Width-Based Planning and Learning

Miquel Junyent Barbany

DOCTORAL THESIS UPF / 2021

THESIS SUPERVISORS
Dr. Anders Jonsson
Dr. Vicenç Gómez

Dept. of Information and Communication Technologies

Al meu pare.

Abstract

Optimal sequential decision making is a fundamental problem to many diverse fields. In recent years, Reinforcement Learning (RL) methods have experienced unprecedented success, largely enabled by the use of deep learning models, reaching human-level performance in several domains, such as the Atari video games or the ancient game of Go. In contrast to the RL approach in which the agent learns a policy from environment interaction samples, ignoring the structure of the problem, the planning approach for decision making assumes known models for the agent's goals and domain dynamics, and focuses on determining how the agent should behave to achieve its objectives. Current planners are able to solve problem instances involving huge state spaces by precisely exploiting the problem structure that is defined in the state-action model.

In this work we combine the two approaches, leveraging fast and compact policies from learning methods and the capacity to perform lookaheads in combinatorial problems from planning methods. In particular, we focus on a family of planners called width-based planners, that has demonstrated great success in recent years due to its ability to scale independently of the size of the state space. The basic algorithm, Iterated Width (IW), was originally proposed for classical planning problems, where a model for state transitions and goals, represented by sets of atoms, is fully determined. Nevertheless, width-based planners do not require a fully defined model of the environment, and can be used with simulators. For instance, they have been recently applied in pixel domains such as the Atari games.

Despite its success, IW is purely exploratory, and does not leverage past reward information. Furthermore, it requires the state to be factored

into features that need to be pre-defined for the particular task. Moreover, running the algorithm with a width larger than 1 in practice is usually computationally intractable, prohibiting IW from solving higher width problems.

We begin this dissertation by studying the complexity of width-based methods when the state space is defined by multivalued features, as in the RL setting, instead of Boolean atoms. We provide a tight upper bound on the amount of nodes expanded by IW, as well as overall algorithmic complexity results. In order to deal with more challenging problems (i.e., those with a width higher than 1), we present a hierarchical algorithm that plans at two levels of abstraction. A high-level planner uses abstract features that are incrementally discovered from low-level pruning decisions. We illustrate this algorithm in classical planning PDDL domains as well as in pixel-based simulator domains. In classical planning, we show how IW(1) at two levels of abstraction can solve problems of width 2.

To leverage past reward information, we extend width-based planning by incorporating an explicit policy in the action selection mechanism. Our method, called π-IW, interleaves width-based planning and policy learning using the state-actions visited by the planner. The policy estimate takes the form of a neural network and is in turn used to guide the planning step, thus reinforcing promising paths. Notably, the representation learned by the neural network can be used as a feature space for the width-based planner without degrading its performance, thus removing the requirement of pre-defined features for the planner. We compare π-IW with previous width-based methods and with AlphaZero, a method that also interleaves planning and learning, in simple environments, and show that π-IW has superior performance. We also show that the π-IW algorithm outperforms previous width-based methods in the pixel setting of Atari games suite. Finally, we show that the proposed hierarchical IW can be seamlessly integrated with our policy learning scheme, resulting in an algorithm that outperforms flat IW-based planners in Atari games with sparse rewards.

Resum

La presa seqüencial de decisions òptimes és un problema fonamental en diversos camps. En els últims anys, els mètodes d'aprenentatge per reforç (RL) han experimentat un èxit sense precedents, en gran part gràcies a l'ús de models d'aprenentatge profund, aconseguint un rendiment a nivell humà en diversos dominis, com els videojocs d'Atari o l'antic joc de Go. En contrast amb l'enfocament de RL, on l'agent aprèn una política a partir de mostres d'interacció amb l'entorn, ignorant l'estructura del problema, l'enfocament de planificació assumeix models coneguts per als objectius de l'agent i la dinàmica del domini, i es basa en determinar com ha de comportar-se l'agent per aconseguir els seus objectius. Els planificadors actuals són capaços de resoldre problemes que involucren grans espais d'estats precisament explotant l'estructura del problema, definida en el model estat-acció.

En aquest treball combinem els dos enfocaments, aprofitant polítiques ràpides i compactes dels mètodes d'aprenentatge i la capacitat de fer cerques en problemes combinatoris dels mètodes de planificació. En particular, ens enfoquem en una família de planificadors basats en el *width* (ample), que han tingut molt èxit en els últims anys gràcies a que la seva escalabilitat és independent de la mida de l'espai d'estats. L'algorisme bàsic, Iterated Width (IW), es va proposar originalment per problemes de planificació clàssica, on el model de transicions d'estat i objectius ve completament determinat, representat per conjunts d'àtoms. No obstant, els planificadors basats en width no requereixen un model de l'entorn completament definit i es poden utilitzar amb simuladors. Per exemple, s'han aplicat recentment a dominis gràfics com els jocs d'Atari.

Malgrat el seu èxit, IW és un algorisme purament exploratori i no

aprofita la informació de recompenses anteriors. A més, requereix que l'estat estigui factoritzat en característiques, que han de predefinirse per a la tasca en concret. A més, executar l'algorisme amb un width superior a 1 sol ser computacionalment intractable a la pràctica, el que impedeix que IW resolgui problemes de width superior.

Comencem aquesta tesi estudiant la complexitat dels mètodes basats en width quan l'espai d'estats està definit per característiques multivalor, com en els problemes de RL, en lloc d'àtoms booleans. Proporcionem un límit superior més precís en la quantitat de nodes expandits per IW, així com resultats generals de complexitat algorísmica. Per fer front a problemes més complexos (és a dir, aquells amb un width superior a 1), presentem un algorisme jeràrquic que planifica en dos nivells d'abstracció. El planificador d'alt nivell utilitza característiques abstractes que es van descobrint gradualment a partir de decisions de poda en l'arbre de baix nivell. Il·lustrem aquest algorisme en dominis PDDL de planificació clàssica, així com en dominis de simuladors gràfics. En planificació clàssica, mostrem com IW(1) en dos nivells d'abstracció pot resoldre problemes de width 2.

Per aprofitar la informació de recompenses passades, incorporem una política explícita en el mecanisme de selecció d'accions. El nostre mètode, anomenat π-IW, intercala la planificació basada en width i l'aprenentatge de la política usant les accions visitades pel planificador. Representem la política amb una xarxa neuronal que, al seu torn, s'utilitza per guiar la planificació, reforçant així camins prometedors. A més, la representació apresa per la xarxa neuronal es pot utilitzar com a característiques per al planificador sense degradar el seu rendiment, eliminant així el requisit d'usar característiques predefinides. Comparem π-IW amb mètodes anteriors basats en width i amb AlphaZero, un mètode que també intercala planificació i aprenentatge, i mostrem que π-IW té un rendiment superior en entorns simples. També mostrem que l'algorisme π-IW supera altres mètodes basats en width en els jocs d'Atari. Finalment, mostrem que el mètode IW jeràrquic proposat pot integrar-se fàcilment amb el nostre esquema d'aprenentatge de la política, donant com a resultat un algorisme que supera els planificadors no jeràrquics basats en IW en els jocs d'Atari amb recompenses distants.

Resumen

La toma secuencial de decisiones óptimas es un problema fundamental en diversos campos. En los últimos años, los métodos de aprendizaje por refuerzo (RL) han experimentado un éxito sin precedentes, en gran parte gracias al uso de modelos de aprendizaje profundo, alcanzando un rendimiento a nivel humano en varios dominios, como los videojuegos de Atari o el antiguo juego de Go. En contraste con el enfoque de RL, donde el agente aprende una política a partir de muestras de interacción con el entorno, ignorando la estructura del problema, el enfoque de planificación asume modelos conocidos para los objetivos del agente y la dinámica del dominio, y se basa en determinar cómo debe comportarse el agente para lograr sus objetivos. Los planificadores actuales son capaces de resolver problemas que involucran grandes espacios de estados precisamente explotando la estructura del problema, definida en el modelo estado-acción.

En este trabajo combinamos los dos enfoques, aprovechando políticas rápidas y compactas de los métodos de aprendizaje y la capacidad de realizar búsquedas en problemas combinatorios de los métodos de planificación. En particular, nos enfocamos en una familia de planificadores basados en el *width* (ancho), que han demostrado un gran éxito en los últimos años debido a que su escalabilidad es independiente del tamaño del espacio de estados. El algoritmo básico, Iterated Width (IW), se propuso originalmente para problemas de planificación clásica, donde el modelo de transiciones de estado y objetivos viene completamente determinado, representado por conjuntos de átomos. Sin embargo, los planificadores basados en width no requieren un modelo del entorno completamente definido y se pueden utilizar con simuladores. Por ejemplo, se han aplicado recientemente en dominios gráficos como los juegos de Atari.

A pesar de su éxito, IW es un algoritmo puramente exploratorio y no aprovecha la información de recompensas anteriores. Además, requiere que el estado esté factorizado en características, que deben predefinirse para la tarea en concreto. Además, ejecutar el algoritmo con un width superior a 1 suele ser computacionalmente intratable en la práctica, lo que impide que IW resuelva problemas de width superior.

Empezamos esta tesis estudiando la complejidad de los métodos basados en width cuando el espacio de estados está definido por características multivalor, como en los problemas de RL, en lugar de átomos booleanos. Proporcionamos un límite superior más preciso en la cantidad de nodos expandidos por IW, así como resultados generales de complejidad algorítmica. Para hacer frente a problemas más complejos (es decir, aquellos con un width superior a 1), presentamos un algoritmo jerárquico que planifica en dos niveles de abstracción. El planificador de alto nivel utiliza características abstractas que se van descubriendo gradualmente a partir de decisiones de poda en el árbol de bajo nivel. Ilustramos este algoritmo en dominios PDDL de planificación clásica, así como en dominios de simuladores gráficos. En planificación clásica, mostramos cómo IW(1) en dos niveles de abstracción puede resolver problemas de width 2.

Para aprovechar la información de recompensas pasadas, incorporamos una política explícita en el mecanismo de selección de acciones. Nuestro método, llamado π-IW, intercala la planificación basada en width y el aprendizaje de la política usando las acciones visitadas por el planificador. Representamos la política con una red neuronal que, a su vez, se utiliza para guiar la planificación, reforzando así caminos prometedores. Además, la representación aprendida por la red neuronal se puede utilizar como características para el planificador sin degradar su rendimiento, eliminando así el requisito de usar características predefinidas. Comparamos π-IW con métodos anteriores basados en width y con AlphaZero, un método que también intercala planificación y aprendizaje, y mostramos que π-IW tiene un rendimiento superior en entornos simples. También mostramos que el algoritmo π-IW supera otros métodos basados en width en los juegos de Atari. Finalmente, mostramos que el IW jerárquico propuesto puede integrarse fácilmente con nuestro esquema de aprendizaje de la política, dando como resultado un algoritmo que supera a los planificadores no jerárquicos basados en IW en los juegos de Atari con recompensas distantes.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Futurists have always envisioned a world where artificial intelligent agents would relieve managers and professionals from realising certain tasks. Such autonomous agents focus on solving sequential decision making problems, where successive decisions are made in a dynamic environment in order to meet a certain goal or criteria. For instance, in the city of Barcelona, water is automatically distributed through the drainage network among different tanks, minimizing flooding in periods of heavy rain (García et al. 2015). A similar approach can be found in power grid management (Mohsenian-Rad et al. 2010), and wind energy conversion (Yaramasu and Wu 2016). Traffic light control (Arel et al. 2010), route planning in transportation networks (Bast et al. 2016; Flórez et al. 2011), battery usage (Fox, Long, and Magazzeni 2011), and resource management in computer clusters (Mao et al. 2016) are all examples of optimal decision strategies. Solutions to sequential decision making problems go far beyond system control. For instance, they have also been used in news recommendations (Zheng et al. 2018), to optimize chemical reactions (Zhou, Li, and Zare 2017), or to automatically design the physical layout of computer chips (Mirhoseini et al. 2021). The endless list of examples will definitely continue growing in the near future, for instance in the field of robotics (Kober, Bagnell, and Peters 2013; Levine et al. 2016) or with self-driving cars (Paden et al. 2016).

1.1 Planning and Learning

We distinguish two types of methods that provide solutions to sequential decision making problems, depending on whether the decision has been computed before or at decision time (Bertsekas 2019):

- *Offline* methods: where a decision is determined by a *policy* or decision strategy that has been learned from past experience, which can either be stored in memory or obtained by a fast function evaluation.
- *Online* methods: where a decision (and possibly the whole sequence of decisions) is computed at the moment it is required. In this case, the solution is usually computed within a certain computational budget, such as a time limit.

Learning methods fall within the first category: a policy is learned either from expert decisions, as in imitation learning (Guo et al. 2014; Ross, Gordon, and Bagnell 2011), or by step-by-step interaction with the environment, as in reinforcement learning (RL) (Sutton and Barto 2018). In the RL setting, the desirability of actions and their effects is usually encoded in a reward function, and the objective of the agent is to find a policy that maximizes some measure of expected cumulative future reward. Often, an estimate of this future reward measure, the value function, is learned alongside or as a substitute of the policy. RL algorithms are usually model-free, i.e., they do not assume a model of the dynamics or the reward function. Instead, they rely entirely on samples from these functions.

On the second category we find planning methods, which compute a sequence of actions that aim to drive the system from an initial to a goal state, ideally through the shortest path, by foreseeing and choosing from many possibilities. In that regard, and differently from RL methods, a model of the environment dynamics, actions, and goals is usually assumed to be known, and is often exploited to derive heuristics that help in solving problems with huge state spaces (Geffner and Bonet 2013).

During the last decade, a great deal of the progress in artificial intelligence have been on account of deep learning methods, which has been driven by an exponential increase in data availability and computational resources, combined with advances in powerful function approximation

techniques. Reinforcement learning has been no exception: it has experienced unprecedented success in the recent years by leveraging function approximation of key components (e.g., the policy or value function) on deep learning models (Arulkumaran et al. 2017; Li 2018). One advantage of such models is that they are trained end-to-end, producing automatic feature extraction in high-dimensional domains. This has allowed major achievements such as reaching human-level performance in Atari videogames (Mnih et al. 2015) or beating the world-champion in the ancient game of Go (Silver et al. 2016).

The policies produced by RL methods, including the ones in the form of deep learning models, are fast to evaluate but specific to a problem instance. On the other hand, planning methods are not tied to a specific instance, but they are slow because actions are computed online.

1.2 System 1 and System 2

The learning and planning approaches (or offline and online methods, in general) are akin to the two processes that are used in psychology to describe judgement and decision making: System 1 and System 2 (Kahneman 2011)[1]. System 1 is fast, unconscious and effortless, and provides answers from associations that have been constructed with experience, as well as from innate skills that we share with other animals. On the other hand, System 2 is slow, effortful, and requires attention. It provides with answers produced after an analytical process. The parallelism with planning and learning is direct: learned policies represent System 1, since they are based on associations from experience and can provide a fast answer. Contrarily, planners represent System 2 since they provide a slower, more "deliberate" solution by looking ahead, discerning between multiple possible answers.

"Thinking" is usually understood as the result of System 2. However, System 2 heavily relies on System 1 to develop a well-thought answer. The latter provides with impressions, intuitions, intentions and feelings that are used by System 2 as heuristics during the analytical process. At the same time, System 2 can change biases, misconceptions or mistakes

[1]The relation to System 1 and System 2 is borrowed from Hector Geffner's talk at IJCAI-ECAI 2018.

produced by System 1 up to some extent by rearranging associations. The division of labor between the two systems is highly efficient: it minimizes effort and optimizes performance.

Planning and learning have been mostly used separately as two different solutions to rather similar problems. However, planning alone with short computational limits may provide poor solutions. Similarly, relying entirely on a possibly inaccurate learned estimate may also result in bad decisions. The combination of the two: thinking ahead before taking a decision, and using learned heuristics and beliefs, seems a plausible approach to provide an informed answer, as the insights from the two system theory suggest.

A natural way to combine planning and learning is to treat the planner as a "teacher" that provides *correct* transitions that are used to learn a policy, as in imitation learning (Guo et al. 2014; Ross, Gordon, and Bagnell 2011). A prominent example of this approach is AlphaGo (Silver et al. 2016), which achieved superhuman performance in the game of Go by combining supervised learning from expert moves and self-play. AlphaZero (Silver et al. 2018), a version of the same algorithm that learns solely from self-play, has outperformed previous variants, also showing stunning results in Chess and Shogi (Silver et al. 2018). The latest version, MuZero (Schrittwieser et al. 2020), learns a model of the environment dynamics, and shows competitive performance with AlphaZero in the mentioned games as well as impressive results in the Atari benchmark. However, it fails to obtain a positive score in challenging sparse reward games such as Venture, Pitfall, or Montezuma's Revenge.

1.3 Sparse Rewards

Sparse or delayed reward tasks are those that require a long sequence of non-rewarded actions before a highly rewarded state is found. Finding such a long sequence of actions requires efficient exploration techniques, which is a central research topic in both planning and reinforcement learning.

One way to incentivize safe online exploration in RL is by adding an explicit bonus in the objective or the reward function. This approach is known under different names, e.g., reward shaping (Ng, Harada, and Rus-

sell 1999), optimism in the face of uncertainty (Kearns and Singh 2002), intrinsic motivation (Chentanez, Barto, and Singh 2005), curiosity-driven RL (Pathak et al. 2017; Still and Precup 2012), pseudo-counts (Bellemare et al. 2016), or entropy-regularized MDPs (Neu, Gómez, and Jonsson 2017). An alternative approach introduces noise directly in the parameter space of the learned policy or value function (Fortunato et al. 2018; Plappert et al. 2018). While these approaches offer significant improvements over classical exploration techniques such as ϵ-greedy or Boltzmann exploration, none of them makes explicit use of the representation of the state, which is treated as a black box.

Typical sparse reward tasks are those that are goal-oriented, with a reward function that does not provide any guidance toward the goal (e.g., the agent *only* receives a reward when achieving the goal). Planning methods precisely aim at finding a sequence of actions that lead the agent from an initial to a goal state. Current planners are able to solve problem instances involving huge state spaces by precisely exploiting the problem structure that is defined in the state-action model. However, many planning techniques highly rely on the state-action model to derive heuristics, and are therefore not akin to the RL setting.

1.4 Width-Based Planning

Iterated Width (IW) (Lipovetzky and Geffner 2012) is a search algorithm that makes use of the state space feature representation to perform structured exploration. The original IW algorithm consists of successive breadth-first searches in which states are pruned if they fail to meet a novelty criterion. In particular, IW(k) only considers k features at a time, and prunes those states for which all combinations of k features are made true in previously generated states. As a consequence, IW(k) runs in time and space that are exponential in k, but independent of the size of the state space.

In contrast with many planning approaches, width-based planners do not require a model of the internal state-action dynamics, and can be used with simulators (Frances et al. 2017). While originally proposed for classical planning problems, i.e., deterministic, goal-driven problems with a fully defined model, width-based planners have evolved closer to the RL

setting, and have been recently applied to the Arcade Learning Environment (Bellemare et al. 2013). First, the internal RAM state was used as features, achieving remarkable results (Lipovetzky, Ramirez, and Geffner 2015), and more recently a rollout version of the algorithm reached comparable results with learning methods in almost real-time, using pixel-based features (Bandres, Bonet, and Geffner 2018). These methods, however, are purely exploratory, and do not exploit past reward experience. The combination with learning methods could potentially redress this shortcoming.

In practice, IW is mostly used with $k = 1$ with complexity linear in the number of features (Bandres, Bonet, and Geffner 2018; Dittadi, Drachmann, and Bolander 2021; Geffner and Geffner 2015; Ramirez et al. 2018). In many challenging problems, even $k = 2$ with quadratic complexity is unfeasible (Geffner and Geffner 2015). Finding ways to run IW with an effective larger value of k while keeping a low complexity can further extend the applicability of this class of planners.

The use of hierarchies in planning has proven to be a very successful way for significantly reducing the computational cost of finding good plans. Traditional methods include Hierarchical Task Networks (Currie and Tate 1991; Erol, Hendler, and Nau 1996), macro-actions (Fikes, Hart, and Nilsson 1972; Korf 1985), and state abstraction methods (Knoblock 1990; Sacerdoti 1974). Hierarchical planning can lead to exponential gains in complexity by exploiting the structure of a problem involving a reduced subset of the state components.

1.5 Goals and Thesis Structure

The general aim of this thesis is to combine the power of structured exploration of width-based planning with a learning approach that provides information from past experience. We wish to apply such a combination to pixel-based, sparse reward tasks. We hypothesize that hierarchies can aid the search in tasks that suppose a major challenge.

The dissertation is structured in three main parts: Background (Chapters 2 and 3), Planning (Chapters 4 and 5), and Learning (Chapters 6 and 7). In the first part we introduce the methods on which our work is based on, as well as recent related work. In the second part, we detail our contri-

butions to width-based planning, without any learning involved. In the third part, we integrate width-based planning with policy learning, and assess our methods experimentally.

I Background

- In Chapter 2 we cover the background in planning from a reinforcement learning (RL) perspective. We define Markov decision processes (MDP) as general models for sequential decision making problems, followed by methods that provide solutions to them such as RL methods and Monte-Carlo Tree Search (MCTS), as well as the latest advances when combined with deep learning models (e.g., DQN and AlphaZero).
- In Chapter 3 we summarize previous work in width-based planning. We begin by describing the classical planning setting, connecting it to the MDP formulation. Then, we describe the basic Iterated Width (IW) algorithm as well as many subsequent improvements such as Best-First Width Search and Rollout IW.

II Planning

- In Chapter 4, we analyze the complexity of the IW(k) algorithm when the state space is represented by a set of multi-valued features (as it is common in MDPs), and relate the complexity results to the ones that appear in the literature, which assume that the state is represented by a set of atoms (prevalent in classical planning).
- In Chapter 5, we describe our hierarchical planning approach for blind-search methods. We detail the implications to the problem width when using IW at different levels of abstraction, and provide an algorithm that discovers high-level features incrementally. We show theoretically and experimentally in classical planning problems that the width of a problem can be reduced when the appropriate high-level features are used.

III Learning

- In Chapter 6, we present π-IW: an algorithm that combines width-based planning and learning. We learn a compact policy in the form

of a neural network that guides the search, resulting in an *informed* IW search. Moreover, we leverage the representation capabilities of deep learning to automatically extract features for IW, waiving the need to predefine them. We show that our approach outperforms MCTS methods in simple sparse reward domains as well as previous width-based planners in the Atari benchmark.

- In Chapter 7, we integrate our hierarchical and policy guidance approaches to tackle more complex tasks. We describe a new high-level width-based planner, and show how the hierarchical version is suitable for sparser reward tasks, achieving a positive score in the famous Atari game Montezuma's Revenge.

Finally, we present our conclusions and future perspectives in Chapter 8. The order in which chapters are presented, motivated by our separation of contributions into the two parts for greater clarity, is different to their chronological order. The work of this thesis has been presented in the following conferences:

- Preliminary results of Chapter 6: ICML / IJCAI / AAMAS 2018 Workshop on Planning and Learning (Junyent, Jonsson, and Gómez 2018)

- Chapter 6: International Conference on Automated Planning and Scheduling (ICAPS) 2019 (Junyent, Jonsson, and Gómez 2019)

- Chapters 4 (sec 4.1), 5 and 7: ICAPS 2021 (Junyent, Gómez, and Jonsson 2021)

Part I

Background

Chapter 2

Planning in Reinforcement Learning

2.1 Markov Decision Processes

Definition 1. *A Markov decision process (MDP) can be modeled as a tuple $M = \langle \mathcal{S}, \mathcal{A}, P, r \rangle$, where:*

- *$\mathcal{S}$ is a finite set of states,*
- *$\mathcal{A}$ is a finite set of actions,*
- *$P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is a transition function that maps state-action pairs to probability distributions over next states,*
- *$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function that maps states and actions to real-valued rewards.*

Here, we use $\Delta(\mathcal{X}) := \{\mu \in \mathbb{R}^{|\mathcal{X}|} : \sum_x \mu(x) = 1, \mu(x) \geq 0 \ (\forall x \in \mathcal{X})\}$ to denote the probability simplex over a set $\mathcal{X}$, i.e. the set of all probability distributions on $\mathcal{X}$.

At each time step t, the agent observes state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$, which leads the environment to a new state $s_{t+1} \in \mathcal{S}$, and the agent receives a reward $r_{t+1} \in \mathbb{R}$. The next state s_{t+1} is sampled following the probability distribution $P(s_t, a_t)$, while r_{t+1} is a sample from the underlying reward distribution such that $\mathbb{E}[r_{t+1}] = r(s_t, a_t)$. The state

and the reward at time step t are solely defined by functions of the state and action at the previous time step, known as the *Markov property*, and allows the agent to take decisions based on the current observed state.

The aim of the agent is to compute a decision strategy or *policy* $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, i.e. a mapping from states to probability distributions over actions, that maximizes some measure of expected future reward. The expected future reward associated with policy π is governed by a value function V^π, defined in each state s as

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \middle| S_0 = s \right]. \tag{2.1}$$

Here, S_t and A_t are random variables representing the state and action at time t, respectively, satisfying $A_t \sim \pi(S_t)$ and $S_{t+1} \sim P(S_t, A_t)$ for each $t \geq 0$, and $\gamma \in (0, 1]$ is a discount factor.

This value function considers the case of never-ending or *continuing tasks*, since the final time step is ∞. However, there are tasks with a natural ending, for instance when a goal is reached. In this case, it is common to run different instances of the same task iteratively, presumably with different initial conditions. We call each iteration an *episode* and such tasks *episodic tasks*. The states where the task ends are called *terminal states*, and in this dissertation we consider them to have value zero by definition. For the purpose of having a common notation with continuing tasks, terminal states are considered to be *absorbing*, i.e. states that only transition to themselves, generating rewards of zero.

A fundamental property of value functions, exploited by many RL algorithms, is that they can be expressed as a recursive function that depicts the relation between the value of a state and the values of its successors, known as the *Bellman equation*:

$$V^\pi(s) = \mathbb{E}_\pi \left[r(S_t, A_t) + \gamma V^\pi(S_{t+1}) \middle| S_t = s \right], \tag{2.2}$$

where, again, the expectation is on both the policy and the transition function, such that $A_t \sim \pi(S_t)$ and $S_{t+1} \sim P(S_t, A_t)$, and the infinite sum of Equation 2.1 can be obtained by unrolling Equation 2.2.

Many algorithms learn an *action-value function* $Q^\pi(s,a)$ instead, that denotes the value of taking an action a in state s under policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \middle| S_0 = s, A_0 = a \right] \tag{2.3}$$

$$= \mathbb{E}_\pi \left[r(S_t, A_t) + \gamma V(S_{t+1}) \middle| S_t = s, A_t = a \right]. \tag{2.4}$$

The *optimal value function* V^* is given by $V^*(s) = \max_\pi V^\pi(s)$ for all $s \in \mathcal{S}$, and the *optimal action-value function* Q^* is consequently specified by $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ for all state-action pairs $s \in \mathcal{S}$, $a \in \mathcal{A}$. The *optimal policy* π^* is then given by the argument achieving these maximums, i.e., $\pi^* = \arg\max_\pi V^\pi = \arg\max_\pi Q^\pi$.

The agent usually keeps an estimate of the optimal policy or value function (either V^* or Q^*). For large state spaces, it is common to define these estimates on a set of *variables* or *features* $\mathcal{F}$, instead of states. For simplicity, and without loss of generality, we assume that each feature has the same domain $\mathcal{D}$, e.g., binary ($\mathcal{D} = \{0,1\}$) or real-valued ($\mathcal{D} = \mathbb{R}$). States are mapped onto feature vectors using a function $\phi : S \to \mathcal{D}^{|\mathcal{F}|}$. For each feature $f \in \mathcal{F}$ and state $s \in \mathcal{S}$, let $\phi(s)[f] \in \mathcal{D}$ be the value that s assigns to f.

2.2 Models and Simulators

There is a variety of methods that provide solutions to MDPs. One way to classify these methods is by the amount of information that they require from the environment. For instance, we may need to know the transition probabilities and the reward function explicitly, or samples from these functions may be sufficient. We distinguish four different cases:

1. *Explicit models* or *distribution models* (Sutton and Barto 2018), where the transition function $P(s,a)$ and the reward function $r(s,a)$ are known.

2. *Generative models* or *simulators* (Kearns, Mansour, and Ng 2002), also called *sample models* (Sutton and Barto 2018), that given an

arbitrary state-action pair (s, a) return a randomly sampled next state and reward. Compared to the previous type, generative models are *black box* models, since the internal dynamics are unknown.

3. *Resettable simulators*, where at each step, the agent applies an action for the current simulator state, and receives in turn a randomly sampled next state and reward. Optionally, the state of the simulator can be restored to a previously visited state, from which the simulation continues. In that regard, a state is typically retrieved and saved for later use (e.g., a checkpoint in a video game). In contrast with generative simulators, the state cannot be arbitrary chosen.

4. *Episodic simulators*, where random samples of next states and rewards are generated sequentially, without the possibility of resetting the simulator state. The only requirement is that it should allow restarting (i.e., resetting the simulator state to an initial state to begin a new episode).

Note that each type requires less information from the environment. An explicit model can be turned into a generative model by sampling from the distributions, and calling a generative simulator repeatedly with the state produced at the previous time step yields a resettable simulator (since seen states can be reused). Finally, ignoring the possibility to reset states in a resettable simulator results in an episodic simulator. Thus, methods suitable for one type are also suitable for all previous types.

Methods that require a model (explicit or generative) are usually classified as *planning* methods, since the model can be used to look ahead and pre-compute a plan of action. For instance, *dynamic programming* algorithms use the transition and reward functions to compute the expectation in Equation 2.2, to find π^* and/or V^*. *Blind-search* methods do not require an explicit model or any domain information, but can still perform a lookahead by sampling from a generative or resettable simulator. Resettable simulators are mostly suitable for *rollout* algorithms, that produce entire trajectories to estimate V^π. Finally, *reinforcement learning* algorithms learn an estimate of the optimal policy or value function by interacting with the environment, and therefore only require an episodic simulator.

2.3 Reinforcement Learning

Many methods that provide solutions to MDPs fall within the scope of a *policy iteration* scheme, which is a general approach for learning the optimal policy or value function. It consists of two processes that are alternated until convergence: *policy evaluation*, that aims at making the value function consistent with the current policy, and *policy improvement*, which improves the policy by making it greedy with respect to the current value function. The original algorithm, in the context of dynamic programming, performs policy evaluation by iteratively applying the Bellman equation to all states in $\mathcal{S}$ until convergence. It thus requires an explicit model, since the transition probabilities and the true reward mean are needed to compute the expectation in Equation 2.2. The policy is then improved by defining it as the greedy action with respect to the current value function, for all states in $\mathcal{S}$, and the overall process restarts. Many methods, such as *value iteration*, improve $\pi(s)$ immediately after the value function has been updated for state s, instead of waiting for the policy evaluation step to converge to V^π. This is also the case for the methods we describe next. Apart from requiring a distributional model at hand, dynamic programming algorithms are computationally expensive, with a complexity that grows linearly in the number of states (and exponential in the number of features).

Reinforcement learning (RL) algorithms typically assume that the transition and reward functions are unknown to the agent, and rely on a simulator that provides samples of these functions. Moreover, although RL methods rely on the same principles as dynamic programming methods, they are generally more computationally efficient. A typical policy evaluation approach in value-based RL methods is to keep an estimate $\widehat{V}$ that is iteratively improved towards V^π by applying the update:

$$\widehat{V}(S_t) \leftarrow \widehat{V}(S_t) + \alpha_t \left[Y_t - \widehat{V}(S_t) \right], \tag{2.5}$$

where Y_t is the *target* made up of reward samples R_t (and often the value prediction of a future state, e.g., $\widehat{V}(S_{t+1})$) used to improve the prediction $\widehat{V}(S_t)$, and $\alpha_t \in [0, 1]$ is a learning rate parameter. The same update can be built for Q^π.

Monte-Carlo (MC) methods learn V^π by averaging sample *returns*.

The return Z_t is defined as the cumulative sum of rewards over the current trajectory, i.e., $Z_t = \sum_{t=0}^{\infty} \gamma^t R_t$. Thus, MC methods learn the value function approximating Equation 2.1 by replacing the expectation with the empirical mean, i.e., setting $Y_t = Z_t$ and applying updates with many instances of Z_t. On the other hand, *Temporal Difference* (TD) methods take advantage of the Bellman equation to update $\widehat{V}$. For instance, a one-step update uses $Y_t = R_{t+1} + \gamma \widehat{V}(S_{t+1})$ as target for $\widehat{V}(S_t)$, *bootstrapping* from the current value estimate at time step $t+1$. TD methods are typically preferred over MC methods since the update can be performed without waiting until the end of the episode. Nevertheless, the one-step TD target, while presenting a low variance, is biased towards the bootstrapped value. In contrast, MC methods are unbiased (since $V^\pi = \mathbb{E}_\pi [Z_t]$), but present a high variance.

Following a policy iteration scheme, the update in Equation 2.5 is generally combined with a policy improvement step by defining the policy toward greediness with respect to the value function. In that regard, the action-value Q^π is usually learned instead of V^π, since it allows to define the policy over the value function without requiring the transition function. This approach converges to π^* and Q^* if the following conditions are satisfied: all states are visited an infinite number of times, the policy converges in the limit to the greedy policy, and α_t is decayed appropriately, satisfying $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$ (e.g., $\alpha_t = 1/t$). A typical approach is to use a soft-greedy policy. For instance, the ϵ-greedy strategy selects a greedy action with probability $1 - \epsilon$ and a random action with probability ϵ, where ϵ can be decayed to zero.

The *Sarsa* algorithm applies the TD update defined above, with $Y_t = R_{t+1} + \gamma \widehat{Q}(S_{t+1}, A_{t+1})$, selecting actions with respect to $\widehat{Q}$ in a soft-greedy style. The algorithm can be extended to n-step returns, i.e., unrolling $\widehat{Q}(S_{t+1}, A_{t+1})$ for n steps, and using $\sum_i^{n-1} \left[\gamma^i R_{t+i}\right] + \gamma^n \widehat{Q}(S_{t+n}, A_{t+n})$ as target. Here, the parameter n can be thought of as a bias-variance tradeoff. A special case is Sarsa(λ), that uses a weighted average of n-step returns, according to an exponentially decaying parameter λ. Sarsa is an *on-policy* method, since it interacts with the environment with the same policy that it is learning. The *off-policy* version, called *Q-learning*, uses $R_t + \gamma \max_a \widehat{Q}(S_{t+1}, a)$ as target in order to improve the prediction from the current most valuable next action.

These methods store each action-value in a vector of $|\mathcal{S}| \times |\mathcal{A}|$ entries, which is usually unfeasible for environments with a large state or action space. In these cases, the state is often represented by a set of features $\mathcal{F}$, and the action-value function is approximated using a vector of weights φ that is combined with state feature valuations, producing an estimate of Q which we denote by $\widehat{Q}_\varphi$. The weights φ are then learned by gradient descent. For instance, one-step Sarsa uses the following update:

$$\varphi \leftarrow \varphi + \alpha_t \left[R_t + \gamma \widehat{Q}_\varphi(S_{t+1}, A_{t+1}) - \widehat{Q}_\varphi(S_t, A_t) \right] \nabla_\varphi \widehat{Q}_\varphi(S_t, A_t). \quad (2.6)$$

The gradient is only computed on the prediction $\widehat{Q}_\varphi(S_t, A_t)$, and not on the next state action-value, which is part of the target. Off-policy methods can also be used with function approximation, but they do not converge as robustly as on-policy learning, specially when approximated with a non-linear model. In Section 2.6 we describe DQN, an algorithm that effectively applies Q-learning with non-linear function approximation.

Policy gradient methods directly learn a parameterized policy. They are based on the policy gradient theorem, that states that:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\widehat{\pi}_\theta} \left[\nabla_\theta \log \widehat{\pi}_\theta(S_t, A_t) Q^{\widehat{\pi}_\theta}(S_t, A_t) \right] \quad (2.7)$$

where θ are the parameters of the policy, and J is the RL objective, e.g., $J = V^{\widehat{\pi}_\theta}(S_0)$ for the episodic setting. The theorem can also be generalized to include an arbitrary baseline $b(S_t)$, so that $\left(Q^{\widehat{\pi}_\theta}(S_t, A_t) - b(S_t)\right)$ is used as the policy performance measure, instead of $Q^{\widehat{\pi}_\theta}(S_t, A_t)$. When replacing the expectation with the empirical mean, different policy gradient algorithms emerge by using one estimate of $Q^{\widehat{\pi}_\theta}(S_t, A_t)$ or another. For instance, the REINFORCE algorithm uses the Monte-Carlo return Z_t, possibly truncated at some given horizon. Although actions are no longer based on the action-value estimate, a value function $\widehat{V}_\varphi$ may still be learned to evaluate the policy. This family of algorithms, known as *actor-critic methods*, uses a TD target as an estimate of $Q^{\widehat{\pi}_\theta}(S_t, A_t)$, for example the one-step version uses $R_t + \gamma \widehat{V}_\varphi(S_{t+1})$, or $R_t + \gamma \widehat{V}_\varphi(S_{t+1}) - \widehat{V}_\varphi(S_t)$ in case that the value function is used as the baseline. In off-policy learning, when samples are taken from a behavioral policy π_b (i.e., the one used to apply actions) that is different from $\widehat{\pi}_\theta$, the magnitude of $Q^{\widehat{\pi}_\theta}$ is usually corrected with an *importance sampling* factor of the form $\frac{1}{\pi_b(A_t|S_t)}$.

2.4 Online Replanning

The reinforcement learning methods presented so far rely on samples from a continuous stream of data, and are thus akin to episodic simulators. In this section, we present a family of methods that do not learn from experience, but instead select an action by looking ahead into the future, considering multiple paths, hence requiring a generative or resettable simulator. These methods are known as *planning at decision time* algorithms (Sutton and Barto 2018) or *online replanning* methods (Bertsekas 2019).

In online replanning, the policy $\pi(s)$ is computed *online for each state* $S_t = s$ at run time, i.e., most of the computation is performed just after the current state S_t is known. The process has an anytime behavior, i.e., it can be stopped at any time, although usually it is run until some computational budget is exhausted, and returns the best sequence of actions so far. Then, the first action is applied, the rest are usually discarded, and the process restarts from the next state S_{t+1}.

This procedure results in a *closed-loop* policy (since the plan is recomputed for each S_t), in contrast to regular planning, where the solution is a whole sequence of actions that are applied in an open-loop manner. Some algorithms compute a new plan at regular intervals, and apply more than one action from the computed sequence at each step in order to reduce the computation costs. This approach of selecting actions is slower compared to methods that use a previously learned compact policy, but works right out of the box, without requiring any training. These methods are highly related to Model Predictive Control (MPC) (Camacho and Alba 2013), where the control signal at a state S_t is optimized for a given model and time window. In MPC, however, we assume a *white-box* model at hand (e.g., the differential equations governing the dynamic system), to which different optimization methods can be applied.

Planning algorithms usually work with deterministic processes. This is the case for our work as well. Throughout this dissertation, we consider *deterministic* Markov Decision processes, where there is no randomness source in the transition and reward functions. Therefore, $P(s, a)$ is 1 for some next state s' and 0 at all other possible states $\mathcal{S} \setminus \{s'\}$, and r_{t+1} is equal to $r(s_t, a_t)$. In this case, the transition function is usually defined to directly map states into next states. We define such a function as $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Note that the stochastic setting could still be considered

by simply treating the sampling procedure as if it was encapsulated in T, such that $s' = T(s, a) \sim P(s, a)$.

2.4.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search is a best-first search method that aims to find the most promising action to take from the current state s (Browne et al. 2012). This is achieved by iteratively building a partial search tree to estimate the action values of s, which is represented by the root node. At each iteration, the most promising node is selected and expanded. The selection process is based on randomized exploration, being purely random at the beginning and becoming more informed at each iteration, based on statistics gathered from different simulations. An iteration consists of four steps:

1. Selection: Starting at the root node, which represents the current state of the environment, the search tree is traversed by successively selecting child nodes according to a *tree policy* (or selection policy). The traversal stops when a state-action pair (s, a) is encountered, such that the successor state $s' = T(s, a)$ is not in the tree. The tree policy only selects *expandable* nodes. A node is expandable if it is non-terminal and not all of its successors have been generated.

2. Expansion: A successor node belonging to an unexplored action is added to the tree. It is also possible to find implementations where more than one successor node is generated, or where the expansion step is performed at regular intervals.

3. Simulation: A simulation of the complete episode is run from the generated node, or from the selected node in case the expansion step is skipped, by applying actions according to a *rollout policy* (also called default policy). The states and actions visited by the rollout policy are only used to generate an outcome of the simulated episode (e.g. win/loss or cumulative reward), and therefore are not saved.

4. Backpropagation: The simulation result is backpropagated through the branch from the newly generated (or selected) node to the root.

The statistics of each node of the branch are updated, which will in turn inform future tree policy decisions.

The process continues until some computational budget is exhausted, at which point the most promising action for the root node is selected and applied in the environment. The selection criteria for this action may vary among different implementations, for instance we can choose the one with higher value, or the one with higher visit counts in order to avoid outliers. Then, the whole process restarts from the next state, which becomes the root node. Often, the tree from the previous search is reused by keeping the subtree containing all descendants from the applied action.

2.4.2 UCT

MCTS crucially relies on its tree policy to select promising nodes. In order to balance exploration and exploitation, Kocsis and Szepesvári (2006) proposed to use UCB1 (Auer, Cesa-Bianchi, and Fischer 2002), an upper confidence bound designed for multiarmed bandit problems, resulting in the Upper Confidence bounds applied to Trees (UCT) algorithm. In UCT, a child node is selected following:

$$\arg\max_{s'} \frac{W(s')}{N(s')} + 2C_p\sqrt{\frac{2\ln N(s)}{N(s')}},$$

evaluated on all possible next states $s' = T(s, a)$, for $a \in A(s)$. Here $N(s)$ is the number of times state s has been visited, and $W(s')$ is the sum of the rewards obtained from all simulated trajectories so far containing state s'. The left term encourages exploitation, since it is the average reward from s'. On the other hand, the right term encourages exploration of less visited nodes, which is controlled by a constant $C_p > 0$. $N(s') = 0$ yields a UCT value of ∞, which ensures that unvisited children nodes are considered at least once.

2.5 The Atari 2600 Benchmark

The Arcade Learning Environment (ALE) (Bellemare et al. 2013) has become a standard benchmark for comparing reinforcement learning algorithms. It consists of a collection of some of the Atari 2600 arcade

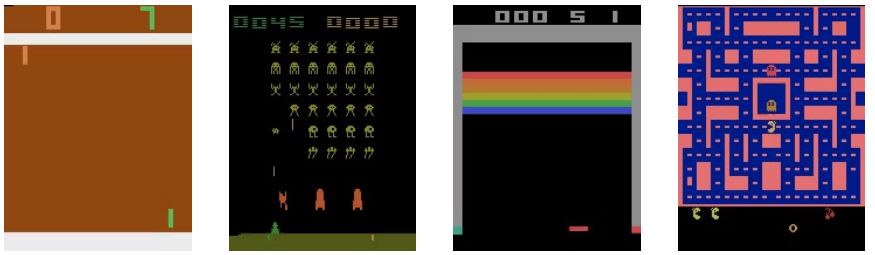

Figure 2.1: Screenshots of four Atari 2600 games. From left to right: Pong, Space Invaders, Breakout and Ms. Pac-man.

video games that became massively popular during the 1980s. Some well-known games include Ms. Pac-man, Pong, Breakout, and Space Invaders, for which we show example screenshots in Figure 2.5.

The game screen observation is 160 pixels wide and 210 pixels high, with 128 possible colors. There are 18 possible actions with deterministic effects: the combination of a two-axes joystick with three different positions per axis and a single button. The internal game state is represented by a RAM vector of 128 bytes that is also optionally accessible to the agent. ALE is a resettable simulator, since it allows to save and restore the internal state at any time, but has been mostly used as an episodic simulator. The platform is actually challenging for planning algorithms, where exhaustive search is prohibitive (e.g., one second of simulation, with 18 actions at 60 frames per second requires $18^{60} \approx 10^{75}$ steps).

Bellemare et al. (2013) present results for the learning and the planning settings. In the learning setting, they use Sarsa(λ) with linear function approximation, and propose several feature sets that are computed from the game screen. For instance, the *Basic* set of features encodes color positions in a subsampled grid of 16×14 tiles, without taking into account the game background. Other feature sets include pair-wise combinations of Basic features, position and velocity inference of objects, locally sensitive hashing, or the bits of the internal RAM state. In Table 2.1 we show results of Sarsa(λ) with Basic and RAM features in 5 games. In the planning setting, the authors provide results of BFS and UCT (columns 4-5). In the latter, duplicate states are detected and pruned. We observe that UCT clearly outperforms BFS and the learning algorithms, while

Game	Basic	RAM	BFS	UCT
Asterix	862.3	943.0	2135.7	290700.0
Beam Rider	929.4	729.8	693.5	6624.6
Freeway	11.3	19.1	0.0	0.4
Seaquest	579.0	593.7	288.0	5132.4
Space Invaders	203.6	226.5	112.2	2718.0

Table 2.1: Results of Sarsa(λ) with Basic and RAM features, BFS and UCT in 5 Atari games, from Bellemare et al. (2013).

using Basic or RAM features with Sarsa(λ) produce similar results. In a later work, Lipovetzky, Ramirez, and Geffner (2015) take into account the bit correlations and report better results using the RAM *bytes* as features in the planning setting (see Section 3.4).

ALE has become popular due to the rise of deep reinforcement learning methods, which we describe in the next section, where agents are trained end-to-end, i.e., the game screen is provided as input to the agent and features are automatically extracted. Because of this popularity, there have been some well established common practises (Machado et al. 2018). For instance, to reduce complexity, an action is applied repeatedly for some steps and all intermediate frames are ignored. This is known as *frameskipping*, and the frameskip value is determined by the amount of frames skipped minus 1 (i.e., with a frameskip of 1 no frame is actually skipped). The length of the episodes is usually limited to 18,000 frames (corresponding to 5 minutes of gameplay at 60 frames per second). In our work we use a frameskip of 15 and also limit the episode length to 18,000 frames. Other more interceding practices that we do not use in our work include ending the episode when losing a life (sometimes giving a negative reward) or using a specific subset of actions for a game. Finally, Henderson et al. (2018) report significantly different results of deep RL methods by varying random seeds, which suggest that some results may not be reliable or even that random seeds could be optimized.

There are many other popular benchmarks apart from ALE. For instance, Minecraft, where the agent has to navigate through a world made of 3D blocks (Guss et al. 2019). Typical goals are to find special blocks or move to a certain position. A similar game is Deepmind Lab (Beattie

et al. 2016), that consists of customizable 3D labyrinths. Another popular simulator is the Open Racing Car Simulator, known as TORCS (Wymann et al. 2014), where the reward is typically related to the speed that the agent achieves. In ViZDoom, the goal of the agent is to navigate the environment to find ammunition and shoot enemies (Kempka et al. 2016). Animal AI is a collection of tasks inspired by animal behaviors (Crosby et al. 2020). MuJoCo, Multi-Joint dynamics with Contact (Todorov, Erez, and Tassa 2012), is a physics simulator commonly used as a continuous action space environment, with popular tasks that range from the classic inverted pendulum to making a humanoid walk. Finally, OpenAI Gym is a widely used platform that collects some of these environments such as ALE, Doom and many other games (Brockman et al. 2016).

2.6 Deep Reinforcement Learning

The combination of reinforcement learning methods with deep learning models has led to unprecedented success in the recent years. Deep reinforcement learning methods (Arulkumaran et al. 2017; Li 2018) have reached super-human performance in many domains, such as the Atari games (Mnih et al. 2015), the game of Go (Silver et al. 2017) or in robotics (Levine et al. 2016). In the next section we briefly introduce the common building blocks of deep learning models.

2.6.1 Deep Learning

A feedforward deep neural network (Goodfellow, Bengio, and Courville 2016), or simply a neural network (NN), defines an input-output mapping. It is composed of an input layer, one or more hidden layers, and an output layer, where the depth stands for the amount of layers. Each layer consists of units that process the output of the previous layer. There is a variety of layer types, the most common being the *fully connected layer*, where each unit is connected to all units from the previous layer. The processing at each unit is a weighted sum of the unit outputs at which it is connected, plus a non-linear function such as logistic, tanh or the rectified linear unit (ReLU). The computation goes *forward*, from input to output, to produce a result that is compared with a desired output in an error or *loss*

function, which usually includes a *regularization* term to avoid overfitting to the training data. The gradient of the loss with respect to the weights is computed and flows *backward*, so that the weights are updated to minimize the loss function. Adaptive gradient descent techniques are usually used, such as *RMSProp*, *Adagrad* or *Adam*. Computations are usually performed in specialized hardware such as Graphic Processing Units (GPU) using automatic differentiation software, e.g., Tensorflow (Abadi et al. 2016) or PyTorch (Paszke et al. 2019).

Convolutional neural networks (CNN) are designed to process data with a grid-like topology, e.g., time-series (1D grids) or images (2D grids). They consist of *convolutional layers*, *pooling layers* and fully connected layers. Convolutional layer units perform a weighted average localized on a small part of the grid, by applying a *kernel* or *filter* whose weights are shared among units. Pooling layers perform similar operations (e.g., average or max) that are constant, i.e., without any learnable weight. *Recurrent neural networks* (RNN) are used to process sequential data such as speech and language. RNNs replicate the network structure for as many time-steps as necessary, sharing weights among all time-steps. *Long short term memory* (LSTM) layers consist of gating mechanisms that control the information flow through recurrent units.

Other types of layers are designed to make training more efficient. For example, *batch normalization layers* help to stabilize training by normalizing the output with the mean and variance of the current batch of inputs to the layer. *Residual networks* learn residual functions by including shortcut connections between layers, which serve to ease training of very deep networks.

Deep neural networks automatically learn representations from raw high-dimensional inputs, recovering patterns that compose different signals, such as edges or parts of objects in images. Deep learning models have demonstrated to be extremely successful, achieving state-of-the-art performance in many domains such as image classification and segmentation, object detection, text generation, translation, and many more, including reinforcement learning, which we contemplate in the next sections.

2.6.2 Deep Q-Learning

Q-learning with function approximation is known to be unstable or even to diverge (Baird 1995). This may become even more evident when the model is non-linear, such as a neural network. In particular, we can identify three problems when combining Q-learning and deep learning (Mnih et al. 2015):

1. High data correlation, since training data is generated sequentially by the agent-environment interaction.

2. Continuous changes in the data distribution, because the policy that generates the data is changing.

3. Using a non-constant target: the same network produces both the Q estimate and its target, which changes during training.

Mnih et al. (2013) overcomes the first problem by storing trajectories in a buffer D, called *experience replay*, so that at every training iteration, a randomly shuffled batch of samples is drawn. Later, Mnih et al. (2015) tackles 2-3 by using a stationary target. For that, a copy of the NN that approximates Q with parameters from a previous iteration is used to provide the target value. The algorithm then minimizes the following loss function:

$$\mathbb{E}_{(s,a,r,s',d)\sim U(D)}\left[\left(r+\gamma(1-d)\max_{a'}\widehat{Q}_{\theta^-}(s',a')-\widehat{Q}_{\theta}(s,a)\right)^2\right], \quad (2.8)$$

where d is the *done* signal that indicates whether s' is terminal, θ are the weights of the network representing the current action-value estimate, and θ^- are the weights of the target network, which are fixed to a previous snapshot of θ that gets updated at regular intervals.

This leads to human-level performance in many of the Atari 2600 games, training a different neural network for each game. The input to the network are the last four gray-scale frames, given as an image with four channels, fed into a convolutional neural network in order to capture spatio-temporal features such as the movement of the ball in "Pong" or "Breakout". The output is a vector representing the value of each action for the input state. We illustrate the neural network structure in Figure 2.2, which consists of convolutional layers followed by fully connected

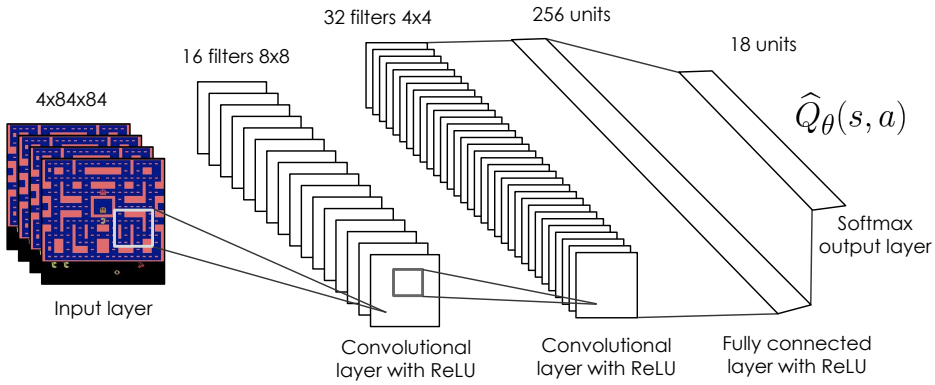

Figure 2.2: Representation of the neural network used in Mnih et al. (2013). The NN used in Mnih et al. (2015) contains an additional convolutional layer and an extra fully connected layer with 512 units.

layers. The authors used one-step Q-learning with an ϵ-greedy policy. The network was trained for 200 million frames in a GPU, taking about 8 days of computation, evaluating every million frames, and storing the best resulting parameters.

There are many follow-up papers that extend the DQN algorithm. *Double-DQN* (Van Hasselt, Guez, and Silver 2016) aims at correcting overoptimism in the action-value prediction with a target that selects the next action from the current Q estimate: $r+\gamma\widehat{Q}_{\theta^-}(s', \arg\max_{a'} \widehat{Q}_\theta(s', a'))$. The *dueling architecture* (Wang et al. 2016) separates the NN into two streams, one for predicting $V(s)$, and another for predicting the advantage $A(s,a)$ of each action, such that $Q(s,a) = V(s) + A(s,a)$. Schaul et al. (2016) draw the experience batches from the replay memory with different priority measures. To improve exploration, noise can be added to the parameters of the NN (Fortunato et al. 2018; Plappert et al. 2018). Performance is also boosted by learning the reward distribution instead of the mean (Bellemare, Dabney, and Munos 2017; Dabney et al. 2018a,b). All aforementioned techniques are combined into one single agent in *Rainbow* (Hessel et al. 2018). Hester et al. (2018) improves DQN with expert demonstrations. Finally, Hausknecht and Stone (2015) introduce an LSTM layer and show similar results to DQN with only one frame as in-

put. Kapturowski et al. (2018) further combines LSTM with distributed training in the *R2D2* method, which is later improved in *Agent57* (Puigdomènech Badia et al. 2020) by introducing UCB-based intrinsic motivation exploration, that led to super-human performance in all 57 Atari games for the first time.

DQN, as any other value-based method, relies on approximating the action-value function in order to derive a policy. Learning this function might be a lot more difficult than directly learning the policy. For instance, in robotic grasping, an agent may learn to close the end effector properly, while learning the value of different closings may be much complicated. Moreover, computing $\arg\max_a Q(s, a)$ with continuous or high-dimensional action spaces is a challenging task.

Many works use a deep learning model to directly learn the policy. *Trust region policy optimization* (TRPO) and *proximal policy optimization* (PPO) (Schulman et al. 2015, 2017) constrain a stochastic policy to change slowly by ensuring a small Kullback–Leibler divergence. A variant of TRPO with bias correction is applied in *actor-critic with experience replay* (ACER) (Wang et al. 2017). *Asynchronous advantage actor-critic*, or A3C (Mnih et al. 2016), maintains several parallel actors learning an entropy-regularized policy that make asynchronous updates to shared NN parameters. Acting and learning is separated in *IMPALA* (Espeholt et al. 2018), while *soft actor-critic* (SAC) (Haarnoja et al. 2018) incorporates policy entropy into the reward function. *Deep deterministic policy gradient* (DDPG) (Lillicrap et al. 2016) directly outputs the most promising action together with its value, which makes it suitable for continuous tasks. DDPG has also been extended to handle multiple agents (Lowe et al. 2017), to use parameter perturbation (Plappert et al. 2018), to learn the reward distribution (Barth-Maron et al. 2018), and to handle overestimation bias (Fujimoto, Hoof, and Meger 2018).

Finally, there are many other approaches that depart from model-free RL which also use deep learning models. For instance, I2A (Racanière et al. 2017), that learns a model of the environment, Go-Explore (Ecoffet et al. 2019, 2021), which makes use of a resettable simulator to explore previously abandoned but promising paths, or MCTS methods such as AlphaGo, AlphaZero and MuZero, described in Sections 2.7 and 2.8.

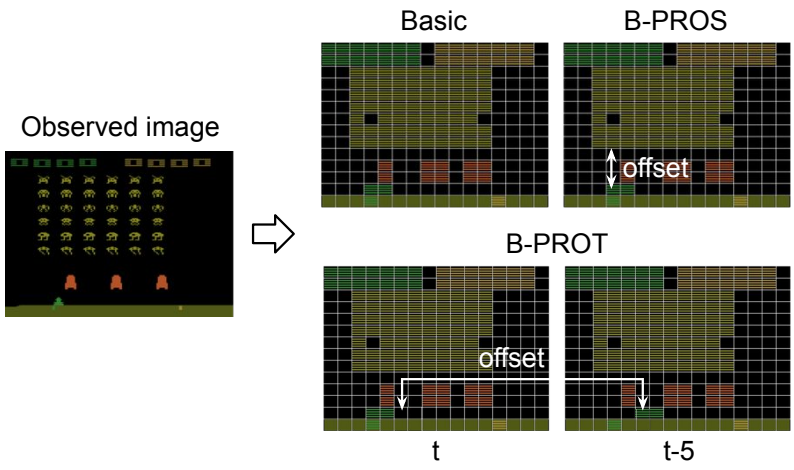

Figure 2.3: B-PROST features in the game of Space Invaders. Adapted from *The Success of DQN Explained by "Shallow" Reinforcement Learning, Amii News.*

2.6.3 Playing Atari with Shallow Reinforcement Learning

The neural network of DQN is trained end-to-end, i.e., features are automatically extracted from input images and are learned to maximize state-action values. Liang et al. (2016) aim at discerning which representational biases encoded in the network architecture are key to the success of DQN. They identify two main aspects: spatial invariance from the convolutional layers and the use of non-Markovian information induced by stacking several frames together. To produce a feature set with such characteristics, they extended the Basic features proposed by Bellemare et al. (2013), producing the B-PROST features.

The B-PROST features are computed as follows. First, the image background is removed from the current frame. The background of each game is pre-computed offline, from 18,000 observations collected from human played trajectories. Then, the image is divided in 16×14 tiles, from which three sets of binary features are extracted:

- Basic features: whether or not a color is present in a tile. Since the color palette consists of 128 colors, there are $16 \times 14 \times 128 = 28{,}672$

Basic features that can take values in $\{0, 1\}$.

- Basic Pair-wise Relative Offsets in Space (B-PROS): whether or not a pair of colors is present for each spatial relation between tiles. They are inspired by the convolutional operation, and intuitively they capture information like "there is a yellow pixel three tiles below a red pixel". There are $\left((16 \times 2 - 1) \times (14 \times 2 - 1) \times 128^2 - 128\right)/2 + 128 = 6{,}856{,}832$ B-PROS features after eliminating redundancy.

- Basic Pair-wise Relative Offsets in Time (B-PROT): whether or not a pair of colors is present for each spatial relation when comparing the current image with the one of the last decision point (e.g., five steps in the past). It captures the temporal relation introduced by the usual practise of stacking several frames and giving them as input to the neural network, with the intention to account for non-Markovian environments. There are $(16 \times 2 - 1) \times (14 \times 2 - 1) \times 128^2 = 13{,}713{,}408$ B-PROT features (in this case there are no redundant offsets).

Figure 2.3 shows the discretization of an image of the Space Invaders Atari game into the Basic, B-PROS and B-PROT features. The union of these three sets form the B-PROST feature set, which consists of a total of 20,598,848 binary features.

With the aim of detecting objects, Liang et al. (2016) defined another set of features called Blob-PROST, where groups of contiguous pixels (blobs) are considered, forming the spatial and temporal relations with the blob centroid points. With these enhanced features, the authors propose to use Sarsa(λ) with a linear model and an ϵ-greedy exploration strategy. The result is an algorithm that is competitive with DQN while using linear regression to encode the Q function.

2.7 AlphaGo: MCTS with Deep Neural Networks

Deep learning models have also been successfully exploited by MCTS methods. Silver et al. (2016) proposed to use different neural networks to reduce the effective depth and breadth of the search tree. In particular, a distinct neural network is used at each stage of the search:

1. Selection: a policy network p_σ is used to provide prior probabilities.

2. Expansion: a rather small linear policy network p_τ decides which action to expand.

3. Simulation: a similar network p_π is used for fast rollout evaluations.

4. Backpropagation: a value network v_θ also evaluates leaf nodes.

The networks p_σ and v_θ have the same structure, with the only difference being that the last layer produces a single prediction value for v_θ, and a probability distribution for p_σ, where the latter consists in a softmax layer with as many outputs as possible actions. The input for both networks consists of 49 binary feature planes of size 19×19 resulting from preprocessing the state. The board positions are encoded into 3 feature planes, corresponding to the player stones, the opponent stones and empty slots. The other 46 planes correspond to specific game information such as how many opponent stones would be captured, how many turns have passed since a move was played, or whether or not a basic known sequence of moves can be successfully performed.

The network structure of p_σ consists of 12 convolutional hidden layers with 192 filters and a ReLU as activation function, the first layer with a filter size of 5×5 and the rest with 3×3, and a convolutional output layer with a 1×1 filter followed by a softmax function. The value function has 2 extra convolutional hidden layers and an extra fully connected layer with 256 rectifier units. The output layer is a fully connected layer with a single tanh unit.

The rollout policy p_π and the tree policy p_τ are smaller, linear networks with different inputs than the feature planes described above. Compared to p_σ, the rollout policy p_π has a lower accuracy predicting expert moves (24.2% vs 55.7% after training) but produces a result faster (2μs vs 3ms). The input to the rollout policy p_π consists of 109,747 handcrafted binary features that encode local information of specific stone patterns centered on the given action, involving the previous and current moves. These features also encode other information such as whether a move saves stones from being captured or not. The tree policy p_τ receives the same binary patterns as p_π plus 32,242 extra patterns, and both networks output a

probability distribution over all possible actions produced by a softmax layer.

2.7.1 Training Pipeline

The training pipeline of AlphaGo consists of three phases. First, the policy network p_σ is trained through supervised learning (SL) to classify board positions from a dataset of expert moves. The dataset consists of 29.4 million moves extracted from 160,000 expert player games, that are augmented by considering all eight reflections and rotations. Training is done through stochastic gradient ascent by maximizing the likelihood of selecting move a in state s:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}.$$

The smaller policy p_π is trained similarly from the 8 million positions. The second phase consists in improving the policy network p_σ by policy gradient reinforcement learning (RL). At the beginning of this phase, a copy of p_σ is made, generating a new neural network with the same structure p_ρ, where the parameters ρ are initialized to σ, i.e., the ones resulting from the SL phase. The reason is that p_σ is later used in MCTS for action selection, while the improved version p_ρ is used to train the value network in the third phase. The second phase then consists in playing games between the current policy network p_ρ and the one of a randomly selected previous iteration, which prevents overfitting the current policy. Weights ρ are trained through stochastic gradient ascent to maximize the expected outcome:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t,$$

where z_t is the terminal reward at the end of the game ($+1$ for winning and -1 for losing).

Finally, the last phase of the training pipeline involves learning to predict the outcome z of the policy network p_ρ. According to Silver et al. (2016), the reason for not learning to predict the value from the dataset of expert moves is that positions of the same game share the same outcome z while being highly correlated. The authors report that v_θ trained from this dataset resulted in overfitting. Instead, they generated a new dataset

with 30 new million positions by self-play using p_ρ, each position extracted from a different game.

The value network is trained by gradient descent to minimize the mean squared error (MSE) between the outcome z and the corresponding predicted value $v_\theta(s)$:

$$\Delta\rho \propto \frac{\partial v_\theta(s)}{\partial \rho}(z - v_\theta(s)).$$

2.7.2 Search

Each edge of MCTS, corresponding to taking an action a at a state s, stores a set of statistics $\{P, N_v, N_r, W_v, W_r, Q\}$, where $P(s,a)$ is the prior probability of taking action a in state s, and $W_v(s,a)$ and $W_r(s,a)$ are the sum of Monte-Carlo action-value estimates over $N_v(s,a)$ and $N_r(s,a)$ leaf evaluations, respectively. Specifically, W_v and N_v are evaluations from the value network v_θ, while W_r and N_r are evaluations from running the rollout policy p_π. The reason for having two separate counts N_v and N_r is that W_v and W_r are updated asynchronously. To exploit Go symmetries, the prior $P(s,a)$ is initialized by applying a randomly chosen reflection or rotation d in a dihedral group of 8 symmetries before evaluating the policy network, such that $P(s,a) = d^{-1}(p_\sigma(a \mid d(s)))$. At each time step t, MCTS selects nodes according to $a_t = \arg\max_a Q(s_t,a) + u(s_t,a)$, where $u(s_t,a)$ is computed using a variant of the pUCT algorithm (Rosin 2011):

$$Q(s,a) = (1-\lambda)\frac{W_v(s,a)}{N_v(s,a)} + \lambda\frac{W_r(s,a)}{N_r(s,a)},$$
$$u(s,a) = c_{\text{puct}}P(s,a)\frac{\sqrt{\sum_b N_r(s,b)}}{1+N_r(s,a)},$$

with $Q(s_t,a)$ being an average of the outcomes resulting from rollouts and evaluations of the value network, weigthed by a parameter λ. During rollouts, actions are chosen using both the rollout policy p_π and a hash table that accumulates moves played by MCTS.

As previously mentioned, MCTS is implemented in an asynchronous distributed manner. There is a master machine that runs the main search, many remote CPUs running rollouts, and many GPUs evaluating the policy and value networks p_σ and v_θ. Note that the RL policy network p_ρ

is only used for policy evaluation, and is not used in the search, where the SL policy network p_σ is used instead. However, the outcome of p_ρ is approximated by the value network v_θ. Finally, the expansion step of MCTS is only performed at regular intervals, when the visit count $N_r(s, a)$ exceeds a given threshold.

Once the search budget is exhausted, AlphaGo selects the action with maximum visit count, and the search tree is reused for subsequent time steps, i.e., the child node corresponding to the played action becomes the new root node, and the subtree below this child node is kept, retaining all its statistics, while the rest is discarded. AlphaGo resigns if $\max_a Q(s, a) < -0.8$, corresponding to an estimated 10% probability of winning the game.

2.8 Policy Iteration MCTS

In this section, we describe three relevant works. First, Silver et al. (2017) presented AlphaGo Zero, an algorithm that defeats its predecesor AlphaGo and is trained solely by self-play, without any information from expert players. Then, Silver et al. (2018) further generalized the AlphaGo Zero approach, removing handcrafted, domain-specific knowledge to produce a competitive algorithm in many domains, called AlphaZero. Finally, Schrittwieser et al. (2020) presents a version of the algorithm that is suitable for episodic simulators, where MCTS is run on a learned state-action model.

2.8.1 AlphaGo Zero

AlphaGo distinguishes two phases: a training phase, where the policy and value models are pretrained using expert data and reinforcement learning, and a search phase, were the the models are exploited to guide the search. AlphaGo Zero (Silver et al. 2017) mixes the two phases instead, so that the policy and value estimates are continuously improved with the experience generated by MCTS, and does not use any expert data. In particular, AlphaGo Zero uses a single neural network with two outputs that, given a state, produces both the policy and the value estimates (p, v) such that $(d^{-1}(p), v) = f_\theta(d(s))$, where d is, once more, a randomly sampled symmetry function. Then, p and v are used to aid MCTS in

a similar manner to AlphaGo. A key difference, though, is that the parameters θ are randomly initialized and improved from the data produced by the tree search. MCTS outputs action probabilities π that result in stronger moves than the ones of p. MCTS can then be seen as a *policy improvement operator*, while the value output by f_θ can be interpreted as a *policy evaluation operator*. These operators are then used repeatedly in a policy iteration procedure: the neural network parameters θ are updated to match more closely the improved action probabilities and the win/loss outcome (π, z) produced by the search. More precisely, f_θ is trained by stochastic gradient descent to minimize the loss function:

$$\mathcal{L} = (z - v)^2 - \pi^T \log p + c_{\ell 2} \|\theta\|^2$$

where ℓ is the loss function over a single datapoint (π, z) and the gradient is actually computed over a minibatch of policy and value targets, sampled from an experience replay buffer. Once more, the experience replay buffer is augmented to take into account all 8 symmetries.

MCTS is also simpler than its predecessor: there are no rollouts. Instead, AlphaGo Zero relies on its value function entirely to evaluate leaf nodes. Thus, each time a new state s is generated, $f_\theta(s)$ is evaluated producing the value estimate, that is backed up along the trajectory. The set of statistics is simpler in this case: $\{P, N, W, Q\}$. The selection step is similar to AlphaGo, selecting actions according to:

$$\begin{aligned} a_t &= \arg\max_a Q(s_t, a) + u(s_t, a), \\ Q(s, a) &= \frac{W(s, a)}{N(s, a)}, \\ u(s, a) &= c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}. \end{aligned}$$

Other differences with AlphaGo include that leaf nodes are always expanded (instead of when a certain visit threshold is reached) and that threads are synchronised, waiting for the neural network evaluation instead of performing asynchronous evaluation and backups. Moreover, additional exploration is achieved by adding Dirichlet noise to the prior probabilities in the root node s_0, $P(s_0, a) = (1 - \epsilon)p_a + \epsilon n_a$ where $n \sim \text{Dir}(0.03)$ and $\epsilon = 0.25$. Finally, AlphaGo Zero resigns if both the root value and its

best child value are lower than a certain threshold. The neural network structure is also different from AlphaGo. It consists of 40 residual blocks each composed by two ReLU convolutional layers of 256 filters with batch normalization, followed by a shortcut connection that adds the input to the block output. The network is finally divided into two heads, one for the value and another for the policy, with 3 and 2 additional layers, respectively.

When the computational budget is consumed, the action with highest visit counts is selected according to $\pi(a \mid s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$, similar to AlphaGo. However, for the first 30 moves, τ is set to 1 to ensure that a diverse set of positions is selected. Then it is changed to a small constant, $\tau \to 0$, to select the most visited action for the rest of the game.

AlphaGo Zero is trained by self-play against its best-performing neural network parameters from all previous iterations. After 1,000 steps, the current network is evaluated by playing 400 games against the previous best network, using $\tau \to 0$. If the current network wins by a margin of at least 55%, then it becomes the best network and is subsequently used for self-play. The network that defeated its predecessor AlphaGo, winning $100 - 0$, was trained during 40 days.

2.8.2 AlphaZero

AlphaZero (Silver et al. 2018) is a generalization of AlphaGo Zero to other two-player domains that results in a simpler approach, removing domain-specific knowledge. For instance, board positions are no longer transformed using rotation or reflection symmetries, either to augment the training dataset or before they are evaluated by the neural network. AlphaZero demonstrates state-of-the-art performance in the game of Go compared to a 3-day training version of its predecessor, AlphaGo Zero, as well as in Shogi and Chess compared to the world-champion programs Elmo and Stockfish.

The input features to the neural network is a binary tensor of size $N \times N \times (MT + L)$: a concatenation of T sets of M planes of size $N \times N$ representing the board positions of the last T time steps, with M planes per time step, one per piece type. The additional L planes denote the player's color, total move count, and other special rules. The output representing the policy also consists of a stack of $N \times N$ planes that varies

among games. For instance, the policy in chess is represented by a $8 \times 8 \times 73$ tensor encoding the probability distribution over 4672 possible moves.

Its predecessor AlphaGo Zero is trained by self-play against its best-performing neural network parameters from all previous iterations. This requires an evaluation step to determine whether the current neural network outperforms the best-performing version by a sufficient margin. AlphaZero, instead, maintains a single neural network which is updated continually, and self-play games are generated using the latest parameters of the neural network, removing the evaluation step to select the best parameters. The result is a simpler algorithm that can be applied to a variety of games.

2.8.3 MuZero

The MCTS methods presented so far consider that either an explicit model or a resettable simulator is available. Schrittwieser et al. (2020) further generalizes AlphaZero to episodic simulators, i.e., to simulators without the capacity to rewind to a previously seen state, by learning a model of the environment. To avoid any confusion, in this section we use *observation* to refer to the state produced by the actual simulator, which need to be distinguished from the hidden states produced by the model, that we denote by *state embedding*, or simply by *embedding*. The resulting algorithm, called MuZero, relies on a recurrent neural network that produces modelled trajectories given two arrays: the observations produced so far by the simulator, and the actions that we wish to try in the model. There is a single neural network, with parameters θ, that can be decomposed into three functions:

- An encoder h_θ, that produces a state embedding from an array of observations.
- A dynamics model g_θ, that maps state embeddings and actions to next state embeddings and rewards.
- An actor-critic model f_θ, that maps a state embedding into a policy and a value estimate.

A representation of the neural network with its three functions is given in Figure 2.4. The functions are related as follows. First, the array of past

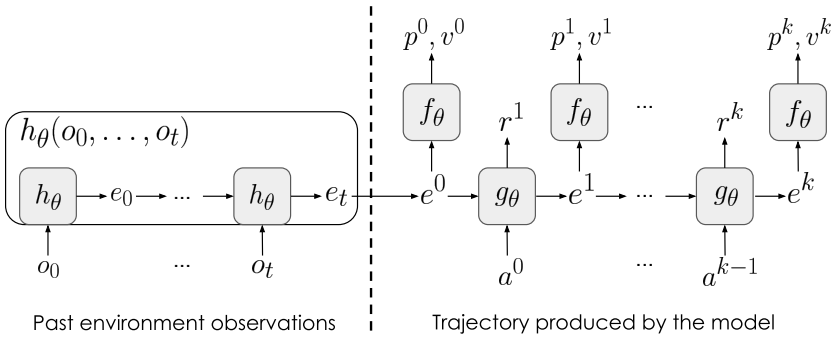

Figure 2.4: MuZero recurrent neural network representation. Information flows from bottom to top (output sequence) and from left to right (recurrent state). The depth of the model trajectory is given by supperscript numbers, and are to be distinguished from the time step of interaction with the simulator (denoted by an subscript). For simplicity, subscripts t in the model trajectory are omitted.

observations is encoded into an embedding $e_t = h_\theta(o_0, \ldots, o_t)$ using the encoder network, where $o_0, \ldots, o_t$ are the observations up to the current time step t. Then, the embedding e_t is used to initialize the recurrent state of the dynamics network g_θ. At this point, g_θ is ready to produce a sequence of rewards and next state embeddings given the sequence of actions resulting from the MCTS selection phase. For each generated embedding, the actor-critic model f_θ is evaluated to produce a policy and value estimate, that will be in turn used in the action selection process, similarly to AlphaZero. Note that the embeddings generated correspond to the recurrent state of the overall neural network.

MCTS keeps statistics $\{N(s,a), Q(s,a), P(s,a), R(s,a), S(s,a)\}$, where $s^{k+1} = S(s^k, a^k)$ and $r^{k+1} = R(s^k, a^k)$ are the state and reward computed using g_θ, and N, Q, P are the visit counts, the average value, and the policy, respectively, computed similarly as in AlphaZero. Differently from previous works, Q values are normalized before being used in the action selection phase, where an action a^k is selected according to $\arg\max_a [Q(s,a) + U(s,a)]$, with the exploration factor $U(s,a)$ being:

$$U(s,a) = P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \left(c_1 + \log \left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right).$$

Instead of producing a scalar for the value and the reward, the neural network output consists in a softmax layer that produces a vector of size 601, where each position of the vector represents a discrete support for every integer in $[-300, 300]$. A scalar x is then represented by a vector filled with zeros except in two positions: the two adjacent supports $\lfloor x \rfloor$ and $\lceil x \rceil$, that contain values $p \in [0,1]$ and $(1-p)$, respectively, such that $x = p \cdot \lfloor x \rfloor + (1-p) \lceil x \rceil$. Here, $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ and $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ are the floor and ceil functions, that map a real-valued number x to the highest integer that is lower or equal to x, and to the lowest integer that is higher or equal to x, respectively. The output vectors of the neural network for the value and the reward are linearly combined with the integer vector $(-300, \ldots, 300)$ to produce the scalars r^k and v^k. Similarly, the targets for the value and the reward are converted to their vector representations before their loss functions are evaluated. The loss function for a data-point collected at time step t is as follows:

$$\mathcal{L}_t(\theta) = \sum_{k=0}^{K} \left(\pi_{t+k}^\top \log p_t^k + \psi(y_{t+k})^\top \log v_t^k + \psi(u_{t+k})^\top \log r_t^k \right) + c_{\ell 2} \|\theta\|,$$

where y_t can be either the n-step or Monte-Carlo return, and $\phi(\cdot)$ is the vector mapping described above. Here, we follow the notation of Schrittwieser et al. (2020) and denote the reward received from the environment by u_t to distinguish it from the reward r^k generated by the model.

MuZero is trained and executed in a massively parallel computing environment: several actors playing against many instances of the same environment sharing the neural network parameters and replay buffer. The results reported show that, after 1M training steps, MuZero reaches the performance of AlphaZero in Chess, Shogi and Go, and clearly outperforms the model-free method R2D2 (Kapturowski et al. 2018) in Atari.

Chapter 3

Width-Based Planning

3.1 The Classical Planning Model

In this section, we start by defining the classical planning model (Geffner and Bonet 2013), which we then encapsulate into a deterministic Markov decision process. Finally, we describe the standard factored representation for classical planning, the STRIPS model, and its generalization to multivalued features.

Definition 2. *A classical planning model is defined by the tuple* $\mathcal{C} = \langle \mathcal{S}, \mathcal{A}, s_0, \mathcal{S}_G, A, T, c \rangle$*, where:*

- $\mathcal{S}$ *is a finite set of states,*
- $\mathcal{A}$ *is a finite set of actions,*
- $s_0 \in \mathcal{S}$ *is an initial state,*
- $\mathcal{S}_G \subseteq \mathcal{S}$ *is a set of goal states,*
- $A(s) \subseteq \mathcal{A}$ *is a set of applicable actions in each state* $s \in \mathcal{S}$*,*
- T *is a deterministic transition function that maps state-action pairs to next states:* $s' = T(s, a)$*,* $a \in A(s)$*,*

Starting from the initial state s_0, the goal of the planner is to generate a plan $\Pi = a_0, a_1, \ldots, a_{n-1}$, consisting of a sequence of actions that generate

a state trajectory $s_1, \ldots, s_n$ such that $s_{i+1} = T(s_i, a_i)$ for all $i = 0, \ldots, n-1$, and $s_n \in \mathcal{S}_G$. The plan Π represents an open-loop controller that solves the classical planning problem, i.e., the solution consists of a fixed sequence of actions that is computed once, where actions do not depend on the observed states as in the closed-loop policies seen in the previous chapter. Optionally, classical planning problems can have a non-negative function $c(s, a)$ representing the cost of applying action a in state s. If no such function is provided, actions are assumed to have unit cost. The solution is Π *optimal* if it represents the shortest path to achieve the goal, i.e., if the cumulative cost $\sum_i c(s_i, a_i)$ is minimized.

3.1.1 Classical Planning Problems as MDPs

We can use deterministic Markov Decision Processes to model goal-directed planning tasks. For that, we need to draw a correspondence from the elements of $\mathcal{C}$ to the elements of the tuple that defines a deterministic MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, r \rangle$. Note that we have reused notation for those elements with a direct equivalence, i.e., the state space $\mathcal{S}$, the action space $\mathcal{A}$, the initial state s_0 and the deterministic transition function T. To model the reward function, we choose to define it as follows:

$$r(s, a) = \begin{cases} 1, & \text{if } s' = T(s, a) \in \mathcal{S}_G \\ 0, & \text{otherwise} \end{cases}$$

In words, a reward is received only when a transition to a goal state occurs. Here, action costs are considered to be 0, and in order to ensure that the shortest path is the one with maximum cumulative sum of rewards, we define the discount factor γ to be strictly lower than 1. Furthermore, we make each goal state $s_G \in \mathcal{S}_G$ absorbing by defining the transition function at each s_G as $T(s_G, a) = s_G$ for each action $a \in A(s_G)$. Hence an optimal policy attempts to reach any goal state as quickly as possible and then stay there.

In MDPs, the size of the state space is usually too large to be represented explicitly, and a set of features $\mathcal{F}$ is typically used to represent it. This is also the case for classical planning, where it is common to use a factored representation, which we detail next.

3.1.2 Factored Representation

One of the most simple and common representations is known as STRIPS (Fikes and Nilsson 1971), where the state space is factored into Boolean variables, called *predicates*, *atoms*, *facts*, or *fluents*, that express whether a proposition about the environment holds in a given state.

Definition 3. *A planning problem in STRIPS is defined as a tuple* $P = \langle \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ *where*

- $\mathcal{P}$ *is a finite set of atoms,*
- $\mathcal{A}$ *is a finite set of operators or actions,*
- $\mathcal{I} \subseteq \mathcal{P}$ *represents the initial state,*
- $\mathcal{G} \subseteq \mathcal{P}$ *represents the goal.*

A classical planning problem is encoded in STRIPS as follows. A state s is represented by a non-empty subset of atoms in $\mathcal{P}$, i.e., propositions about the environment that are *true*, while atoms that are not in s are assumed to be *false*. The state space $\mathcal{S}$ then becomes the possible collections of atoms over $\mathcal{P}$. Hence, there are $2^{|\mathcal{P}|} - 1$ possible states, where $|\mathcal{P}|$ is the number of atoms of the problem, considering that a state contains at least one atom. The initial state s_0 is encoded by atoms in $\mathcal{I}$, and the set of goal states is induced by the partial state $\mathcal{G}$, such that $\mathcal{S}_G = \{s \mid \mathcal{G} \subseteq s\}$.

Each action $a \in \mathcal{A}$ is represented by three sets of atoms over $\mathcal{P}$, called *Precondition*, *Add*, and *Delete* lists, which are denoted by *Pre(a)*, *Add(a)*, and *Del(a)*, respectively. The applicable actions in a state s, $a \in A(s)$, are the ones in $\mathcal{A}$ with $Pre(a) \subseteq s$. The state transition function is defined as $s' = T(s, a) = (s \setminus Del(a)) \cup Add(a)$, so that atoms in $Del(a)$ are deleted from s and atoms in $Add(a)$ are added, producing the next state s'.

The Planning Domain Definition Language (PDDL) (McDermott et al. 1998), employed in the International Planning Competitions (IPC) (McDermott 2000), is a language reminiscent of Lisp that has been widely used to encode STRIPS problems. Problems in PDDL consist of two parts: a *domain* and an *instance* file. The domain defines the actions using *schemas*, which describe the preconditions and effects over generic variables, that are later replaced with specific objects. On the other hand,

the instance describes these specific objects, as well as the atoms in $\mathcal{I}$ and $\mathcal{G}$, that represent the initial state and the set of goal states, respectively.

For MDPs, we defined a factorization of the state space using a set of features $\mathcal{F}$ with an arbitrary domain size $|\mathcal{D}|$. In STRIPS we can also accommodate variables that are not Boolean. In general, if $f \in \mathcal{F}$ is a multivalued variable or feature with domain $\mathcal{D}$, we can define propositions such as "$f = v$" for each value $v \in \mathcal{D}$. This extension is used in other formulations, e.g., in SAS$^+$ (Bäckström and Nebel 1995). The set of atoms is then defined as $\mathcal{P} = \mathcal{F} \times \mathcal{D}$. Note that in this case each state contains *exactly* $|\mathcal{F}|$ atoms, instead of being a collection of atoms of arbitrary size. Here we use a common domain $\mathcal{D}$ for simplicity, which can be finite or not, but we can extend the formulation to the case where each feature f has a different domain $\mathcal{D}_f$ without loss of generality.

Note that this formulation also holds for binary domains, where we can explicitly indicate the values "$f = true$" or "$f = false$" for each variable. In fact, there are many extensions of STRIPS that accommodate negative atoms, e.g., propositional STRIPS with negative atoms (PSN) (Bäckström 1995). Let us denote the multivalued features formulation with propositions of the type "$f = v$" as $\mathcal{P}_{\mathcal{F}}$, and use $\mathcal{P}_{true}$ for the regular binary formulation where the state is a collection of true propositions, and other missing atoms are assumed to be false. For binary domains, we distinguish these two formulations explicitly because using one or the other will affect the output of our algorithms (see Section 3.2) and their complexity (see Chapter 4). In the next section we describe Iterated Width, a search algorithm that makes use of the state factorization to structure the search.

3.2 Iterated Width

Iterated Width (IW) is a forward search algorithm originally developed for goal-directed planning problems with deterministic actions (Lipovetzky and Geffner 2012). It requires the state space to be factored into a set of atoms $\mathcal{P}$, and exploits the structure of the state space defined by $\mathcal{P}$ to explore efficiently. Here, we consider the generic formulation $\mathcal{P}_{\mathcal{F}}$ defined above, and for simplicity assume that the set of atoms is built from multivalued features $\mathcal{F}$ with a common domain $\mathcal{D}$, e.g., binary ($\mathcal{D} = \{0, 1\}$), integer ($\mathcal{D} = \mathbb{Z}$) or real-valued ($\mathcal{D} = \mathbb{R}$).

The IW algorithm consists of a sequence of calls to the procedures IW(k), for $k = 0, 1, 2, \ldots$, until the problem is solved or k exceeds $|\mathcal{F}|$. Each call IW(k) performs a standard breadth-first search (BrFS) from a given initial state s_0, pruning states that are not *novel*. The variable k is called the *width* of the breadth-first search.

Definition 4. *A state s is considered novel for width k if any tuple of size k of atoms in s appears for the first time in the search.*

Hence, when a new state s is generated, IW(k) contemplates all k-tuples of atoms in s and, if all the tuples appear in previously generated states, the state s is pruned. Otherwise, it is added to the OPEN list, i.e., the list of nodes to be expanded. The breadth-first search thus expands a number of nodes that is exponential in the width parameter k. Note that for $k = 0$ no state is considered novel, and IW(0) only solves a problem if the initial state is a goal state. We next define the novelty of a state, which is a quantity induced from the previous definition:

Definition 5. *The novelty of a state s, denoted as $w(s)$, is the minimum k that makes s novel in an IW(k) search, for $k = 1, \ldots, |\mathcal{F}|$. If s is not novel for any k, then $w(s) = |\mathcal{F}| + 1$.*

Although $w(s)$ may be a quantity of interest, it can be expensive to compute since it requires to iterate through all feature tuples of size $1, ..., |\mathcal{F}|$ until one that makes the state novel is found. In practice, either $w(s)$ is computed until a maximum value k, and the novelty of states that are not novel for such k is set to $|\mathcal{F}|+1$, or it is not computed at all. IW(k) usually checks whether the state is novel or not for the current k, i.e., we are interested in $novel(s, k) = (w(s) <= k)$ where $novel(s, k)$ is either true or false, without computing the exact novelty measure $w(s)$. Note that all iterations to check tuples of lower size than k can be skipped, since any tuple of atoms is contained in another tuple of larger size. Thus, a state that is novel for width i will also be novel for width k, for $i < k$. This notion of state novelty has been extended in many follow-up works (Bandres, Bonet, and Geffner 2018; Lipovetzky and Geffner 2017a; Shleyfman, Tuisov, and Domshlak 2016), as we describe in the next sections.

Example. In Figure 3.1 we illustrate IW(k) using a small example that involves three binary features (i.e. $\mathcal{D} = \{0, 1\}$) and four actions. Here, we

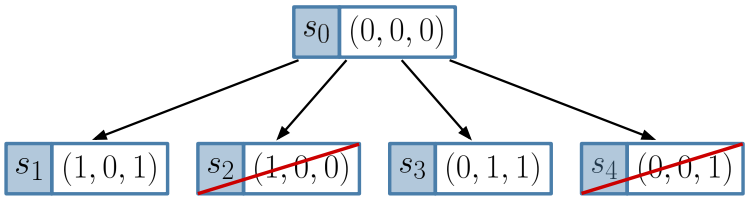

Figure 3.1: Example run of IW(1) at depth 1. States, expanded left to right, are represented by their feature vectors, and actions correspond to edges.

show an IW(1) execution until depth 1, i.e., only the initial state s_0 is expanded, which is represented by the feature vector $(f_0, f_1, f_2) = (0, 0, 0)$. Assume that breadth-first search expands the states in the order left-to-right. Since $k = 1$, we look at whether individual feature values appear for the first time in the search. State s_1 generates two new feature values: f_0 and f_2 have value 1, and therefore the state is not pruned. State s_2 does not generate any new feature values, and is thus pruned. State s_3 is novel since it assigns 1 to f_1 for the first time, while s_4 is pruned. The algorithm would continue expanding the nodes that have not been pruned in a breath-first manner until all nodes are pruned or the goal state is reached. If we run IW(2) instead, state s_2 would not be pruned, since the tuple $(f_0 = 1, f_2 = 0)$ appears for the first time in the search. When considering width $k = 3$, no states would be pruned since all triplets of features are different. In general, IW(k) is equivalent to a BrFS with duplicate state detection when $k = |\mathcal{F}|$.

The pruning mechanism of IW is directly affected by how the state space is factorized. For instance, in the previous example with binary features we have considered the STRIPS formulation for multivalued variables $\mathcal{P}_{\mathcal{F}}$. This formulation with $\mathcal{D} = \{0, 1\}$ actually generates a different IW(k) search tree than when considering the regular formulation for binary domains $\mathcal{P}_{true}$. The reason is that, when using $\mathcal{P}_{\mathcal{F}}$, features that are false also contribute to the novelty test, which is not the case in $\mathcal{P}_{true}$. As a side note, the example shown in Figure 3.1 cannot be represented by $\mathcal{P}_{true}$ as is, since the root node would be the empty set of atoms, which is not allowed by the STRIPS formulation. $\mathcal{P}_{true}$ is used in most works

where IW is applied to classical planning problems, since they are defined in PDDL. In this work, we only consider $\mathcal{P}_{true}$ when the environment is defined in PDDL (as in Chapter 5), and the multivalued variable formulation otherwise, even if the features are binary. In Chapter 4, we discuss the implications in algorithmic complexity of using one formulation or the other.

The IW algorithm is *sound* and *complete*, i.e., it outputs a valid solution if there is one. However, the resulting plan may not be *optimal*, i.e., it may not be the shortest path. Nevertheless, IW(k) is guaranteed to be optimal for a certain value of $k = w$. This value w is called the *problem width*, which we describe in the next section. The reason why IW does not always output an optimal solution is because the sequence of calls to IW(k) may solve the problem (optimally or not) before reaching $k = w$.

3.2.1 Problem Width

IW(k) eventually traverses the entire state space when k is large enough. The traversal depends on two components:

- The set of features $\mathcal{F}$ that defines how the states are structured.
- An ordering $O(s)$ that defines in which order the available actions $A(s)$ are taken for each state s that is being expanded. This ordering is usually random, producing different results for different IW(k) calls.

As previously mentioned, whether or not IW(k) is guaranteed to solve a given problem P optimally depends on the underlying notion of problem width that we define next. Here, we borrow the definition from Bonet and Geffner (2021), which is based on tuple sequences, instead of using the original one that is defined over graphs of tuples (Lipovetzky and Geffner 2012).

Definition 6. *The width of a solvable problem P, denoted as $w(P)$, is the minimum k for which there is a sequence of atom tuples $t_0, t_1, \ldots, t_m$, each with at most k atoms, such that:*

1. *t_0 is true in the initial state of P*

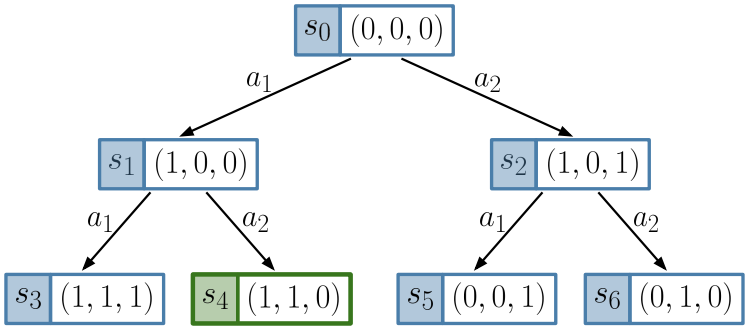

Figure 3.2: Example of a full tree expanded as in BrFS with three binary features and two actions to illustrate the notion of width. The goal state is s_4.

2. *any optimal plan for t_i can be extended into an optimal plan for t_{i+1} by adding a single action, $i = 1, \ldots, m-1$*

3. *any optimal plan for t_m is an optimal plan for P*

If P is unsolvable, then $w(P) = |\mathcal{F}| + 1$.

The tuples t_i that satisfy the above conditions (and therefore are part of an optimal trajectory) can be thought of stepping stones or subgoals that need to be achieved at each step. The maximum size of these tuples, $w(P)$, bounds the size of the search space, and therefore $w(P)$ can be considered a measure of problem complexity. For problems whose width is bounded by k, IW(k) is guaranteed to find an optimal solution (i.e. the shortest path), if there exists one, in time that is exponential in k.

It is important to note that the definition above requires that *any* optimal plan for t_i should be part of an optimal plan for t_j, where $i < j$. Take the example of Figure 3.2, that shows states represented by their feature vectors expanded by BrFS, and consider width $k = 1$. We look at a sequence of tuples (in this case single atoms, since $k = 1$) of the trajectory that leads to the goal state, given by the optimal plan $\Pi = \langle a_1, a_2 \rangle$. Take the atom $f_1 = 1$, which is the one that changes from s_0 to s_1. At depth 1, both states s_1 and s_2 appear with $f_1 = 1$, and therefore there are two optimal plans for reaching such atom: $\Pi_1 = a_1$ and $\Pi_2 = a_2$. However,

only Π_1 is part of Π, and the second condition is therefore not satisfied. In fact, this problem has width 2, and the tuple $(f_1 = 1, f_3 = 0)$ satisfies the condition. Note that the optimal plan Π can be obtained from an IW(1) search, with ordering $O(s) = \langle a_1, a_2 \rangle$, i.e. for a given state s, a_1 will be applied before a_2. However, IW(1) with the order of actions reversed will prune s_1 and will not solve the problem.

We could thus define $w(P)$ as the minimum k for which P can be solved by IW(k) *in all possible orderings* $O(s)$ used to apply actions when expanding each state s. Let us define the *effective* width, that may vary among IW searches depending on the ordering used in the search (which is usually random):

Definition 7. *The effective width of a problem P, denoted as $w_e(P)$, is the minimum k for which P can be solved by a specific IW search, i.e., by the procedures IW(k) for $k = 1, 2, \ldots$, with a specific ordering O.*

The effective width is then the minimum state novelty required to solve the problem for a specific IW search. While obtaining the width of a problem is hard, the effective width can be a good approximation of the actual width, and can be obtained by running IW. Note that $w_e(P)$ is actually a lower bound on the actual width $w(P)$, since an IW(k) search may solve a problem P while $k < w(P)$. Interestingly, most classical planning domains present a low width, at least when their goal consists of a single atom, and in practice can be solved in linear or quadratic time.

3.2.2 Width of Single-Atom Goal Problems

Lipovetzky and Geffner (2012) showed that most classical planning problems turn out to have a very small width when the goal consists of a single atom. The authors actually proved that the domains Blocks, Logistics and n-puzzle have a bounded width, independent of the problem size and initial state when considering one goal atom. They also experimentally showed that, in practice, most benchmarks present an effective width of 1 or 2 when $|G| = 1$. The experiment consisted in applying IW(1) and IW(2) in single-atom goal instances of 37 benchmarks from the International Planning Competitions (IPC), prior to 2012. To generate single-atom goal instances, each IPC instance with G goal atoms was divided into G instances with one goal atom, producing a total of 37,921

Effective width	# Domains	# Inst.	Inst. IW(1)	Inst. IW(2)
$w_e(D) = 1$	4	24,382	100%	100%
$w_e(D) = 2$	22	9,916	30.6%	100%
$w_e(D) > 2$	11	3,623	26.8%	60.5%
Total	37	37,921	37.0%	88.2%

Table 3.1: IW results in single-atom goal benchmarks (summary of Table 1 of Lipovetzky and Geffner (2012)). Domains are classified by their effective width, i.e., according to the minimum k for which IW(k) solves all the given instances of the domain in this experiment. Most of the instances of $w_e(D) = 1$ come from the domain VisitAll (21,859).

instances. The authors reported the amount of instances solved by IW(k) with $k = 1$ and $k = 2$, as well as the amount of unsolved instances for which $k > 2$ is necessary.

We show a summary of the results in Table 3.1. Here, we classify each domain D by whether all its instances are solved by $IW(1)$, $IW(2)$ or whether it requires $k > 2$. We refer to this quantity as the effective width of the domain, $w_e(D)$, for the provided set of domain instances for this experiment. We observe that IW(1) is able to solve all instances in 4 domains, while IW(2) remarkably solves 26 out of 37 domains. Moreover, IW(1) solves more than a quarter of instances from domains with $w_e > 1$. For IW(2), the percentage of instances solved from domains with higher effective width grows to 60.5%. In general, IW(2) can solve almost 9 out of 10 instances, which means that we only go beyond quadratic time in 11.8% of the cases.

3.2.3 Serialized IW

The results in single atom goal instances, where most domains present a low width, suggest that the complexity of classical planning benchmarks comes from the conjunction of goal atoms. Serialized IW (SIW) is a hill climbing algorithm that consists of a series of IW searches. At each iteration i, SIW runs an IW search until at least one new goal atom is true. It keeps a set $\mathcal{G}_i$ of goal atoms that have been achieved such that, at each iteration, a new goal atom is achieved while preserving the ones obtained in previous iterations: $\mathcal{G}_i \subset \mathcal{G}_{i+1} \subseteq \mathcal{G}$, starting with $\mathcal{G}_0 = \emptyset$. The initial

state for the IW search at each iteration i becomes the state where $\mathcal{G}_{i-1}$ is achieved, i.e., when at least one new goal atom is achieved in a state $s_{\mathcal{G}_i}$, SIW starts a new IW search at iteration $i+1$ with $s_{\mathcal{G}_i}$ as the initial state. The initial state at iteration $i=1$ is $s_{\mathcal{G}_0} = s_0$, and SIW stops when all goal atoms have been achieved, i.e., when $\mathcal{G}_i = \mathcal{G}$.

There are several variations of SIW which include using a counter of goal atoms still to be achieved $\#g(s) = |\{p \mid p \in \mathcal{G} \setminus s\}|$, and whenever a new minimum of $\#g(s)$ is reached, a new IW search starts (Bonet and Geffner 2021). Note that in this version there is no requirement of $\mathcal{G}_i \subset \mathcal{G}_{i+1}$, but rather $|\mathcal{G}_i| < |\mathcal{G}_{i+1}|$. In contrast with IW, SIW is incomplete, since the subgoals need to be achieved monotonically, and even if there is such a goal-monotonic path, SIW may not be able to find it because it is greedy, i.e., it commits to the first newly achieved goal atom not present in the current $\mathcal{G}_i$. Nevertheless, experimental results show that SIW is competitive with planners that exploit goal information, like the ones we describe in the next section.

3.3 Best-First Width Search: Beyond Pure Exploration

Although Iterated Width performs remarkably well in problems where the goal consists of a single atom, it does not match the performance of state-of-the-art planners in instances with $|\mathcal{G}| > 1$. This is mainly because IW is a pure exploration method that does not make use of goal information and does not exploit the state-action model.

Best-first search (BFS) algorithms explore a graph by expanding the most promising node at each time, according to an evaluation function. When the evaluation function is given by a heuristic, i.e., an approximation of the cost to the goal, we refer to such algorithms as *greedy* best-first search methods (GBFS). Usually, several heuristics are taken into consideration with lexicographic preferences, e.g., if we choose nodes according to heuristics $\langle h_1, h_2, h_3 \rangle$, then the node with the lowest value of h_1 will be selected, breaking ties with the value of h_2, and then with h_3.

In classical planning, state-of-the-art planners use heuristics derived from the STRIPS state-action model definition. Some examples of heuristics that have proved to be successful are given next:

- The additive heuristic h_{add} (Bonet and Geffner 2001), where two functions are defined, $c_x(s, x)$ and $c_{\mathcal{X}}(s, \mathcal{X})$, that represent the cost to achieve an atom x and a set of atoms $\mathcal{X} = \{x_1, ..., x_m\}$, respectively, from state s. The core assumption is that the cost of reaching a set of atoms $\mathcal{X}$ is estimated as the sum of the costs to achieve each atom in $\mathcal{X}$, i.e., $c_{\mathcal{X}}(s, \mathcal{X}) = \sum_{i=1}^{m} c_x(s, x_i)$, and each individual atom cost $c_x(s, x)$ is in turn recursively computed using $c_{\mathcal{X}}$. Then, the heuristic is defined as the estimated cost to get to the set of atoms representing the goal state, $h_{add} = c_{\mathcal{X}}(s, \mathcal{G})$.

- The relaxed planning graph heuristic h_{ff} (Hoffmann and Nebel 2001), where a delete-free relaxed plan is computed, i.e., a plan Π_d where the delete effects $\text{Del}(o)$ of each operator o are ignored, and therefore each state s in the relaxed problem accumulates all previously seen atoms in the trajectory from s_0 to s. Then, the cost to the goal is estimated by the number of actions of the computed plan, $h_{\text{ff}} = |\Pi_d|$.

- The landmarks heuristic, h_L (Richter, Helmert, and Westphal 2008), that computes a set of landmarks or subgoals from the relaxed plan, and estimates the goal distance by the number of landmarks left.

In order to provide a planner that competes with state-of-the-art planners, Lipovetzky and Geffner (2012) presented a GBFS algorithm that combines the novelty measure with goal-directed heuristics. Note that BFS using state novelty $w(s)$ as an evaluation function is equivalent to IW, since it expands nodes with novelty 1 first, then novelty 2, and so on, while avoiding the repeated work of each IW(k). Nevertheless, as we discussed before, computing $w(s)$ may be expensive, and it is usually computed up to a certain k.

In particular, Lipovetzky and Geffner (2012) use as heuristic a linear combination of state novelty with an indicator function of whether the action leading to such state is *helpful*. Helpful actions are the ones that are useful to reach a goal atom in the relaxed plan (Hoffmann and Nebel 2001). Ties are broken with the number of goal atoms left, denoted by $\#g(s)$, and then with h_{add}. The result is a planner that achieves state-of-the-art performance.

The possibilities of using state novelty as a heuristic are further explored in Lipovetzky and Geffner (2017a), focusing on GBFS algorithms

that make use of state novelty as a heuristic, which they call Best-First Width Search (BFWS) methods. They empirically show in classical planning benchmarks that using the novelty measure as the main heuristic consistently increases the planner performance (in this case, without combining it with helpful actions). For instance, BFWS($\langle w, h_{add} \rangle$) and BFWS($\langle w, h_{\text{ff}} \rangle$) solve roughly twice as many instances, in less time (on average), as their GBFS versions (i.e. using h_{add} and h_{ff} alone, without any novelty measure). Here, the evaluation order of the heuristic functions is left to right.

Lipovetzky and Geffner (2017a) extend the notion of novelty to handle subproblems defined by heuristic valuations (or by any given collection of functions) as follows.

Definition 8. *Given a set of functions $h_1, \dots h_m$, a state s is novel for width w if there exists a tuple t that appears for the first time among the previously generated states s' that have the same function values, i.e., with $h_i(s) = h_i(s')$ for $i = 1, \dots m$.*

We use $w_{h_1,\dots h_m}$ to explicitly denote that the novelty measure considers the subspace induced by the values of functions $h_1, \dots, h_m$. For instance, IW(k) with $w_{\#g}$ would need to keep a different novelty table for each set of states with the same number of goal atoms, and would prune states with all k-tuples of features appearing in previously seen states that have the same number of goal atoms.

Note that the set of functions used in the novelty measure can be different from the list of heuristic functions f taken into account by BFWS(f). In fact, one of the best performing planners in Lipovetzky and Geffner (2017a) is BFWS(f_5), where $f_5 = \langle w_{\#r,\#g}, \#g \rangle$. Here $\#g(s)$ and $\#r(s)$ are two counters representing the number of atoms of sets $\mathcal{G}$ and $\mathcal{R}$ that are contained in s, respectively. $\mathcal{G}$ is the set of goal atoms, and $\mathcal{R}$ is a set of *relevant* atoms that capture potential subgoals not explicit in $\mathcal{G}$. More precisely, $\mathcal{R}$ is the union of all atoms collected along the last relaxed plan that has been computed. In BFWS(f_5), a new relaxed plan is computed every time the goal counter decreases.

BFWS(f_5) matches the performance of state-of-the-art planners, while a more sophisticated version, Dual-BFWS, clearly outperforms all previous planners. As the name suggests, it consists of two consecutive searches.

First, an incomplete version of BFWS(f_5) that prunes nodes with novelty greater than 1. Then a BFWS(f_4), with $f_4 = \langle w_{h_L,h_{\text{ff}}}, h_L, h_{\text{ff}} \rangle$. The incomplete version of BFWS(f_5) is further explored in Lipovetzky and Geffner (2017b).

Frances et al. (2017) generalized BFWS(f_5) by considering other sets of relevant atoms $\mathcal{R}$ that do not require to compute a relaxed plan, and that do not require the state-action model, in general. Their best-performing planner uses a set of atoms $\mathcal{R} = \mathcal{R}^*_G$ extracted from an initial IW(1) search. For those problems where it is not too computationally expensive, IW(2) is run instead. Specifically, $\mathcal{R}^*_G$ is the set of atoms collected from all the trajectories in the IW(1) (or IW(2)) search tree that lead to states containing at least one goal atom. The counter $\#r$ is then computed from $\mathcal{R}^*_G$, and the novelty test considers the subspace induced by both $\#r$ and $\#g$. Following BFWS(f_5), the search selects nodes according to the heuristic $w_{\#r,\#g}$ breaking ties with $\#g$.

The resulting planner, which they call BFWS($\mathcal{R}^*_G$), compares well with BFWS(f_5). Importantly, it does not require the state-action model at any time, and is appropriate to be used with simulators. This approach, however, cannot be directly used in the MDP setting, since it requires to know the goal representation in advance to compute the heuristic $\mathcal{R}^*_G$.

3.4 Width-Based Planning in MDPs

Several extensions of the original IW algorithm have been developed over the last years, as the ones described in the previous section. One line of research has focused on the popular Atari 2600 benchmark (Bellemare et al. 2013), predominantly in the online replanning setting, where actions are selected after a lookahead.

Lipovetzky, Ramirez, and Geffner (2015) extended the original IW algorithm to deterministic MDPs by associating a reward $R(s)$ to each state s during search, equivalent to the reward $\sum_{t=0}^{d-1} \gamma^t r_{t+1}$ accumulated on the path from s_0 to $S_d = s$, where d is the depth of s in the search tree. Here, the discount factor γ has the effect of favoring earlier rewards. After the search completes, the state s^* with highest cumulative reward $R(s^*)$ that has been generated but not pruned by IW is identified. Then, the first action on the path from s_0 to s^* is applied, similarly to MCTS.

Instead of dealing with the game image, the internal state of the ALE simulator is used to represent the state, namely the RAM bytes, resulting in $|\mathcal{F}| = 128$ features with domain size $|\mathcal{D}| = 256$. The authors state that the bit correlations are important, and that using the individual bits instead ($|\mathcal{F}| = 1024$, $|\mathcal{D}| = 2$) would result in poor results for IW(1).

Their results show that IW(1) outperforms UCT in 31 out of 54 games. Both planners use a planning horizon of 150,000 frames, distributed in 500 rollouts of depth 300 in the case of UCT, and are restricted to a maximum search depth of 1,500 nodes. In IW(1), however, a frameskip of 5 is used, resulting in a budget of $30,000$ nodes, whereas in UCT no frame is skipped (i.e., frameskip of 1; each frame is assigned to one node). The maximum episode duration for both algorithms is 18,000 frames as is common practice in ALE.

All algorithms reuse frames generated in previous searches, i.e., the subtree of the previous lookahead rooted at the selected child s' is kept, while all siblings of s' and their descendants are discarded. This is common practice in online replanning for deterministic problems, which we refer to as *subtree caching*. Such reused frames are not counted in the budget for the subsequent search. In addition, cached states are ignored for the computation of the novelty of new states, i.e., we start each search with an empty novelty table, that is not re-initialized with the atom tuples of the cached states.

Shleyfman, Tuisov, and Domshlak (2016) introduced prioritized-IW (p-IW), modifying the online replanning version of IW in two ways: changing the novelty measure, and using a novelty queue that breaks ties with rewards. Their novelty measure is as follows:

Definition 9. *A state s is considered novel for width k if there exists a tuple of atoms t of size k in s such that the cumulative reward $R(s)$ is higher than the one previously recorded for t.*

Thus, in this case, the novelty table keeps the highest cumulative reward so far, denoted as $\hat{r}(t)$, for each possible tuple of atoms t (initially $\hat{r}(t) = -\infty$ for all tuples t), and considers a state novel if $R(s) > \hat{r}(t)$. Then, it updates the novelty table with $\hat{r}(t) = \max\{R(s), \hat{r}(t)\}$. With this modification, the cumulative reward directly affects the pruning mechanism, ensuring that p-IW(k) will never prune a state where $R(s)$ is the highest seen so far, which may be the case for IW(k).

The second modification involves the order in which states in the OPEN queue are selected for expansion. Instead of using a first in first out (FIFO) queue, p-IW(k) uses a priority queue, where states are expanded by depth (to maintain the BrFS expansion) and ties are broken by cumulative reward. Jinnai and Fukunaga (2017) further extended p-IW by learning to avoid actions that lead to the same successor state. Both considerably outperform IW in the RAM setting of the Atari suite.

The literature revised so far contemplates a deterministic setting, which is also the case in our work. To the best of our knowledge, there are no works where width-based planning is applied to the stochastic version of the Atari games (Machado et al. 2018). Geffner and Geffner (2015), however, present results of IW in the General Video-Game AI competition (GVG-AI) benchmarks, where the state space is represented by an engineered set of atoms $\mathcal{P}$, and actions consist of procedures with stochastic outcomes. To handle stochasticity, the possible actions at each state are applied many times and labelled as either *safe* or *unsafe*. Safe actions are those that have minimum risk of the avatar dying when applied. Unsafe actions are then considered as not applicable.

Apart from IW(1) and IW(2), the authors also present results in a mixture between the two, which they call IW(3/2), that consists in contemplating tuples of size 2 only for a subset of atoms. Specifically, they consider all pairs of atoms where one is a proposition about the avatar (i.e., the agent). The results show that IW(1) clearly outperforms other algorithms such as MCTS, even with a very small time budget (40ms). When a slightly larger budget is allowed (300ms or 1s), IW(3/2) performs the best in a subset of the games, scaling better than IW(2), similar to IW(1). In the next section, we describe an extension of IW that is suitable for very small planning horizons.

3.5 Rollout IW

Bandres, Bonet, and Geffner (2018) introduced a Rollout version of IW that is equivalent to IW but presents a better anytime behavior in the presence of sparse rewards, i.e., it produces a similar result to that of IW without time limitations, and returns a better result when interrupted promptly. The algorithm constructs the IW search tree in a depth-first

manner, by repeatedly generating trajectories. However, it maintains the width notion by keeping track of the minimum depth at which each tuple of features is found, and generalizes the notion of novelty accordingly.

Definition 10. *A newly generated state s is considered novel for width k if any tuple of size k of atoms in s appears at a lower depth than previously recorded.*

The main difference with IW is that only one action is applied in each state, generating trajectories instead of fully expanding nodes in a breadth-first manner. Instead of keeping an OPEN list of nodes that need to be expanded, the algorithm repeatedly traverses the search tree, revisiting partially expanded nodes, and generating successor nodes when necessary. In order to only keep expanding non-pruned nodes, and therefore maintain the width notion, the algorithm distinguishes between two types of nodes in the search tree: *solved* and *unsolved* nodes. Solved nodes are those that do not need to be visited anymore, either because they have been pruned or because all of their successors have already been generated and solved, i.e., all actions have been applied and lead to solved successor nodes.

The tree traversal is done by choosing actions at random among those that do not lead to solved nodes. At each step, the simulator is called depending on whether or not the successor for the chosen action at a given node is already present in the tree. In order to prune nodes that are not novel, a novelty table D keeps track of the depths at which all tuples of features are encountered. We then compare the depth d of a node n to all its feature tuples of size w, and if d is lower than the depths $D[t]$ previously recorded at the novelty table for some tuple t, then n is considered novel. When revisiting unsolved nodes of the search tree, we also need to check that they are still novel before continuing the traversal. In that case, we will find a tuple t that has depth *equal* to $D[f]$. The original algorithm distinguishes four cases to determine whether a rollout continues or not, given a node n with depth d (Bandres, Bonet, and Geffner 2018):

- Case 1: if node n is new in the tree and makes some tuple t of features in n true, with $d < D[t]$, update novelty table and **continue** rollout.
- Case 2: if node n is new in the tree but $d \geq D[t]$ for each tuple t of features in n, **terminate** rollout.

Algorithm 3.1 Rollout IW(k)

```
function LOOKAHEAD(tree, k)
    Initialize_labels(tree)
    D := Novelty_table()
    while within_budget and ¬tree.root.solved do
        n, a := Select(tree.root, D, k)
        if a ≠ ⊥ then
            Rollout(n, a, D, k)

function SELECT(n, D, k)
    loop
        novel := Check_novelty(D, k, n.atoms, n.depth)
        if is_terminal(n) or ¬novel then
            Solve_and_propagate_label(n)
            return n, ⊥
        a := Sample_action(n)
        if n[a] in tree then
            n := n[a]
        else
            return n, a

function ROLLOUT(n, a, D, k)
    while within_budget do
        n := Successor(n, a)
        n.solved := false
        novel := Check_and_update_novelty(D, k, n.atoms, n.depth)
        if is_terminal(n) or ¬novel then
            Solve_and_propagate_label(n)
            return
        a := Sample_action(n)
```

- Case 3: if node n is already in the tree but all feature tuples of n are recorded at a lower depth, $d > D[t]$, **terminate** rollout.

- Case 4: if node n is already in the tree and makes some tuple true with $d = D[t]$, **continue** rollout.

The Rollout IW algorithm repeatedly selects a node-action pair (n, a) that has not yet been expanded, and then performs a rollout from (n, a). In the original pseudocode, however, the selection and rollout phases are intertwined. In this work, we split the pseudocode presented in Bandres, Bonet, and Geffner (2018) into three distinct functions: *Lookahead*, *Select*, and *Rollout*, which are detailed in Algorithm 3.1. The main function *Lookahead* iteratively interleaves calls of *Select* and *Rollout*, which contemplate cases 3-4 and 1-2, respectively. *Select* samples actions to traverse the tree until a node-action pair (n, a) is reached such that there is no successor in the tree for (n, a). For that, it relies on function *Sample_action* to select actions at random from those that do not lead to solved nodes (i.e., either they lead to unsolved nodes or to nodes not yet generated). *Rollout* then samples actions starting from (n, a), calling the *Successor* function to generate new nodes that are added to the search tree, until a state is reached that is either terminal or not novel. At that point, the final node is marked as *solved* and the process restarts until all nodes have been solved or a maximum budget of time or nodes is exhausted.

Pruning is done in both the *Select* and *Rollout* functions. *Select* only needs to check the novelty table but does not need to update it. On the other hand, the *Rollout* function checks the novelty table and updates it with the feature tuples of all newly generated nodes that have a lower depth than previously recorded. The pseudocode reported in Bandres, Bonet, and Geffner (2018) does not update the novelty table with features of terminal nodes. This is an algorithm design choice, which we have not seen to have an effect on performance. In our experiments in Chapter 6, we use Rollout IW as is detailed in Algorithm 3.1, which updates the novelty table with terminal node features (i.e., we perform the novelty check and update as soon as the successor state is generated).

We detail both functions for checking and updating the novelty table with feature tuple depths in Algorithm 3.2. Every time a node is labelled as solved, we try to propagate the label along the branch to the root

Algorithm 3.2 Depth-based novelty check and update

```
function CHECK_NOVELTY(D, k, atoms, d)
    for t in Combinations(atoms, k) do
        if d ≤ D[t] then
            return true
    return false

function CHECK_AND_UPDATE_NOVELTY(D, k, atoms, d)
    novel := false
    for t in Combinations(atoms, k) do
        if d < D[t] then
            D[t] := d
            novel := true
    return novel
```

(in function *Solve_and_propagate_label*). Each node of the branch will be solved if all of its children have been generated (i.e., all available actions have been tried) and appear as solved. Thus, the label propagation stops when at least one child has not yet been solved. With subtree caching, all nodes of the cached tree are initially marked as not solved, except for the ones that are terminal. This is done by function *Initialize_labels*.

There is a subtle difference between the exploration performed in Rollout IW and IW, even when enough time is given. The pruning mechanism of IW depends on the ordering in which actions are applied when expanding nodes. Thus, a tuple t of features that appears twice at the same depth will only make novel the state that has been generated first. However, Rollout IW may keep more than one node unsolved due to the same tuple t. This is because there may be several nodes at the same depth that satisfy the condition $d = D[t]$. Although this condition will only be checked for unsolved nodes, there may be nodes that were novel at generation due to other tuples different than t, and were therefore not marked as solved. If those tuples were later found at a lower depth, such nodes would still be considered novel at the selection phase due to tuple t.

Rollout IW is the first width-based algorithm that unties the order of generating states from the novelty test. This actually allows for many new width-based algorithms that generate nodes in different orders and

use the functions in Algorithm 3.2 to check and update the depth-based novelty table. For example, in Chapter 5 we present an algorithm based on Rollout IW that uses this novelty test. Note, however, that Rollout IW needs to perform many novelty checks, compared to IW, because the order is not breadth-first. In Chapter 4, we analyse the complexity of just checking, or checking and updating the novelty table.

3.5.1 Results in Atari Games

Compared to IW, the rollout version has a better anytime behavior. This becomes very convenient in simulator environments such as the Atari games, where the algorithm is applied iteratively in an online replanning manner. Bandres, Bonet, and Geffner (2018) provide results in Atari games in almost real time, using a budget of 0.5 seconds for planning in each iteration, that compares well with previous approaches. For the first time, visual features extracted from the game images are used, which are arguably more complex and less informative than the internal RAM state of the game used in previous approaches. Specifically, the features used are the B-PROST features, described in Section 2.6.3, which consists of a total of 20,598,848 binary features. In this case, Blob-PROST features also described in Liang et al. (2016), that aim at detecting objects, were not considered for the Rollout IW experiments.

Bandres, Bonet, and Geffner (2018) presented results in 58 Atari games for IW(1) and Rollout IW(1) with B-PROST features, using a frameskip of 15 frames, and time budgets of 0.5 and 32 seconds. They compare their approach to IW(1) using RAM features (Lipovetzky, Ramirez, and Geffner 2015), which used a budget of 30,000 nodes and a frameskip of 5, DQN, and human scores (Mnih et al. 2015). For Rollout IW, the authors consider two extensions, which they call Risk Aversion (RA) and Risk Aversion with Subscoring (RAS). In the risk aversion scenario, negative rewards are multiplied by a large constant $\alpha = 50{,}000$ and a high negative reward of -10α is given when the agent loses a life. Inspired by the serialization of goals, the subscoring approach also considers many novelty tables, and each state s is considered novel by looking up the value in the i-th novelty table, where i is obtained depending on the cumulative reward of the

Algorithm	Features	Budget	$\geq$ Human
DQN	-	-	46.9%
Sarsa	Blob-PROST	-	36.7%
IW(1)	RAM	15,000 nodes	77.6%
IW(1)	B-PROST	0.5s	14.2%
Rollout IW(1)			38.7%
RA Rollout IW(1)			44.8%
RAS Rollout IW(1)			51.0%
IW(1)		32s	44.8%
Rollout IW(1)			69.3%
RA Rollout IW(1)			71.4%
RAS Rollout IW(1)			75.5%

Table 3.2: Summary of Tables 1 and 2 from Bandres, Bonet, and Geffner (2018). The first two rows are results of reinforcement learning methods (Liang et al. 2016; Mnih et al. 2015) and the third row belongs to IW(1) with RAM features (Lipovetzky, Ramirez, and Geffner 2015).

trajectory from the root to s, $R(s)$, as follows:

$$i = \begin{cases} 0, & \text{for } R(s) \leq 0 \\ \lfloor \log_2(R(s)) \rfloor, & \text{for } 0 < R(s) < 1 \\ \lfloor \log_2(R(s)) + 1 \rfloor, & \text{for } 1 \leq R(s) \end{cases}$$

where $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$, once more, is the floor function.

Table 3.2 shows a summary of the results. The rollout version clearly outperforms IW(1) with both time budgets, achieving a higher score than the one reported by a human player in almost 70% of the games, with the larger budget. Interestingly, it performs comparably to RL methods only using 0.5s of budget, even when using a simpler set of features, although risk aversion is needed to match the performance of DQN. The version of the algorithm with risk aversion and subscoring, with 32 seconds of budget, is able to match the score of IW(1) with RAM features, which is remarkable since the later uses the internal simulator state as features and a bigger budget. Nevertheless, when we compare the absolute scores of both algorithms (instead of the relative score compared to a human player) we find that IW(1) outperforms Rollout IW in 34 out of 51 games.

Part II

Planning

Chapter 4

Complexity of IW

IW(k) involves two main loops: an outer loop that expands states, and an inner loop for the novelty check. Let N be a bound on the number of (novel) states *expanded* by IW(k), and C a bound on the number of *checks* to the novelty table performed for each generated state. Let b bound the number of actions applicable at each state. Then, bN is a bound on the number of states *generated* by IW(k), and the overall complexity is bounded by bNC.

There are well-known results regarding the complexity of IW(k) given the atoms of the problem $\mathcal{P}$. The number of (novel) nodes expanded by IW(k) is bounded by $N = |\mathcal{P}|^k$, while at each node generation at most $C = \binom{|\mathcal{P}|}{k}$ tuples of atoms are checked, which is also bounded by $|\mathcal{P}|^k$. Therefore, IW(k) runs in time $\mathcal{O}(b|\mathcal{P}|^{2k})$. Furthermore, if the novelty table is a perfect hash, it requires at most $\binom{|\mathcal{P}|}{k} = \mathcal{O}(|\mathcal{P}|^k)$ memory space.

When using the formulation $\mathcal{P} = \mathcal{F} \times \mathcal{D}$, we can get tighter bounds than the ones presented above. This is because atoms resulting from different values of the same feature cannot be found at the same time. We divide this Chapter in two sections. First, we provide a tighter bound on the number of expanded states N. Then, we discuss the role of the novelty check C when using features, the impact it has on the overall complexity when a short planning horizon is considered, and when rollouts are used instead of BrFS. We end the Chapter providing a lower and an upper bound on C.

4.1 Expanded Nodes

In this section we provide a tighter upper bound on the number of states N expanded by IW(k). We use $n = |\mathcal{F}|$ to denote the number of features, and $d = |\mathcal{D}|$ to denote the domain size. The existing complexity result with atoms composed by features that we know of is based on $|\mathcal{P}| = nd$, yielding that IW(k) expands at most $(nd)^k$ nodes (Lipovetzky, Ramirez, and Geffner 2015). Bandres, Bonet, and Geffner (2018) give a bound based on the number of features, but do not take into account the domain size.

Proposition 1. *Let $N(n, d, k)$ denote the maximum number of novel states visited by IW(k) for a given pair (n, d). Then, $N(n, d, k)$ is given by the recursive formula*

$$
\begin{aligned}
&N(n, d, 0) = 1, \\
&N(n, d, n) = d^n, \\
&N(n, d, k) = N(n-1, d, k) + (d-1) \cdot N(n-1, d, k-1).
\end{aligned}
$$

There are two base cases: $k = 0$, in which case no state is novel apart from s_0, i.e., $N(n, d, 0) = 1$, and $k = n$, in which case all states are novel, i.e., $N(n, d, n) = d^n$. The intuition for the recursion is as follows. Consider the case where IW(k) visits the maximum number of states. Given a feature $f \in \mathcal{F}$, we can partition the subset of novel states into two subsets:

- $\mathcal{S}_f$: states that are novel solely due to tuples that include f.
- $\mathcal{S}_{\neg f}$: states that are novel (in part) due to tuples that exclude f.

The initial state s_0 is included in $\mathcal{S}_{\neg f}$, since it is novel due to all features. Since f is irrelevant in $\mathcal{S}_{\neg f}$, IW(k) would consider novel the same states in this group even if we removed f. Thus, the maximum amount of novel states in $\mathcal{S}_{\neg f}$ is $N(n-1, d, k)$. Regarding $\mathcal{S}_f$, we can further divide it into $d-1$ subsets, each corresponding to a value of f different from its initial value $v_0 = \phi(s_0)[f]$. In each subset, since the value of f is the same, the novelty test can be simplified to checking tuples of size $k-1$ of features different than f. Therefore, the maximum number of novel states in $\mathcal{S}_f$ is given by $(d-1) \cdot N(n-1, d, k-1)$.

It is important to note that we are not decomposing the problem into multiple subproblems with one less feature, where we could apply many IW searches, but rather using a divide and conquer strategy to recursively define an upper bound on the number of novel states in each subset. If $n > k > 0$, the maximum number of novel states that can be produced by an IW(k) search in a problem with n features is the same as the maximum number of novel states generated in d IW searches in some other problem with $n - 1$ features: $(d - 1)$ times the one of IW($k - 1$) plus the one of IW(k).

	(f_0, f_1, f_2, f_3)	f_0f_1	f_0f_2	f_0f_3	f_1f_2	f_1f_3	f_2f_3
$\mathcal{S}_{\neg f_0}$	**(0, 0, 0, 0)**	**00**	**00**	**00**	**00**	**00**	**00**
	(0, 0, 0, 1)	00	00	**01**	00	**01**	**01**
	(0, 0, 1, 1)	00	**01**	01	**01**	01	**11**
	(0, 0, 1, 0)	00	01	00	01	00	**10**
	(0, 1, 1, 0)	**01**	01	00	**11**	**10**	10
	(0, 1, 1, 1)	01	01	01	11	**11**	11
	(0, 1, 0, 1)	01	00	01	**10**	11	01
	(0, 1, 0, 0)	01	00	00	10	10	00
$\mathcal{S}_{f_0}$	**(1, 1, 0, 0)**	**11**	**10**	**10**	10	10	00
	(1, 1, 0, 1)	11	10	**11**	10	11	01
	(1, 1, 1, 1)	11	**11**	11	11	11	11
	(1, 1, 1, 0)	11	11	10	11	10	10
	(1, 0, 1, 0)	**10**	11	10	01	00	10
	(1, 0, 1, 1)	10	11	11	01	01	11
	(1, 0, 0, 1)	10	10	11	00	01	01
	(1, 0, 0, 0)	10	10	10	00	00	00

Table 4.1: List of all possible d^n states, with $n = 4$ features and domain size $d = 2$. We list them in Gray code (i.e., only one bit changing at a time) and consider this as the order of expansion of IW(2), ensuring the worst case scenario where the maximum number of states are considered novel. The last column shows the tuple combinations taken into account in the novelty test. Novel states and the feature tuples that make them novel are shown in bold. Values of f_0 and tuples that are irrelevant for the novelty test in subsets $\mathcal{S}_{\neg f_0}$ and $\mathcal{S}_{f_0}$ are shown in gray.

In the worst case, all states in $\mathcal{S}_{\neg f}$ present the same value in feature f, i.e., the one appearing in the initial state v_0, otherwise $|S_f|$ would not be maximal. The intuition is that, to produce the maximum number of novel states, all feature values of f need to be *key*, at some point, to make a state novel. This is the case in the example that we show in Table 4.1, where all d^n states are visited by IW(2) in such an order that only one feature changes at a time. In this case, the problem has $n = 4$ features with domain size $d = 2$, but such list of *Gray code* states can be generated for any pair (n, d) (Guan 1998).

If we take $f = f_0$, we observe that the first 7 states are novel, in part, due to tuples of features f_1, f_2, and f_3, and therefore belong to $\mathcal{S}_{\neg f_0}$. Indeed, they can be counted using $N(n-1, d, k) = N(4-1, 2, 2)$:

$$\begin{aligned} N(3,2,2) &= N(3-1,2,2) + (2-1) \cdot N(3-1,2,1) \\ &= 2^2 + N(2,2,1) \\ &= 2^2 + N(2-1,2,1) + (2-1) \cdot N(2-1,2,0) \\ &= 2^2 + 2^1 + 1 = 7. \end{aligned}$$

The last 4 novel states are novel exclusively due to tuples containing f_0 ($\mathcal{S}_{f_0}$), and can be counted using $(d-1) \cdot N(n-1, d, k-1) = 1 \cdot N(4-1, 2, 2-1)$:

$$\begin{aligned} N(3,2,1) &= N(3-1,2,1) + (2-1) \cdot N(3-1,2,0) \\ &= N(2,2,1) + 1 \\ &= N(2-1,2,1) + (2-1) \cdot N(2-1,2,0) + 1 \\ &= 2^1 + 1 + 1 = 4. \end{aligned}$$

Theorem 1. *For n features of size d, the maximum number of novel states visited by IW(k), $0 \leq k < n$, is*

$$N(n,d,k) = \sum_{i=0}^{k} \left[\binom{n-1-i}{k-i} d^i (d-1)^{k-i} \right].$$

Proof. The proof is by induction on pairs of integers (n, k). The base case is given by $(n, 0)$, in which case we have

$$N(n,d,0) = \sum_{i=0}^{0} \left[\binom{n-1-i}{0-i} d^0 (d-1)^{0-i} \right] = \binom{n-1}{0} d^0 (d-1)^0 = 1.$$

For (n,k) such that $0 < k < n-1$, by hypothesis of induction we assume that Theorem 1 holds for $(n-1,k-1)$ and $(n-1,k)$. Applying the recursive definition yields

$$\begin{aligned}
N(n,d,k) &= (d-1)N(n-1,d,k-1) + N(n-1,d,k) \\
&= (d-1)\sum_{i=0}^{k-1}\left[\binom{n-2-i}{k-1-i}d^i(d-1)^{k-1-i}\right] \\
&\quad + \sum_{i=0}^{k}\left[\binom{n-2-i}{k-i}d^i(d-1)^{k-i}\right] \\
&= \sum_{i=0}^{k-1}\left[\left(\binom{n-2-i}{k-1-i} + \binom{n-2-i}{k-i}\right)d^i(d-1)^{k-i}\right] \\
&\quad + \binom{n-2-k}{0}d^k(d-1)^0 \\
&= \sum_{i=0}^{k-1}\left[\binom{n-1-i}{k-i}d^i(d-1)^{k-i}\right] + \binom{n-1-k}{0}d^k(d-1)^0 \\
&= \sum_{i=0}^{k}\left[\binom{n-1-i}{k-i}d^i(d-1)^{k-i}\right].
\end{aligned}$$

Here, we used the identities $\binom{n-1}{m-1} + \binom{n-1}{m} = \binom{n}{m}$, $0 < m < n$, and $\binom{n}{0} = 1 = \binom{n+1}{0}$.

For (n,k) such that $k = n-1$, by hypothesis of induction we assume that Theorem 1 holds for $(n-1,k-1)$. Applying the recursive definition yields

$$\begin{aligned}
N(n,d,k) &= (d-1)N(n-1,d,k-1) + N(n-1,d,k) \\
&= (d-1)\sum_{i=0}^{k-1}\left[\binom{n-2-i}{k-1-i}d^i(d-1)^{k-1-i}\right] + d^k \\
&= \sum_{i=0}^{k-1}\left[\binom{n-1-i}{k-i}d^i(d-1)^{k-i}\right] + \binom{n-1-k}{0}d^k(d-1)^0 \\
&= \sum_{i=0}^{k}\left[\binom{n-1-i}{k-i}d^i(d-1)^{k-i}\right].
\end{aligned}$$

Here, we used the definition $N(n-1,d,k) = N(k,d,k) = d^k$ and the identity $\binom{n}{n} = 1 = \binom{n+1}{n+1}$, which is applicable since $(k-1-i) = (n-1-1-i) = (n-2-i)$. □

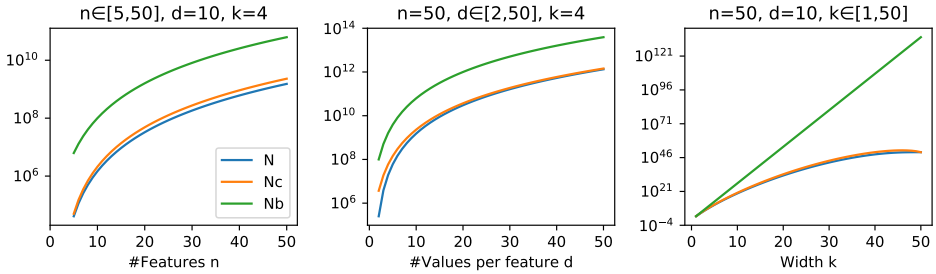

Figure 4.1: Comparison example between $N(n, d, k)$, its succinct upper bound $N_c = d^k \binom{n}{k}$, and the baseline $N_b = (nd)^k$. Plots from left to right vary n, d, and k, respectively. The y axis shows the number of states for the given parameters (n, d, k).

To obtain a more concise upper bound on N, we can write

$$N(n,d,k) = \sum_{i=0}^{k} \left[\binom{n-1-i}{k-i} d^i (d-1)^{k-i} \right]$$
$$\leq d^k \sum_{i=0}^{k} \binom{n-1-i}{k-i} = d^k \sum_{i=0}^{k} \binom{n-k-1+i}{i} = d^k \binom{n}{k}.$$

where equality only holds for the case $k = 0$. In this case, we used identities $\binom{m}{r} = \binom{m}{m-r}$ and $\sum_{i=0}^{m} \binom{r+i}{i} = \binom{r+m+1}{m}$. The original bound can be obtained by approximating the binomial coefficient as an exponential $d^k \binom{n}{k} \leq d^k n^k = (nd)^k$.

Figure 4.1 shows a comparison example between $N(n, d, k)$, its compact version $N_c = d^k \binom{n}{k}$, and the (loose) baseline bound $N_b = (nd)^k$. In particular, we compare N, N_c, and N_b in three plots where, from left to right, we vary parameters n, d, and k individually, while leaving the rest fixed. First, we observe that the difference between N and its compact version N_c is not significant compared to the baseline, in all three plots. The left-most plot shows a constant difference of two orders of magnitude between N_b and our tighter bounds N and N_c for this particular example, where the number of features varies from 1 to 50, each feature can take 10 values, and we are considering width $k = 4$. If we compare N and N_c we

observe a slight increasing difference due to the term -1 that we ignored in the process to obtain the compact bound. If we vary d instead (middle plot), with $n = 50$ features, we see a difference between 1 and 3 orders of magnitude between N_b and N. In this case, the small difference between N and N_c reduces when increasing d. When varying k (right-most plot), the difference between N_b and our two bounds N and N_c grows exponentially. Here, the difference between N and N_c gets to almost two orders of magnitude, which is not significant when compared to the baseline.

4.2 Novelty Check and Update

In this section we discuss the impact of the second bound C on the overall bound bNC. The novelty check for width k of a state s consists of many look-ups to the novelty table and, if there is one tuple of atoms in s that does not appear in the table, the state is considered novel. The novelty table is then updated with all the newly seen atom tuples in s. If we consider $\mathcal{P}_{true}$, where states are collections of propositions of the environment that are true, then there are at most $C = \binom{|\mathcal{P}|}{k}$ atom tuples. On the other hand, when considering $\mathcal{P}_{\mathcal{F}}$, since the atoms are restricted to one value per feature, there are $C = \binom{n}{k}$ possible tuples to be checked, where $n = |\mathcal{F}|$.

When considering a problem with binary features, using $\mathcal{P}_{\mathcal{F}}$ or $\mathcal{P}_{true}$ (i.e., explicitly stating false statements or not, respectively) makes a difference in the output of IW(k), as discussed in Chapter 3, but also in the complexity of checking and updating the novelty table. Using $\mathcal{P}_{true}$ results in a quicker novelty check and update, since only a fraction of the possible atoms are checked (those that are true). Moreover, in practise, the number of atoms used to represent each state is small compared to $|\mathcal{P}|$. This is because the dynamics of the environment forbid two atoms from appearing simultaneously, e.g., in Blocksworld, if one object is being held, no other object can be held. In order to get a novelty check nearly as efficient with $\mathcal{P}_{\mathcal{F}}$, atoms that cannot appear simultaneously should be encoded into the domain of the same feature, e.g., a feature that tells us which block is being held instead of having a binary feature for each block. In other words, using $\mathcal{P}_{true}$ does not affect the novelty check complexity (assuming that the number of true atoms is small), while in $\mathcal{P}_{\mathcal{F}}$ it is important how the

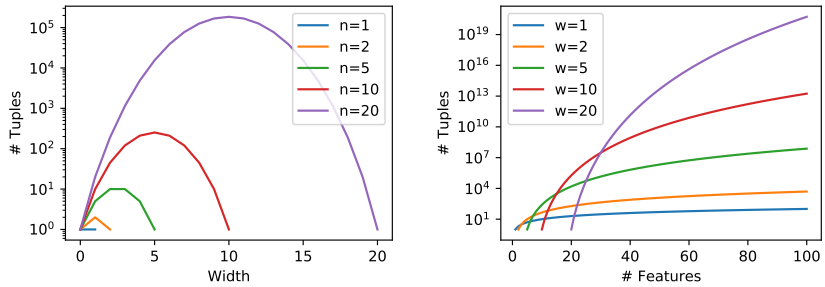

Figure 4.2: Number of tuples that IW(k) checks and updates in a problem, varying the width k for different number of features (left), and varying the number of features n for different widths (right).

features are defined (since all of them will be checked).

Note that *checking* the novelty in $\mathcal{P}_{\mathcal{F}}$ takes *at most* $\binom{n}{k}$ look-ups, while *updating* the novelty table takes *exactly* $\binom{n}{k}$. The reason is that finding *one* novel tuple is enough to tell that a state is novel, and therefore the novelty check loop can stop at that point. In contrast, *all* new tuples need to be recorded to update the novelty table. Usually, both check and update operations are done at the same time. The complexity for checking and updating the novelty table then grows asymptotically at a rate $\Theta\left(n^k\right)$, since $\binom{n}{k}$ is bounded by n^k. The overall complexity bound for IW(k) when using $\mathcal{P}_{\mathcal{F}}$ is $bNC = bN(n, d, k)\binom{n}{k}$, which is in turn upper-bounded by $bd^k\binom{n}{k}^2$ or $b(dn^2)^k$. Note that even this last (loose) bound is tighter, by a factor d, than the one considering $|\mathcal{P}_{true}| = nd$, as in Lipovetzky, Ramirez, and Geffner (2015), which gives $b(dn)^{2k}$.

In terms of novelty check and update, having a high width may actually imply a low complexity, depending on the amount of features n. For instance, checking and updating the novelty table in IW(n) is quicker than in IW(1). The reason is that there is only a single tuple to be checked at each state when $k = n$, compared to n tuples when $k = 1$. In general, the complexity has a bell shape when increasing k, given by the binomial coefficient $\binom{n}{k}$. This is illustrated in Figure 4.2 for some examples of n, varying $k < n$. The right plot shows the effect of increasing the amount of features for some fixed widths. No matter how informative such new

features are, they will have an impact on the complexity. It is therefore not desirable to have redundant features, even though the result of IW(k) would be the same with or without them. As discussed above, in terms of novelty check, it is preferable to introduce new feature values instead of new features, since the bound is linear in d but quadratic in n.

The impact of the novelty check and update is more notorious in the online replanning setting, since only a fraction of the state space that IW(k) can explore is actually visited. In this case, the amount of expanded nodes N becomes bounded by the planning budget B, resulting in the overall complexity bound bBC. Here, C plays a bigger role than initially presumed, assuming $B < N$. This is aggravated in Rollout IW. Since nodes are not expanded as in BrFS, the novelty of all revisited states needs to be rechecked at every traversal to maintain the notion of width. Then, the complexity of Rollout IW becomes $\mathcal{O}\left(b(dn^3)^k\right)$ (Bandres, Bonet, and Geffner 2018), which is worse by a factor of n^k, compared to IW. All this motivates the use of width $k = n$ later in Chapter 6, where we plan at two levels of abstraction, and use Rollout IW(n) to explore the high-level state space at a low cost. This way of planning hierarchically is described in the next Chapter.

4.2.1 Checking only features that change

The novelty check and update can be optimized by only checking tuples with features that have changed in a transition $s' = T(s, a)$ (Bonet and Geffner 2021), i.e., tuples that contain features in s' that have a different value from its parent state s. In STRIPS, it is easy to check for new atoms: they are given by the $Add(a)$ list. In other formulations, we may need to iterate over the state atoms or the feature vector. However, this extra step may pay off, as we analyze next.

Theorem 2. *When ignoring tuples of unchanged features, the amount of feature tuples checked in IW(k) grows asymptotically at a rate of at least* $\mathcal{O}(n^{k-1})$ *and at most* $\mathcal{O}(n^k)$.

Proof. Let c denote the features that *changed* value, and u denote the *unchanged* features, such that $n = c + u$. Then, for width k, $c < k$ and

$u < k$, we have that the amount of tuples to be checked is

$$\binom{c+u}{k} - \binom{u}{k} = \sum_{i=0}^{k} \binom{c}{i} \cdot \binom{u}{k-i} - \binom{u}{k} = \sum_{i=1}^{k} \binom{c}{i} \cdot \binom{u}{k-i}, \quad (4.1)$$

where the first equality follows from applying Vandermonde's identity. From Equation 4.1 we observe that the complexity is $\mathcal{O}(c^k + cu^{k-1})$. When only checking tuples of features that have changed, we have that the amount of feature tuples to be checked is lower-bounded by $\binom{n-1}{k-1} = \mathcal{O}(n^{k-1})$ (in the case that only one feature has changed) and upper-bounded by $\binom{n}{k} = \mathcal{O}(n^k)$ (in the case that all features have changed). □

If we consider that c is bounded by a constant, then the complexity of checking and updating the novelty table grows asymptotically at a rate $\mathcal{O}(u^{k-1}) = \mathcal{O}(n^{k-1})$. When using $\mathcal{P}_{true}$, the complexity becomes $\mathcal{O}(|\mathcal{P}|^{k-1})$ under the same assumption, as described by Bonet and Geffner (2021).

Chapter 5

Hierarchical Iterated Width

In this chapter, we present our hierarchical approach to width-based planning. We start by defining a simple algorithm for hierarchical blind search. Then, we consider using width-based planners at two levels of abstraction, and show its effect on the width compared to planning at a single level, drawing connections with related work. Finally, we present a method to discover high-level features incrementally, that we test in classical planning domains.

For simplicity, and without loss of generality, we assume a two-level hierarchy: a high level (h) and a low level (ℓ). Each level is defined by its own feature set (F_h and F_ℓ, with domains $\mathcal{D}_h$ and $\mathcal{D}_\ell$, respectively) and feature mapping ($\phi_h : \mathcal{S} \to \mathcal{D}_h^{|\mathcal{F}_h|}$ and $\phi_\ell : \mathcal{S} \to \mathcal{D}_\ell^{|F_\ell|}$, respectively). Each state s maps to a high-level state $s_h = \phi_h(s)$ and a low-level state $s_\ell = \phi_\ell(s)$.

5.1 A Hierarchical Approach to Blind Search

Blind search methods require two components: a successor function, that given a state and an action returns a successor state (e.g. a simulator), and a stopping condition, that will stop the search, for instance, when the goal is reached or after a budget is exhausted. In order to have different search levels, we modify these two components as follows:

- **High-level successor function:** Each call to this function triggers

a low level search, that runs until a new high-level state is found (i.e., a state s that maps to a different $\phi_h(s)$).

- **Low-level stopping condition:** When a different high-level state is encountered, the search is stopped, returning control to the high-level planner. This stopping condition is added to the existing stopping conditions.

The control goes back and forth between the high and low-level planners. Each time that the high-level successor function is called, the according low-level search is resumed, generating new states until a new high-level state is found. We achieve this by storing a low-level search tree for each high-level state. If the low-level search terminates without finding a new high-level state, the high-level successor function returns *null*, and the high-level state is marked as *expanded*. The high-level planner will only generate successors from non-expanded high-level states, and can resume search from any state by retrieving it from memory.

The proposed framework allows many levels of abstraction, as well as the possibility to have different search methods at each level. For instance, we could have a breadth-first search at the high level and a depth-first search at the low level, or combine different width-based search methods.

5.2 Hierarchical Width

The framework in the previous section partitions the states into subsets based on high-level features. To plan over the subsets, we can use any width-based search method as a high-level planner. For instance, we can apply IW(2) at the high level and IW(1) at the low level. An example representation of this hierarchical search, which we denote by HIW$(2, 1)$ is given in Figure 5.1. In general, we call HIW(k_h, k_ℓ) the hierarchical combination of two IW searches, where k_h is the width at the high-level and k_ℓ the width at the low level. We next define a type of high-level feature that we call *splitting*, and compare HIW with flat IW, showing the effect of the hierarchy on the width of the problem.

Definition 11. *A high-level feature $f \in \mathcal{F}_h$ is splitting if, for each value $v \in \mathcal{D}_h$, the states of the induced subset $\{s \in \mathcal{S} : \phi_h(s)[f] = v\}$ and their edges form a connected graph.*

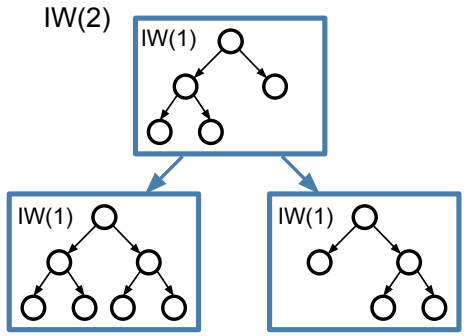

Figure 5.1: Illustrative example of HIW$(2, 1)$. Each high-level node of the IW(2) search (in blue), contains a low-level IW(1) search tree (in black).

Example: consider a simple problem, depicted in Figure 5.2 (left), where an agent needs to move along a corridor of length L, pick up a key, and go back along the same path to open a door. We can describe this problem using two features: p (the position) and h (whether or not the key is held). Initially $p = 0$ and $h = 0$. The goal is $p = 0$ and $h = 1$. If $h \in \mathcal{F}_h$, then h is splitting: when h is false, the agent can still visit all the positions of the corridor, and likewise when h is true. In Figure 5.2 (right) we show a similar problem in 2D. Let us split the space of positions, which in this case we represent with two features (p_x, p_y), into two tiles. Then, let us define a feature $t \in \{0, 1\}$ that tells us whether the agent is in one tile or the other. If we take the dashed line in blue to define such tiles, then t is splitting, since the agent can visit all positions of the same tile without needing to move to the other tile. Note that not all possible positions (p_x, p_y) need to be reachable from each value v of feature t (as it is the case of the corridor example), rather, all positions (p_x, p_y) in the subspace $t = v$ need to be reachable from a state of the same subspace, for all $v \in \{0, 1\}$. If we take the dashed line in red to define the tiles instead, the wall (in gray) prevents the agent from moving from the lower to the upper positions of the left tile, and thus t would not be splitting in this case.

Theorem 3. *If all features in $\mathcal{F}_h$ are splitting, HIW(k_h, k_ℓ) is equivalent to a restricted version of IW$(k_h + k_\ell)$ with tuples of k_h features from $\mathcal{F}_h$ and k_ℓ features from $\mathcal{F}_\ell$.*

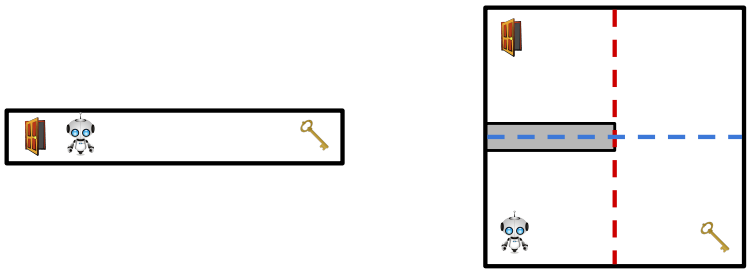

Figure 5.2: Navigation problem examples to illustrate splitting features, where an agent needs to pick up a key to open a door. Left: corridor example (1D). Right: 2D example with a wall represented by the rectangle in gray. The dashed lines represent possible ways of splitting the 2D state space.

Proof. Since each feature in $\mathcal{F}_h$ is splitting, when we apply IW(k_ℓ) in a high-level state s_h, the subset of states induced by s_h is connected. Since the restricted version of IW($k_h + k_\ell$) considers exactly k_ℓ features in $\mathcal{F}_\ell$, it will explore the same low-level states as IW(k_ℓ). At the high-level, the restricted version of IW($k_h + k_\ell$) considers exactly k_h features in $\mathcal{F}_h$, so it will explore the same high-level states as IW(k_h). Since the tuples in IW($k_h + k_\ell$) involve features in both $\mathcal{F}_h$ and $\mathcal{F}_\ell$, each state in the low-level search of a new high-level state is novel. Hence HIW(k_h, k_ℓ) explores the same states as the restricted version of IW($k_h + k_\ell$). □

Example: The corridor example of Figure 5.2 with features p and h has width 2, since IW needs to keep track of the key and visited position jointly to be able to reach the goal. This example can be solved by HIW(1, 1) using $\mathcal{F}_h = \{h\}$ and $\mathcal{F}_\ell = \{p\}$, after two low-level searches (one for $h = 0$ and one for $h = 1$), and visits the same states as IW(2).

Theorem 3 compares HIW(k_h, k_ℓ) to flat IW($k_h + k_\ell$) when all the features in $\mathcal{F}_h$ are splitting. However, this is not a necessary condition for HIW(k_h, k_ℓ) to solve problems of width $k_\ell + k_h$. Without splitting features, HIW will not generate the same nodes as the restricted version of IW, but may still find the goal. We empirically show this in the experiments section.

Notice that HIW may not generate the states in the same order than

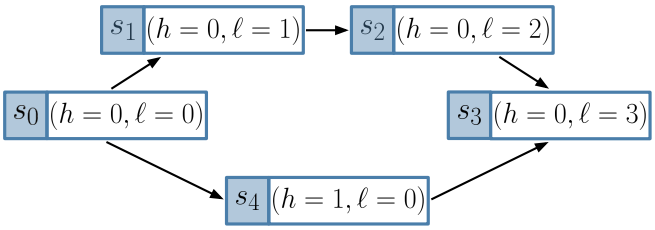

Figure 5.3: Example of MDP with two splitting features where HIW$(1, 1)$ and IW(2) visit the state space in a different order.

the restricted flat version. For instance, consider the MDP example shown in Figure 5.3, where the state space is described by two features, (h, ℓ), with domains $\{0, 1\}$ and $\{0, 1, 2, 3\}$, respectively. In this case, both features are splitting, since states with $h = 0$ (i.e., s_0, s_1, s_2, and s_3) and states with $\ell = 0$ (i.e., s_0 and s_4) are connected, and all other feature values appear in a single state. Let us select feature h as the high-level feature. Because h is splitting, HIW$(1, 1)$ will visit the whole state space. However, the order at which states are visited is determined by the high-level splitting: s_0, s_1, s_2, s_3 for the high-level state represented by $h = 0$, and s_4 for $h = 1$. If we run flat IW(2) instead, s_3 will be visited at depth 3 instead of at depth 4. Finally, note that in this case an IW(1) search would be sufficient to explore the state space, because the subspace induced by $h = 1$ consists of a single state.

Theorem 4. *Let $n_h = |\mathcal{F}_h|$ and $d_h = |\mathcal{D}_h|$ be the number of high-level features and domain sizes, and define (n_ℓ, d_ℓ) analogously. The maximum number of novel states expanded by HIW(k_h, k_ℓ) is $N(n_h, d_h, k_h) \cdot N(n_\ell, d_\ell, k_\ell)$.*

Proof. At the high level, HIW(k_h, k_ℓ) applies IW(k_h), which expands a maximum of $N(n_h, d_h, k_h)$ novel high-level states due to Theorem 1. For each novel high-level state, HIW(k_h, k_ℓ) applies IW(k_ℓ), which expands a maximum of $N(n_\ell, d_\ell, k_\ell)$ novel low-level states. □

Note that the maximum number of novel states expanded by the unrestricted version of IW$(k_h + k_\ell)$ on the feature set $\mathcal{F} = \mathcal{F}_h \cup \mathcal{F}_\ell$ is $N(n_h + n_\ell, \max(d_h, d_\ell), k_h + k_\ell)$, which is much larger, in general, than $N(n_h, d_h, k_h) \cdot N(n_\ell, d_\ell, k_\ell)$.

Example: The RAM memory in Atari, used in Lipovetzky, Ramirez, and Geffner (2015), consists of $n = 128$ features with $d = 256$ values. For IW(2), an upper bound on the number of novel states is $N(n, d, w) \sim 5 \cdot 10^8$. If we identify a splitting feature and define $n_h = 1$, $n_\ell = 127$, and $k_h = k_\ell = 1$, the upper bound due to Theorems 2 and 3 is $N(n_h, d, k_h) \cdot N(n_\ell, d, k_\ell) \sim 8 \cdot 10^6$, an improvement of almost two orders of magnitude.

5.3 Connections to Related Work

Planning with IW at different levels of abstraction in the presented hierarchical framework has an effect on the width required to solve a problem. If the high-level features are chosen correctly, the problem is effectively decomposed into subproblems that can be solved using a lower width than the one initially required (i.e., using the flat version). Decomposing the problem and running IW in the resulting subproblems is not a new concept. For instance, Serialized IW (SIW) performs a sequence of IW searches, starting a new search each time the number of unachieved goal atoms $\#g$ decreases. To draw a connection with our method, let us consider $\#g$ as a high-level feature. HIW would then generate a tree where each high-level node corresponds to a search for a different value of $\#g$. Note that here $\#g$ is a special feature that tracks the *minimum* amount of unachieved goal atoms so far (i.e., $\#g$ only changes when a state that has more goal atoms than the low-level root node is generated). In this case, a particular instance of SIW would correspond to a trajectory of the HIW tree. Note that the high-level pruning of HIW would favor trajectories where $\#g$ decreases more abruptly.

The decomposition exploited by HIW and SIW can also be achieved in a single search. Best-First Width Search (BFWS) methods consider a different novelty table depending on the valuation of a given set of functions. A similar approach is taken in Rollout IW by using subscoring, where a different novelty table is used depending on the obtained reward. In order to relate HIW and BFWS, let us consider the high-level feature mappings as the functions used to determine the novelty test in BFWS, which we denote by $w_{\mathcal{F}_h}$. In this case, a BFWS($w_{\mathcal{F}_h}$) search (i.e., where no other heuristic is used to break ties with the novelty test $w_{\mathcal{F}_h}$) would be similar to some extent to the one produced by HIW. However, there is one im-

portant difference: once more, the tree produced by BFWS would not be affected by the high-level pruning as in HIW. In the case of HIW, a state with new low-level feature tuples would not be explored if its high-level feature tuples have already appeared before in the high-level search. In contrast, BFWS would consider novel all states with newly seen low-level feature tuples. This is also the case for Rollout IW or any other (flat) width-based search algorithm that uses multiple novelty tables.

Theorem 3 relates HIW(k_h, k_ℓ) with a restricted version of IW$(k_h + k_\ell)$. The restricted flat IW works with all features $\mathcal{F}_h \cup \mathcal{F}_\ell$, but only considers tuples composed by k_h features of $\mathcal{F}_h$ and k_ℓ features of $\mathcal{F}_\ell$. Thus, the higher width $k_h + k_\ell$ is only effective between high- and low-level features. Geffner and Geffner (2015) also take into account tuples of higher size for a certain type of features. More precisely, in a variant of the algorithm called IW(3/2), they run IW(1) including also tuples of size 2 whenever one of the features is related to the game avatar. HIW$(1, 1)$ has a similar effect to IW(3/2) in the presence of splitting features, although one central difference is, again, the high-level pruning.

5.4 Incremental Hierarchical IW (IHIW)

In classical planning, the states are defined by a set of atoms and, although one atom may be more informative than others, there is no hierarchical structure. In this section, we present a simple method for identifying relevant features that may split the state space. Then, we introduce an algorithm that performs a sequence of hierarchical searches, using the aforementioned method to discover new high-level feature candidates at each step. In the experiments section, we test the algorithm in a range of classical planning domains.

5.4.1 Discovering High-Level Features

Consider a search tree generated by IW(1) for a problem of width 2. Is it possible to identify features that split the state space, so that the problem can be solved by HIW$(1, 1)$? In this section, we present a simple method for detecting candidate abstract features from a set of features $\mathcal{F}$.

We consider all trajectories in the tree and hypothesize that a feature

Algorithm 5.1 Method for finding high-level features

Input: node n
$N = \emptyset$
if IsLeaf(n) & Depth(n) > 2 **then**
  $P = \text{Atoms}(n) \cap \text{Atoms}(\text{Parent}(n))$ ▷ common atoms
  if $|P| < |\text{Atoms}(n)|$ **then** ▷ ensure different state
    $b = \text{Branch}(tree, n)$ ▷ get branch root→n
    $B = \bigcup_{i=1}^{\text{Depth}(n)-2} \text{Atoms}(b[i])$ ▷ all branch atoms
    $N = P - B$ ▷ keep (branch) novel atoms
return N

that changes only once before a trajectory is pruned is a good candidate for a high-level feature. Consider again the corridor example in which an agent has to use a key to open a door. IW(1) prunes any trajectory that repeats a position p, and will not solve the problem. However, feature h splits the state space into two sub-problems: reaching the key (h=false), and going back to the door (h=true).

We can detect high-level features using the method detailed in Algorithm 5.1. For each pruned leaf node, we retrieve the features that are shared with its parent that have not appeared in that branch before. The intuition is that when a splitting feature f changes value, from v_0 to v_1, the *next* state is likely to be pruned by IW(1), since v_1 has just been observed for f, and all other features may have been visited when f took value v_0.

5.4.2 An incremental approach

A simple algorithm that takes advantage of the previous method would be:

1. Perform an IW(1) search, if the goal is found, return.
2. Run Algorithm 5.1 on the IW(1) tree to find high-level features.
3. Run HIW$(1, 1)$ with the discovered high-level features.

Algorithm 5.2 Incremental Hierarchical IW Search

```
Initialize: H = ∅, P = List(), solved = false
while not solved do
    pruned, solved = HIW(w_h, w_l)
    if not solved then
        Append(P, pruned)
        while H == ∅ do
            if P is empty then
                return
            n = Pop(P)                          ▷ Sample pruned node
            H = FindAbstractFeatures(n)         ▷ Algorithm 5.1
        h = Pop(H)                              ▷ Sample candidate atom
        RestructureTree(h)                      ▷ Create high-level nodes
```

This algorithm actually finds promising candidate features for small problems. For instance, it can solve the simple corridor example. However, it fails on more complex problems, possibly because a single IW(1) search may not be sufficient to visit states that contain relevant features.

To address this, we propose a slightly more sophisticated approach, Incremental HIW (Algorithm 5.2), that runs a series of HIW searches. It maintains a set of high-level feature candidates H, exploits one feature candidate at a time, and discovers new relevant features when necessary. First, we run HIW$(1, 1)$, which is equivalent to IW(1) since we start with $H = \emptyset$. While the task is not solved, we randomly sample a pruned node and update H using Algorithm 5.1. We may repeat this operation until new feature candidates are found or there are no more pruned nodes to sample from, in which case we stop the search. Then, a feature candidate is sampled from H, and the current search tree is restructured accordingly, in order to reuse the tree in the subsequent search.

Restructuring the tree mainly involves two operations: detaching subtrees at the low level and inserting new nodes at the high level. We illustrate this process with an example in Figure 5.4 where, after a new high-level feature is added, low-level nodes marked in blue present a different value for such feature compared to their root node, and are therefore detached from their tree to form a new high-level node, that is inserted in

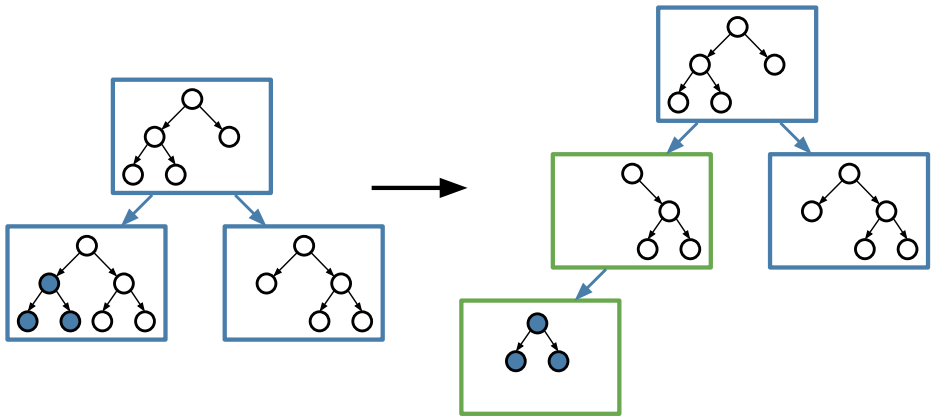

Figure 5.4: Illustration of IHIW tree restructure after adding a new high-level feature. Nodes filled in blue have a different high-level feature value than their root node. Novelty tables of the low-level searches of the resulting nodes (marked in green) need to be reset before resuming the search.

the high-level tree. Note that the inserted node does not need to be a leaf node as in this example, and we may need to insert several nodes at once. Although this process may seem costly, both operations consist of modifying the data structure, while leaving the data untouched. Modifying a search tree, however, implies that the associated novelty table cannot be reused. Thus, we generate a new novelty table, if necessary, when the according tree search is resumed.

5.5 Experiments in Classical Planning

In this section, we evaluate experimentally the proposed hierarchical approach. We address the following questions:

- In practice, can HIW$(1, 1)$ solve problems of width 2?
- Can Algorithm 5.1 find good high-level feature candidates?
- Is IHIW$(1, 1)$ a good alternative to IW(2)?

Lipovetzky and Geffner (2012) empirically showed that most classical planning problems with atomic goals present a low width. In Table 5.1, we reproduce such results, and compare them to our algorithm. The table consists of 36 domains from the International Planning Competitions, prior to 2012. For each domain, we show the amount of single goal instances (I), generated by splitting each instance with G goal atoms into G single goal instances. Columns 3-11 show the amount of instances solved (C), together with the average number of nodes (N) and time (T) per solved instance, for IW(1), IW(2) and IHIW$(1, 1)$. Here, IHIW$(1, 1)$ consists of two standard IW(1) searches, one at each level of abstraction. All algorithms have a planning budget of 10,000 nodes.

In some domains, IW(1) has greater coverage than IW(2), e.g., in Woodworking. This is because we set a budget of $10K$ nodes, and IW(2) may exhaust the budget before finding the goal. We observe that IHIW$(1, 1)$ outperforms IW(1) in all but five domains, where they perform exactly the same: Barman, OpenStacks, Parking, Scannalyzer and Woodworking. Besides, there are two domains, Storage and VisitAll, where both reach the maximum coverage. Compared to IW(2), IHIW$(1, 1)$ covers more or the same number of instances in 24 out of 36 domains. In 12 cases the average number of nodes per solved instance is lower in IHIW(1,1) than in IW(2), and in 18 cases IHIW(1,1) solved it faster.

Note that Table 5.1 only reports the average time for solved instances. Thus, we may find that IHIW(1,1) is quicker than IW(2) even when solving more instances. In fact, there are only three domains in which IHIW(1,1) has a higher average time than IW(2) (Elevator, Transport, and VisitAll), but the average time for IHIW(1,1) is actually computed over more instances (those that have been solved) than the one of IW(2) (e.g., in VisitAll, the 1.83s in average of IHIW(1,1) are over *all* instances, and therefore cannot be fairly compared to the 1.34s IW(2), that are over only 16.9% of the instances).

With these results we can conclude that HIW$(1, 1)$ can solve problems of width 2 in practice, and that Algorithm 5.1 is a good approach to identify promising high-level features. Finally, we can state that IHIW$(1, 1)$ is an efficient alternative to IW(2).

Domain	I	IW(1)			IW(2)			IHIW(1, 1)		
		C	N	T	C	N	T	C	N	T
8puzzle	32	40.6	34	0.00	**100**	475	0.04	**100**	137	0.01
Barman	232	**9.1**	215	0.02	**9.1**	215	0.13	**9.1**	215	0.02
Blocks World	302	37.4	91	0.01	79.5	1696	0.23	**96.4**	869	0.06
Cybersecurity	86	65.1	64	0.01	65.1	64	0.22	**67.4**	158	0.02
Depots	189	10.6	494	0.28	23.8	2393	1.58	**28.0**	2268	0.97
Driverlog	259	44.0	996	0.12	53.3	1249	0.18	**62.9**	1085	0.11
Elevator	510	0.0	-	-	11.4	5875	1.38	**16.9**	4752	1.79
Ferry	8	0.0	-	-	**100**	10	0.00	**100**	11	0.00
Floortile	538	96.3	515	0.04	93.5	1115	0.63	**99.3**	567	0.04
Freecell*	68	8.8	192	0.14	**22.1**	3558	4.00	19.1	504	0.48
Grid	19	5.3	2	0.00	**36.8**	2071	6.45	15.8	1244	2.51
Gripper	460	0.0	-	-	**100**	3355	1.70	**100**	2140	0.36
Logistics	249	18.1	2	0.00	**100**	763	0.16	28.5	87	0.01
Miconic	2325	0.0	-	-	0.0	-	-	**100**	2751	0.24
Mprime	50	8.0	2	0.01	18.0	3316	0.75	**20.0**	2600	0.48
Mystery	45	8.9	2	0.01	**37.8**	1200	0.57	31.1	1903	0.37
NoMystery	210	0.0	-	-	**80.0**	1917	1.61	24.8	1487	1.22
OpenStacks*	455	**0.0**	-	-	**0.0**	-	-	**0.0**	-	-
OpenStacks6	1230	5.1	176	0.20	**14.2**	2637	11.46	13.8	2332	0.37
PSRsmall	316	89.9	2	0.00	92.1	2	0.00	**94.0**	3	0.00
ParcPrinter	990	85.6	195	0.01	84.6	695	0.63	**92.0**	464	0.03
Parking	540	**66.3**	2770	2.28	65.2	2963	5.79	**66.3**	2770	2.27
Pegsol	990	92.6	4	0.00	**100**	9	0.01	97.8	7	0.00
Pipes-NoTan	259	45.6	299	0.08	55.6	1937	0.85	**57.5**	683	0.17
Rovers*	488	31.6	2520	0.37	23.2	2504	1.59	**35.2**	2576	0.37
Satellite*	1324	5.7	367	0.19	7.2	675	0.23	**7.9**	1433	0.22
Scanalyzer	648	**99.1**	370	0.29	96.6	322	0.66	**99.1**	370	0.28
Sokoban	154	35.1	37	0.01	**74.0**	1049	5.36	40.3	84	0.01
Storage	240	**100**	327	1.87	**100**	1035	15.76	**100**	327	1.88
Tpp*	118	0.0	-	-	**44.9**	3313	26.01	35.6	1476	0.19
Transport	330	0.0	-	-	11.8	3765	1.20	**18.5**	4230	1.96
Trucks	345	0.0	-	-	**11.6**	5158	0.77	1.7	3342	0.47
Visitall	21880	**100**	2918	1.83	16.9	2912	1.34	**100**	2918	1.83
Woodworking	1801	**91.6**	1110	0.29	88.3	1063	3.43	**91.6**	1110	0.29
Zeno	219	21.0	10	0.00	**36.5**	1740	0.18	29.2	1035	0.10
		7			17			24	12	18

Table 5.1: Results of IW(1), IW(2), and IHIW(1, 1) in 35 classical planning domains. Column I shows the number of single goal instances. Columns 3-11 show the coverage (C) in percentage (best in bold), the average amount of expanded nodes (N), and the average time (T) in seconds, for each algorithm. In blue: IHIW(1,1) with lower N or T than IW(2). In domains with *, not all instances were evaluated due to time or memory constraints.

Part III

Learning

Chapter 6

Policy-Guided Iterated Width

In spite of its success, IW does not learn from experience, so its performance does not improve over time. When expanding a node, IW generates all possible states, one per action. The recently proposed Rollout IW algorithm (Bandres, Bonet, and Geffner 2018) generates whole branches by expanding one state per node, but selects actions randomly. Even though both algorithms favor novel states with previously unseen feature values, random action selection does not take into account previous experience and results in uninformed exploration. As a result, reaching a distant reward in a specific search may be arbitrary.

We begin this chapter by presenting a modified version of Rollout IW that uses a policy estimate, modelled using a neural network (NN), for action selection. We then present a novel approach to feature selection that uses the node activation of the neural network as features. Finally, we evaluate our algorithm experimentally in simple sparse reward problems as well as in the Atari benchmark.

6.1 Policy-Guided Iterated Width (π-IW)

We now present our algorithm, Policy-Guided Iterated Width (π-IW), that enhances Rollout IW by incorporating an action selection policy, resulting in an *informed* IW search. More precisely, we leverage the exploration

capacity of IW to train a policy estimate $\widehat{\pi}_\theta$, which is used in turn to guide subsequent search. We consider tuples of size 1, i.e., IW(1), which keeps planning tractable. Similar to Rollout IW, π-IW requires both a resettable simulator, that provides the successor of a state s, and a representation of s in terms of features $\mathcal{F}$. Also, π-IW operates in an online replanning scheme, i.e., at each time-step, a planning step is followed by an action execution step. Importantly, the online replanning setting allows us to solve problems of width higher than one, since we reinitialize the novelty table at each planning step.

Below we describe the two basic steps of the π-IW algorithm, and present a mechanism for extracting a feature space from the policy $\widehat{\pi}_\theta$. This second use of the policy is beneficial if no feature representation is initially available.

6.1.1 Planning Step

The planning step of π-IW is very similar to Rollout IW, which we briefly outline here for clarity. In Algorithm 3.1, we chose to restructure the original pseudocode of Bandres, Bonet, and Geffner (2018) into three main functions: *Lookahead*, *Select*, *Rollout*. The function *Lookahead* iteratively alternates calls of *Select* and *Rollout* until a computational budget is exhausted or the root node is solved. *Select* samples actions to traverse the tree until a state-action pair (n, a) is reached that has not yet been expanded. Then, *Rollout* samples actions starting from (n, a) until a state is reached that is either terminal or not novel. At that point, the final node is marked as *solved* and the process restarts. Following Bandres, Bonet, and Geffner (2018), a state is considered novel if one of its atoms is true at a smaller depth than the one registered so far in the novelty table. A node that was already in the tree will not be pruned if its depth is exactly equal to the one in the novelty table for one of its atoms. This corresponds to functions *Check_novelty* and *Check_and_update_novelty* of Algorithm 3.2.

The only difference between Rollout IW and π-IW is how the function *Sample_action* is defined. In the original Rollout IW, *Sample_action* returns an action sampled with uniform probability, whereas π-IW uses a softmax policy $\widehat{\pi}_\theta(a|s_n) \propto \exp\left(h_a(s_n, \theta)/\tau\right)$, where h_a, $a \in A$, are the output logits of the neural network and τ is a temperature parameter. Our planning step thus becomes the original Rollout IW in the limit $\tau \to \infty$.

Just as in Rollout IW, actions that lead to nodes labelled as solved should not be considered. Thus, we set probability $\widehat{\pi}_\theta(a|s_n) = 0$ for each solved action a and normalize $\widehat{\pi}_\theta$ over the remaining actions before sampling.

Every time a node is labelled as solved, we try to propagate the label along the branch to the root (function *Solve_and_propagate_label* of Algorithm 3.1). Each node of the branch will be marked as solved if all of its children appear as solved. Thus, the propagation of the label stops when at least one child has not yet been pruned. In the case of caching the subtree of the previous plan, which is the case in our experiments, all nodes of the cached tree are initially marked as not solved, except for the ones that are terminal (function *Initialize_labels*).

6.1.2 Action Execution Step

Once the tree has been generated, the discounted rewards are backpropagated to the root: $R_i = r_i + \gamma \max_{j \in \text{children}(i)} R_j$. In general, a policy $\pi_t(\cdot|s_t)$ can be induced from the returns at the root node by applying another softmax function, although in our experiments we applied the deterministic version (with $\tau \to 0$), that we define as follows:

$$\pi(a|s_t) = \begin{cases} 1/m, & \text{for } R(s_t, a) = \max_b R(s_t, b) \\ 0, & \text{otherwise} \end{cases}$$

where m is the amount of entries that present maximum return, i.e., the amount of actions a that satisfy $R(s_t, a) = \max_b R(s_t, b)$. Note that in the case that $m = 1$, the result is a deterministic policy (i.e., one entry has probability one and the rest have probability zero), whereas in the case $m > 1$ π_t is a stochastic policy. A clear example is when all entries present the same return (e.g., zero, when no reward at all is found), in which case the resulting policy is the uniform policy. Sampling from this policy is similar to taking the argument that achieves the maximum. The reason for not using the *argmax* function, however, is that it is usually implemented deterministically (e.g., by choosing the index of the first entry that matches the maximum value), and using a stochastic policy in cases where $m > 1$ is crucial to ensure a proper exploration.

The current root state s_t is stored together with the target policy in an experience replay dataset to train the model in a supervised manner.

Finally, a new root is selected from the nodes at depth 1 by selecting an action $a_t \sim \pi_t(\cdot|s_t)$, and the resulting subtree is kept for the next planning step. In previous work, it has been argued that not adding cached nodes to the novelty table of the subsequent search increases exploration and hence performance (Lipovetzky, Ramirez, and Geffner 2015). However, in our experiments, we found that including feature tuples from cached nodes in the novelty table actually boosted the performance in sparse reward tasks, which are precisely the tasks that require more exploration. Therefore, we reinitialize the novelty table with nodes from the cached tree before each search (this is done in function *Initialize_labels*). For that, cached nodes are revisited in the order they were generated, and their feature tuples are added to the novelty table. Note that cached nodes will contain outdated information. We did not find this to have a great impact on performance. If that would ever become an issue, one possibility could be to rerun the model on all nodes of the tree at regular intervals (this is not done in our experiments).

6.1.3 Learning Step

In this step, we use the policy extracted from the search tree as a target policy to train the policy estimate $\widehat{\pi}_\theta$. As previously mentioned, transitions consisting of a state and a tree policy $\langle s, \pi \rangle$ are stored in an experience replay dataset D. The learning step consists in performing stochastic gradient descent on a randomly sampled a batch B of these transitions. We use the cross-entropy error between the induced target policy $\pi_i(\cdot|s_i)$ and the current policy estimate $\widehat{\pi}_\theta(\cdot|s_i)$ to update the policy parameters θ, defining a loss function

$$\mathcal{L}(\theta) = \sum_{i}^{|B|} -\pi_t(\cdot|s_i)^\top \log \widehat{\pi}_\theta(\cdot|s_i) + c_{\ell 2}\|\theta\|^2,$$

where the last term is an ℓ-2 regularization term to avoid overfitting and help convergence, weighted by hyperparamter $c_{\ell 2}$.

The planning and learning steps can be run in parallel, as in AlphaZero, or sequentially. For simplicity, in our experiments we choose the latter, sampling a batch of transitions at each iteration. We keep a maximum of $|D|$ transitions, discarding outdated transitions in a FIFO manner.

6.2 Dynamic Features

The quality of the transitions recorded by IW greatly depends on the feature set $\mathcal{F}$ used to define the novelty of states. For example, even though IW has been applied directly to visual (pixel) features (Bandres, Bonet, and Geffner 2018), it tends to work best when the features are *symbolic*, e.g., when the RAM state is used as a feature vector (Lipovetzky, Ramirez, and Geffner 2015). Symbolic features make planning more effective, since the width of a problem is effectively reduced by the information encoded in the features. However, how to automatically learn powerful features for this type of structured exploration is an open challenge.

Unlike previous width-based methods, π-IW can use the representation learned by the policy NN to define a feature space, as in representation learning (Goodfellow, Bengio, and Courville 2016). With this dependence, the behavior of IW effectively changes when interleaving policy updates with runs of IW. If appropriately defined, these features should help to distinguish between important parts of the state space. In this work, we extract $\mathcal{F}$ from the last hidden layer of the NN. In particular, we use the output of the rectified linear units that we subsequently discretize in the simplest way, resulting in binary features (0 for zero outputs and 1 for positive outputs).

6.3 Experiments

In this section, we evaluate the performance of Policy-Guided Iterated-Width (π-IW) in different settings. First, we consider a simple problem where we compare our method against AlphaZero and current width-based methods. Second, we present results in the Atari 2600 testbed. The following questions are addressed:

- How do the different types of exploration (structured for π-IW and unstructured for MCTS) affect the performance of both algorithms?
- What is the benefit of learning a policy to guide the IW planner?
- Are the learned (dynamic) features effective? Is it possible to learn them without degrading the performance?

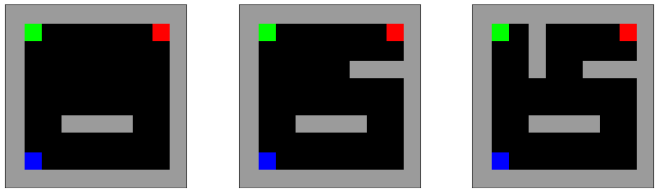

Figure 6.1: Snapshot of three versions of the maze. The blue, red and green squares represent the agent, the key and the door, respectively.

To answer the previous questions, we use a simple pixel-based version of the corridor and maze environment of Figure 5.2, where an agent (represented by a blue square) has to navigate to first pick up a key (red square) and then go through a door (green square). An episode terminates with a reward of $+1$ when the goal is accomplished, with a reward of -1 when a wall is hit, or with no reward after a maximum of 200 steps is reached. Intermediate states are *not* rewarded (including picking up the key), which makes the the problem more challenging. The observation is an 84×84 RGB image and possible actions are no-op, going up, down, left or right. See Figure 6.1 for an example.

We first analyze π-IW using static (pre-defined) and dynamic (learned) features. For the first case, we take the set of Basic features (Bellemare et al. 2013) described in Section 2.6.3, where the input image is divided in tiles and an atom, represented by a tuple (i, j, c), is true if color c appears in the tile (i, j). In our simple environment, we make the tiles coincide with the grid, and there is no background subtraction. We call this variant π-IW(1)-Basic. For the second case, we take the (discretized) outputs of the last hidden layer of the policy network as binary feature vectors. We call this variant π-IW(1)-Dynamic.

6.3.1 π-IW Can Reduce the Width of a Problem

Our first example is the simple corridor task, where the agent is located between a key and a door (see Figure 6.2). Using the Basic features, this problem has width 2, since the agent needs to keep track of the paired features *having the key* (final position in blue/black or red) and *visited position* (color blue or other) jointly. Therefore, it is not solvable by

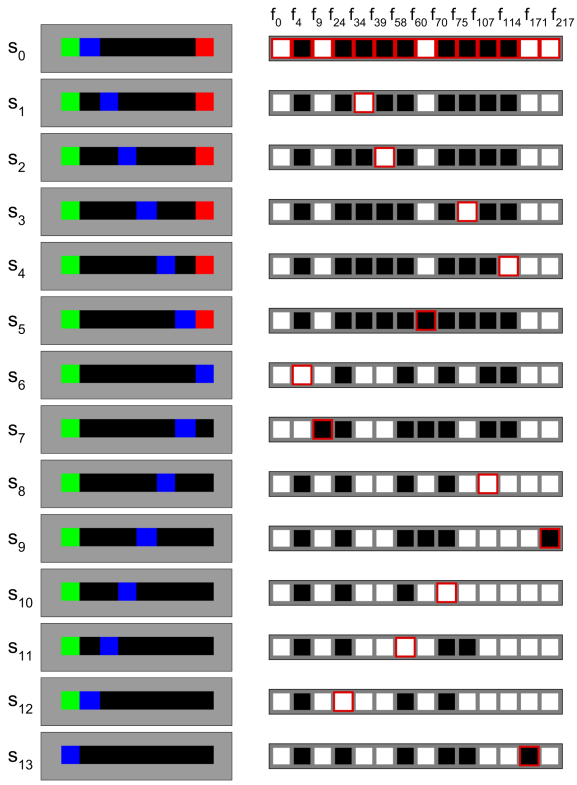

Figure 6.2: Feature learning in the corridor task. Left: from top to bottom, states expanded by π-IW(1) in one planning step once the policy has been trained. Right: subset of learned features for each state. Novel features at each step are marked in red.

IW(1) in the classical (offline) setting, i.e., in one planning step. However, in the online replanning setting, where novelty tables are reset after each action execution step, the task is solvable by IW(1). We are interested in analyzing the behavior of π-IW using dynamic features.

As expected, one planning step of π-IW using the set of Basic features does not reach the goal, and results in a trajectory pruned at s_8, two steps after picking up the key. Similarly, π-IW(1)-Dynamic is unable to generate the optimal plan in its first planning step, since the initial features are uninformed. However, using a sufficient number of features (13 binary features in this example), after learning, π-IW(1)-Dynamic *solves the task using a single (offline) planning step* in five out of five cases. This shows that the problem width is *reduced* from 2 to 1 in the learned representation of the policy. This is remarkable, since there is no explicit term in the loss function that encourages the policy to generate such a representation.

6.3.2 π-IW Improves MCTS Exploration

We now compare the two π-IW variants with current width-based methods and AlphaZero (Silver et al. 2018) in a more complex task. We consider three variants of the maze, with increasing difficulty, shown in Figure 6.1. Although AlphaZero was originally designed for two-player, zero-sum games, it can be easily extended to the MDP setting. Each time a node n is generated, the statistics W_m of all nodes in the trajectory from the root to n are updated with the value of n. Since in the MDP setting there are rewards in the intermediate edges, we update W_m with the discounted sum of rewards of all edges between nodes m and n, including the value of n (e.g. $W_1 \leftarrow W_1 + r_1 + \gamma r_2 + \gamma^2 v_3$).

AlphaZero controls the balance between exploration and exploitation by a parameter p_{uct} together with a temperature parameter in the target policy τ, similar to ours. In the original paper, τ is set to 1 for a few steps at the beginning of every episode, and then it is changed to an infinitesimal temperature $\tau = \epsilon$ for the rest of the game (Silver et al. 2017). Nevertheless, we achieved better results in our experiments with AlphaZero using $\tau = 1$ for the entire episode.

Both π-IW and AlphaZero algorithms share the same NN architecture and hyperparameters, specified in Table 6.1. We use two convolutional and two fully connected layers as in Mnih et al. (2013) (see Figure 2.2), which

Hyperparameter	Value	Algorithm
Discount factor	0.99	Both
Batch size	32	Both
Learning rate	0.0005	Both
Clip gradient norm	40	Both
RMSProp decay	0.99	Both
RMSProp epsilon	0.1	Both
Tree budget nodes	50	Both
Dataset size $\|D\|$	10^3	Both
L2 reg. loss factor	10^{-3}	Both
Tree policy temp. τ	1	Both
p_{uct}	0.5	AlphaZero
Diritchlet noise α	0.03	AlphaZero
Noise factor	0.25	AlphaZero
Value loss factor	1	AlphaZero

Table 6.1: Hyperparameters used for π-IW and AlphaZero.

are trained using the non-centered version of the RMSProp algorithm. All hyperparameters of AlphaZero and π-IW have been optimized for the second version of the game, with two walls.

Figure 6.3 shows results comparing π-IW against existing width-based algorithms and AlphaZero for the three mazes (we also performed experiments using different random configurations of the walls and the results remained consistent). The top row shows the average reward as a function of the number of interactions with the environment. As expected, the number of interactions required to solve the problem increases with the level of difficulty. While π-IW variants reach top performance in less than $2 \cdot 10^5$ interactions in the first and the second versions of the game, they require nearly 10^6 to fully solve the third version.

The performance of the two other width-based algorithms (IW and Rollout IW) is independent of the number of interactions. These algorithms, despite using the structured exploration of IW, are limited to width 1 and do not learn from previous visited states. Consequently, only in a very few cases (depending on the tie breaking, basically), they find

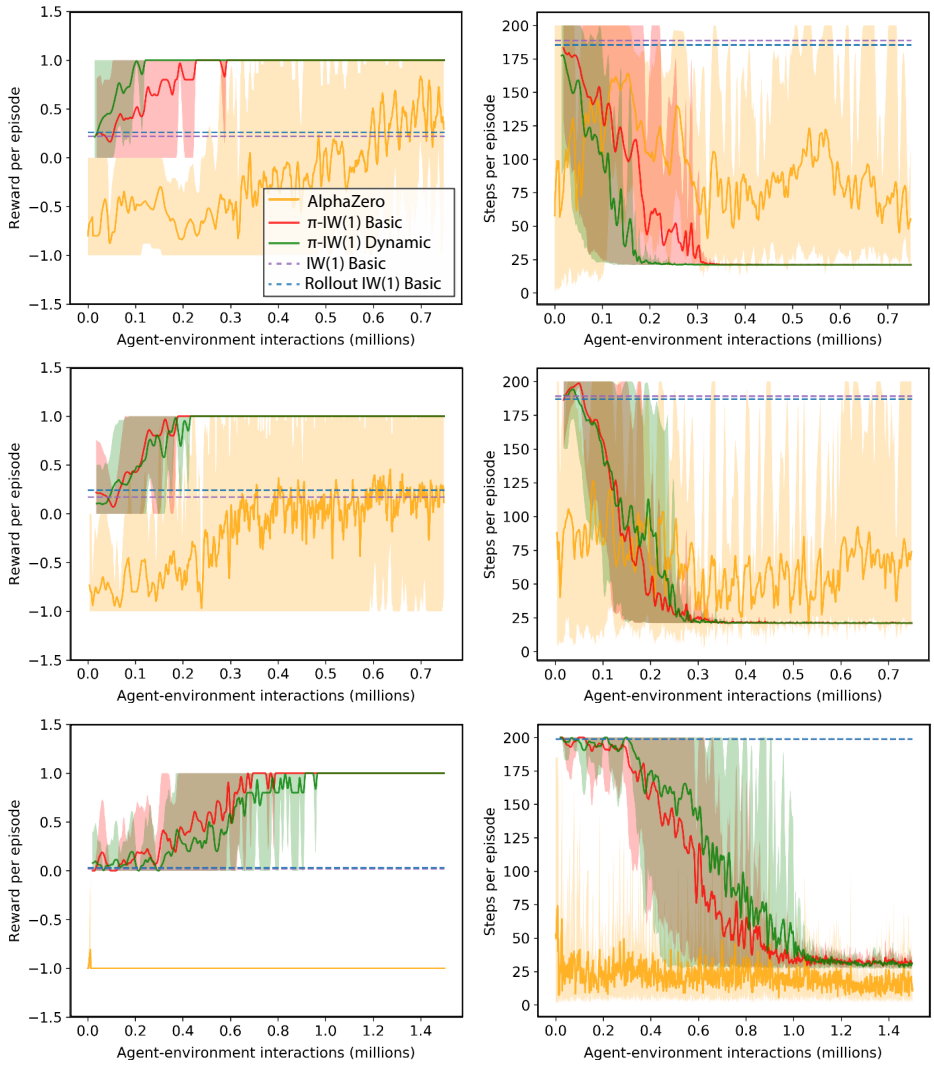

Figure 6.3: Performance in terms of reward (left column) and transitions per episode (right column) of width-based planners and AlphaZero in the three simple mazes (from top to bottom: 1, 2, and 3 walls, respectively). Plots are averages over 5 runs and shades show the minimum and maximum values. Results of IW and Rollout IW are averages over 100 runs.

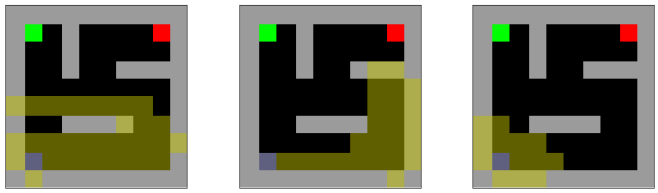

Figure 6.4: Three illustrative examples of visited states (in yellow) during one planning step before learning: π-IW(1)-Basic (left), π-IW(1)-Dynamic (middle), and AlphaZero (right). The π-IW variants explore a larger region of the state space than AlphaZero.

the correct sequence of actions. Finally, AlphaZero shows less stable behavior and is unable to solve the most difficult scenario (rightmost plot). Comparing the two π-IW variants we observe few differences. This is remarkable, since it indicates that in this simple maze the features used by IW can be learned easily from scratch.

Figure 6.3 (bottom row) compares the number of steps per episode as a function of the number of interactions. For all versions of the game, we observe a decreasing trend in the π-IW variants until the optimal policy is learned. In this case, episodes require progressively fewer steps because the learned policy converges to the optimal one. The convergence occurs in alignment with the reward per episode (top row) and again, we observe no significant difference between π-IW(1)-Basic and π-IW(1)-Dynamic. In contrast, AlphaZero shows a more irregular behavior. In the hardest instance, it ends up hitting a wall most of the time.

To illustrate the benefits of the structured exploration of π-IW compared to MCTS, we analyze the sample trajectories from the different algorithms after the first planning step, before any learning. We calculate the length of the trajectories (removing repeated states). On average, AlphaZero produces a tree whose longest trajectory is 3.83 (stdev $=$ 2.15). On the other hand, the longest trajectories of π-IW are 7.3 (stdev $= 2.5$) and 7.02 (stdev $= 2.25$), respectively for Basic and dynamic features (averages over 100 runs). Figure 6.4 illustrates this. Although there is no guidance towards rewarding states for any of the three algorithms (since the NNs have not been trained yet), π-IW reaches deeper parts of the state space than AlphaZero thanks to the structured exploration. As before, we

do not observe significant differences between using handcrafted features or the ones extracted from the NN.

From these results we can draw the following conclusions. First, existing width-based algorithms can be significantly improved by incorporating a guiding policy, as in π-IW. Second, we have shown that for this simple problem, a small set of features can be effectively learned without degrading the performance of π-IW. Finally, π-IW outperforms AlphaZero because it uses the structure of the state to explore more systematically and reach deeper states. In contrast, the exploration of MCTS needs to go through an optimal branch several times to increase its probability for action selection, since the policy estimate is based on counts.

6.3.3 π-IW on Atari Games

We end this experimental section presenting results of π-IW(1)-Dynamic on the pixel setting of the Atari suite. The aim of this section is to compare the performance of π-IW on a more challenging benchmark with existing width-based algorithms that use (predefined) pixel-based features. In particular, we consider IW and Rollout IW from Bandres, Bonet, and Geffner (2018). We do not provide results of AlphaZero in this benchmark, since our preliminary analysis showed poor performance, and the amount of hyperparameters to tune is considerably higher compared to π-IW.

We focus on a similar setting as in Bandres, Bonet, and Geffner (2018), where a short budget is given to the planner, although we do not aim to plan in real time. In our case, we set the budget of expanded nodes to 100, resulting in approximately one second per transition, considering *both planning and learning steps*. Note that this budget is very small compared to existing RAM-based methods, that allow 30,000 expanded nodes at each step (Lipovetzky, Ramirez, and Geffner 2015; Shleyfman, Tuisov, and Domshlak 2016).

Table 6.2 shows results in 54 Atari games (pixel setting) comparing π-IW using dynamic features with IW and Rollout IW using the B-PROST feature set. Results of π-IW are an average of the performance in the last 10 episodes of 5 independent runs, and results of IW and Rollout IW are taken from Bandres, Bonet, and Geffner (2018). All hyperparameters are kept the same as in Table 6.1 except for tree budget = 100, $|D| = 10^4$, $\tau = 0.5$, and frameskip = 15. The inputs of the NN are the last 4

Game	IW 0.5s	IW 32s	Rollout IW 0.5s	Rollout IW 32s	π-IW #100
Alien	1316.0	**14010.0**	4238.0	6896.0	3969.8
Amidar	48.0	1043.2	659.8	**1698.6**	950.4
Assault	268.8	336.0	285.6	319.2	**1574.9**
Asterix	1350.0	262500.0	45780.0	66100.0	**346409.1**
Asteroids	840.0	**7630.0**	4344.0	7258.0	1368.5
Atlantis	33160.0	82060.0	64200.0	**151120.0**	106212.6
Bank Heist	24.0	739.0	272.0	**865.0**	567.2
Battle zone	6800.0	14800.0	39600.0	**414000.0**	69659.4
Beam rider	715.2	1530.4	2188.0	2464.8	**3313.1**
Berzerk	280.0	1318.0	644.0	862.0	**1548.2**
Bowling	30.6	**49.2**	47.6	45.8	26.3
Boxing	99.4	79.0	75.4	79.4	**99.9**
Breakout	1.6	56.0	82.4	36.0	**92.1**
Centipede	88890.0	**143275.4**	36980.2	65162.6	126488.4
Chopper command	1760.0	1800.0	2920.0	5800.0	**11187.4**
Crazy climber	16780.0	44340.0	39220.0	43960.0	**161192.0**
Demon attack	106.0	23619.0	2780.0	9996.0	**26881.1**
Double dunk	-22.0	-22.4	3.6	**20.0**	4.7
Enduro	2.6	229.2	169.4	359.4	**506.6**
Fishing derby	-83.8	-39.0	-68.0	-16.2	**8.9**
Freeway	0.6	**25.0**	2.8	12.6	0.3
Frostbite	106.0	182.0	220.0	**5484.0**	270.0
Gopher	1036.0	**18472.0**	7216.0	13176.0	18025.9
Gravitar	380.0	1630.0	1630.0	**3700.0**	1876.8
HERO	2034.0	7432.0	13709.0	28260.0	**36443.7**
Ice hockey	-13.6	-7.0	-6.0	**6.6**	-11.7
James bond 007	40.0	180.0	450.0	**22250.0**	43.2
Kangaroo	160.0	3820.0	1080.0	**5780.0**	1847.5
Krull	3206.8	5611.8	1892.8	1151.2	**8343.3**
Kung-fu master	440.0	8980.0	2080.0	14920.0	**41609.0**
Montezuma's revenge	**0.0**	**0.0**	**0.0**	**0.0**	0.0
Ms. Pac-man	2578.0	**20622.8**	9178.4	19667.0	14726.3
Name this game	7070.0	**13478.0**	6226.0	5980.0	12734.8
Phoenix	1266.0	5550.0	5750.0	**7636.0**	5905.1
Pitfall!	**-8.6**	-92.2	-81.4	-130.8	-214.8
Pong	-20.8	0.8	-7.4	**17.6**	-20.4
Private eye	2690.8	-526.4	-322.0	**3157.2**	452.4
Q*bert	515.0	16505.0	3375.0	8390.0	**32529.6**
Road Runner	200.0	0.0	2360.0	37080.0	**38764.8**
Robotank	3.2	32.8	31.0	**52.6**	15.7
Seaquest	168.0	356.0	980.0	**10932.0**	5916.1
Skiing	-16511.0	-15962.0	**-15738.8**	-16477.0	-19188.3
Solaris	1356.0	2300.0	700.0	1040.0	**3048.8**
Space invaders	280.0	1963.0	2628.0	1980.0	**2694.1**
Stargunner	840.0	1340.0	13360.0	**15640.0**	1381.2
Tennis	-23.4	-22.2	-18.6	**-2.2**	-23.7
Time pilot	2360.0	5740.0	7640.0	8140.0	**16099.9**
Tutankham	71.2	172.4	128.4	184.0	**216.7**
Up'n down	928.0	62378.0	36236.0	44306.0	**107757.5**
Venture	0.0	**240.0**	0.0	80.0	0.0
Video pinball	28706.4	441094.2	203765.4	382294.8	**514012.5**
Wizard of wor	5660.0	**115980.0**	37220.0	73820.0	76533.2
Yars' revenge	6352.6	10808.2	5225.4	9866.4	**102183.7**
Zaxxon	0.0	15080.0	9280.0	22880.0	**22905.7**
# best	1	10	1	17	24

Table 6.2: Scores of width-based methods in 54 Atari games.

grayscale frames stacked to form a 4-channel image. Results of π-IW are an average of the last 10 episodes for 5 runs with different random seeds. Performance is measured after 20M generated nodes, i.e., interactions with the simulator (excluding skipped frames).[1]

First, if we compare π-IW against the methods that use a budget of 0.5 seconds (2nd and 4th columns vs 6th column), we observe that π-IW systematically outperforms both IW and rollout variants (see values in blue). Only in 13 games is either IW or Rollout IW better than our method, and only in five cases are both better. This shows that better performance can indeed be achieved at the cost of training the policy and learning the features.

Second, we observe (in bold) that π-IW outperforms all other methods in 24 games, and performance is comparable to the non-guided approaches in most other games (3rd and 5th columns vs 6th column). Remarkably, this is achieved with significantly less computational budget (approximately 30 times less) and without the need of predefined features.

These results suggest that a guiding policy can be beneficial, not only in terms of computational budget, but also in terms of the learned representation (in our case, the simple discretized features of the hidden layer) that can be directly exploited by the IW planner.

[1]The results initially published for π-IW in Junyent, Jonsson, and Gómez (2019) were partially affected by input noise due to a software bug (frames were incorrectly stacked after resetting the simulator to a previously seen state). Table 6.2 shows results with the bug amended. Since experiments are computationally expensive, we decided to rerun them with half the total interactions budget (20M instead of 40M). Despite the budget reduction, results of Table 6.2 are arguably better than those initially published.

Chapter 7

Hierarchical π-IW

The π-IW algorithm introduced in Chapter 6, effectively combines planning and learning by learning a policy π from the rewards observed in the IW tree, and uses π to guide future searches. However, in sparse-reward tasks, IW(1) may not reach any reward, especially when the planning horizon is too short. First, we extend the original π-IW in two ways: adding a better tie breaking mechanism, and a value function estimate. Then, we present a width-based method for high-level search that selects nodes according to visitation counts. Finally, we combine our hierarchical approach with policy guidance, resulting in the π-HIW algorithm, that we test in gridworld environments with sparser rewards as well as in the Atari suite.

7.1 Improvements to π-IW

The π-IW algorithm introduced in Chapter 6, effectively combines planning and learning by learning a policy π from the rewards observed in the IW tree, and uses π to guide future searches. However, in sparse-reward tasks, IW(1) may not reach any reward, especially when the planning horizon is too short. Here we extend the original π-IW in two ways: adding a better tie breaking mechanism, and a value function estimate. In experiments, we call this version π-IW+.

When no reward is found during planning, the target policy for the learning step becomes the uniform distribution, and π-IW behaves as Roll-

out IW. In this case, π-IW may take a step towards a region of the search tree with low node count, and presumably with less novel states, losing valuable structure information provided by the IW search. To avoid that, we modify the target policy of π-IW to use the node counts in the search tree for tie-breaking (i.e., the amount of descendants per action at the root node). The new target policy takes the form $\pi^{\text{target}} \propto \pi^{\text{rewards}} \cdot \pi^{\text{counts}}$, where the product is element-wise, and π^{counts} is a softmax distribution:

$$\pi^{\text{counts}}(a|s) = \frac{\exp\left(1/(\tau c(s,a)+1)\right)}{\sum_{a' \in A} \exp\left(c(s,a')\right)}$$

where τ is a temperature parameter and $c(s,a)$ is the amount of nodes in the subtree of action a. The temperature parameter for π^{rewards}, which is also defined as a softmax distribution but proportional to the returns $R(s,a)$, is typically close to zero to ensure a greedy target policy. Therefore, by performing the product, we achieve the effect of tie-breaking, especially if the temperature parameter for the counts is some orders of magnitude higher than the one for the rewards.

This tie-breaking may help finding deeper rewards. However, π-IW will not exploit this information in subsequent episodes, since π^{target} is still based on the rewards of the *current* planning horizon. To amend this, we learn a value function, which we combine with the observed rewards to generate a better estimate of π^{rewards}. When backpropagating the rewards from the leaves to the root, we take the maximum between the observed rewards and our value estimate.

To learn the parameterized policy estimate $\widehat{\pi}_\theta$, we take the same approach as before, using the cross-entropy loss to update the parameters θ, with the only difference being that target policy now uses visitation counts for tie-breaking. To learn the value function, we take a similar approach to MuZero (Schrittwieser et al. 2020) and use a support vector to represent the value function, that we also learn with the cross-entropy loss function. We keep using the Monte-Carlo return z_t as target instead of an n-step return as in MuZero. The overall loss is:

$$\mathcal{L} = -\pi_t^{\text{target}}(\cdot|s_t)^\top \log \widehat{\pi}_\theta(\cdot|s_t) - \psi(z_t)^\top \log(\widehat{v}_\theta) + c_{\ell 2}\|\theta\|^2, \tag{7.1}$$

where $\psi(\cdot) : \mathbb{R} \to [0,1]^{x_{\text{sup}}}$ is a mapping from Monte-Carlo returns z_t to their vector representations of $|x_{\text{sup}}| = 601$ integer supports, as described in Section 2.8.3, and once more the last term is an ℓ-2 regularization term.

7.2 Learning with Hierarchy

In this section we show how to combine Hierarchical IW (HIW) with our learning-based approach that uses a policy to direct search. First, we define a variation of Rollout IW for the high-level search. Then, we outline the aspects that need to be considered when using HIW in an online replanning setting.

7.2.1 Count-Based Rollout IW

Rollout IW(k) performs breadth-first search implicitly, from independent rollout trajectories. It maintains the notion of width by modifying the definition of novelty: a state s is considered novel if any w-tuple of features of s has not appeared at a lower depth. With this, the authors achieve an algorithm that is equivalent to IW(k), but with better anytime behavior. This novelty measure actually allows for many width-based algorithms, since it unties the order of expanding nodes from the novelty test.

In our hierarchical framework, a subset of states is encapsulated under the same high-level state (i.e., a set of high-level features). Selecting one high-level state or another directly determines which low-level states are generated. In order to balance exploration within high-level states, we extend Rollout IW with a selection method that depends on state visitation counts.

Our method, named Count-Based Rollout IW, is detailed in Algorithm 7.1. Similar to Rollout IW, it consists of two phases, node selection and rollout, that are iteratively called in function *Lookahead*. Differently from Rollout IW, we keep a list of OPEN nodes O (i.e., nodes that have not been pruned and need to be expanded), and we remove nodes from O whenever they need to be pruned, i.e., when a new node at a lower depth is generated such that it makes deeper nodes not novel anymore. The list of OPEN nodes is initialized with the root node and all unpruned cached nodes, in the case we are caching nodes from previous searches. Apart from O, we keep two mappings: N and C. N maps feature tuples to novel nodes, and is used for pruning, while C maps feature vectors to visit counts, and is used in the selection phase. N is initially empty, and C defaults to zero for all new feature vectors, and is reset to a new empty mapping at every episode.

Algorithm 7.1 Count-Based Rollout IW

initialize O: list of open nodes
initialize N: mapping from tuples to novel nodes
initialize C: mapping from feature vectors to counts
initialize D: depth-based novelty table

function LOOKAHEAD(O, N, C, D)
  while not StopCondition() **and not** IsEmpty(O) **do**
    n = Select(O, C)
    Rollout(n, O, N, C, D)

function SELECT(O, C)
  c = GetCounts(O, C) ▷ Get feature counts of nodes in O
  $p \propto \exp(1/\tau(c+1))$
  n = Sample(O, p)
  return n

function ROLLOUT(n, O, N, C, D)
  while not StopCondition() **do**
    $C[n.features]$++
    s = Successor(n)
    if s == *null* **then**
      Remove(n, O) ▷ Remove if n has been fully expanded
      return
    $s.T$ = CheckUpdateNovelty(D, $s.features$, $s.depth$)
    if IsEmpty($s.T$) **or** IsTerminal(s) **then**
      return
    PruneOther(s, O, N)
    Append(O, n) ▷ Add node to OPEN list
    $n = s$

function PRUNEOTHER(n, O, N)
  for t **in** $n.T$ **do**
    $o = N[t]$ ▷ Get previous node that had t as novel tuple
    Remove(t, $o.T$) ▷ Remove t from its novel tuples list
    if IsEmpty($o.T$) **then** ▷ If o has no more novel tuples
      PruneSubtree(o, O) ▷ Remove o and its descendants from O
    $N[t] = n$ ▷ Set n as the node that is novel due to tuple t

In function *Select*, a node n from O is selected according to a softmax probability distribution inversely proportional to the visitation counts of its feature vector, stored in mapping C. Then, a rollout is performed from n, generating a new successor node s at each iteration, until one that does not pass the novelty test is found. In this case, instead of returning a Boolean value, the novelty test function returns a set of novel tuples T. If the set is empty, then the node should not be added to the list of open nodes O, and a new node should be selected. If the set is not empty, we add the newly generated node s to the open list. We also store its novel tuples to recheck its novelty in subsequent iterations.

For each new novel node n, with novel tuples $n.T$, there may be other nodes deeper in the tree that were initially novel due to one or more tuples of T, which may need to be pruned. This is done in function *PruneOther*. We identify such nodes with N, that maps feature tuples to unpruned nodes. Then, for each tuple $t \in n.T$, we can check which other node $o = N[t]$ was novel due to t, and remove t from its set of novel nodes $o.T$. In the case the set becomes empty, then we should prune the node, removing it from the open list, together with its descendants. This is done in function *PruneSubtree*, which traverses the subtree rooted at o in a breadth-first manner removing all visited nodes from the open list O.

When pruning a node n, we leave the visitation count for the feature vector of n untouched. Thus, a new node with the same feature mapping generated at a lower depth will be selected according to the existing visitation count. With this, we ensure a balance between different high-level states.

7.2.2 Policy-Guided Hierarchical IW (π-HIW)

Hierarchical IW can be straightforwardly used for online replanning. At each step, we sample an action $a \sim \pi^{\text{target}} \propto \pi^{\text{rewards}} \cdot \pi^{\text{counts}}$. To generate π^{rewards}, we need to backpropagate the rewards through the hierarchical tree. Starting from the high-level leaf nodes, we first backpropagate the rewards of the associated low-level trees. Then, to propagate this return between two high-level nodes, we feed it to the corresponding low-level leaf nodes of the high-level parent, and repeat until we reach the high-level root. See Figure 7.1 (left) for an illustration. To generate π^{counts}, we backpropagate the counts in a similar manner.

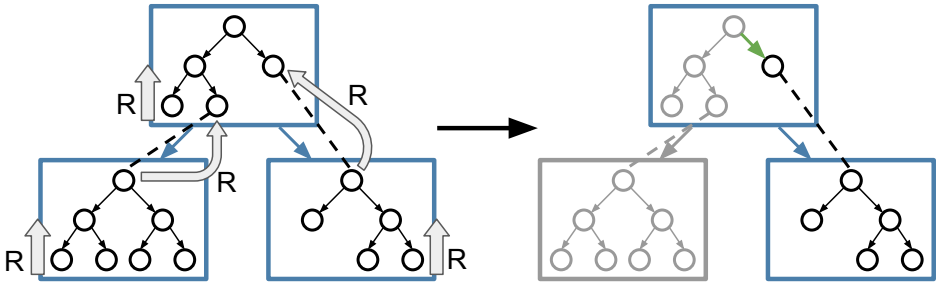

Figure 7.1: Illustration of reward propagation and action execution in a hierarchical search. Dashed lines represent low-level connections between leaves of the high-level parent nodes and roots of their high-level children. Left: rewards are propagated from the leaves to the root and through the low-level connections. Right: when the selected action (in green) is taken, nodes in gray become unreachable and can be discarded.

After executing an action a, we cache the resulting subtree for subsequent searches, similar to previous work. In this case, we need to take into account that some high-level states will not be reachable anymore, and we should thus remove them from the high-level tree before resuming the search. This is illustrated in Figure 7.1 (right), where the left high-level node together with its low-level tree is discarded (shown in gray) due to an action taken inside the high-level root node.

7.3 π-HIW in Pixel-Based Testbeds

In this section, we test our approach, π-HIW, in pixel-based gridworld environments and Atari games. We use two levels of abstraction: the high-level planner is Count-based Rollout IW (Algorithm 7.1) and the low-level planner is π-IW+ (i.e., Rollout IW guided by the current policy estimate). The set of abstract features $\phi_h(s)$ consists of a discretization of the image, similar to the one used in Go-Explore (Ecoffet et al. 2019, 2021), where the image is divided into tiles and the mean pixel value of each tile is taken as the feature value. Usually, this is further quantized into a smaller subset (e.g. 8 pixel values). For the low-level set of features, we follow our

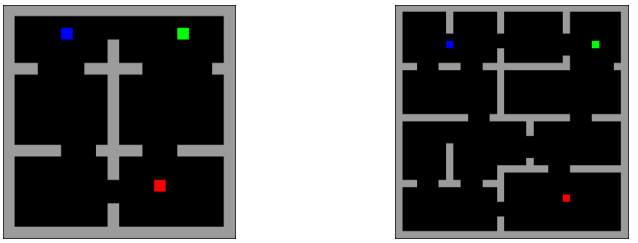

Figure 7.2: Snapshots of the two larger gridworld environments. Colors blue, red, green and gray represent the agent, key, door, and walls, respectively. The optimal policy takes 36 and 62 steps for the smaller (left) and larger (right) tasks, respectively.

previous approach and define $\phi_\ell(s)$ as the boolean discretization of $h(s)$, where h is the last layer of the neural network representing $\widehat{\pi}_\theta$.

7.3.1 Gridworld Environments

We test our algorithm in two gridworld environments depicted in Figure 7.2. The setting is similar to the one of Section 6.3.2, but with larger environments and therefore sparser rewards. The agent (blue square) is rewarded with $+1$ only when the door (green square) is reached while holding the key (red square). Any other state has a reward of 0, except if the agent hits a wall in which case the reward is -1. We also end the episode after 200 and 500 steps for the smaller and larger environment, respectively. Once more, the observation is a $84 \times 84 \times 3$ image and possible actions are {no-op, up, down, left, right}.

We compare our hierarchical approach, π-HIW$(1, 1)$, to two baselines: π-IW, and the modified version, which we call π-IW+, that uses a value estimate and the subtree size for tie-breaking. For the latter, we use a temperature of 1 to generate π^{counts}. In order to bound the memory used by the planner, we set a maximum of 5000 nodes that we keep in memory per step. The visitation count temperature used by the high-level planner (Algorithm 7.1) is set to 0.005. All other hyperparameters are the same as in Section 6.3.2.

Figure 7.3 shows results for both environments. We observe that π-IW

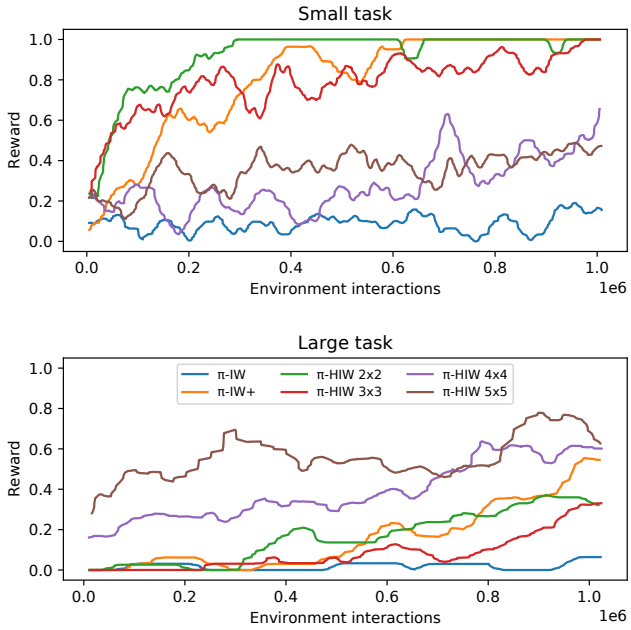

Figure 7.3: Comparison between π-IW, π-IW+, and π-HIW$(1, 1)$ with different discretizations in the gridworld environments.

does not perform well in these larger environments, obtaining a reward close to zero in both tasks. π-IW+ takes advantage of the value function and the tie-breaking counts and learns to solve the first task, while achieving a mean score of 0.5 for the second one in 10^6 interactions with the environment. For the hierarchical version, which also includes the aforementioned modifications, we report results of π-HIW$(1, 1)$ using different number of tiles in $\phi_h(s)$, and 256 values per tile. We observe how, for the smaller task, 2x2 tiles is enough to get a good performance, similar to the baseline π-IW+, and the performance degrades when increasing the number of tiles. In the larger task, π-HIW$(1, 1)$ outperforms the baseline, but it needs at least 4x4 tiles to perform well.

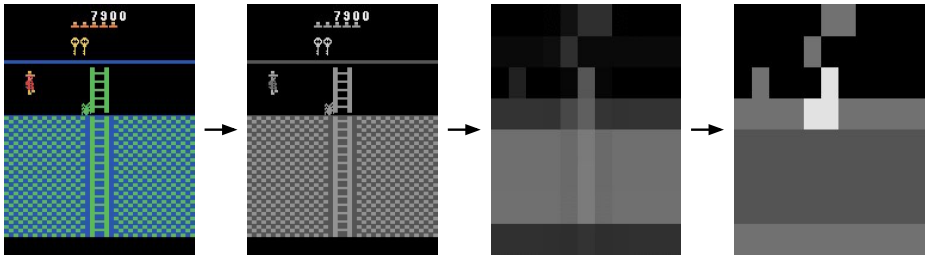

Figure 7.4: Downsampling of a frame in the game Montezuma's Revenge. After converting the image to grayscale, the regions formed by a 8×11 grid are averaged and subsampled, from 128 to 32 pixel values.

7.3.2 Atari Games

We finish this section with a set of experiments using the Atari simulator. In this case, we do not optimize the hyper-parameters and define $\mathcal{F}_h$ using 32 pixels values and 8×11 tiles. An example of this downsampling is given in Figure 7.4 Moreover, we use width $k_h = n = |\mathcal{F}_h|$ at the high level, i.e., π-HIW$(n, 1)$. Even though IW(n) explores the entire high-level state space, there is a single combination of n features, which makes the novelty check efficient. In the original IW algorithm, IW(n) is equivalent to a breadth-first search without state duplicates. Nevertheless, we use Count-Based Rollout IW, described in Algorithm 7.1. With this, we aim to achieve effective widths larger than 2.

Figure 7.6 shows a comparison between π-HIW$(n, 1)$ and π-IW using the same setup as in Section 6.3.3. Relative improvement is computed as $(s_{\pi\text{-HIW}} - s_{\text{rand}})/(s_{\pi\text{-IW}} - s_{\text{rand}})$, where $s_{\pi\text{-IW}}$ and $s_{\pi\text{-HIW}}$ are the scores of the flat and hierarchical versions, respectively, and s_{rand} is the score of a random agent taken from Wang et al. (2016). In the game Skiing, $s_{\pi\text{-IW}} < s_{\text{rand}}$, and therefore the relative improvement of π-HIW is not shown. All scores are shown in Table 7.3.2 except for the game Freeway, because the simulator was excessively slow compared to other games. Since π-HIW includes the improvements described in Section 7.1, we also present results of π-IW+ (i.e, the flat version with a value estimate and counts for breaking ties) in Table 7.3.2 for comparison.

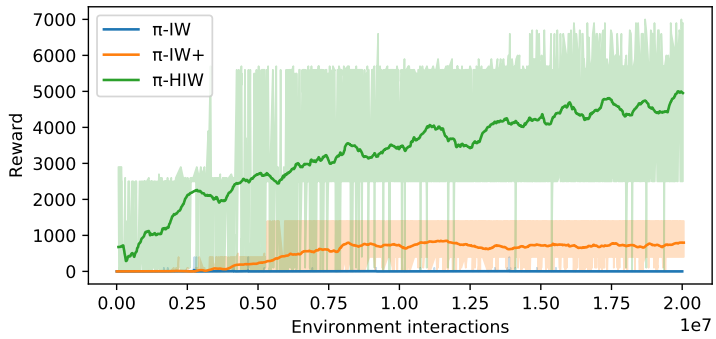

Figure 7.5: Performance of π-IW, π-IW+ and π-HIW in the Atari game Montezuma's Revenge. Average over 5 runs with different random seeds. Shades show the maximum and minimum values.

We observe that π-HIW improves over its predecessor π-IW in 28 games. Interestingly, games consisting of an agent moving in a fixed background present the best results e.g., James Bond, Private Eye, Pong, Frostbite, Chopper Command, etc. Within this type of games, π-HIW remarkably achieves a positive score in hard exploration games such as Montezuma's Revenge and Venture, a score not yet reported for any width-based planner (in Figure 7.6, the improvement for these games is ∞). Figure 7.5 shows the learning curve in the game of Montezuma's Revenge.

We also see an improvement in games with a moving background where the agent stays at a fixed position of the screen, for instance in Battle zone, Beam Rider, Road Runner, or Time Pilot. However, the size of the downsampling plays a big role. For instance, we find that performance drops in shooting games where the bullet is very small such as in Wizard of Wor, Asteroids, or Fishing derby, and specially if there is no agent moving as in Atlantis. We also observe poor performance in fight games where the agent may occupy many tiles, as in Boxing and Kung-fu master, or in noisy games (e.g., constantly changing background) as in Enduro and Krull. Finally, we can conclude that these results confirm that π-HIW can benefit from the state abstractions provided by a simple downsampling of the image, specially in navigation games.

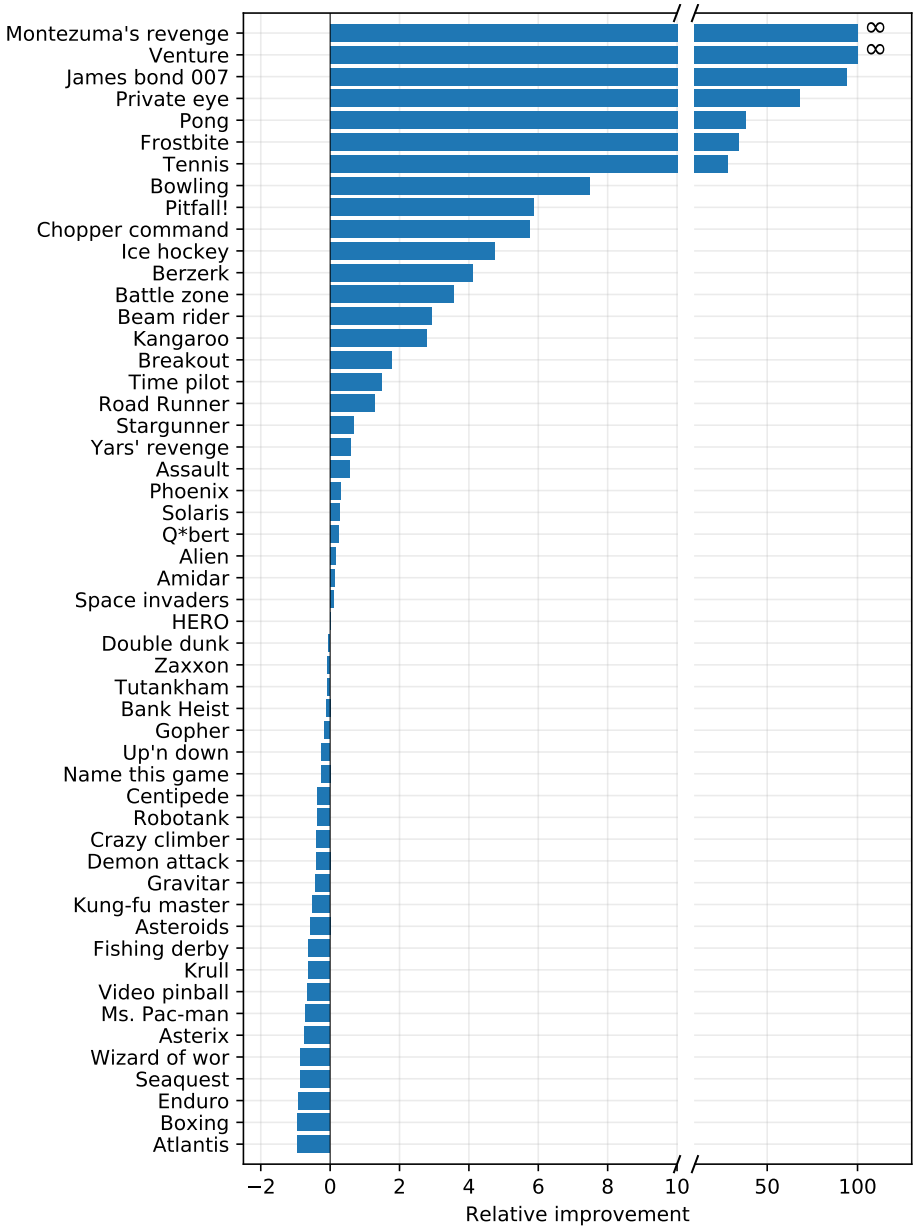

Figure 7.6: Relative improvement of π-HIW over π-IW in the Atari suite.

Game	π-IW(1)	π-IW(1)+	π-HIW$(n, 1)$
Alien	3969.78	2585.77	**4609.18**
Amidar	950.45	374.20	**1076.17**
Assault	1574.91	922.30	**2344.28**
Asterix	**346409.11**	247063.36	90017.25
Asteroids	1368.55	**1490.87**	990.95
Atlantis	106212.63	**143177.73**	17539.22
Bank Heist	**567.16**	256.29	501.68
Battle zone	69659.40	30848.95	**309137.79**
Beam rider	3313.11	8428.96	**11931.41**
Berzerk	1548.23	960.03	**7417.26**
Bowling	26.28	**78.18**	50.09
Boxing	**99.88**	88.19	6.81
Breakout	92.07	107.64	**252.88**
Centipede	126488.35	**141070.19**	80685.48
Chopper command	11187.44	3431.74	**70787.12**
Crazy climber	**161192.01**	138648.58	102205.99
Demon attack	26881.13	**35022.64**	16007.64
Double dunk	**4.68**	-16.80	3.51
Enduro	**506.59**	63.83	44.47
Fishing derby	**8.89**	-28.02	-53.76
Frostbite	270.00	1636.51	**7242.60**
Gopher	**18025.91**	7061.76	15001.18
Gravitar	**1876.80**	1532.33	1154.01
HERO	**36443.73**	22097.39	36231.21
Ice hockey	-9.66	-4.02	**-2.36**
James bond 007	43.20	205.91	**1380.13**
Kangaroo	1847.46	2918.98	**6861.57**
Krull	8343.30	**13014.77**	4121.81
Kung-fu master	**41609.03**	24871.94	20680.65
Montezuma's revenge	0.00	810.49	**5275.89**
Ms. Pac-man	**14726.33**	5916.86	4523.47
Name this game	12734.85	**18167.55**	9977.12
Phoenix	5905.12	**7647.67**	7508.63
Pitfall!	-214.75	**-2.46**	-128.82
Pong	-20.42	**2.14**	-9.70
Private eye	452.40	1766.13	**29548.76**
Q*bert	32529.60	23337.90	**40449.72**
Road Runner	38764.81	43813.29	**87953.53**
Robotank	**15.66**	9.68	10.63
Seaquest	**5916.05**	559.28	867.51
Skiing	-19188.32	**-13852.04**	-15417.86
Solaris	3048.78	1832.93	**3524.69**
Space invaders	2694.09	1622.49	**2946.18**
Stargunner	1381.24	1642.82	**1864.64**
Tennis	-23.67	**-8.26**	-20.00
Time pilot	16099.92	11126.86	**34610.25**
Tutankham	**216.67**	181.44	199.06
Up'n down	**107757.51**	59497.75	80991.07
Venture	0.00	**15.68**	10.73
Video pinball	**514012.51**	387308.60	184720.01
Wizard of wor	**76533.18**	30383.68	12027.43
Yars' revenge	102183.67	64544.51	**159496.20**
Zaxxon	**22905.73**	10159.01	21135.58
# best	19	14	21

Table 7.1: Comparison of π-IW(1), π-IW(1)+ and π-HIW$(n, 1)$ over 53 Atari games. Best score given in bold.

Chapter 8

Conclusions

First, we have provided new complexity bounds regarding the amount of nodes expanded by IW and its novelty check when the state space is represented by multivalued features, as it is common in the MDP setting. The use of IW with multivalued features is not new (Bandres, Bonet, and Geffner 2018; Lipovetzky, Ramirez, and Geffner 2015; Ramirez et al. 2018), but the complexity results are usually directly taken from the classical planning setting, where the state space is factored into a set of atoms, instead. Our recursive formula to count the maximum amount of novel states, given in Proposition 1, is based on two basic premises: two feature values cannot appear simultaneously, and a feature value appears in many tuples at the same time.

The recursion supposes a divide and conquer strategy for counting novel states, where we divide states into two groups (the two terms of the recursion): those that are novel exclusively due to one feature, and those partially due to other features. In Theorem 1, we provide a general (non-recursive) formula for counting novel states, and show that it can differ many orders of magnitude from previous bounds which are based on atoms that do not consider feature value constraints. We have also discussed the implications of the novelty check and update in the overall IW(k) complexity, providing upper and lower complexity bounds when only features that change are checked.

We have presented a hierarchical approach to blind search. Our method uses different feature mappings to create several levels of abstraction, al-

lowing different search algorithms at different levels of the planning hierarchy. Specifically, we propose to use Iterated Width at two levels, resulting in the hierarchical search algorithm HIW(k_h, k_ℓ). We show that HIW(k_h, k_ℓ) can solve problems of width $k_h + k_\ell$ with the right choice of high-level features, effectively reducing the algorithm complexity.

When high-level features are not provided, we show that they can be extracted from a previous IW search. Our method for finding abstract features extracts feature candidates from pruned branches of the IW search tree that meet a certain criteria. We combine this automatic feature extraction approach with an incremental method that performs HIW searches, discovering new high-level features at each iteration. We test our approach in classical planning benchmarks, showing that our incremental HIW$(1, 1)$ algorithm outperforms IW(2) in domains with goals consisting of a single atom. Specifically, our approach presents a better anytime behavior than IW(2), solving more instances while generally expanding less nodes and taking less time. This shows the power of hierarchies to reduce search complexity.

In the third part of the dissertation, we have presented π-IW, an algorithm that effectively combines width-based planning and learning. Our approach learns a compact policy using the exploration power of IW(1), which helps reaching distant high-reward states. We use the transitions recorded by IW(1) to train a policy in the form of a neural network. Simultaneously, the search is informed by the current policy estimate, reinforcing promising paths. We have shown that the learned representation by the policy network can be used as a feature space for IW, removing the need of pre-defined features. Interestingly, π-IW can even learn representations that reduce the width of a task.

Our algorithm operates in a similar manner to AlphaZero, except that the exploration relies on the pruning mechanism of IW, it does not keep a value estimate (in our first approach), and the target policy is based on observed rewards rather than visitation counts. Compared to AlphaZero and previous width-based methods, π-IW has superior performance in simple sparse reward environments. In the Atari 2600 benchmark, π-IW achieves a similar performance to Rollout IW with a much smaller planning budget and without the need to provide pre-defined features. Our first approach, however, is not able to achieve a positive score in very

challenging tasks such as Montezuma's Revenge.

We improve our algorithm by learning a value function that is used to augment the policy target with distant reward information. Furthermore, we take advantage of the IW structured exploration when no rewards are present by breaking ties with state visit counts, effectively caching branches with a higher number of states for the subsequent search. With this improvements, π-IW is able to achieve a positive score in Montezuma's Revenge.

Finally, we show that our policy learning and guidance scheme can be seeamlessly integrated with the proposed hierarchical approach. For the high-level search, we have presented a novel width-based algorithm that selects nodes according to a visitation count, leveraging the depth-based novelty tables from Rollout IW. The algorithm favors newly generated nodes (as in Rollout IW) and, in contrast with previous algorithms that select already visited nodes at random, it maintains a balance to revisit already generated nodes. In contrast with other algorithms, it does not use the tree structure to select a new node.

In experiments, we show that features extracted from a simple grid downsampling can boost performance in navigation tasks, where π-HIW outperforms the flat version when we make the reward sparser. In Atari games, because of the downsampled high-level features, our method presents the best improvements on tasks that consist of either an agent moving in a fixed background, or a moving background with the agent in a fixed position. Among these games, we find Montezuma's Revenge, where π-HIW improves over the value-based flat version by a factor of five.

8.1 Future Perspectives

The algorithms that we have presented contemplate environments where actions have deterministic effects. However, many processes are stochastic. In the future, we would like to bring width-based planning methods closer to the RL setting by allowing stochastic state transitions. If IW(1) can successfully explore stochastic domains, then our approach for training a policy as well as the presented hierarchical framework should still apply. As discussed in Section 3.4, width-based algorithms have already been applied in stochastic domains, but the approach is based on applying

an action several times in order to gather statistics before generating a node (Geffner and Geffner 2015). Another approach would be to not reuse cached nodes for the subsequent search so that noise is cancelled out throughout the online replanning. However, in situations with small computational budgets, both approaches would perform poorly, and we think that further research on this direction is necessary.

Width-based planning algorithms as well as Monte-Carlo tree search methods require a resettable simulator. Although this is a milder requirement than assuming explicit models at hand, as in most planning algorithms, it makes online replanning methods less appealing than RL algorithms. One line of research would be to explore the performance of width-based methods with a learned model, similar to the approach of MuZero (Schrittwieser et al. 2020). Another aspect from AlphaZero and MuZero that could be incorporated to our algorithms is to distribute learning across many machines, as well as to produce experience by interacting with many instances of the same environment concurrently.

Together with our policy learning scheme, we propose to automatically extract the feature set from the last hidden layer of the neural network. There have been other approaches that learn features for IW. For instance, Dittadi, Drachmann, and Bolander (2021) use the states generated by an initial IW search with B-PROST features to train a variational autoencoder. Then, they use the learned features in a second search, achieving better results than in the initial one. The main drawback of their approach is that features still need to be specified in the first search. A promising line of research would be to train the autoencoder online. For instance, we could start with the random features produced by the neural network (initialized with random parameters), as in our approach, and train the autoencoder with experience from the same search. This would waive the need of predefining any features, or perform an initial search. Here, we could add our policy guidance mechanism, and even combine features extracted from the policy with features generated by the autoencoder. Both approaches should learn different types of features: we should expect autoencoder features to capture information about objects in the image, and policy features to capture goal-related information. For instance, in the game of Pong, the ball represents a very small part of the image, and may not have a big impact in autoencoder features. However, it is key for the

policy, and therefore should be highly represented by the policy features.

All these approaches for learning features are generic, and do not take into account how they are used in the IW search. A very promising research line is how to actively learn features that reduce the width of the problem. In our approach, this is a consequence of the policy learning, but there is no term in the loss function that encourages this behavior. How to define such a loss term is still an open question. We believe that learning such width-reduction features would definitely boost the potential of width-based methods.

In our hierarchical approach, the levels of abstraction are determined by different feature sets. In Atari games, for instance, we use features extracted from a simple downsampling, and show that these features help in navigation games. However, this is not a general approach, and ideally, we wish to not have to predefine the high-level feature set. The quality of the features at the high level has a direct impact on the low level searches, and therefore on the overall performance of our algorithm. Although we provide an algorithm to identify promising high-level feature candidates, looking at those that may be splitting and hopefully reduce the problem width, how to integrate this or any other approach to identify features with online learning is still an open research question that we would like to explore in the future.

Finally, IW relies on the notion of state novelty to perform structured exploration. Recent RL methods have proposed to exploit other notions of novelty for systematic exploration (Bellemare et al. 2016; Martin et al. 2017; Ostrovski et al. 2017). In these approaches, an RL agent is rewarded when it faces novel experience. Burda et al. (2019) presents what has been known as the *noisy TV problem*, where a TV showing random noisy images attracts the attention of the agent forever, that becomes a "couch potato", and fails to make any progress. In this scenario, IW would dedicate a huge amount of resources exploring the noisy TV. In fact, this may happen with any feature that is not *controllable*, i.e., that the agent cannot modify with its actions. Thus, it would be desirable to identify such features, in order to concentrate exploration in controllable regions of the state space. One possibility would be to use extract such information from a learned model, or to use the already generated states as a heuristic of whether or not a certain feature can be changed by an action.

Bibliography

Abadi, Martín et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, pp. 265–283.

Arel, Itamar et al. (2010). "Reinforcement learning-based multi-agent system for network traffic signal control". In: *IET Intelligent Transport Systems* 4.2, pp. 128–135.

Arulkumaran, Kai et al. (2017). "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6, pp. 26–38.

Auer, Peter, Nicolo Cesa-Bianchi, and Paul Fischer (2002). "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2, pp. 235–256.

Bäckström, Christer (1995). "Expressive equivalence of planning formalisms". In: *Artificial Intelligence* 76.1-2, pp. 17–34.

Bäckström, Christer and Bernhard Nebel (1995). "Complexity results for SAS+ planning". In: *Computational Intelligence* 11.4, pp. 625–655.

Baird, Leemon (1995). "Residual algorithms: Reinforcement learning with function approximation". In: *Machine Learning Proceedings 1995*, pp. 30–37.

Bandres, Wilmer, Blai Bonet, and Hector Geffner (2018). "Planning With Pixels in (Almost) Real Time". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Barth-Maron, Gabriel et al. (2018). "Distributed Distributional Deterministic Policy Gradients". In: *International Conference on Learning Representations*.

Bast, Hannah et al. (2016). "Route planning in transportation networks". In: *Algorithm engineering*, pp. 19–80.

Beattie, Charles et al. (2016). "Deepmind lab". In: *arXiv preprint arXiv:1612.03801*.

Bellemare, Marc G, Will Dabney, and Rémi Munos (2017). "A distributional perspective on reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 449–458.

Bellemare, Marc G et al. (2013). "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bellemare, Marc G et al. (2016). "Unifying Count-Based Exploration and Intrinsic Motivation". In: *Advances in Neural Information Processing Systems*. Im, pp. 1–26.

Bertsekas, Dimitri P (2019). *Reinforcement learning and optimal control*.

Bonet, Blai and Héctor Geffner (2001). "Planning as heuristic search". In: *Artificial Intelligence* 129.1-2, pp. 5–33.

Bonet, Blai and Hector Geffner (2021). "General Policies, Representations, and Planning Width". In: *Proc. AAAI*.

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: `arXiv:1606.01540`.

Browne, Cameron B et al. (2012). "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1, pp. 1–43.

Burda, Yuri et al. (2019). "Exploration by random network distillation". In: *International Conference on Learning Representations*.

Camacho, Eduardo F and Carlos Bordons Alba (2013). *Model predictive control*.

Chentanez, Nuttapong, Andrew G. Barto, and Satinder P. Singh (2005). "Intrinsically Motivated Reinforcement Learning". In: *Advances in Neural Information Processing Systems 17*. Ed. by L. K. Saul, Y. Weiss, and L. Bottou, pp. 1281–1288.

Crosby, Matthew et al. (2020). "The Animal-AI Testbed and Competition". In: *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*. Ed. by Hugo Jair Escalante and Raia Hadsell. Vol. 123. Proceedings of Machine Learning Research, pp. 164–176.

Currie, Ken and Austin Tate (1991). "O-Plan: the open planning architecture". In: *Artificial intelligence* 52.1, pp. 49–86.

Dabney, Will et al. (2018a). "Distributional reinforcement learning with quantile regression". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Dabney, Will et al. (2018b). "Implicit quantile networks for distributional reinforcement learning". In: *International conference on machine learning*. PMLR, pp. 1096–1105.

Dittadi, Andrea, Frederik K. Drachmann, and Thomas Bolander (2021). "Planning from Pixels in Atari with Learned Symbolic Representations". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.6, pp. 4941–4949.

Ecoffet, Adrien et al. (2019). "Go-explore: a new approach for hard-exploration problems". In: *arXiv preprint arXiv:1901.10995*.

Ecoffet, Adrien et al. (2021). "First return, then explore". In: *Nature* 590.7847, pp. 580–586.

Erol, Kutluhan, James Hendler, and Dana S Nau (1996). "Complexity results for HTN planning". In: *Annals of Mathematics and Artificial Intelligence* 18.1, pp. 69–93.

Espeholt, Lasse et al. (2018). "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures". In: *International Conference on Machine Learning*. PMLR, pp. 1407–1416.

Fikes, Richard E, Peter E Hart, and Nils J Nilsson (1972). "Learning and executing generalized robot plans". In: *Artificial intelligence* 3, pp. 251–288.

Fikes, Richard E. and Nils J. Nilsson (1971). "Strips: A new approach to the application of theorem proving to problem solving". In: *Artificial Intelligence* 2.3, pp. 189–208. DOI: https://doi.org/10.1016/0004-3702(71)90010-5.

Flórez, José et al. (2011). "Planning multi-modal transportation problems". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 21. 1.

Fortunato, Meire et al. (2018). "Noisy Networks For Exploration". In: *International Conference on Learning Representations*.

Fox, Maria, Derek Long, and Daniele Magazzeni (2011). "Automatic construction of efficient multiple battery usage policies". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 21. 1.

Frances, Guillem et al. (2017). "Purely declarative action descriptions are overrated: Classical planning with simulators". In: *IJCAI 2017. Twenty-Sixth International Joint Conference on Artificial Intelligence; 2017 Aug 19-25; Melbourne, Australia.[California]: IJCAI; 2017. p. 4294-301.*

Fujimoto, Scott, Herke Hoof, and David Meger (2018). "Addressing function approximation error in actor-critic methods". In: *International Conference on Machine Learning*. PMLR, pp. 1587–1596.

García, L. et al. (2015). "Modeling and real-time control of urban drainage systems: A review". In: *Advances in Water Resources* 85, pp. 120–132. DOI: https://doi.org/10.1016/j.advwatres.2015.08.007.

Geffner, Hector and Blai Bonet (2013). *A Concise Introduction to Models and Methods for Automated Planning*.

Geffner, Tomas and Hector Geffner (2015). "Width-based planning for general video-game playing". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 11. 1.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*.

Guan, Dah-Jyh (1998). "Generalized Gray codes with applications". In: *Proceedings of the National Science Council, Republic of China. Part A, Physical science and engineering* 22.6, pp. 841–848.

Guo, Xiaoxiao et al. (2014). "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27.

Guss, William H. et al. (2019). "MineRL: A Large-Scale Dataset of Minecraft Demonstrations". In: *Twenty-Eighth International Joint Conference on Artificial Intelligence*.

Haarnoja, Tuomas et al. (2018). "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International Conference on Machine Learning*. PMLR, pp. 1861–1870.

Hausknecht, Matthew and Peter Stone (2015). "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15)*.

Henderson, Peter et al. (2018). "Deep reinforcement learning that matters". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Hessel, Matteo et al. (2018). "Rainbow: Combining improvements in deep reinforcement learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Hester, Todd et al. (2018). "Deep q-learning from demonstrations". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Hoffmann, Jörg and Bernhard Nebel (2001). "The FF planning system: Fast plan generation through heuristic search". In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Jinnai, Yuu and Alex S Fukunaga (2017). "Learning to Prune Dominated Action Sequences in Online Black-Box Planning". In: *AAAI Conference on Artificial Intelligence*.

Junyent, Miquel, Vicenç Gómez, and Anders Jonsson (2021). "Hierarchical Width-Based Planning and Learning". In: *Proceedings of the 31st International Conference on Automated Planning and Scheduling*. Vol. 31. ICAPS, pp. 519–527.

Junyent, Miquel, Anders Jonsson, and Vicenç Gómez (2018). *Improving width-based planning with compact policies*. ICML / IJCAI / AAMAS Workshop on Planning and Learning.

Junyent, Miquel, Anders Jonsson, and Vicenç Gómez (2019). "Deep Policies for Width-Based Planning in Pixel Domains". In: *Proceedings of the 29th International Conference on Automated Planning and Scheduling*. Vol. 29. ICAPS, pp. 646–654.

Kahneman, Daniel (2011). *Thinking, fast and slow*.

Kapturowski, Steven et al. (2018). "Recurrent experience replay in distributed reinforcement learning". In: *International conference on learning representations*.

Kearns, Michael, Yishay Mansour, and Andrew Y Ng (2002). "A sparse sampling algorithm for near-optimal planning in large Markov decision processes". In: *Machine learning* 49.2, pp. 193–208.

Kearns, Michael and Satinder Singh (2002). "Near-optimal reinforcement learning in polynomial time". In: *Machine learning* 49.2-3, pp. 209–232.

Kempka, Michal et al. (2016). "ViZDoom: A Doom-based AI research platform for visual reinforcement learning". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8. DOI: 10.1109/CIG.2016.7860433.

Knoblock, Craig A. (1990). "Learning Abstraction Hierarchies for Problem Solving". In: *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 2*, pp. 923–928.

Kober, Jens, J Andrew Bagnell, and Jan Peters (2013). "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274.

Kocsis, Levente and Csaba Szepesvári (2006). "Bandit based monte-carlo planning". In: *European conference on machine learning*. Springer, pp. 282–293.

Korf, Richard E (1985). "Macro-operators: A weak method for learning". In: *Artificial intelligence* 26.1, pp. 35–77.

Levine, Sergey et al. (2016). "End-to-End Training of Deep Visuomotor Policies". In: *Journal of Machine Learning Research* 17.39, pp. 1–40.

Li, Yuxi (2018). "Deep reinforcement learning". In: *arXiv preprint arXiv:1810.06339*.

Liang, Yitao et al. (2016). "State of the Art Control of Atari Games Using Shallow Reinforcement Learning". In: *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems* Aamas, pp. 485–493.

Lillicrap, Timothy P. et al. (2016). "Continuous control with deep reinforcement learning." In: *ICLR*.

Lipovetzky, Nir and Hector Geffner (2012). "Width and Serialization of Classical Planning Problems". In: *Proceedings of the 20th European Conference on Artificial Intelligence*, pp. 540–545.

Lipovetzky, Nir and Hector Geffner (2017a). "Best-first width search: Exploration and exploitation in classical planning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1.

Lipovetzky, Nir and Hector Geffner (2017b). "A polynomial planning algorithm that beats LAMA and FF". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 27. 1.

Lipovetzky, Nir, Miquel Ramirez, and Hector Geffner (2015). "Classical Planning with Simulators: Results on the Atari Video Games."

In: *International Joint Conference on Artificial Intelligence*. Vol. 15, pp. 1610–1616.

Lowe, Ryan et al. (2017). "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA, pp. 6382–6393.

Machado, Marlos C et al. (2018). "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents". In: *Journal of Artificial Intelligence Research* 61, pp. 523–562.

Mao, Hongzi et al. (2016). "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM workshop on hot topics in networks*, pp. 50–56.

Martin, Jarryd et al. (2017). "Count-Based Exploration in Feature Space for Reinforcement Learning". In: *International Joint Conference on Artificial Intelligence*.

McDermott, Drew (2000). "The 1998 AI Planning Systems Competition". In: *AI Magazine* 21.2, p. 35. DOI: 10.1609/aimag.v21i2.1506.

McDermott, Drew et al. (1998). *PDDL - The Planning Domain Definition Language*. Tech. rep. CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control.

Mirhoseini, Azalia et al. (2021). "A graph placement methodology for fast chip design". In: *Nature* 594.7862, pp. 207–212.

Mnih, Volodymyr et al. (2013). "Playing Atari With Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop*.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Mnih, Volodymyr et al. (2016). "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR, pp. 1928–1937.

Mohsenian-Rad, Amir-Hamed et al. (2010). "Autonomous Demand-Side Management Based on Game-Theoretic Energy Consumption Scheduling for the Future Smart Grid". In: *IEEE Transactions on Smart Grid* 1.3, pp. 320–331. DOI: 10.1109/TSG.2010.2089069.

Neu, Gergely, Vicenç Gómez, and Anders Jonsson (2017). "A Unified View of Entropy-Regularized Markov Decision Processes". In: *Deep Reinforcement Learning Symposium, NIPS*.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In:

Ostrovski, Georg et al. (2017). "Count-Based Exploration with Neural Density Models". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70, pp. 2721–2730.

Paden, Brian et al. (2016). "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles". In: *IEEE Transactions on Intelligent Vehicles* 1.1, pp. 33–55. DOI: 10.1109/TIV.2016.2578706.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32.

Pathak, Deepak et al. (2017). "Curiosity-driven exploration by self-supervised prediction". In: *International Conference on Machine Learning*. PMLR, pp. 2778–2787.

Plappert, Matthias et al. (2018). "Parameter Space Noise for Exploration". In: *International Conference on Learning Representations*.

Puigdomènech Badia, Adrià et al. (2020). "Agent57: Outperforming the atari human benchmark". In: *International Conference on Machine Learning*. PMLR, pp. 507–517.

Racanière, Sébastien et al. (2017). "Imagination-augmented agents for deep reinforcement learning". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 5694–5705.

Ramirez, Miquel et al. (2018). "Integrated Hybrid Planning and Programmed Control for Real Time UAV Maneuvering". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 1318–1326.

Richter, Silvia, Malte Helmert, and Matthias Westphal (2008). "Landmarks Revisited." In: *AAAI*. Vol. 8, pp. 975–982.

Rosin, Christopher D (2011). "Multi-armed bandits with episode context". In: *Annals of Mathematics and Artificial Intelligence* 61.3, pp. 203–230.

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell (2011). "A reduction of imitation learning and structured prediction to no-regret online learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR, pp. 627–635.

Sacerdoti, Earl D (1974). "Planning in a hierarchy of abstraction spaces". In: *Artificial intelligence* 5.2, pp. 115–135.

Schaul, Tom et al. (2016). "Prioritized Experience Replay". In: *International Conference on Learning Representations*. Puerto Rico.

Schrittwieser, Julian et al. (2020). "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839, pp. 604–609.

Schulman, John et al. (2015). "Trust region policy optimization". In: *International conference on machine learning*. PMLR, pp. 1889–1897.

Schulman, John et al. (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Shleyfman, Alexander, Alexander Tuisov, and Carmel Domshlak (2016). "Blind Search for Atari-Like Online Planning Revisited". In: *International Joint Conference on Artificial Intelligence*, pp. 3251–3257.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, pp. 484–489.

Silver, David et al. (2017). "Mastering the game of go without human knowledge". In: *nature* 550.7676, pp. 354–359.

Silver, David et al. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419, pp. 1140–1144. DOI: 10.1126/science.aar6404.

Still, Susanne and Doina Precup (2012). "An information-theoretic approach to curiosity-driven reinforcement learning". In: *Theory in Biosciences* 131.3, pp. 139–148.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*.

Todorov, Emanuel, Tom Erez, and Yuval Tassa (2012). "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1.

Wang, Ziyu et al. (2016). "Dueling network architectures for deep reinforcement learning". In: *International conference on machine learning*. PMLR, pp. 1995–2003.

Wang, Ziyu et al. (2017). “Sample Efficient Actor-Critic with Experience Replay”. In: *5th International Conference on Learning Representations, ICLR 2017*.

Wymann, Bernhard et al. (2014). *TORCS, The Open Racing Car Simulator*. http://www.torcs.org.

Yaramasu, Venkata and Bin Wu (2016). *Model predictive control of wind energy conversion systems*.

Zheng, Guanjie et al. (2018). “DRN: A deep reinforcement learning framework for news recommendation”. In: *Proceedings of the 2018 World Wide Web Conference*, pp. 167–176.

Zhou, Zhenpeng, Xiaocheng Li, and Richard N Zare (2017). “Optimizing chemical reactions with deep reinforcement learning”. In: *ACS central science* 3.12, pp. 1337–1344.