffner, H. (2009). Plan recognition as planning. In *IJCAI*, pages 1778–1783.

[195] Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177.

[196] Rintanen, J. (2008). Regression for classical and nondeterministic planning. In *ECAI*, pages 568–572. IOS Press.

[197] Rovner, A., Sievers, S., and Helmert, M. (2019). Counterexample-guided abstraction refinement for pattern selection in optimal classical planning. In *ICAPS*, volume 29, pages 362–367.

[198] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nat.*, 323(6088):533–536.

[199] Russell, S. J. and Norvig, P. (2020). *Artificial Intelligence: a Modern Approach (4th Edition)*. Pearson.

[200] Sacerdoti, E. D. (1975). The nonlinear nature of plans. Technical report, Stanford Research Inst. Menlo Park CA.

[201] Safaei, J. and Ghassem-Sani, G. (2007). Incremental learning of planning operators in stochastic domains. In *SOFSEM*, pages 644–655.

[202] Sanner, S. (2010). Relational dynamic influence diagram language (rddl): Language description. Technical Report 32, National ICT Australia Ltd (NICTA), Machine Learning Group. Uncertainty Track, Seventh International Planning Competition.

[203] Saxena, A., Prasad, M., Gupta, A., Bharill, N., Patel, O. P., Tiwari, A., Er, M. J., Ding, W., and Lin, C.-T. (2017). A review of clustering techniques and developments. *Neurocomputing*, 267:664–681.

[204] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80.

[205] Schäpers, B., Niemueller, T., Lakemeyer, G., Gebser, M., and Schaub, T. (2018). Asp-based time-bounded planning for logistics robots. In *ICAPS*, volume 28, pages 509–517.

[206] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *ICLR*.

[207] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. P., and Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nat.*, 588(7839):604–609.

[208] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *ICLR*.

[209] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[210] Segura-Muros, J. Á., Pérez, R., and Fernández-Olivares, J. (2021). Discovering relational and numerical expressions from plan traces for learning action models. *Appl. Intell.*, 51:7973–7989.

[211] Shakya, A. K., Pillai, G., and Chakrabarty, S. (2023). Reinforcement learning algorithms: A brief survey. *Expert Syst. Appl.*, page 120495.

[212] Shen, W., Trevizan, F., and Thiébaux, S. (2020). Learning domain-independent planning heuristics with hypergraph networks. In *ICAPS*, volume 30, pages 574–584.

[213] Shen, W. M. and Simon, H. A. (1989). Rule creation and rule learning through environmental exploration. In *IJCAI*, pages 675–680. Morgan Kaufmann.

[214] Shen, Y., Zhao, N., Xia, M., and Du, X. (2017). A deep q-learning network for ship stowage planning problem. *Pol. Marit. Res.*, 24(s3):102–109.

[215] Sheth, A. P. and Roy, K. (2024). Neurosymbolic value-inspired artificial intelligence (why, what, and how). *IEEE Intell. Syst.*, 39(1):5–11.

[216] Shi, X., Chen, Z., Wang, H., Yeung, D., Wong, W., and Woo, W. (2015). Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In *NeurIPS*, pages 802–810.

[217] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

[218] Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D. P., Rabinowitz, N. C., Barreto, A., and Degris, T. (2017). The predictron: End-to-end learning and planning. In *ICML*, pages 3191–3199.

[219] Singh, S., Jaakkola, T. S., and Jordan, M. I. (1994). Reinforcement learning with soft state aggregation. In *NeurIPS*, pages 361–368.

[220] Sohrabi, S., Riabov, A. V., and Udrea, O. (2016). Plan recognition as planning revisited. In *IJCAI*, pages 3258–3264.

[221] Srinivas, A., Jabri, A., Abbeel, P., Levine, S., and Finn, C. (2018). Universal planning networks: Learning generalizable representations for visuomotor control. In *ICML*, pages 4732–4741.

[222] Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *SARA*, pages 212–223.

[223] Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bull.*, 2(4):160–163.

[224] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[225] Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211.

[226] Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. H. (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *UAI*, pages 528–536.

[227] Tadepalli, P., Givan, R., and Driessens, K. (2004). Relational reinforcement learning: An overview. In *ICML workshop on relational reinforcement learning*, pages 1–9.

[228] Tamar, A., Levine, S., Abbeel, P., Wu, Y., and Thomas, G. (2016). Value iteration networks. In *NeurIPS*, pages 2146–2154.

[229] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.

[230] Tate, A. (1977). Generating project networks. In *IJCAI*, pages 888–893.

[231] Thomaz, A. L. and Breazeal, C. (2006). Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *AAAI*, volume 6, pages 1000–1005.

[232] Torralba, A., Seipp, J., and Sievers, S. (2021). Automatic instance generation for classical planning. In *ICAPS*, volume 31, pages 376–384.

[233] Toyer, S., Trevizan, F., Thiébaux, S., and Xie, L. (2018). Action schema networks: Generalised policies with deep learning. In *AAAI*, volume 32, pages 6294–6301.

[234] Trevizan, F. W. and Veloso, M. M. (2014). Depth-based short-sighted stochastic shortest path problems. *Artif. Intell.*, 216:179–205.

[235] ús Virseda, J., Borrajo, D., and Alcázar, V. (2013). Learning heuristic functions for cost-based planning. *Plan. Learn.*, 4.

[236] Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M., and Sanner, S. (2015). The 2014 international planning competition: Progress and trends. *Ai Mag.*, 36(3):90–98.

[237] van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100.

[238] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NeurIPS*, pages 5998–6008.

[239] Vidal, T. and Ghallab, M. (1996). Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *ECAI*, pages 48–54.

[240] Wahlström, N., Schön, T. B., and Deisenroth, M. P. (2015). From pixels to torques: Policy learning with deep dynamical models. *arXiv preprint arXiv:1502.02251*.

[241] Walsh, T. J. and Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. In *AAAI*, volume 8, pages 714–719.

[242] Wang, W. Y., Li, J., and He, X. (2018). Deep reinforcement learning for nlp. In *ACL: Tutorial Abstracts*, pages 19–21.

[243] Wang, X. (1996). *Learning planning operators by observation and practice.* PhD thesis, Carnegie Mellon University.

[244] Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* PhD thesis, King's College.

[245] Weber, B., Mateas, M., and Jhala, A. (2012). Learning from demonstration for goal-driven autonomy. In *AAAI*, volume 26, pages 1176–1182.

[246] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256.

[247] Xin, L., Song, W., Cao, Z., and Zhang, J. (2021). Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In *NeurIPS*, pages 7472–7483.

[248] Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan examples using weighted max-sat. *Artif. Intell.*, 171(2-3):107–143.

[249] Yoon, S. and Kambhampati, S. (2007). Towards model-lite planning: A proposal for learning & planning with incomplete domain models. In *ICAPS Workshop on Artificial Intelligence Planning and Learning*.

[250] Yoon, S. W., Fern, A., and Givan, R. (2006). Learning heuristic functions from relaxed plans. In *ICAPS*, volume 2, pages 162–171.

[251] Yoon, S. W., Fern, A., and Givan, R. (2007). Ff-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, pages 352–359.

[252] Yoon, S. W., Fern, A., and Givan, R. (2008). Learning control knowledge for forward search planning. *J. Mach. Learn. Res.*, 9:683–718.

[253] Younes, H. L. and Littman, M. L. (2004). Ppddl 1.0: An extension to pddl for expressing planning domains with probabilistic effects. Technical Report CMU-CS-04-167, Carnegie Mellon University.

[254] Yu, C., Zheng, X., Zhuo, H. H., Wan, H., and Luo, W. (2023). Reinforcement learning with knowledge representation and reasoning: A brief survey. *arXiv preprint arXiv:2304.12090*.

[255] Zambaldi, V. F., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D. P., Lillicrap, T. P., Lockhart, E., Shanahan, M., Langston, V., Pascanu, R., Botvinick, M. M., Vinyals, O., and Battaglia, P. W. (2019). Deep reinforcement learning with relational inductive biases. In *ICLR*.

[256] Zheng, Y., Liu, Q., Chen, E., Ge, Y., and Zhao, J. L. (2014). Time series classification using multi-channels deep convolutional neural networks. In *WAIM*, volume 8485, pages 298–310.

[257] Zhuo, H. H. and Yang, Q. (2014). Action-model acquisition for planning via transfer learning. *Artif. Intell.*, 212:80–103.

Part V

Appendix

Appendix A

Heuristic Learning with Admissible Bounds

A.1 Truncated Gaussian implementation

This Appendix explains several important implementation details of the Truncated Gaussian $\mathcal{TN}$ distribution used in the experiments of this chapter. The code for this implementation can be found in a GitHub repository [177].

A.1.1 Numerically stable formulas for Truncated Gaussian

In order to train and use a system that involves a Truncated Gaussian, we need to compute several properties, such as its mean and the log-probability of some value x under the distribution. However, the naive implementation of the formulas for calculating these quantities is numerically unstable due to floating-point rounding errors, especially when μ lies outside the interval (l, u). In this subsection, we briefly explain the source of instability and provide numerically stable formulas for calculating these values.

Given a Truncated Gaussian distribution $\mathcal{TN}(\mathrm{x} \mid \mu, \sigma, l, u)$, its mean $\mathbb{E}[\mathrm{x}]$ is given by the following formula:

$$\mathbb{E}[\mathrm{x}] = \mu + \frac{\phi(\alpha) - \phi(\beta)}{\Phi(\beta) - \Phi(\alpha)}\sigma,$$

$$\text{where } \alpha = \frac{l-\mu}{\sigma}, \quad \beta = \frac{u-\mu}{\sigma} \quad (\beta \geq \alpha)$$

The expression $\frac{\phi(\alpha)-\phi(\beta)}{\Phi(\beta)-\Phi(\alpha)}$ should not be evaluated directly because it involves subtractions between values that could be potentially very close to each other, causing floating-point rounding errors.

We now describe a stable implementation of this formula introduced in [66]. Let us define the following function:

$$F_1(x, y) = \frac{e^{-x^2} - e^{-y^2}}{erf(y) - erf(x)}$$

Then, we reformulate the mean as follows:

$$\mathbb{E}[\mathrm{x}] = \mu + \sqrt{\frac{2}{\pi}} F_1\left(\frac{\alpha}{\sqrt{2}}, \frac{\beta}{\sqrt{2}}\right)\sigma$$

F_1 can be evaluated in a numerically stable manner by using the formulas below:

$$\begin{aligned}
&F_1(x,y) \\
&= F_1(y,x), && \text{if } |x| > |y| \\
&= P_1(x, y-x), && \text{if } |x-y| = |\epsilon| < 10^{-7} \\
&= \frac{1-\Delta}{\Delta erfcx(-y) - erfcx(-x)} && \text{if } x, y \leq 0 \\
&= \frac{1-\Delta}{erfcx(x) - \Delta erfcx(y)} && \text{if } x, y \geq 0 \\
&= \frac{(1-\Delta)e^{-x^2}}{erf(y) - erf(x)} && \text{otherwise}
\end{aligned}$$

where $\Delta = e^{x^2-y^2}$, $erfcx(x) = e^{x^2} erfc(x) = e^{x^2}(1 - erf(x))$ is a function that is commonly available in mathematical packages, and P_1 is a Taylor expansion of $F_1(x, x+\epsilon) = P_1(x, \epsilon)$ where $y = x + \epsilon$:

$$P_1(x,\epsilon) = \sqrt{\pi}x + \frac{1}{2}\sqrt{\pi}\epsilon - \frac{1}{6}\sqrt{\pi}x\epsilon^2 - \frac{1}{12}\sqrt{\pi}\epsilon^3 + \frac{1}{90}\sqrt{\pi}x(x^2+1)\epsilon^4$$

Next, we provide a numerically stable method for computing the log-probability $\log \mathcal{TN}(x \mid \mu, \sigma, l, u)$. Let us assume $l \leq x \leq u$, since otherwise the probability is 0 (whose logarithm is $-\infty$). The value is given by the following expression:

$$\begin{aligned}
&\log \mathcal{TN}(x \mid \mu, \sigma, l, u) = \log\left(\frac{1}{\sigma}\frac{\phi(\xi)}{\Phi(\beta) - \Phi(\alpha)}\right) = \\
&-\log\sigma - \log\sqrt{2\pi} - \frac{\xi^2}{2} - \log\bigl(\Phi(\beta) - \Phi(\alpha)\bigr), \\
&\text{where} \quad \xi = \frac{x-\mu}{\sigma}
\end{aligned}$$

Let $Z = \Phi(\beta) - \Phi(\alpha)$. We obtain $\log(Z)$ from the stable formula for $\mathbb{E}[\mathrm{x}]$. When $\alpha, \beta \geq 0$,

$$\log(Z) = -\log\frac{\mathbb{E}[\mathrm{x}] - \mu}{\sigma} - \log\sqrt{2\pi} - \frac{\alpha^2}{2} + \log\left(1 - e^{\frac{\alpha^2-\beta^2}{2}}\right)$$

When $\alpha, \beta \leq 0$,

$$\log(Z) = -\log\frac{\mu - \mathbb{E}[\mathrm{x}]}{\sigma} - \log\sqrt{2\pi} - \frac{\beta^2}{2} + \log\left(1 - e^{\frac{\beta^2-\alpha^2}{2}}\right)$$

Otherwise,

$$\log(Z) = -\log 2 + \log\left[erf\left(\frac{\beta}{\sqrt{2}}\right) - erf\left(\frac{\alpha}{\sqrt{2}}\right)\right]$$

A.1.2 Truncated Gaussian with missing bounds

A Truncated Gaussian distribution can be defined with either the lower l or upper bound u missing, as $\mathcal{TN}(\mu, \sigma, -\infty, u)$ or $\mathcal{TN}(\mu, \sigma, l, \infty)$, respectively. It can also be defined with no bounds at all as $\mathcal{TN}(\mu, \sigma, -\infty, \infty)$, in which case it is equivalent to an untruncated Gaussian $\mathcal{N}(\mu, \sigma)$.

In our implementation, we use $l = -1e5$ and $u = 1e5$ as the parameters of a Truncated Gaussian with no lower and/or upper bound, respectively. We have observed that these values result indistinguishable from $l \to -\infty$ and $u \to \infty$ when calculating the mean $\mathbb{E}[\mathrm{x}]$ and log-probability $\log p(x)$, as long as $-1e5 \ll \mu \ll 1e5$, $\sigma \ll 1e5$ and $-1e5 < x < 1e5$ (since $p(x) = 0$ for any x outside the interval (l, u)).

A.1.3 Truncated Gaussian with open bounds

When defining a Truncated Gaussian distribution $\mathcal{TN}(\mu, \sigma, l, u)$, we need to specify whether the bounds l, u are contained in the support of the distribution or not, i.e., whether the support is equal to $[l, u]$ (they are contained) or (l, u) (they are *not* contained). When the support is $[l, u]$ we say that the Truncated Gaussian has *closed bounds* and that it has *open bounds* otherwise.

Our first Truncated Gaussian implementation used closed bounds, but we discovered that this decision would sometimes lead to learning issues since the ML model would tend to output $\mu \ll 0$ (e.g., $\mu = -100$). We believe the reason for that behavior is that a highly accurate lower bound l (e.g., h^{LMcut}) can be sometimes equal to h^* and the ML model is encouraged to maximize $\log p(l) = \log p(h^*)$. In order to do so, it can simply output $\mu \ll 0$, as the smaller (more negative) μ gets, the higher $\log p(l)$ becomes. Therefore, using closed bounds would often result in a learned heuristic equivalent to $l = h^{\text{LMcut}}$, as the mean of $\mathcal{TN}(\mu, \sigma, l, u)$ is almost equal to l when $\mu \ll l$.

For this reason, we switched to open bounds in our implementation. To do so, we simply subtracted a small value $\epsilon = 0.1$ from l, obtaining a new distribution $\mathcal{TN}(\mu, \sigma, l - \epsilon, u)$. This made sure that x was never equal to $l' = l - \epsilon$ when calculating $\log p(x = h^*)$, which prevented the ML model from predicting $\mu \ll 0$. Finally, in order to obtain a Truncated Gaussian where the upper bound u is also *open*, we can add ϵ to u, which results in a new distribution $\mathcal{TN}(\mu, \sigma, l - \epsilon, u + \epsilon)$.

A.2 Parameter details

A.2.1 Model hyperparameters

In this Appendix, we detail the hyperparameter values used for the different models: NLM, HGN, and linear regression. In general, we did not perform extensive hyperparameter tuning for the different models.

For the NLM, we used a model with breadth 3 and depth 5, where every inner layer outputs 8 predicates for each arity. The multi-layer perceptrons used in the network employed sigmoid as their activation function and contained no hidden layer. AdamW [147] with weight decay 0.01 helped suppress overfitting.

For the HGN, we employed a hidden size of 32 and 4 recursion steps. We note that using more recursion steps did not improve the performance significantly while being slower. As mentioned in the main paper, the learning rate for the HGN is $1e^{-3}$, which corresponds to the value used in [212].

Finally, we report that we initially tested an L2 weight decay penalty for the linear regression model which was removed later as it did not help.

blocksworld	
train	seed $\in$ 1..38, blocks $\in$ 5..16
val	seed $\in$ 1..11, blocks $\in$ 5..16
test	seed $\in$ 1..11, blocks $\in$ 11..22
ferry	
train	seed $\in$ 1..16, locations $\in$ 2..6, cars $\in$ 2..6
val	seed $\in$ 1..4, locations $\in$ 2..6, cars $\in$ 2..6
test	seed $\in$ 1..16, locations $\in \{10, 15, 20, 25, 30\}$, cars $\in \{10, 15, 20, 25, 30\}$
gripper	
train	seed $\in$ 1..80, balls $\in \{2, 4, 6, 8, 10\}$
val	seed $\in$ 1..20, balls $\in \{2, 4, 6, 8, 10\}$
test	seed $\in$ 1..20, balls $\in \{20, 40, 60, 80, 100\}$
visitall	
train	seed $\in$ 1..70, $x \in$ 3..5, $x = y$, ratio $\in \{0.5, 1.0\}$
val	seed $\in$ 1..17, $x \in$ 3..5, $x = y$, ratio $\in \{0.5, 1.0\}$
test	seed $\in$ 1..17, $x, y \in$ 5..7, ratio $\in \{0.5, 1.0\}$

Table 6: **List of parameters for instance generators.**

A.2.2 Parameters of instance generators

We generated the problems used in our experiments with parameterized generators [63]. Table 6 shows the range of parameter values used for each generator. We note that the *gripper* generator was modified to select the initial states randomly from the entire state space, unlike traditional instances whose initial state specifies that all balls are in the left room.

A.3 Full experimental results for NLM models

In Table 4, we only showed the NLM planning results for our proposed configuration (learn/h^{FF}) and the baseline configuration (fixed/none). Complete results for NLM models are shown in Table 7, which includes the learn/none and fixed/h^{FF} configurations.

Regarding the fixed/h^{FF} configuration, we observe that $\mathcal{TN}$ only obtains better planning results (when both the number of solved instances and evaluated nodes are considered) than $\mathcal{N}$ and $\mathcal{N}$+clip on *visitall* and *gripper*, but not on *ferry* and *blocksworld*. These results are not surprising. After all, as commented in Section V.4.3, this configuration is problematic for $\mathcal{TN}$ since σ affects its heuristic prediction $\mathbb{E}[\mathrm{x}]$ and, therefore, needs to learn σ in order to achieve good performance. This is also why it does not obtain better accuracy than the other approaches (see Table 3).

Regarding the learn/none configuration, $\mathcal{TN}$ is the best model overall: It obtains better planning performance than the alternative approaches on *visitall* and *gripper*, comparable performance with $\mathcal{N}$+clip on *ferry* and is outperformed on *blocksworld*.

		learn/h^{FF}			learn/none		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)							
blocks	.13	.84±.18	.85±.18	**.88±.17**	**.85±.37**	.57±.36	.51±.37
ferry	.82	.91±.00	.91±.01	**.98±.00**	.01±.05	**.60±.12**	.59±.15
gripper	.96	**1**	**1**	**1**	0	.75±.34	**1**
visitall	.86	.97±.07	**.98±.08**	**.98±.06**	.79±.16	**1**	**1**
Average node evaluations (smaller is better)							
blocks	9309	2690±2111	2681±2115	**2060±1823**	**3225±3401**	5754±2825	6492±2695
ferry	5152	3216±532	3117±557	**2477±369**	9952±299	**6751±610**	6780±952
gripper	3918	1642±198	1643±210	**1637±244**	10000±0	4313±1685	**1480±1484**
visitall	3321	2156±1458	2148±1491	**1683±1269**	3860±2393	818±353	**385±616**

	fixed/h^{FF}			fixed/none		
domain	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)						
blocks	**.90±.13**	.89±.13	.87±.12	**.79±.34**	.50±.34	.55±.35
ferry	**.99±.00**	**.99±.00**	.94±.01	.01±.06	.57±.16	**.58±.08**
gripper	**1**	**1**	**1**	.00±.00	.92±.26	**1**
visitall	.98±.03	.99±.03	**1**	.82±.14	**1**	**1**
Average node evaluations (smaller is better)						
blocks	**1791±1678**	1809±1674	2171±1489	**4118±3005**	6268±2649	5903±2835
ferry	2218±255	**2197±255**	3471±482	9933±340	6675±1019	**6475±593**
gripper	1631±65	1635±60	**1301±48**	10000±0	2941±1360	**1709±428**
visitall	1437±1114	1355±1119	**1142±929**	3384±2035	**591±151**	612±160

Table 7: **Planning performance of NLM heuristics (complete results).** For each model, we use the weights that resulted in the best validation MSE loss during training. Table columns and rows have the same meaning as in Table 4. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$+clip and $\mathcal{TN}$ is highlighted in **bold**.

Finally, we note that the best configuration is the one we proposed: learn/h^{FF}.

A.4 Full experimental results for HGN models

In this Appendix, we provide the results of the experiments conducted on the STRIPS-HGN models. The accuracy metrics are shown in Table 8, whereas Table 9 contains planning results.

The training of HGN models was highly unstable even with a reduced learning rate $1e^{-3}$ (compared to $1e^{-2}$ in other models) and the results vary significantly across domains. For example, in *ferry*, HGN models generally failed to learn effective heuristics, as shown in Table 8 where the MSE exceeds 10^7. In contrast, in *visitall*, it generally achieved better accuracy than NLMs.

One potential reason for the failure in *ferry* is the limited expressivity of HGN. Unlike NLM, HGN is designed to only receive the delete-relaxed information about the problem instance, which may harm its ability to learn a heuristic for the original instance. Another potential reason for its weak performance is the training length: due to the reduced learning rate, it may need more training steps in order to converge.

Focusing on the planning results, Table 9 shows improvements of HGN-based heuristics over h^{FF} in *blocksworld* and *visitall*. In the case of *ferry*, results are poor due to the low accuracy of learned heuristics.

Finally, $\mathcal{TN}$ often achieves better accuracy and planning performance than $\mathcal{N}$ and $\mathcal{N}$+clip for equivalent configurations.

				learn/h^{FF}		learn/none		fixed/h^{FF}		fixed/none	
domain	metric	h^{FF}	h^{LMcut}	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$
blocks	MSE	22.8	25.06	2.3±2.6	**1.5±.4**	**2.3±.8**	3.9±2.4	**2.6±2.3**	6.5±10.8	5.2±2.6	**1.8±.6**
	+clip			2.3±2.6		**2.3±.8**		**2.6±2.3**		3.7±1.8	
ferry	MSE	9.77	11.10	(8.9±2)e^8	**(2.3±5)**e^7	**(4.9±7)**e^7	(5.2±9)e^7	(3.0±4)e^5	**(1.7±2)**e^7	(4.2±6)e^6	**(2.7±5)**e^7
	+clip			(8.9±2)e^8		**(4.9±7)**e^7		(2.4±3)e^5		(4.2±6)e^6	
gripper	MSE	9.93	15.82	1.1±.4	1.5±.6	59.2±91.3	**7.5±12.6**	3.7±2.5	4.4±6.0	9.4±12.8	3.0±2.7
	+clip			**1.0±.4**		58.5±90.4		**3.5±2.6**		**2.5±.8**	
visitall	MSE	13.9	36.4	4.0±.4	**3.6±.5**	**.3±.0**	**.3±.0**	4.3±.5	4.1±.4	**.3±.0**	**.3±.1**
	+clip			3.8±.4		**.3±.0**		**4.0±.5**		**.3±.0**	

Table 8: **Test accuracy of HGN heuristics.** Table columns and rows have the same meaning as in Table 3. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$ + *clip*, $\mathcal{TN}$ is highlighted in **bold**.

		learn/h^{FF}			learn/none		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)							
blocks	.13	.70±.30	**.72±.29**	.48±.26	.39±.22	.42±.25	**.57±.35**
ferry	.82	.01±.02	.01±.02	**.02±.06**	.00±.00	.00±.00	**.00±.02**
gripper	.96	.36±.14	.37±.15	**.39±.13**	.24±.20	.24±.20	**.27±.11**
visitall	.86	**.99±.03**	.97±.04	.97±.04	**1**	**1**	**1**
Average node evaluations (smaller is better)							
blocks	9309	3984±2675	**3906±2649**	5844±2088	6636±1859	6456±1969	**5186±2915**
ferry	5152	9916±132	9915±134	**9834±424**	10000±0	10000±0	**9968±105**
gripper	3918	7078±971	7008±1058	**6949±1020**	8050±1490	8048±1492	**7952±564**
visitall	3321	1512±1192	1555±1149	**1472±1182**	204±144	155±24	**146±12**

	fixed/h^{FF}			fixed/none		
domain	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)						
blocks	.57±.28	.62±.32	**.66±.29**	.38±.29	.39±.34	**.62±.35**
ferry	.02±.04	**.06±.12**	.02±.04	.01±.02	**.02±.03**	.01±.03
gripper	.25±.14	.27±.18	**.35±.25**	.19±.06	.38±.28	**.63±.40**
visitall	.98±.03	.99±.03	**1**	**1**	**1**	**1**
Average node evaluations (smaller is better)						
blocks	5235±2262	5007±2439	**4335±2548**	6649±2554	6576±2812	**4581±3096**
ferry	9897±248	**9659±644**	9834±258	9924±169	**9856±250**	9941±175
gripper	8074±801	7917±1138	**7271±1852**	8264±474	7300±1633	**4997±3206**
visitall	1158±955	1063±858	**875±761**	166±41	**152±21**	153±15

Table 9: **Planning performance of HGN heuristics.** Table columns and rows have the same meaning as in Table 4. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$+clip and $\mathcal{TN}$ is highlighted in **bold**.

A.5 Full experimental results for linear regression models

Table 10 shows both the MSE and the NLL metrics obtained by the linear regression (LR) models on the test problems. To evaluate the NLL, we used the best validation NLL loss checkpoints. We show the NLL in Table 10 since all the models (including $\mathcal{N}$) are trained by minimizing the NLL loss, and not the MSE. For the previous experiments, we used the MSE to measure heuristic accuracy, and not the NLL, for two main reasons. First, MSE is easier to understand and interpret than NLL. For example, if a model obtains an MSE of 2 this means that its learned heuristic deviates from h^* by 2 units on average, whereas an NLL of 2 has no straightforward interpretation. Second, the MSE and NLL tend to correlate. However, this latter condition is violated in the LR models, so we decided to show both the NLL and MSE metrics. We observe that $\mathcal{TN}$ outperforms $\mathcal{N}$ according to the NLL metric.

We now detail additional observations. The LR models are remarkably accurate in *gripper* compared to the NLM models despite their simplicity. Residual learning provided no benefit to LR models because they already receive the h^{FF} heuristic as one of their inputs, so using h^{FF} again as the basis for the residual does not provide any additional information. We also observe that learning σ tends to improve the NLL.

Table 11 shows the planning performance of the LR models, using the checkpoints with the best validation MSE. $\mathcal{TN}$ and $\mathcal{N}$ models showed comparable performance, except in fixed σ configurations where $\mathcal{TN}$ models suffered. Due to the simple model architecture (composed of a single linear layer), models with different random seeds converged to the same search behavior, despite their different weight initializations. Indeed, the standard deviations in Table 10 also tend to be significantly smaller than those obtained by other models.

A.6 Experimental results with different lower bounds

Tables 12 and 13 show the ablation study of the NLM models using $h^{\max}$ and h^{blind} instead of h^{LMcut} as the lower bound l, with our proposed (and best) learn/h^{FF} configuration.

We observe that the $\mathcal{N}$+clip models obtain almost identical MSE regardless of the heuristic used for clipping, which makes sense. As previously commented, $\mathcal{N}$+clip is identical to $\mathcal{N}$ except for those cases where the model makes a *really bad* prediction, i.e., when it predicts a cost-to-go that is very far off from the target h^*. If the ML model has been trained correctly, this should seldom occur, meaning that $\mathcal{N}$+clip will be equivalent to $\mathcal{N}$ and, thus, no matter the heuristic (h^{LMcut}, $h^{\max}$ or h^{blind}) used for clipping, the performance will be on average the same.

Regarding the $\mathcal{TN}$ models, we observe that they obtain comparable test MSE on *blocksworld*, *ferry* and *gripper*. However, on *visitall*, the $l = h^{\text{LMcut}}$ configuration clearly outperforms the other two: $l = h^{\text{LMcut}}$ obtains an MSE of 5.3, $l = h^{\max}$ a value of 7.01 and $l = h^{\text{blind}}$ of 7.15. We believe this may be due to the difficulty of

				learn/h^{FF}		learn/none		fixed/h^{FF}		fixed/none	
domain	metric	h^{FF}	h^{LMcut}	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$
blocks	MSE	22.8	25.06	**6.5±.1**	6.7±.4	**6.3±.0**	6.8±.4	7.0±.1	7.8±.2	6.9±.1	7.8±.1
	+clip			**6.5±.1**		**6.3±.0**		**6.9±.1**		**6.8±.1**	
ferry	MSE	9.77	11.10	1.3±.1	3.1±.6	**1.1±.3**	3.0±.6	**1.0±.3**	8.5±.8	**1.0±.2**	8.5±.6
	+clip			**1.2±.1**		**1.1±.3**		**1.0±.3**		**1.0±.2**	
gripper	MSE	9.93	15.82	**.5±.2**	.7±.3	**.6±.2**	.7±.3	**.5±.2**	1.2±.3	**.5±.3**	1.1±.4
	+clip			**.5±.2**		**.6±.2**		**.5±.2**		**.5±.3**	
visitall	MSE	13.9	36.4	6.6±.1	**5.0±.2**	2.0±.0	2.9±.0	6.5±.1	**6.4±.1**	2.0±.0	2.3±.1
	+clip			6.6±.1		**1.9±.0**		6.4±.1		**1.9±.0**	
blocks	NLL	22.8	25.06	2.2±.0	**1.6±.0**	2.1±.0	**1.6±.0**	3.0±.0	**2.8±.0**	3.0±.0	**2.8±.0**
	+clip			2.2±.0		2.1±.0		3.0±.0		3.0±.0	
ferry	NLL	9.77	11.10	1.6±.1	**1.4±.1**	1.5±.1	**1.4±.1**	**1.5±.1**	3.1±.2	**1.5±.1**	3.1±.2
	+clip			1.6±.1		1.5±.1		**1.5±.1**		**1.5±.1**	
gripper	NLL	9.93	15.82	1.0±.1	**.9±.2**	1.4±.2	**.9±.2**	1.4±.1	**1.2±.1**	1.4±.1	**1.1±.1**
	+clip			1.0±.1		1.4±.2		1.4±.1		1.4±.1	
visitall	NLL	13.9	36.4	2.1±.0	**1.8±.0**	**1.5±.0**	**1.5±.1**	2.9±.0	**2.7±.0**	1.8±.0	**1.6±.0**
	+clip			2.1±.0		**1.5±.0**		2.9±.0		1.7±.0	

Table 10: **Test accuracy of LR heuristics.** Table columns and rows have the same meaning as in Table 3. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$ + *clip*, $\mathcal{TN}$ is highlighted in **bold**.

visitall, as this is the domain where the different models obtain highest MSE. Due to its high difficulty, the use of a more informative bound l, i.e., h^{LMcut} instead of $h^{\max}$ or h^{blind}, may make a difference for the $\mathcal{TN}$ models, resulting in more accurate predictions.

A.7 Experimental results with different residuals

Tables 14 and 15 show the ablation study of the NLM models using h^{LMcut} as the residual base. While h^{LMcut} is not theoretically ideal (compared to h^{FF}) as it is a biased estimator that is always smaller than the target h^*, in practice it also worked well as the residual base for heuristic learning.

We observe that, for the $\mathcal{TN}$ model, the h^{FF} residual results in better test MSE than the h^{LMcut} one in every domain except *visitall*. Conversely, we observe that h^{LMcut} residual works better for the $\mathcal{N}$ model, as it obtains better MSE in every domain except *gripper*. Therefore, it seems that the best heuristic to use as a base for the residual (h^{FF} vs h^{LMcut}) depends on the chosen model ($\mathcal{TN}$ or $\mathcal{N}$).

Regardless of the heuristic employed, we observe that the residual-based configurations (learn/h^{FF} and learn/h^{LMcut}) outperform learn/none, thus showing the benefits of using residual learning.

A.8 Planning domain descriptions

In this Appendix, we provide detailed descriptions and PDDL encodings for the four planning domains employed in our experiments: *blocksworld*, *ferry*, *gripper* and *visitall*.

		learn/h^{FF}			learn/none		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)							
blocks	.13	**.20±.00**	.18±.00	.17±.00	**.21±.00**	.20±.00	.19±.00
ferry	.82	**1**	**1**	**1**	**1**	**1**	**1**
gripper	.96	**1**	**1**	**1**	**1**	**1**	**1**
visitall	.86	.91±.00	.91±.00	**.96±.00**	**.92±.00**	.91±.00	**.92±.00**
Average node evaluations (smaller is better)							
blocks	9309	**8770±0**	8920±0	8989±0	**8658±0**	8795±0	8951±0
ferry	5152	1891±0	**1886±0**	1944±0	1894±0	**1891±0**	2025±0
gripper	3918	**964±0**	968±0	973±0	972±0	**970±0**	971±0
visitall	3321	2829±0	3048±0	**2035±0**	3040±0	2952±0	**2783±0**

	fixed/h^{FF}			fixed/none		
domain	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)						
blocks	**.20±.00**	.19±.00	.19±.00	**.21±.00**	.17±.00	.19±.00
ferry	**1**	**1**	.45±.00	**1**	**1**	.45±.00
gripper	**1**	**1**	.76±.00	**1**	**1**	.70±.00
visitall	**.91±.00**	**.91±.00**	**.91±.00**	.90±.00	**.91±.00**	**.91±.00**
Average node evaluations (smaller is better)						
blocks	**8844±0**	8849±0	8984±0	**8816±0**	8992±0	8944±0
ferry	**1862±0**	1900±0	7652±0	1877±0	**1860±0**	7630±0
gripper	**966±0**	**966±0**	4587±0	969±0	**964±0**	4775±0
visitall	3132±0	**2772±0**	2859±0	3079±0	2947±0	**2853±0**

Table 11: **Planning performance of LR heuristics.** For each model, we use the weights that resulted in the best validation MSE loss during training. Table columns and rows have the same meaning as in Table 4. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$+clip and $\mathcal{TN}$ is highlighted in **bold**.

				learn/h^{FF} ($l = h^{\text{LMcut}}$)		learn/h^{FF} ($l = h^{\max}$)		learn/h^{FF} ($l = h^{\text{blind}}$)	
domain	metric	h^{FF}	h^{LMcut}	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$
blocks	MSE	22.8	25.06	.76±.1	**.65±.1**	**.76±.1**	**.76±.2**	.76±.1	**.66±.1**
	+clip			.76±.2		**.76±.1**		.76±.1	
ferry	MSE	9.77	11.10	3.73±.7	**3.45±.8**	3.98±.9	**3.13±.9**	3.73±.8	**3.21±.9**
	+clip			3.72±.6		3.98±.9		3.73±.8	
gripper	MSE	9.93	15.82	**3.65±.9**	3.70±.9	**3.66±.9**	3.75±1.0	**3.65±.9**	3.69±1.1
	+clip			**3.65±.7**		**3.66±.9**		**3.65±.9**	
visitall	MSE	13.9	36.4	7.67±.4	**5.30±.6**	7.58±.5	**7.01±.3**	7.55±.4	**7.15±.5**
	+clip			7.60±.4		7.55±.5		7.55±.4	

Table 12: **Test accuracy of heuristics with alternative lower bounds (h^{LMcut}, $h^{\max}$, h^{blind}).** Results are obtained using NLMs and the *learn/h^{FF}* configuration. Table columns and rows have the same meaning as in Table 3. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N} + clip$, $\mathcal{TN}$ is highlighted in **bold**. Results for $\mathcal{N}$ models without clipping may vary across different lower bounds due to the use of different seeds.

A.8.1 Blocksworld

Blocksworld is one of the oldest domains in the planning literature. It represents a table with a collection of blocks that can be stacked on top of each other. The

		learn/h^{FF} ($l = h^{\text{LMcut}}$)			learn/h^{FF} ($l = h^{\text{max}}$)			learn/h^{FF} ($l = h^{\text{blind}}$)		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher is better)										
blocks	.13	.84±.19	.85±.19	**.88±.14**	.82±.18	.82±.18	**.85±.16**	.83±.17	.83±.17	**.89±.13**
ferry	.82	.91±.19	.91±.19	**.98±.05**	.97±.06	.97±.06	**.98±.05**	.90±.20	.90±.20	**.98±.02**
gripper	.96	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
visitall	.86	.97±.07	**.98±.06**	**.98±.05**	.98±.05	**.99±.03**	.98±.04	.97±.05	.98±.04	**.99±.02**
Average node evaluations (smaller is better)										
blocks	9309	2690±2128	2681±2121	**2060±1607**	2871±2235	2863±2227	**2480±1783**	2816±2065	2805±2084	**1987±1400**
ferry	5152	3216±1964	3117±1967	**2477±1093**	2802±1115	2772±1080	**2414±1051**	3277±1989	3291±1998	**1907±479**
gripper	3918	1642±139	1643±141	**1637±492**	1891±553	1890±556	**1763±361**	1889±544	1896±560	**1495±69**
visitall	3321	2156±1451	2148±1511	**1683±1290**	2007±1301	**1777±1154**	1894±1313	1899±1292	1864±1330	**1755±1132**

Table 13: **Planning performance of heuristics with alternative lower bounds (h^{LMcut}, h^{max}, h^{blind}).** Results are obtained using NLMs and the *learn/h^{FF}* configuration. Table columns and rows have the same meaning as in Table 4. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$+clip and $\mathcal{TN}$ is highlighted in **bold**. Results for $\mathcal{N}$ models without clipping may vary across different lower bounds due to the use of different seeds.

				learn/h^{FF}		learn/h^{LMcut}		learn/none	
domain	metric	h^{FF}	h^{LMcut}	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{TN}$
blocks	MSE	22.8	25.06	.76±.1	**.65±.1**	.72±.1	**.66±.1**	3.26±.6	**2.71±.4**
	+clip			.76±.1		.72±.1		2.91±.4	
ferry	MSE	9.77	11.10	3.73±.8	**3.45±.8**	**3.23±1.1**	3.58±1.3	141.05±29.6	**8.63±2.7**
	+clip			3.72±.8		**3.23±1.1**		10.44±1.8	
gripper	MSE	9.93	15.82	**3.65±.9**	3.70±1.1	3.94±1.0	**3.88±1.1**	68.12±15.3	**5.65±1.1**
	+clip			**3.65±.9**		3.94±1.0		13.37±2.2	
visitall	MSE	13.9	36.4	7.67±.4	**5.30±.6**	4.40±.8	**4.03±.5**	25.31±7.9	**9.70±1.6**
	+clip			7.60±.4		4.40±.8		18.79±4.2	

Table 14: **Test accuracy of heuristics with alternative residuals (h^{FF}, h^{LMcut}, none).** Results are obtained using the NLM models. Table columns and rows have the same meaning as in Table 3. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N} + clip$, $\mathcal{TN}$ is highlighted in **bold**.

		learn/h^{FF}			learn/h^{LMcut}			learn/none		
domain	h^{FF}	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$	$\mathcal{N}$	$\mathcal{N}$+clip	$\mathcal{TN}$
Ratio of solved instances under 10^4 evaluations (higher the better)										
blocks	.13	.84±.18	.85±.18	**.88±.15**	**.88±.14**	**.88±.15**	.85±.19	**.85±.22**	.57±.36	.51±.37
ferry	.82	.91±.19	.91±.19	**.98±.04**	.64±.11	.66±.15	**.68±.12**	.01±.01	**.60±.12**	.59±.13
gripper	.96	**1**	**1**	**1**	**1**	**1**	**1**	.00±.00	.75±.42	**1**
visitall	.86	.97±.06	**.98±.05**	**.98±.04**	**1**	**1**	**1**	.79±.34	**1**	**1**
Average node evaluations (smaller the better)										
blocks	9309	2690±2193	2681±2192	**2060±1673**	2461±1747	**2390±1757**	2735±2106	**3225±2446**	5754±2723	6492±2693
ferry	5152	3216±2033	3117±1991	**2477±1209**	6107±664	6088±739	**5944±680**	9952±70	**6751±740**	6780±863
gripper	3918	1642±212	1643±218	**1637±390**	1392±349	**1390±341**	1477±482	10000±0	4313±3475	**1480±378**
visitall	3321	2156±1431	2148±1550	**1683±1245**	508±205	531±232	**483±162**	3860±3249	818±562	**385±112**

Table 15: **Planning performance of heuristics with alternative residuals (h^{FF}, h^{LMcut}, none).** Results are obtained using the NLM models. Table columns and rows have the same meaning as in Table 4 of the main paper. For each configuration, the best metric among $\mathcal{N}$, $\mathcal{N}$+clip and $\mathcal{TN}$ is highlighted in **bold**.

goal in this domain is to rearrange the blocks to achieve a specific configuration, starting from some initial block arrangement. The initial and goal configurations are randomized. Blocks can be placed on top of another block or on the table, and every block can never have more than a single block on top of it. The arm/crane used to move the blocks around can only carry a single block at a time. Listing 1 contains the PDDL description of this domain.

Listing 1: **PDDL domain for *blocksworld*.**

```
(define (domain blocksworld)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block))

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

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

  (:action stack
         :parameters (?x - block ?y - block)
         :precondition (and (holding ?x)
                            (clear ?y))
         :effect (and (not (holding ?x))
                      (not (clear ?y))
                      (clear ?x)
                      (handempty)
                      (on ?x ?y)))

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

A.8.2 Ferry

Ferry is another classical domain. The goal is to use a ferry to transport a series of cars from their starting to their final locations, which are randomized. Each location is connected to every other location and the ferry can only carry one car at a time. Listing 2 contains the PDDL description of this domain.

Listing 2: **PDDL domain for *ferry*.**

```
(define (domain ferry)
  (:predicates (not-eq ?x ?y)
   (car ?c)
   (location ?l)
   (at-ferry ?l)
   (at ?c ?l)
   (empty-ferry)
   (on ?c))

  (:action sail
      :parameters  (?from ?to)
      :precondition (and (not-eq ?from ?to)
                         (location ?from) (location ?to) (at-ferry
                            ?from))
      :effect (and  (at-ferry ?to)
        (not (at-ferry ?from))))

  (:action board
      :parameters (?car ?loc)
      :precondition  (and  (car ?car) (location ?loc)
         (at ?car ?loc) (at-ferry ?loc) (empty-ferry))
      :effect (and (on ?car)
       (not (at ?car ?loc))
       (not (empty-ferry))))

  (:action debark
      :parameters  (?car  ?loc)
      :precondition  (and  (car ?car) (location ?loc)
         (on ?car) (at-ferry ?loc))
      :effect (and (at ?car ?loc)
       (empty-ferry)
       (not (on ?car)))))
```

A.8.3 Gripper

In the *gripper* domain, a robot with two gripper hands must transport a series of balls from one room to another. Unlike in traditional *gripper* instances, the initial and goal location of each ball is randomized. Each gripper hand can only carry one ball at a time. Listing 3 contains the PDDL description of this domain.

A.8.4 Visitall

This deceptively simple domain describes a square *NxN* grid in which a robot can move in the four directions (up, down, right or left). The goal is for the robot to visit several cells of the grid. The initial robot location, the number of cells to visit and their positions in the grid are all randomized. Listing 4 contains the PDDL description of this domain.

Listing 3: **PDDL domain for *gripper*.**

```
(define (domain gripper)
  (:predicates (room ?r)
   (ball ?b)
   (gripper ?g)
   (at-robby ?r)
   (at ?b ?r)
   (free ?g)
   (carry ?o ?g))

  (:action move
      :parameters  (?from ?to)
      :precondition (and  (room ?from) (room ?to) (at-robby ?from))
      :effect (and  (at-robby ?to)
        (not (at-robby ?from))))

  (:action pick
      :parameters (?obj ?room ?gripper)
      :precondition  (and  (ball ?obj) (room ?room) (gripper ?
         gripper)
         (at ?obj ?room) (at-robby ?room) (free ?gripper))
      :effect (and (carry ?obj ?gripper)
       (not (at ?obj ?room))
       (not (free ?gripper))))

  (:action drop
      :parameters  (?obj  ?room ?gripper)
      :precondition  (and  (ball ?obj) (room ?room) (gripper ?
         gripper)
         (carry ?obj ?gripper) (at-robby ?room))
      :effect (and (at ?obj ?room)
       (free ?gripper)
       (not (carry ?obj ?gripper)))))
```

Listing 4: **PDDL domain for *visitall*.**

```
(define (domain visit-all)
  (:requirements :typing)
  (:types        place - object)
  (:predicates (connected ?x ?y - place)
         (at-robot ?x - place)
         (visited ?x - place)
  )

  (:action move
  :parameters (?curpos ?nextpos - place)
  :precondition (and (at-robot ?curpos) (connected ?curpos ?nextpos
     ))
  :effect (and (at-robot ?nextpos) (not (at-robot ?curpos)) (
     visited ?nextpos))
  ))
```

Appendix B

Problem Generation with Neuro-Symbolic AI

B.1 Problem generation example

In this Appendix we provide a simple, handcrafted example of our problem generation method that illustrates how a single planning problem is created from start to finish. For this example, we will use *blocksworld* as our domain and, at each step (state), we will assume a random action is chosen from the set of applicable actions $App(s)$. In the initial state generation phase, an action $a \in App(s)$ corresponds to adding an atom to the (current) initial state s_{ic}, where this atom is obtained by instantiating a domain predicate (*on*, *ontable*, *handempty*, *holding* and *clear* in blocksworld) on objects of the correct type (*block* type in blocksworld, as all predicates are only instantiated on objects of this type) so that s_{ic} remains continuous-consistent. These objects can be in s_{ic} or not. In the latter case, we call them *virtual* objects and they are added to s_{ic} along with their corresponding atom. In the goal generation phase, an action $a \in App(s)$ corresponds to executing a domain action (*stack*, *unstack*, $pick - up$ and $put - down$ in blocksworld) in the (current) goal state s_{gc}, modifying the atoms in s_{gc} according to the positive and negative effects of a, which are encoded in the PDDL description of the domain. The selected action a must be grounded, i.e., instantiated, on objects of the correct type (*block* type in blocksworld, as all actions are applied to objects of this type) present in s_{gc} and, also, its preconditions must be true in s_{gc}.

Additionally, we will assume we randomly choose when to stop generating the states s_{ic} and s_{gc}. In a real scenario, the generative policies would be in charge of both selecting the next action to execute and when to stop generating s_{ic} and s_{gc} (i.e., sampling the termination action *end*). We represent the states of the MDP, corresponding to (incomplete or fully-generated) planning problems, as a tuple (s_{ic}, s_{gc}). We represent s_{ic} and s_{gc} as another tuple (O, P), where O is a set containing the objects present in the state (with their respective types), and P is a set containing the atoms of the state. We now detail the process followed to generate the example problem:

1. At the start of the generation process, the goal state s_{gc} is empty and the initial state s_{ic} can be either empty or equal to another state predefined by the user. In blocksworld, s_{ic} also begins empty. Therefore, the generation process starts

from an empty MDP state $(\_,\_)$, where the first element of the tuple represents s_{ic} and the second one represents s_{gc}. We assume the action *add ontable*$(o1)$ is sampled, where $o1$ is an object of type *block*. Since $o1$ corresponds to a virtual object, we also need to add it to s_{ic}. Thus, the resulting state is $(\,(\{block\ o1\}, \{ontable(o1)\}), \_)$, which corresponds to a continuous-consistent state.

2. The action *add on*$(o2, o1)$ is selected, where $o2$ is a virtual object of type *block*. The resulting state is $(\,(\{block\ \ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1)\}),\ \_)$, which corresponds to a continuous-consistent state.

3. The action *add clear*$(o2)$ is selected and the resulting state is $(\,(\{block\ \ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1), clear(o2)\}),\ \_)$, which corresponds to a continuous-consistent state.

4. The action *add handempty*$()$ is selected and the resulting state is $(\,(\{block\ o1, block\ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), \_)$, which corresponds to a continuous-consistent state.

5. The termination action *end* is sampled, so the initial state generation phase concludes, i.e., $s_i = s_{ic}$. Then, the consistency evaluator checks if s_i meets the eventual consistency rules, which it does (otherwise, s_i would be discarded and no goal would be generated). Therefore, the goal generation phase can start. The goal state is initialized to the initial state, i.e., $s_{gc} = s_i$, so the resulting state is $(\,(\{block\ \ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ \ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}))$.

6. As we are now in the goal generation phase, the set of applicable actions $App(s)$ corresponds to the domain actions whose preconditions are met in s_{gc}. Assume the action *unstack*$(o2, o1)$ is selected. Then, the current goal state s_{gc} is modified with the effects of the selected action. Thus, the next state is $(\,(\{block\ \ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ o1, block\ o2\}, \{ontable(o1), holding(o2), clear(o1)\}))$.

7. The action $put - down(o2)$ is selected and the resulting state is $(\,(\{block\ o1, block\ \ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ \ o1, block\ o2\}, \{ontable(o1), clear(o1), clear(o2), handempty(), ontable(o2)\}))$.

8. The termination action *end* is sampled, so the goal generation phase concludes, i.e., $s_g = s_{gc}$. Then, the problem goal g is obtained by selecting a subset of the atoms in s_g according to the goal types and predicates specified by the user. Let us assume the goal types and predicates list is $\{ontable(block), on(block, block)\}$, i.e., g must only contain atoms of predicate type *on* or *ontable* instantiated on objects of type *block*. If that is the case, the goal is $g = \{ontable(o1), ontable(o2)\}$ and NeSIG outputs the problem (s_i, g), which is shown in Listing 1.

Listing 1: **Example problem generated with NeSIG.**

```
(define (problem example_blocksworld_problem)

(:domain blocksworld)

(:objects o1 o2 - block)

(:init (ontable o1) (on o2 o1)
       (clear o2) (handempty))

(:goal (ontable o1) (ontable o2))
)
```

B.2 PDDL domains

In this Appendix we provide the PDDL descriptions for the planning domains employed in our experiments: *logistics*, *sokoban*, *miconic* and *satellite*. The PDDL description of *blocksworld* is provided in Listing 1.

Listing 2: **PDDL description for *logistics*.**

```
(define (domain logistics)
  (:requirements :strips :typing :existential-preconditions)
  (:types city location thing - object
          package vehicle - thing
          truck airplane - vehicle
          airport - location)
  (:predicates (in-city ?l - location ?c - city)
               (at ?obj - thing ?l - location)
               (in ?p - package ?veh - vehicle))

  (:action drive
          :parameters (?t - truck ?from ?to - location)
          :precondition (and (at ?t ?from)
                             (exists (?c - city)
                             (and (in-city ?from ?c)
                                  (in-city ?to ?c))))
          :effect (and (not (at ?t ?from))
                       (at ?t ?to)))

  (:action fly
          :parameters (?a - airplane ?from ?to - airport)
          :precondition (at ?a ?from)
          :effect (and (not (at ?a ?from))
                       (at ?a ?to)))

  (:action load
          :parameters (?v - vehicle ?p - package ?l - location)
          :precondition (and (at ?v ?l)
                        (at ?p ?l))
          :effect (and (not (at ?p ?l))
                       (in ?p ?v)))

  (:action unload
          :parameters (?v - vehicle ?p - package ?l - location)
          :precondition (and (at ?v ?l)
                             (in ?p ?v))
```

```
        :effect (and (not (in ?p ?v))
                     (at ?p ?l))))
```

Listing 3: **PDDL description for *sokoban*.**

```
(define (domain sokoban)
  (:requirements :typing :negative-preconditions)
  (:types loc)
  (:predicates (at-robot ?l - loc)
               (at-box ?l - loc)
               (at-wall ?l - loc)
               (connected-up ?l1 - loc ?l2 - loc)
               (connected-right ?l1 - loc ?l2 - loc))

  (:action move-up
        :parameters (?from - loc ?to - loc)
        :precondition (and (at-robot ?from)
                           (not (at-box ?to))
                           (not (at-wall ?to))
                           (connected-up ?from ?to))
        :effect (and (not (at-robot ?from))
                     (at-robot ?to)))

  (:action move-down
        :parameters (?from - loc ?to - loc)
        :precondition (and (at-robot ?from)
                           (not (at-box ?to))
                           (not (at-wall ?to))
                           (connected-up ?to ?from))
        :effect (and (not (at-robot ?from))
                     (at-robot ?to)))

  (:action move-right
        :parameters (?from - loc ?to - loc)
        :precondition (and (at-robot ?from)
                           (not (at-box ?to))
                           (not (at-wall ?to))
                           (connected-right ?from ?to))
        :effect (and (not (at-robot ?from))
                     (at-robot ?to)))

  (:action move-left
        :parameters (?from - loc ?to - loc)
        :precondition (and (at-robot ?from)
                           (not (at-box ?to))
                           (not (at-wall ?to))
                           (connected-right ?to ?from))
        :effect (and (not (at-robot ?from))
                     (at-robot ?to)))

  (:action push-up
        :parameters (?l1 - loc ?l2 - loc ?l3 - loc)
        :precondition (and (at-robot ?l1)
                           (at-box ?l2)
                           (not (at-box ?l3))
                           (not (at-wall ?l3))
                           (connected-up ?l1 ?l2)
                           (connected-up ?l2 ?l3))
        :effect (and (not (at-robot ?l1))
                     (at-robot ?l2)
                     (not (at-box ?l2))
                     (at-box ?l3)))

  (:action push-down
        :parameters (?l1 - loc ?l2 - loc ?l3 - loc)
        :precondition (and (at-robot ?l1)
                           (at-box ?l2)
                           (not (at-box ?l3))
                           (not (at-wall ?l3))
                           (connected-up ?l2 ?l1)
                           (connected-up ?l3 ?l2))
        :effect (and (not (at-robot ?l1))
                     (at-robot ?l2)
                     (not (at-box ?l2))
                     (at-box ?l3)))

  (:action push-right
        :parameters (?l1 - loc ?l2 - loc ?l3 - loc)
        :precondition (and (at-robot ?l1)
                           (at-box ?l2)
                           (not (at-box ?l3))
                           (not (at-wall ?l3))
                           (connected-right ?l1 ?l2)
                           (connected-right ?l2 ?l3))
        :effect (and (not (at-robot ?l1))
                     (at-robot ?l2)
                     (not (at-box ?l2))
                     (at-box ?l3)))

  (:action push-left
        :parameters (?l1 - loc ?l2 - loc ?l3 - loc)
        :precondition (and (at-robot ?l1)
                           (at-box ?l2)
                           (not (at-box ?l3))
                           (not (at-wall ?l3))
                           (connected-right ?l2 ?l1)
                           (connected-right ?l3 ?l2))
        :effect (and (not (at-robot ?l1))
                     (at-robot ?l2)
                     (not (at-box ?l2))
                     (at-box ?l3))))
```

Listing 4: **PDDL description for *miconic*.**

```
(define (domain miconic)
  (:requirements :strips)
  (:types passenger floor)

;; (above ?f1 - floor ?f2 - floor) -> f1 is on top of f2
(:predicates
  (at ?p - passenger ?f - floor)
  (above ?f1 - floor  ?f2 - floor)
  (boarded ?p - passenger)
  (lift_at ?f - floor)
  (lift_empty))

(:action board
  :parameters (?p - passenger ?f - floor)
  :precondition (and (lift_at ?f)
                     (at ?p ?f)
                     (lift_empty))
  :effect (and  (not (lift_empty))
                (not (at ?p ?f))
                (boarded ?p)))

(:action depart
  :parameters (?p - passenger ?f - floor)
  :precondition (and (lift_at ?f)
                     (boarded ?p))
  :effect (and  (not (boarded ?p))
                (at ?p ?f)
                (lift_empty)))

(:action up
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift_at ?f1)
                     (above ?f2 ?f1))
  :effect (and (lift_at ?f2)
               (not (lift_at ?f1))))

(:action down
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift_at ?f1)
                     (above ?f1 ?f2))
  :effect (and (lift_at ?f2)
               (not (lift_at ?f1)))))
```

Listing 5: **PDDL description for *satellite*.**

```
(define (domain satellite)
  (:requirements :strips :typing)
  (:types satellite direction instrument mode)

  ;; We introduce a ‘dummy’ predicate that is only used for adding
  ;; directions which are not instantiated in any atom of the init
  ;; state. This makes possible to have directions which only
  ;; appear in "have_image" atoms in the goal (but no atom in the
  ;; init state).
  (: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)
        (dummy ?d - direction))

  (:action turn_to
   :parameters (?s - satellite ?d_new - direction ?d_prev -
      direction)
   :precondition (and (pointing ?s ?d_prev))
   :effect (and  (pointing ?s ?d_new)
                 (not (pointing ?s ?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 (?d - direction ?i - instrument ?m - mode)
  :precondition (and (calibrated ?i)
                     (supports ?i ?m)
                     (power_on ?i)
                     (exists (?s - satellite)
                       (and (on_board ?i ?s) (pointing ?s ?d))
                     ))
  :effect (have_image ?d ?m)))
```

B.3 Consistency rules

In this Appendix we detail the consistency rules (i.e., constraints) used for the *blocksworld*, *logistics*, *sokoban*, *miconic* and *satellite* domains. As explained in Section VI.3.1.1, consistency rules are encoded using a novel, semi-declarative language that makes possible to combine standard Python code with a FOL-like syntax. For each domain, we need to provide a consistency evaluator containing the consistency rules, which simply corresponds to a Python class with two methods: *check_continuous_consistency(self, curr_state, atom_pred, atom_obj_consts, atom_obj_inds, atom_obj_types)* and *check_eventual_consistency(self, curr_state)*. The first method receives as inputs the current initial state s_{ic} (*curr_state*) along with some atom a to add to s_{ic}, whose information is encoded in the *atom_pred*, *atom_obj_consts*, *atom_obj_inds* and *atom_obj_types* parameters, and returns whether the state resulting from adding a to s_{ic} is continuous-consistent or not. Both *atom_obj_consts* and *atom_obj_inds* contain the objects instantiated on the atom a but represent this information in a different manner: the former is used by rules encoded as FOL formulas, whereas the latter is employed by rules encoded using standard Python code. Therefore, the user can encode consistency rules in their preferred manner: declarative (i.e., FOL) or imperative (i.e., Python). This choice is completely transparent to NeSIG and does not impact training. The second method receives as its sole input the completely-generated initial state s_i and returns whether it is eventual-consistent or not. We now provide the code for the consistency evaluators of the five domains used in our experimentation. For brevity reasons, we have omitted lines corresponding to aliases (e.g., *ontable = self.ontable*) and declaration of FOL variables (e.g., x = *Variable('x')*). As previously mentioned, consistency rules are chosen so that NeSIG can generate the same set of problems as the domain-specific generators. For instance, problems obtained by the blocksworld generator never contain atoms of type *holding* in their initial state, so this constraint is also encoded in the consistency rules for blocksworld employed by NeSIG. Finally, it can be observed that, thanks to our proposed language, consistency rules can be encoded in an easy and interpretable manner with just a few lines of code.

Listing 6: **Consistency evaluator for *blocksworld*.**

```
class ConsistencyEvaluatorBlocksworld(ConsistencyEvaluator):

    def check_continuous_consistency(self, curr_state, atom_pred,
   atom_obj_consts, atom_obj_inds, atom_obj_types):

        """
        (ontable x)
            - x is new
        """
        if atom_pred == 'ontable':
            a = atom_obj_consts[0]
            formula = virtual(a)
            return self._evaluate(formula)

        """
        (on a b)
            - a is new
            - b is NOT new
            - b does not appear in an atom of type
              clear
            - b does not have a block on top of
              it (on(*,b) does not exist)
        """
        if atom_pred == 'on':
            a, b = atom_obj_consts
            formula = virtual(a) & ~virtual(b) & ~clear(b) & ~TE(x,
    on(x,b))
            return self._evaluate(formula)

        """
        (clear x)
            - x is NOT new
            - x does not appear in holding(x)
            - x does not have a block on top of
              it (on(*,x) does not exist)
        """
        if atom_pred == 'clear':
            a = atom_obj_consts[0]
            formula = ~virtual(a) & ~TE(x, on(x,a))
            return self._evaluate(formula)

        """
        (holding x)
```

```
            - The initial state cannot contain
              atoms of type "holding"
        """
        if atom_pred == 'holding':
            return False

        """
        (handempty)
            No consistency rules to check
            Note that we don't need to check
            whether (handempty) already exists,
            since repeated atoms are implicitly
            discarded
        """
        if atom_pred == 'handempty':
            return True

    def check_eventual_consistency(self, curr_state):
        # The problem must contain at least
        # two blocks (otherwise, they can't be
        # stacked)
        formula_1 = (TE(x, _type(x, block)) >= 2)

        # The initial state must contain
        # (handempty)
        formula_2 = handempty()

        # For all objects x there must exist
        # clear(x), unless they have another
        # block y on top
        formula_3 = FA(x,clear(x) | TE(y, on(y,x)))

        return self._evaluate(formula_1 & formula_2
        & formula_3)
```

Listing 7: **Consistency evaluator for *logistics*.**

```
class ConsistencyEvaluatorLogistics(ConsistencyEvaluator):

    def check_continuous_consistency(self, curr_state, atom_pred,
   atom_obj_consts, atom_obj_inds, atom_obj_types):
        """
        (in-city ?loc - location ?city - city)
            - loc is new
            - if city is new, then loc must be of
              type airport (the first location of
              every city is always an airport)
            - if city is NOT new, then loc must
              be of type location (since each
              city contains one and only one
              airport)
        """
        if atom_pred == 'in-city':
            loc, city = atom_obj_consts
            formula = virtual(loc) &
            (_type(loc,airport) ** virtual(city))
            return self._evaluate(formula)

        """
        (at ?obj - thing ?loc - location)
            - obj is new
            - loc must not be new
            - obj must be of type package, truck or
              airplane
            - if obj is of type airplane, then loc
              must be of type airport
        """
        if atom_pred == 'at':
            obj, loc = atom_obj_consts
            formula = virtual(obj) & ~virtual(loc)
            & ( _type(obj, package) |
            _type(obj, truck) |
            _type(obj, airplane) ) &
            ( _type(obj, airplane) >>
            _type(loc, airport) )
            return self._evaluate(formula)

        """
        (in ?p - package ?veh - vehicle)
            The initial state can have no atoms of
            type "in"
        """
        if atom_pred == 'in':
            return False

    def check_eventual_consistency(self, curr_state):
        # The problem must contain at least one
        # airplane
        formula_1 = TE(x, _type(x, airplane))

        # The problem must contain at least two
        # cities
        formula_2 = TE(x, _type(x, city)) >= 2

        # Every city must contain at least one
        # truck
        # x -> city, y -> location/airport in
        # the city, z -> truck at the
        # location/airport
        # Meaning of the formula: "For every
        # city x, there must exist a
        # location/airport y in the city x,
        # so that there exists a truck z at
        # the location/airport y"
        formula_3 = FA(x, _type(x, city) >>
        TE(y, in_city(y, x) &
        TE(z, _type(z, truck) & at(z, y) ) ) )

        return self._evaluate(formula_1 & formula_2
        & formula_3)
```

Listing 8: **Consistency evaluator for *sokoban*.**

```
class ConsistencyEvaluatorSokoban(ConsistencyEvaluator):

    def check_continuous_consistency(self, curr_state, atom_pred,
   atom_obj_consts, atom_obj_inds, atom_obj_types):
        """
        (connected-up l1 l2)
            - Cannot be added, as the we start
              generation from an empty NxM map
        """
        if atom_pred == 'connected-up':
            return False

        """
        (connected-right l1 l2)
            - Cannot be added, as the we start
              generation from an empty NxM map
        """
        if atom_pred == 'connected-right':
            return False

        """
        (at-robot loc)
            - loc must already exist in the state
            - Only one robot can exist at the same
              time
            - (at-box loc) does not exist
            - (at-wall loc) does not exist
        """
        if atom_pred == 'at-robot':
            loc = atom_obj_consts[0]
            formula = ~virtual(loc) &
            ~TE(x, at_robot(x)) & ~at_box(loc)
            & ~at_wall(loc)
            return self._evaluate(formula)

        """
        (at-box loc)
            - loc must already exist in the state
            - (at-robot loc) does not exist
            - (at-wall loc) does not exist
        """
        if atom_pred == 'at-box':
            loc = atom_obj_consts[0]
            formula = ~virtual(loc) &
            ~at_robot(loc) & ~at_wall(loc)
            return self._evaluate(formula)

        """
        (at-wall loc)
            - loc must already exist in the state
            - (at-robot loc) does not exist
            - (at-box loc) does not exist
        """
        if atom_pred == 'at-wall':
            loc = atom_obj_consts[0]
            formula = ~virtual(loc) &
            ~at_robot(loc) & ~at_box(loc)
            return self._evaluate(formula)

    def check_eventual_consistency(self, curr_state):
        # The initial state must contain one robot
        formula = (TE(x, at_robot(x)) == 1)
        return self._evaluate(formula)
```

Listing 9: **Consistency evaluator for *miconic*.**

```
class ConsistencyEvaluatorMiconic(ConsistencyEvaluator):

    def check_continuous_consistency(self, curr_state, atom_pred,
   atom_obj_consts, atom_obj_inds, atom_obj_types):
        """
        (above ?f1 - floor  ?f2 - floor)
            - If both floors are new, no need to check consistency
            - If there already exist floors in the init state
                - f1 must be new
                - f2 must NOT be new
                - There must be no floor above f1
        """
        if atom_pred == 'above':
            f1, f2 = atom_obj_consts
            # ~virtual(x) is needed so that virtual objects are not
            # considered when testing if the init state already
            # contains an object of type floor
            formula = TE(x, _type(x, floor) & ~virtual(x)) >>
            ( virtual(f1) & ~virtual(f2) & ~TE(y, above(y,f2)) )
            return self._evaluate(formula)

        """
        (at ?p - passenger ?f - floor)
            - p is new
            - f is NOT new
        """
        if atom_pred == 'at':
            p, f = atom_obj_consts
            formula = virtual(p) & ~virtual(f)
            return self._evaluate(formula)

        """
        (boarded ?p - passenger)
            - Cannot be added
        """
        if atom_pred == 'boarded':
            return False

        """
        (lift_at ?f - floor)
            - f is NOT new
        """
        if atom_pred == 'lift_at':
            f = atom_obj_consts[0]
            formula = ~virtual(f)
            return self._evaluate(formula)

        """
        (lift_empty)
            - No need to check whether it has already been added,
              since repeated atoms are implicitly discard
        """
        if atom_pred == 'lift_empty':
            return True
    def check_eventual_consistency(self, curr_state):
        # The init state must contain (lift_empty)
        formula_1 = lift_empty()

        # The lift must be at one and only one floor
        formula_2 = TE(x, lift_at(x)) == 1

        return self._evaluate(formula_1 & formula_2)
```

Listing 10: **Consistency evaluator for *satellite*.**

```
class ConsistencyEvaluatorSatellite(ConsistencyEvaluator):

    def check_continuous_consistency(self, curr_state, atom_pred,
   atom_obj_consts, atom_obj_inds, atom_obj_types):
        """
        (dummy ?d - direction)
            - d is new
        """
        if atom_pred == 'dummy':
            d = atom_obj_consts[0]
            formula = virtual(d)
            return self._evaluate(formula)

        """
        (pointing ?s - satellite ?d - direction)
            - s is new
            - d is NOT new
        """
        if atom_pred == 'pointing':
            s, d = atom_obj_consts
            formula = virtual(s) & ~virtual(d)
            return self._evaluate(formula)

        """
        (on_board ?i - instrument ?s - satellite)
            - i is new
            - s is NOT new
        """
        if atom_pred == 'on_board':
            i, s = atom_obj_consts
            formula = virtual(i) & ~virtual(s)
            return self._evaluate(formula)

        """
        (supports ?i - instrument ?m - mode)
            - i is NOT new
        """
        if atom_pred == 'supports':
            i, m = atom_obj_consts
            formula = ~virtual(i)
            return self._evaluate(formula)

        """
        (calibration_target ?i - instrument ?d - direction)
            - i is NOT new
            - d is NOT new
        """
        if atom_pred == 'calibration_target':
            i, d = atom_obj_consts
            formula = ~virtual(i) & ~virtual(d)
            return self._evaluate(formula)

        """
        (power_avail ?s - satellite)
            - s is NOT new
        """
        if atom_pred == 'power_avail':
            s = atom_obj_consts[0]
            formula = ~virtual(s)
            return self._evaluate(formula)

        """
        (power_on ?i - instrument)
            - Cannot be added
        """
        if atom_pred == 'power_on':
            return False

        """
        (calibrated ?i - instrument)
            - Cannot be added
        """
        if atom_pred == 'calibrated':
            return False

        """
        (have_image ?d - direction ?m - mode)
            - Cannot be added
        """
        if atom_pred == 'have_image':
            return False

    def check_eventual_consistency(self, curr_state):
        # There exists at least one satellite
        formula_1 = TE(x, _type(x, satellite))

        # Each satellite needs to have power available
        formula_2 = FA(x, _type(x, satellite) >> power_avail(x))

        # Each satellite needs to have at least one instrument on
        # board
        formula_3 = FA(x, _type(x, satellite) >> TE(y, on_board(y,
                        x)))

        # Each instrument needs to support at least one mode
        formula_4 = FA(x, _type(x, instrument) >> TE(y, supports(x,
                        y)))

        # Each instrument needs to have at least one calibration
        # target
        formula_5 = FA(x, _type(x, instrument) >> TE(y,
                        calibration_target(x, y)))

        return self._evaluate(formula_1 & formula_2 & formula_3 &
                              formula_4 & formula_5)
```

B.4 Parameters for domain-specific generators

In this Appendix we detail the range of parameters employed for the domain-specific generator of each domain: *blocksworld*, *logistics*, *sokoban*, *miconic* and *satellite*. For each generator parameter, we set a possible range of values and uniformly sample from it to generate each problem. Finally, the number of problems generated per experiment, time and memory limits for the planners, and difficulty of terminated problems is equal to that used for NeSIG (see Table 21). The remaining configuration parameters are detailed in Tables 16, 17, 18, 19 and 20. These values have been selected in order to maximize problem diversity and avoid generation bias towards any particular type of problem.

Name	Value
seed for each experiment	1
blocks range	$[ceil(D/3), D]$

Table 16: **Generator parameters for *blocksworld*.** D denotes maximum problem size and *ceil* is the ceiling function that approximates a real number to the next integer.

Name	Value
seed for each experiment	1
airplanes range	$[1, D]$
cities range	$[2, D]$
city_size range	$[1, D]$
packages range	$[1, D]$
extra_trucks range	$[0, D]$

Table 17: **Generator parameters for *logistics*.** D denotes maximum problem size.

Name	Value
seed for each experiment	1
boxes range	$[1, floor(C * 0.6)]$
walls range	$[0, floor(C * 0.6)]$

Table 18: **Generator parameters for *sokoban*.** C denotes the number of cells (e.g., 25 for a map size of 5x5) and $floor$ is the floor function that approximates a real number to the preceding integer.

B.5 NeSIG hyperparameters

In this Appendix we provide the hyperparameters not detailed in Section VI.4.1. We have utilized the same hyperparameter values for all five domains: *blocksworld*,

Name	Value
seed for each experiment	1
floors range	$[2, D]$
passengers range	$[1, D]$

Table 19: **Generator parameters for *miconic*.** D denotes maximum problem size.

Name	Value
seed for each experiment	1
satellites range	$[1, ceil(D/2)]$
max instruments per satellite	$[1, ceil(D/2)]$
modes	$[1, ceil(D/2)]$
targets	$[1, ceil(D/2)]$
observations	$[1, ceil(D/2)]$

Table 20: **Generator parameters for *satellite*.** D denotes maximum problem size and *ceil* is the ceiling function that approximates a real number to the next integer.

logistics, *sokoban*, *miconic* and *satellite*. The exceptions to this rule are the following: goal types and predicates[1], since this information is domain-dependent; the maximum number of actions in the goal generation phase, as we use more for *sokoban*, *miconic* and *satellite* since these domains are more challenging than the rest; diversity threshold θ (see Equation VI.1), chosen by testing a few values and selecting the one resulting in the best balance between problem difficulty and diversity; number of training steps and problems generated at each step, as *miconic* and *satellite* require larger values to ensure training stability and convergence. The complete list of hyperparameters is detailed in Table 21.

[1]We take into consideration type inheritance. Therefore, the set $\{at(package, location)\}$ is equivalent to $\{at(package, location), at(package, airport)\}$, since type *airport* inherits from *location*.

Name	Value
num problems generated at each train step	
— blocksworld	25
— logistics	25
— sokoban	25
— miconic	50
— satellite	50
seed for each experiment	1-5
goal types and predicates	
— blocksworld	$\{on(block, block)\}$
— logistics	$\{at(package, location)\}$
— sokoban	$\{at\text{-}box(loc)\}$
— miconic	$\{at(passenger, floor)\}$
— satellite	$\{have\_image(direction, mode)\}$
max actions init state generation phase (training/validation)	15
max actions goal generation phase (training/validation)	
— blocksworld	60
— logistics	60
— sokoban	75
— miconic	75
— satellite	75
max actions init state generation phase (size generalization experiments)	
— blocksworld	[10,15,20,25,30,35,40]
— logistics	[10,15,20,25,30,35,40]
— sokoban	[10,15,21,30]
— miconic	[10,15,20,25,30,35,40]
— satellite	[10,15,20,25,30,35,40]
max actions goal generation phase (size generalization experiments)	
— blocksworld	[40,60,80,100,120,140,160]
— logistics	[40,60,80,100,120,140,160]
— sokoban	[50,75,105,150]
— miconic	[50,75,100,125,150,175,200]
— satellite	[50,75,100,125,150,175,200]
diversity threshold θ in Equation VI.1	
— blocksworld	0.02
— logistics	0.2
— sokoban	0.02
— miconic	0.2
— satellite	0.1
minibatch size	64
L2 weight decay	0.0
reward discount factor γ	1.0
GAE factor λ	1.0
min num samples for performing a training step	32
critic loss weight	0.1
gradient clipping value	5.0
PPO epochs per train step	3
PPO ϵ value	0.2
Policy entropy bonus	Linearly annealed from 0.2 to 0 over 2500 training steps
NLM	
— breadth	3
— depth	5
— output predicates for inner layers	8 (for each arity from 0 to 3)
— MLP hidden features	0
— residual type	*all* (concatenate to the input of each layer the outputs of all previous layers)
— exclude-self	True (*reduce* operations ignore repeated tensor positions)
— batch normalization	False
— activation function	sigmoid

Table 21: **NeSIG hyperparameters.**