

Beyond Markov Decision Processes: How to Leverage Structure to Build Efficient Reinforcement Learning Algorithms

I M P E R I A L

Arnaud Robert

Supervisors: Prof. A. Aldo Faisal & Dr. Ciara Pike-Burke

Department of Computing

Imperial College London

This dissertation is submitted in part fulfilment of the requirements for the
degree of *Doctor of Philosophy* in Computing of Imperial College London
and the Diploma of Imperial College London

July 2025

Declaration

I hereby declare that except where explicit reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Arnaud Robert

July 2025

Abstract

A Markov Decision Process (MDP) describes a general framework for modelling decision-making in an uncertain environment. Over the years, researchers have mainly focused on developing Reinforcement Learning (RL) algorithms to learn how to behave optimally in unknown environments without making explicit assumptions about the existence of underlying structures in the MDP.

This general approach has been instrumental to RL’s widespread success across various domains. However, when additional structure is present and exploitable, enabling algorithms to leverage it might lead to significant efficiency improvement. This work investigates the benefits of designing RL algorithms that can efficiently leverage these structures. Specifically, this work focuses on two distinct types of known latent structures.

First, we consider MDPs that exhibit a hierarchical structure; that is, tasks described by such MDPs can be decomposed into a sequence of sub-tasks. In this context, we provide a lower bound on the sample complexity of hierarchical RL algorithms, which allows us to quantify the potential benefit of hierarchical approaches. We also offer a framework for building hierarchical algorithms that leverage a known hierarchical decomposition. The validity of that framework is supported by theoretical guarantees of its efficiency and empirical evidence that it outperforms its monolithic counterpart whenever a hierarchical structure is present.

Second, we consider MDPs that exhibit a graphical structure. The algorithm has access to a graph that encodes the conditional independence between state variables, and the unknown dynamics can be inferred only by local components of the graph. In this context, we

provide a posterior sampling-based algorithm that theoretically and empirically outperforms RL algorithms that do not leverage this structural property. Finally, we provide empirical evidence that this latent graphical structure is present in optimising wind farms' yields and demonstrate the efficiency of our algorithm on that particular task.

Acknowledgements

I am deeply grateful to my two supervisors, Prof. Aldo Faisal and Dr. Ciara Pike-Burke, for their unwavering patience, boundless enthusiasm, and invaluable guidance during all those years. From the very first day, Aldo's trust in me and enthusiasm gave me the confidence to trust my ideas and persevere. The academic freedom I was granted at the Brain and Behaviour lab was instrumental in allowing me to collaborate with Ciara at a later stage. Ciara's patient introduction to the fascinating world of reinforcement learning theory was a turning point in my academic journey. Working with Ciara was a tremendous pleasure, and I learned a lot from her exceptional clarity of thought and dedication.

I am also grateful to have spent these years with brilliant PhD students and researchers in the Brain and Behaviour lab and Ciara's group. I loved the endless discussions about Reinforcement Learning and Science during each lunch break, especially with Max Grogan, Paul Festor, Filippo Valdetaro, Bukeikhan Omarali, Balasundaram Kadirvelu and all the Brain and Behaviour lab members. I also particularly enjoyed the time spent preparing, presenting and attending Ciara's reading group. It took over a year for Pavithra Srinath, Benjamin Howson, Emmeran Johnson, Joseph Lazzaro and I to decipher the 38 chapters of the fascinating Bandit Algorithms book [74].

During my PhD, I was lucky to collaborate with two other PhD students and their advisors. I am grateful to Siyi Li and his supervisor, Prof. Matthew Piggott, for sharing their knowledge about the fascinating world of wind farm optimization. I learned a lot from Siyi's great

expertise in Deep Learning, and the content of Chapter 6 would not have been possible without this collaboration. I am also deeply grateful to Gianluca Drappo and his supervisor, Prof. Marcello Restelli. Gianluca visited Ciara's group in London for three months. I am very proud of the work we did together, which I detail in Chapter 4.

The final version of this thesis also benefited from the feedback of my two examiners, Dr. Antonio Del Rio Chanona and Prof. Long Tran-Thanh. I also wanted to thank them for making my viva such an enjoyable and memorable discussion.

The above could lead someone to believe that the PhD experience is simply a sequence of encounters with brilliant researchers, ultimately resulting in the writing of a thesis. I have been lucky enough for this to be partly true, but I do not forget the countless periods of doubts and challenges that I had to overcome to complete this work. Overcoming these challenges was only possible because of the support I received from my family and friends, particularly my parents, Muriel and Christian Robert.

I am most grateful to my wife, Roxane Pedrazzini.

You made it so easy; every second thought and every doubt became meaningless after a single discussion with you. Your energy, joy, and determination are contagious; this manuscript greatly benefited from these qualities. This work is for you, and I hope that over the years, I will be able to show as much support as you have demonstrated during these last four years.

Table of contents

List of figures	xv
Notation	xix
1 Introduction	1
1.1 Motivation	1
1.2 Approach	2
1.3 Structure of the Thesis	6
1.3.1 RL on Hierarchical MDPs	6
1.3.2 RL on Graphs	8
1.4 Publications and Contributions	9
2 Background	13
2.1 Finite-Horizon Markov Decision Processes	13
2.1.1 Examples	15
2.1.2 Planning in an MDP	17
2.2 Learning in an Unknown MDP	20
2.2.1 Performance Metrics	20
2.2.2 Upper Bounds and Lower Bounds	22
2.3 Algorithms	27
2.3.1 Upper Confidence Bound Value Iteration (UCBVI)	28

2.3.2	Posterior Sampling for Reinforcement Learning (PSRL)	32
2.3.3	Q-learning	34
2.3.4	Q-UCB	37
2.3.5	Upper Confidence Fixed-Horizon Reinforcement Learning (UCFH)	38
3	A Lower Bound on HRL Efficiency	43
3.1	Motivation	44
3.2	General Formulation of Goal-Conditioned HRL	47
3.2.1	Stationary Finite-Horizon Markov Decision Process	47
3.2.2	Episodic Finite-Horizon Hierarchical MDP	49
3.2.3	Probably-Approximately Correct RL	53
3.2.4	Running Example	54
3.3	Lower Bound on the Sample Complexity of HRL	55
3.3.1	Interpretation of the Sample Complexity Bound:	58
3.4	Proof of the Lower Bound	60
3.5	Stationary Hierarchical Q-Learning (SHQL)	64
3.6	Experiments	66
3.6.1	N-Rooms Environment	66
3.6.2	Gymnasium Taxi Task	72
3.7	Related work	75
3.7.1	The Options Framework and Goal-Conditioned HRL	76
3.8	Conclusion	78
4	An Upper Bound on HRL Regret	79
4.1	Introduction	80
4.2	Related work	81
4.3	Problem Setting	83

4.3.1	Goal-based Finite-Horizon Markov Decision Process with Sparse Reward	83
4.3.2	Hierarchical MDP	84
4.3.3	Learning in a Hierarchical MDP	91
4.4	Goal-conditioned Hierarchical Reinforcement Learning Algorithm (GHRL)	93
4.5	Upper Bounds on the Regret of GHRL	98
4.5.1	High-level Plans and Failure Events	99
4.5.2	Optimism	100
4.5.3	Proof of Theorem 4.4.2	104
4.5.4	Proof of Corollary 4.4.3	109
4.6	Experiments	110
4.7	Discussion	114
4.7.1	Connection with Chapter 3	116
4.8	Conclusion	117
5	Posterior Sampling for RL on Graphs	119
5.1	Introduction	120
5.2	Time Inhomogeneous MDP and Directed Acyclic Graphs	122
5.2.1	Time-Inhomogeneous Fixed-Horizon Markov Decision Process	122
5.2.2	Directed Acyclic Graphs	124
5.3	Directed Acyclic Markov Decision Process	125
5.3.1	Relationship Between DAMDPs and MDPs	127
5.3.2	Learning in a DAMDP	130
5.3.3	Additional Properties of DAMDP	132
5.4	Dynamic Programming on Graphs	134
5.5	Posterior Sampling RL on Graphs (PSGRL)	138
5.6	Regret of PSRL on the <i>full</i> MDP	141

5.7	Maximum Leaky Flow of a Graph	142
5.8	Related work	147
5.9	Proof of PSGRL Upper Bound on Regret	149
5.9.1	Confidence Sets	151
5.9.2	Upper Bounding the Regret	158
5.10	Conclusion	163
6	Real-World Example: Wind Farm Yield Optimisation	165
6.1	Introduction	166
6.2	Background	171
6.2.1	Wind Farm Simulations with Engineering Wake Models	171
6.2.2	Graph Neural Networks and Transformers	172
6.3	Methodology	173
6.3.1	Graph Representation of Wind Farms	173
6.3.2	Message-Passing Graph Neural Networks	174
6.3.3	Transformers	176
6.4	Model Training	180
6.4.1	Data Generation	180
6.4.2	Validation of the Learned Latent Graph	182
6.5	Results and Discussion	184
6.5.1	Assessment of Surrogate Model Efficacy	184
6.5.2	Wind Farm Yield Optimisation	188
6.6	Conclusions	190
7	Conclusion	191
7.1	Summary	191
7.2	Further work	195

7.2.1	A Lower Bound on HRL Efficiency	195
7.2.2	An Upper Bound on HRL Regret	196
7.2.3	Posterior Sampling for RL on Graphs	197
7.2.4	Real-World Example: Wind Farm Yield Optimisation	198
7.2.5	Learning the structure	199
References		203

List of figures

2.1	Illustration of a grid world MDP and a wind farm optimization MDP	15
3.1	Diagrams representing the interactions between the components of a goal-conditioned hierarchical agent and the induced MDP decomposition.	47
3.2	Empirical performance of SHQL on N-rooms environment with a single room profile.	67
3.3	Evolution of the lower bound ratios of RL against HRL.	68
3.4	Performance of SHQL in the N -room problem with two room profiles . . .	71
3.5	Performance of SHQL in the N -room problem with three room profiles . . .	71
3.6	Performance of SHQL in the N -room problem with four room profiles . . .	71
3.7	Performance of SHQL on a wide range of maze tasks	71
3.8	This figure illustrates how the added difficulty of learning new room profiles impacts the performance of SHQL. In particular, we observe, as one would expect, that as we increase the number of room profiles, the number of episodes necessary to learn an optimal policy increases, as well as the learning curve variance.	72
3.9	An illustration of the Taxi environment and the empirical performance of SHQL on the Taxi environment.	74

4.1	An illustration of the considered hierarchical decomposition in a grid-world environment	89
4.2	Illustration of path grid-world environment	111
4.3	Empirical comparison of $\text{GHRL}_{\text{Q-UCB}}$ against Q-UCB	112
4.4	Evaluation of $\text{GHRL}_{\text{Q-UCB}}$ with multiple room profiles	113
4.5	Evaluation of $\text{GHRL}_{\text{Q-UCB}}$ in the N -rooms environment	113
5.1	Visualisation of a simple Directed Acyclic Markov Decision Process (DAMDP)	125
5.2	Illustration of the transformation of a Directed Acyclic Graph (DAG) into a layered DAG	133
5.3	Perfromance of PSGRL on the maximum leaky flow problem (1/2)	143
5.4	Perfromance of PSGRL on the maximum leaky flow problem (2/2)	146
5.5	An illustration to highlight the differences between FMDPs and DAMDPs .	147
6.1	Message-passing Graph Neural Network architecture	176
6.2	Transformer architecture	177
6.3	Wind Farm Layout Generation	181
6.4	Comparison Transformer VS GNN	183
6.5	Visualisation of the learned interaction patterns	186
6.6	Transformer performance to predict wind farms yield	187
6.7	PSGRL performance to compute optimal yaw angles	188

List of Algorithms

1	Planning with Backward Induction	20
2	Upper Confidence Bounds Value Iteration (UCBVI)	31
3	Posterior Sampling for Reinforcement Learning (PSRL)	34
4	Q-learning	35
5	Q-learning with UCB-Hoeffding (Q-UCB)	37
6	Upper Confidence Fixed-Horizon episodic RL (UCFH)	41
7	Fixed-horizon Extended Value Iteration (FixedHorizonEVI)	42
8	Stationary Hierarchical Q-learning (SHQL)	65
9	Goal-conditioned Hierarchical Reinforcement Learning (GHRL)	97
10	Planning on a DAMDP	136
11	Posterior sampling on graph MDPs (PSGRL)	140

Notation

General Mathematical Symbols

$\otimes$	denotes the Cartesian product.
$\Delta(\cdot)$	takes as input a set and denotes a distribution over that set.
$\emptyset$	denotes the empty set.
$\mathbb{I}\{\cdot\}$	denotes the indicator function, that returns 1 if its input evaluates to True and 0 otherwise.
$\mathcal{B}$	calligraphic symbols are used to denote sets.
$\mathcal{O}(\cdot)$	is used to define upper bounds on the limiting behaviour of quantities of interest.
$\Omega(\cdot)$	is used to define lower bounds on the limiting behaviour of quantities of interest.
$\tilde{\Omega}(\cdot)$	is used to define lower bounds on the limiting behaviour of quantities of interest, ignoring log terms.
$\tilde{\mathcal{O}}(\cdot)$	is used to define upper bounds on the limiting behaviour of quantities of interest, ignoring log terms.

Markov Decision Processes

Δ_k	denotes the suboptimality gap $V^* - V^{\pi_k}$ at episode k .
------------	--

$\mathcal{A}$	denotes the action space.
$\mathcal{M}$	denotes a set of MDPs
$\mathcal{P}$	denotes a set of transition functions.
$\mathcal{S}$	denotes the state space.
Π	denotes the set of deterministic Markov policies.
π	denotes a policy.
ρ	denotes the initial state distribution.
H	denotes the horizon.
M	is used to denote a Markov Decision Processes (MDP).
P	denotes the transition function.
$Q_{M,1}^\pi(s, a)$	denotes the q-function of policy π on MDP M given that at time step 1 the agent is in state s and executes action a , note that when clear from context we omit superscripts and subscripts.
R	denotes the mean reward function.
$V_{M,1}^\pi$	denotes the value function of policy π on MDP M from time step 1, note that when clear from context we omit superscripts and subscripts.

Hierarchical MDPs

$\mathcal{A}_h$	denotes the high-level action space.
$\mathcal{A}_l$	denotes the low-level action space.

$\mathcal{A}_o$	whenever the action space is modified, we use the subscript o to denote the original action space.
$\mathcal{S}_h$	denotes the high-level state space.
$\mathcal{S}_l$	denotes the low-level state space.
$\mathcal{S}_o$	whenever the state space is modified, we use the subscript o to denote the original state space.
μ	denotes a high-level policy.
ν	denotes a low-level policy.
ρ_h	denotes the high-level initial state distribution.
ρ_l	denotes the low-level initial state distribution.
ρ_o	whenever the initial state distribution is modified, we use the subscript o to denote the original initial state distribution.
H_h	denotes the high-level horizon.
H_l	denotes the low-level horizon.
H_o	whenever the horizon is modified, we use the subscript o to denote the original horizon.
M_h	denotes the high-level MDP.
M_l	denotes the low-level MDP.
M_o	whenever an MDP is modified, we use the subscript o to denote the original MDP.
P_h	denotes the high-level transition function.

P_l	denotes the low-level transition function.
P_o	whenever the transition function is modified, we use the subscript o to denote the original transition function.
$Q_{(M_l, M_h), 1}^{(\nu, \mu)}(s, a)$	denotes the q-function of the pair of hierarchical policies (ν, μ) on the decomposed MDP (M_l, M_h) given that at time step 1 the agent is in state s and executes action s , note that when clear from context we omit superscripts and subscripts.
$Q_{M_h, 1}^{\mu}(s, a)$	denotes the q-function of high-level policy μ on the high-level MDP M_h given that at time step 1 the agent is in state s and executes action s , note that when clear from context we omit superscripts and subscripts.
$Q_{M_l, 1}^{\nu}(s, a)$	denotes the q-function of low-level policy ν on the low-level MDP M_l given that at time step 1 the agent is in state s and executes action s , note that when clear from context we omit superscripts and subscripts.
R_h	denotes the high-level mean reward function.
R_l	denotes the low-level mean reward function.
R_o	whenever the reward function is modified, we use the subscript o to denote the original mean reward function.
$V_{(M_l, M_h), 1}^{(\nu, \mu)}$	denotes the value function of the pair of hierarchical policies (ν, μ) on the decomposed MDP (M_l, M_h) from time step 1, note that when clear from context we omit superscripts and subscripts.
$V_{M_h, 1}^{\mu}$	denotes the value function of high-level policy μ on high-level MDP M_l from time step 1, note that when clear from context we omit superscripts and subscripts.

$V_{M_l,1}^v$ denotes the value function of low-level policy v on low-level MDP M_l from time step 1, note that when clear from context we omit superscripts and subscripts.

Directed Acyclic MDPs

$\mathcal{X}$ denotes the atomic state space, in the context of DAMDP.

$\mathcal{Y}$ denotes the atomic action space, in the context of DAMDP.

ρ_A^v denotes the initial atomic state distribution.

c_t^v denotes the transition context for node v in layer t .

G denotes a graph.

M_G denotes a DAMDP, an MDP with a latent graphical structure encoded in the graph G .

p^i denotes the atomic transition function for a node in equivalence class $i \in [U_\tau]$.

r^j denotes the atomic reward function of node with action profile $j \in [U_r]$.

U_τ denotes the number of different atomic state spaces.

U_r denotes the number of different atomic action spaces.

Chapter 1

Introduction

1.1 Motivation

Reinforcement Learning (RL) has gained significant attention in recent years due to its successful application in a wide range of complex problems, including autonomous systems [141, 66, 34], robotics [75, 9, 36, 53, 7], finance [96, 42, 150], healthcare [151, 69, 126, 90, 111] and games [134, 125, 124, 91, 88, 20, 140]. RL has its roots in behavioural psychology [32], which studies how organisms learn from experience to achieve goals. RL algorithms directly learn through interactions with an environment, enabling agents to make decisions based on feedback from their actions. Despite RL's impressive successes and the breadth of application domains to which this methodology is already applied, several limitations and challenges prevent an even wider adoption of this technology. Arguably, one of the most limiting factors of RL's wider adoption is its data inefficiency [22]. This usually restricts the application of RL to tasks that can be simulated on a computer, allowing the algorithm to interact many times with its environment to learn the optimal behaviour.

In many applications, interaction with the environment can be difficult or too costly (e.g. in robotics [53]); in such scenarios, it would be preferable if the algorithm could learn an optimal behaviour from a limited amount of interactions. Even in cases where datasets

consisting of many interactions are available, we need guarantees that our algorithm will learn the best behaviour possible, given the available data. We want our algorithms to extract all the information they can access efficiently.

Interestingly, one of RL's most resounding successes exposes its sample inefficiency. The breakthrough success of AlphaGo [124], an algorithm that used RL to learn how to play the game of Go and successfully defeated the Go world champion, demonstrated the immense potential of RL when combined with deep learning techniques. AlphaGo's training was composed of two phases. The first was an initial supervised training phase over a dataset containing several hundred thousand games played by humans. The second phase involved self-play, in which two versions of the algorithm played against each other. During that specific training phase, the algorithm played several millions of games. When AlphaGo defeated the Go world champion, this achievement underscored the power of RL to solve intricate decision-making tasks with vast state spaces, reinforcing RL's relevance in research and practical applications. AlphaGo played five matches against the world champion. The algorithm won four and lost one. This single victory of the (human) world champion, who observed and played significantly fewer games than the algorithm, proves that nature can develop algorithms that harness the information in its dataset more efficiently than the current state-of-the-art RL algorithms. Importantly, it shows that more efficient algorithms exist and are waiting to be discovered. The work presented in this thesis aims to improve our understanding of the efficiency gains of RL algorithms when we observe more structure in the MDP the algorithm is solving.

1.2 Approach

An interesting approach to improve the efficiency of RL algorithms is to deliberately make more assumptions about the underlying structure describing the problem at hand. Typically, an RL problem is described by a Markov Decision Process (MDP) (formally defined in

Chapter 2). Beyond the Markov assumption¹, MDPs make little assumptions about the problem they describe. If the problem has more structure, it is possible to explicitly model this structure in the MDP. This structure can then be leveraged by the algorithm to boost its sample efficiency. A wide variety of assumptions about the underlying structure can be made. Below, I highlight some of the most common structural assumptions in Reinforcement Learning and illustrate them with real-world examples.

Hierarchical structure: Hierarchical Reinforcement Learning (HRL) assumes that the task has a hierarchical structure and can be decomposed into a sequence of sub-tasks. This approach has two immediate benefits that can potentially improve the sample efficiency compared to traditional RL algorithms. First, it induces a decomposition of the temporal credit assignment problem, and second, it allows for the re-usability of learned policies. The temporal credit assignment problem is an important challenge in RL. It consists of successfully identifying actions that were instrumental in observing a reward. For example, what critical chess moves allowed the player to win the game? Decomposing a potentially long initial task into a sequence of shorter tasks allows us to break down the credit assignment problem [71]. The initial question of finding the most decisive action in the task's success can be broken down into the following questions. First, which sub-goal was most instrumental in the outcome, and second, which action was most instrumental in the success of that specific sub-goal? The agent's ability to solve a sub-task can be considered as a skill. The agent can re-use learned skills and compose them to solve its tasks. This ability to re-use skills can be seen as an abstraction over action; this mechanism was proven to be helpful in autonomous car driving, where the set of skills consisted of the manoeuvres the car could perform [43] the algorithm only had to determine the sequence of manoeuvres it needed to perform to complete its navigation. This capacity to learn and re-use skills is also useful when the agent faces a completely new task. Often, the set of skills required to successfully learn to solve

¹The Markov assumption stipulates that given the current state, the learner's next state is independent of the previous states it visited.

task A could be helpful to solve another task B. Skill transfer is the ability to select and use skills learned on a different task, this was successfully applied in robotics [70]. This capacity to re-use skills learned on different tasks can as well tremendously boost the efficiency of RL algorithms.

Factored structure: In RL's applications, the state and the action space can become prohibitively large, making exploring all possible trajectories in search of the optimal behaviour tedious. Factored structures allow encoding various types of conditional independence, breaking down these spaces' combinatorial nature.

A state is usually described as a potentially large vector, encoding all the necessary features describing the agent's current state in the environment. For example, consider the state of a robot, which encodes information like the current room or if the robot's hands are free. A factored structure can explicitly encode that the action *pick-up* does not depend on the current room and only on the current state of the robot's hand. When predicting the effect of an action, the algorithm can safely ignore the variables independent of that specific action. This greatly simplifies the learning problem and can tremendously boost the learning efficiency of the algorithm.

Similarly, it is possible to inform the agent of conditional independence between components describing the action for complex action spaces. For example, in a healthcare setting, an action might involve combining several drugs and treatments. As there are exponentially many such combinations, it can be beneficial to identify which component of the action (i.e., which drug treatment) affects which variable describing the patient's state [133]. Considering each drug or treatment individually first reduces the number of actions, and considering only the subset of variables that a drug or treatment can influence also greatly simplifies the prediction problem. Combined, these benefits can further boost the sample efficiency [100].

Multi-agent systems [54] are another natural application of the factored action spaces. In such systems, agents tend to have action with local effects, potentially not impacting a large

portion of the remaining agents. If the relationship between the different agents is known, it can successfully be leveraged to construct more efficient algorithms [54].

Graphical structure: Graphs are a natural representation that encodes the relations between connected entities. RL algorithms which leverage this latent graphical structure have emerged as a promising alternative to traditional methods for discovering better decision-making strategies in various disciplines, including chemistry and computer science. A graph RL approach was used in chemistry to find molecules with desired properties, a critical task in drug design [148]. In this particular example, the state is represented as a graph and depicts the current state of the molecule. Actions are possible modifications of that graph to eventually reach the goal of designing a molecule with some set properties. Constraining the algorithm to work explicitly on a graphical state representation constrains the original problem and considerably reduces the state space size, eventually boosting the algorithm’s efficiency.

Many additional real-world problems can be modelled with a graphical latent structure, and several attempts to use RL to solve these problems exist. Notably, RL finds applications in network routing tasks, where the graph describes the actual communication network and the agent is tasked to find a good routing strategy, depending on the predicted stress on the network [136]. In economics, graphs can typically be used to model shared interests by different parties. An RL algorithm successfully found strategies to solve an n-party social dilemma used to study the tension between decisions that benefit the individual and the common good [31].

In summary, these examples highlight the wide diversity of problems that could be modelled with additional structures such as factored, graphical or hierarchical structures. They also underscore the immediate efficiency benefit of developing RL algorithms that can successfully leverage these structures. In many cases, however, these examples are only empirical, and while providing strong evidence of the potential of such methods, they do

not provide a complete understanding of the underlying reasons behind these efficiency gains. Theoretical results (e.g. [29, 77, 11, 62, 131]) usually provide a deeper understanding of the problem, usually uncovering interesting principles useful for improving algorithm design. For example, the Q-learning algorithm analysed in [62] improved our understanding of exploration strategies in model-free RL and the policy gradient theorem [131] gave birth to a wide range of new RL methods such as PPO [121], DDPG [80], A3C [89], SAC [57], TRPO [120].

In this work, we design and study algorithms that can successfully leverage these latent structures and we provide theoretical guarantees that leveraging these structures leads to efficiency gains. In the next section, we summarize our contribution and the overall structure of this thesis.

1.3 Structure of the Thesis

The work presented in this thesis focuses on developing and analysing Reinforcement Learning (RL) algorithms that can successfully leverage additional structure in the Markov Decision Process (MDP) instance they are solving. In particular, this work focuses on two types of structures: Chapters 3 and 4 focus on MDPs that exhibit a hierarchical structure and on understanding how to algorithmically leverage this structure. Chapters 5 and 6 focus on leveraging graphical latent structure. In the remainder of this chapter, we briefly summarise our contributions to Hierarchical Reinforcement Learning (Sec. 1.3.1) and Reinforcement Learning on graphs (Sec. 1.3.2).

1.3.1 RL on Hierarchical MDPs

Hierarchical Reinforcement Learning (HRL) algorithms can perform planning at multiple levels of abstraction. Empirical results have shown that the state [33, 139] or temporal [132,

15] abstractions provided might significantly improve the sample efficiency of algorithms. Yet, we still do not have a complete understanding of the basis of those efficiency gains nor any theoretically grounded design rules. I dedicate two chapters to Hierarchical RL, each focusing on solving a slightly different problem. In Chapter 3, we focus on understanding hierarchical decompositions and how to identify efficient decompositions. We derive a lower bound on the sample complexity of any HRL algorithm, which provides a theoretical tool to better understand the learning dynamics in hierarchical problems. In that specific context, the sample complexity seems a natural metric for performance. In contrast, in Chapter Chapter 4, we will focus on designing an efficient algorithm for HRL. Here, we are interested in closely monitoring the behaviour of this specific algorithm; hence, providing a guarantee on the regret suffered by our algorithm seems more natural.

In Chapter 3, we derive a lower bound on the sample complexity for the considered class of goal-conditioned HRL algorithms. The proposed lower bound empowers us to quantify the benefits of hierarchical decomposition and leads to designing a simple Q-learning-type algorithm that leverages hierarchical decompositions. We empirically validate the lower bound's behaviour by investigating the sample complexity of the proposed hierarchical algorithm on a spectrum of tasks (hierarchical n -rooms, Gymnasium's Taxi). The hierarchical n -rooms tasks were designed to allow us to dial their complexity over multiple orders of magnitude.

In chapter 4, we make several assumptions about the nature of the hierarchical decomposition. This allows us to propose a new class of algorithms in the goal-based HRL framework with a sparse reward function. Specifically, we consider two layers of hierarchy: a high-level policy instructs a sequence of sub-goals to complete the task, and a low-level policy aims to solve each sub-task individually. The proposed framework's flexibility allows the user to choose, according to their preferences, which algorithm to use to compute the low-level policy. Our analysis shows how this specific algorithmic choice impacts the global problem

of jointly learning the two levels of the hierarchy. In particular, if the algorithm computing the low-level policy is efficient (see UCBVI [13] for example), our results are a significant improvement compared to previous results, such as [146]. The results we provide improve existing results regarding efficiency and the nature of the guarantee. Technically, we provide an upper bound on the frequentist regret, while previous results provided guarantees on a weaker performance metric, the Bayesian regret. These various metric and their relations are formally presented in Chapter 2. The key insight behind our method is that in the goal-conditioned HRL setting with sparse reward, we can improve the algorithm’s performance by leveraging the fact that we can compute the probability with which the low-level agent’s policy can reach a given sub-goal. We empirically validate our framework using Q-UCB [62], as a low-level algorithm, on a set of grid-world navigation tasks, where the proposed framework consistently outperforms the monolithic version of Q-UCB.

Our theoretical and algorithmic findings in Chapters 3 and 4 provide a step towards answering the foundational question of quantifying the improvement hierarchical decomposition offers over monolithic solutions in Reinforcement Learning.

1.3.2 RL on Graphs

In Chapters 5 and 6, we consider the case where the structure of the Markov Decision Process can be modelled using a directed acyclic graph (DAG) composed of nodes and edges, allowing for a more basic or "atomic" representation of the state and action spaces. In this case, each node has a state, and the state transition dynamics are influenced by the states and actions at its parent nodes. In Chapter 5, we propose an MDP framework, *Directed Acyclic Markov Decision Process* (DAMDP), that formalises this model. Crucially, DAMDP retains many of the benefits of MDPs, as we show that Dynamic Programming can be used for planning to find the optimal policy in a known DAMDP. We further show that it is possible to perform Reinforcement Learning in DAMDPs where the transition probabilities and the

reward function are unknown. For this, we derive a posterior sampling-based algorithm that is able to leverage the graph structure to boost learning efficiency. Moreover, we obtain a theoretical bound on the Bayesian regret for this algorithm which directly shows the efficiency gain from considering the graph structure. We conclude by showing empirically how exploiting the DAMDP structure with our algorithm outperforms traditional posterior sampling for Reinforcement Learning in a maximum flow problem and a real-world wind farm optimisation task.

Chapter 6 focuses on providing evidence that the optimization of wind farms exhibits the graphical structure mentioned in Chapter 5. This work proposes a novel data-driven model capable of providing accurate predictions for the power generation of all wind turbines in wind farms of arbitrary layout, yaw angle configurations and wind conditions. The proposed model functions by encoding a wind farm into a fully connected graph and processing the graph representation through a transformer model [137]. The resultant transformer surrogate demonstrates robust generalisation capabilities and effectively uncovers latent structural patterns embedded within the graph representation of wind farms. The versatility of the proposed approach extends to the optimisation of yaw angle configurations through the application of genetic algorithms. This evolutionary optimisation strategy facilitated by the graph transformer surrogate achieves prediction accuracy levels comparable to industrially standard wind farm simulation tools. We report a relative accuracy of more than 99% in identifying optimal yaw angle configurations of previously unseen wind farm layouts. An additional advantage lies in the significant reduction in computational costs, positioning the proposed methodology as a compelling tool for efficient and accurate wind farm optimisation.

1.4 Publications and Contributions

Publications: The work done in this thesis consists of the following scientific publications.

The content of Chapter 3 has been published as Robert, A., Pike-Burke, C., & Faisal, A. A. (2024). Sample complexity of goal-conditioned hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

A preliminary version of this work was presented in the following publications:

Robert, A., Pike-Burke, C., & Faisal, A. A. Sample Complexity of Hierarchical Decompositions in Markov Decision Processes. *In ICML Workshop on New Frontiers in Learning, Control, and Dynamical Systems*.

and

Robert, A., Pike-Burke, C., & Faisal, A. A. (2023). Sample Complexity of Goal-Conditioned Hierarchical Reinforcement Learning. *16th European Workshop on Reinforcement Learning*.

The content of Chapter 4 has been submitted to *AISTATS* as Robert, A*, Drappo, G*, Metelli, A., Faisal, A. A., Restelli, M., & Pike-Burke, C. (2024) Efficient Exploitation of the Hierarchical Structure in Sparse Reward Reinforcement Learning.

The content of Chapter 5 has been submitted to *Transactions on Machine Learning Research* as Robert, A., Faisal, A. A., & Pike-Burke, C. (2024). Posterior Sampling for Reinforcement Learning on Graphs.

The content of Chapter 6 has been published as Li, S., Robert, A., Faisal, A. A., & Piggott, M. D. (2024). Learning to optimise wind farms with graph transformers. *In Applied Energy*.

Contributions: The work presented in Chapters 3 and 5 is my own. The work presented in Chapter 4 was made in close collaboration with Gianluca Drappo; we both contributed to the theoretical findings and the experimental setup. However, the implementation of the

environments and the algorithms are my own. The work presented in Chapter 6 results from a collaboration with Siyi Li. While I designed the initial experimental setting, Siyi significantly improved it; specifically, he implemented the transformer model we used. I implemented the graph neural network, which we use as a baseline. Most of the conceptual ideas, including the experimental setting and the objective of learning the latent graphical structure of the problem, are my own.

Chapter 2

Background

This chapter presents the fundamental concepts necessary to understand the work done in the remaining chapters. In particular, Chapters 3 to 5 consider the problem of finite-horizon Markov Decision Processes. We start this background chapter with a formal definition of finite-horizon MDP. We then define some of the most common performance metrics encountered in this setting: return, expected return, sample complexity, and regret, which will allow us to evaluate our algorithms both theoretically and empirically. Finally, we present four algorithms; the work done in this thesis will then extend each of these algorithms, knowing additional structural information about the problem at hand.

2.1 Finite-Horizon Markov Decision Processes

A Markov Decision Process is a mathematical framework that describes how an agent interacts with its environment. Reinforcement Learning considers the problem of learning how to behave optimally when the environment is unknown and uncertain. In this work, we only consider the finite horizon (or episodic) setting that we define formally now.

Definition 2.1.1. A finite-horizon MDP is defined by the following tuple

$M = \langle \mathcal{S}, \mathcal{A}, H, \{P_h\}_{h=1}^H, \{r_h\}_{h=1}^H, \rho \rangle$ where:

- $\mathcal{S}$ is the state space, the finite set of possible state values.
- $\mathcal{A}$ is the action space, the finite set of possible actions the agent can take.
- $H \in \mathbb{N}$ and $H < \infty$ is the horizon of the problem, i.e., an episode's length.
- $\{P_h\}_{h=1}^H$ is the set of stochastic transition functions. The stochastic transition function maps each state, action and time step tuple to a distribution over states, $P_h : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$, where $\Delta(\mathcal{S})$ is the set of all distribution over the state space $\mathcal{S}$. Specifically, when action $a \in \mathcal{A}$ is executed while in state $s \in \mathcal{S}$ the probability of observing $s' \in \mathcal{S}$ at the next time step (i.e. $h + 1$) is given by $P_h(s'|s, a)$. Note that if the transition functions are independent of the current time step h , the problem is time-homogeneous or stationary. Otherwise, the problem is time-inhomogeneous or non-stationary.
- $\{r_h\}_{h=1}^H$ is the set of reward functions $r_h : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. It associates to each state-action pair an immediate reward bounded between 0 and 1.
- $\rho \in \Delta(\mathcal{S})$ is the initial state distribution; it selects the first state of each episode.

To complete our understanding of finite-horizon MDPs, we now consider the execution of a single episode. The first state of the episode $s_1 \in \mathcal{S}$ is drawn from the initial state distribution $s_1 \sim \rho$. Once the first state is observed, the agent selects an action $a_1 \in \mathcal{A}$ and executes it. The agent then observes the next state $s_2 \in \mathcal{S}$ sampled from the stochastic transition function $s_2 \sim P_1(s_1, a_1)$ and the immediate reward $r_1(s_1, a_1)$. The environment is now in state s_2 , and the agent chooses an action to perform, and it repeats the following steps until it interacts H times with the environment; at this point, the episode ends. We denote by K the total number of episodes we execute, so $T = KH$ represents the total number of interactions with the environment.

In RL, we are concerned with learning the sequence of actions that allows the agent to accumulate the largest expected sum of reward possible. The agent strategy is called a policy, which is defined now.

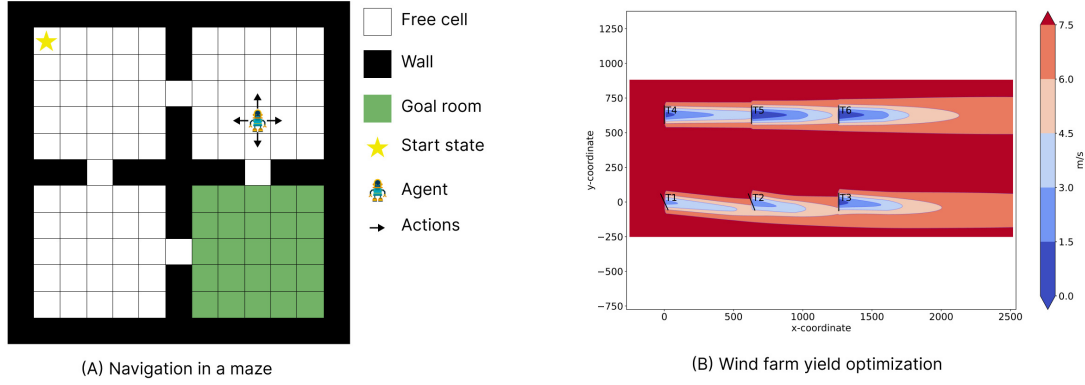


Fig. 2.1 An illustration of two MDPs, with real-world application. On the leftmost figure, we depict a typical navigation task described in example 2.1.3. The starting state is depicted with a yellow star, and the goal is to reach the green room. The rightmost figure illustrates the wind farm optimisation problem. The action is to choose the orientation of each wind turbine. We can see that in the top row of wind turbines, all turbines directly face the wind, and the turbulent wind impacts the yield of the downstream wind turbine. On the contrary, the second row shows how changing the yaw angle can deflect the turbulent winds. This problem is described in example 2.1.4.

Definition 2.1.2. A policy $\pi = \{\pi_h\}_{h=1}^H$ is a set of mappings between a state and a distribution over action $\pi_h(s) : \mathcal{S} \rightarrow \Delta(\mathcal{A})$. Essentially, an agent that follows the policy π selects action $a \in \mathcal{A}$ while in state $s \in \mathcal{S}$ during time step $h \in \{1, \dots, H\}$ with probability $\pi_h(a|s)$. This work is mostly concerned with deterministic policies; in such a setting, policies directly map each state to an action, $\pi_h : \mathcal{S} \rightarrow \mathcal{A}$.

2.1.1 Examples

We now provide examples of two MDPs that will be used throughout this thesis.

Example 2.1.3. An example that we use to illustrate the behaviour of our algorithms in chapter 3 and 4 is a simple maze navigation task that is depicted in the leftmost plot of Figure 2.1. In this setting, the environment consists of a grid world, where each cell represents a state, and the agent has to navigate from one cell to the next until it reaches a specific goal room. The agent's goal is to find the shortest path from the starting state to the goal room

despite two challenges. First, the maze contains walls that cannot be crossed by the agent, so it needs to find a path around them, and second, the transition dynamics are stochastic and unknown. The action space consists of the four cardinal directions $\{N, S, E, W\}$ and the selected action is executed with probability $1 - p$, and one of the remaining actions is executed with probability p . The reward function usually assigns a reward of 1 as long as the agent is in the goal state and a reward of 0 in every other state.

Example 2.1.4. Another example that we will discuss in greater detail in chapter 5 and 6 is the optimisation of the yield of a wind farm. A wind farm contains N wind turbines positioned on a field (i.e. a $2D$ -lattice). We call each column of this lattice a layer, and the wind flows through each layer successively, starting from the first layer (i.e. the most upstream one) to the last layer (i.e. the most downstream one). The farm's yield corresponds to the quantity of energy produced by all the wind turbines. A single wind turbine maximizes its yield when it directly faces the wind, but the rotating turbine alters the airflow downstream of the wind turbine and makes it turbulent; this stream of turbulent wind is called the wake effect. The interactions between the wind turbines composing the farm create an interesting control problem. If the angle between the wind turbine and the wind direction changes, the wake effect can be deflected. Note that there is a direct relation between the wind speed and the power a turbine produces; the higher the wind speed, the more energy the wind turbine will produce. This situation is depicted on the rightmost plot of Figure 2.1. This plot shows the farm from above, and the wind speed is color-coded. Areas with fast winds (7.5 m/s) are red; the lower the wind speed in an area, the closer its colour will be to blue. If we consider the first row of wind turbines, which contains the turbine labelled $T4$, and . All turbines in that row are facing the wind, the three turbines combined yield 2200kW. In contrast, the second row of wind turbines, containing the turbines labelled $T1$, $T2$, and $T3$, has been optimised to maximize the yield, and they jointly produce 3300kW. This significant difference in yield is a direct consequence of the wake effect. One can notice that $T1$ and $T2$

are not facing the wind, which deflects the wake effect and preserves the quality of the wind flow for downstream wind turbines. Globally, while $T1$ yields less power than its counterpart $T4$, the bottom row is significantly more productive than the top row. Finding the optimal wind turbine angles is an interesting control problem that can be defined as an MDP. The state space would consist of the wind condition observed at all turbines in a given layer, the action space would consist of the angle for each turbine in this layer, and the reward function would correspond to the wind turbine's yield. The next state would then consist of the wind conditions at each wind turbine in the next layer.

2.1.2 Planning in an MDP

Whether the problem is the grid world example 2.1.3 or the wind farm example 2.1.4, we want to identify policies with a high total expected reward. To that end, we define the value function of a policy, which can be seen as a metric to evaluate the quality of a policy. Unless specified otherwise, the policy we consider belongs to the set of deterministic Markov policies, Π . In finite-horizon MDP, it is sufficient to consider deterministic Markov policies as if the problem accepts an optimal general policy; then it accepts a deterministic Markov policy [86].

Definition 2.1.5. The value function of a policy π in the first time step $h = 1$ consists of the total expected reward the policy accumulates from time step 1 onwards:

$$V_1^\pi = \mathbb{E} \left[\sum_{h=1}^H r_h(s_h, a_h) \mid s_1 \sim \rho, a_h = \pi_h(s_h), s_{h+1} \sim P_h(\cdot \mid s_h, a_h) \right],$$

where $s_{h+1} \sim P_h(\cdot \mid s_h, a_h)$ denotes that the state at time step $h + 1$ is sampled from the transition distribution $P_h(\cdot \mid s_h, a_h)$. The expectation is taken over the initial state distribution ρ and the stochastic transitions $P_h(\cdot \mid s, a)$.

More generally, the value function of any state $s_h \in \mathcal{S}$ at any time step $h \in \{2, \dots, H\}$ is given by:

$$V_h^\pi(s_h) = \mathbb{E} \left[\sum_{t=h}^H r_t(s_t, a_t) | a_t = \pi_t(s_t), s_{t+1} \sim P_t(\cdot | s_t, a_t) \right],$$

where the expectation is only with respect to the stochastic transition functions $\{P_t\}_{t=h}^H$.

The optimal policy is the policy that has the highest associated value function, that is:

$$\pi^* = \max_{\pi \in \Pi} V_1^\pi.$$

Where Π is the set of deterministic Markov policies, it is proven that an optimal policy always exists for finite horizon MDP. Additionally, it can also be shown that there exists an optimal deterministic policy [114].

Defining the total expected reward after taking a specific action $a \in \mathcal{A}$ is useful. This quantity is called the Q-function, and it is defined as follows:

$$Q_h^\pi(s, a) = \mathbb{E} \left[\sum_{t=h}^H r_t(s_t, a_t) | s_h = s, a_h = a, a_t = \pi_t(s_t), s_{t+1} \sim P_t(\cdot | s_t, a_t) \right]$$

where the expectation is taken with respect to the stochastic transition functions, $\{P_t\}_{t=h}^H$.

The relationship between the value function and the Q-function is described by the Bellman equations, which states that for any deterministic Markov policy, $\pi \in \Pi$, the following holds:

$$\begin{aligned} V_h^\pi(s) &= Q_h^\pi(s, \pi_h(s)) & (\forall (s, h) \in \mathcal{S} \times \{1, \dots, H\}) \\ Q_h^\pi(s, a) &= r(s, a) + \sum_{s' \in \mathcal{S}} P_h(s' | s, a) V_{h+1}^\pi(s') & (\forall (s, a, h) \in \mathcal{S} \times \mathcal{A} \times \{1, \dots, H\}) \\ V_{H+1}^\pi(s) &= 0 & (\forall s \in \mathcal{S}) \end{aligned}$$

As the Bellman equations hold for any policy [114], it holds, in particular, for the optimal policy; we refer to it as the Bellman optimality equations. To lighten the notation, we use the superscript $*$ to denote the value function and the Q-function of the optimal policy, that is, $V_h^* = V_h^{\pi^*}$ and $Q_h^* = Q_h^{\pi^*}$.

$$\begin{aligned} V_h^*(s) &= Q_h^*(s, \pi_h^*(s)) & (\forall (s, h) \in \mathcal{S} \times \{1, \dots, H\}) \\ Q_h^*(s, a) &= r(s, a) + \sum_{s' \in \mathcal{S}} P_h(s'|s, a) V_{h+1}^*(s') & (\forall (s, a, h) \in \mathcal{S} \times \mathcal{A} \times \{1, \dots, H\}) \\ V_{H+1}^*(s) &= 0 & (\forall s \in \mathcal{S}) \end{aligned}$$

Note that we immediately obtain an optimal deterministic policy from the optimal Q-function, $\pi_h^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q_h^*(s, a)$. In general, as long as the MPD is known, that is, we know the reward functions $\{r_h\}_{h=1}^H$ and we know the transition functions $\{P_h\}_{h=1}^H$ computing an optimal policy is a Dynamic Programming problem, called *Planning*. Dynamic programming problems exhibit a particular structure that allows us to find a solution to the original problem by breaking it down into a sequence of *easier* sub-problems. When computing an optimal policy, this particular structure is directly observable from the Bellman optimality equation [114] at time step h :

$$Q_h^*(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}} P_h(s'|s, a) V_{h+1}^*(s') \quad \forall (s, a) \in \mathcal{S} \times \mathcal{A}. \quad (2.1)$$

From Eq. 2.1, we immediately see that knowing a policy that behaves optimally from time step $h+1$ onwards can be used to easily compute a policy that behaves optimally from time step h . If $V_{h+1}^*(s)$ is the optimal value function we can get $\pi_h^*(s) = \operatorname{argmax}_a Q_h^*(s, a)$ and we get the value function at time step h , as $V_h^*(s) = Q_h^*(s, \pi_h^*(s))$. Repeating these steps until $h = 1$ allows us to compute the optimal policy and its associated value function. These steps are detailed in Algorithm 1.

Algorithm 1: Planning with Backward Induction

Input: $\{P_h\}_{h=1}^H, \{r_h\}_{h=1}^H, \mathcal{S}, \mathcal{A}$

 1 **for** $s \in \mathcal{S}$ **do**

 2 $V_H(s) = \max_{a \in \mathcal{A}} r_H(s, a)$

 3 **for** $t \in \{H-1, \dots, 1\}$ **do**

 4 **for** $s \in \mathcal{S}$ **do**

 5 $V_t(s) = \max_{a \in \mathcal{A}} r_t(s, a) + \sum_{s' \in \mathcal{S}} P_t(s'|s, a) V_{t+1}(s')$

 6 $\pi(s, t) = \operatorname{argmax}_{a \in \mathcal{A}} r_t(s, a) + \sum_{s' \in \mathcal{S}} P_t(s'|s, a) V_{t+1}(s')$
Output: V, π

Theorem 2.1.6. *If the inputs of Algorithm 1 is the true transition function, P , and mean reward function, r , then the output consists of the optimal value function $V = V_1^*$ and the optimal policy $\pi = \pi^*$.*

A detailed proof of theorem 2.1.6 is available in [114].

2.2 Learning in an Unknown MDP

Now that we have formally defined MDPs, we can consider a more challenging problem that represents RL's primary focus: learning an optimal policy when the environment is unknown. Let's first present the most common performance metrics that we can use to evaluate the algorithm's performance theoretically and empirically.

2.2.1 Performance Metrics

Performance metrics are crucial to evaluating, understanding, and comparing algorithms. The training of reinforcement learning algorithms in a finite-horizon MDP consists of a sequence of K episodes. At the beginning of each episode k , the algorithm will compute a policy π^k that the algorithm uses to roll out an episode of H time steps. During each episode rollout, the algorithm uses the data collected to update its beliefs about the world. Before starting the

next episode $k + 1$, the algorithm computes the policy π^{k+1} , which better reflects its updated view of the world. A first natural metric for a learning agent is the *return* or the sum of all the rewards accumulated by an agent during an episode.

Definition 2.2.1. The return is the sum of immediate rewards the agent accumulates in an episode while executing the policy π :

$$R^\pi = \sum_{h=1}^H r_h(s_h, a_h).$$

Where $s_1 \sim \rho$ is drawn from the initial state distribution, the action, $a_h = \pi_h(s_h)$, is selected by the policy, and the next state, $s_{h+1} \sim P_h(\cdot | s_h, a_h)$, is drawn from the stochastic transition function.

The return is a random quantity, depending on the initial state distribution and the stochastic transition function. When we take the expectation of the return with respect to these two random sources, we obtain the value function, $V_1^\pi = \mathbb{E}_{\rho, P}[R^\pi]$.

The return is mostly used to compare algorithms' empirical performance. When a theoretical comparison is required, we seek more robust ways to make comparisons that do not depend on the various sources of randomness. Hence, the following metrics are preferred.

Firstly, the regret, which represents the difference in expected return between the optimal policy π^* and the learned policy at episode k , π^k . This difference $V_1^* - V_1^{\pi^k} = \Delta_k$ is called the sub-optimality gap. The regret then consists of the sum of the suboptimality gaps for each learning episode.

Definition 2.2.2. The regret consists of the sum of suboptimality gaps suffered during K episodes of training. That is, if the current policy π^k is selected at the beginning of each episode k by a learning algorithm $\mathfrak{A}$ the regret suffered is:

$$\mathfrak{R}(M, \mathfrak{A}, K) = \sum_{k=1}^K \Delta_k,$$

where M denotes the specific MDP instance with which the algorithm interacts.

Secondly, the Bayesian regret assumes that the transition function and the reward function are drawn from the prior distributions f_p and f_r , respectively.

Definition 2.2.3. The Bayesian regret is the expected regret, given the prior distribution over the transition functions, f_p , and over the reward functions, f_r .

$$\mathbb{E}[\mathfrak{R}(M, \mathfrak{A}, K)] = \mathbb{E}\left[\sum_{k=1}^K \Delta_k\right],$$

where the expectation is taken over the distributions f_p and f_r . These two distributions combined induce a distribution over the set of possible MDPs $\mathcal{M}$.

Lastly, instead of looking at the sum of errors made, it is sometimes interesting to consider only the number of episodes required to learn a policy that achieves an ε -optimal performance. A policy π has an ε -optimal performance if the difference between its value function and the optimal value function is bounded by ε , $|V_1^* - V_1^\pi| \leq \varepsilon$. This metric is called a Probably Approximately Correct (PAC) bound on the number of episodes or simply the sample complexity.

Definition 2.2.4. An algorithm satisfies a PAC bound N for parameters $\varepsilon \in (0, 1)$ and $\delta \in (0, 1)$; that is, for any fixed horizon MDP, the algorithm plays with probability $1 - \delta$ a policy that is at least ε -optimal after at most N episodes, where N is a polynomial of the characteristics of the MDP instance.

More compactly, we can say that an algorithm satisfies a PAC guarantee N if with probability $1 - \delta$, $\max\{k \in \mathbb{N} : \Delta_k > \varepsilon\} \leq N$. Note that N is a function of the MDP characteristics.

2.2.2 Upper Bounds and Lower Bounds

Upper bounds: Typically, the analysis of an algorithm provides an upper bound on its regret or sample complexity. The upper bound then states that in order to learn a near-optimal

policy, the algorithm needs to suffer at most a specific amount of regret or need at most a specific number of episodes.

Lower bounds: Interestingly, it is possible to provide lower bounds for the sample complexity and the regret of any algorithm. As these bounds are valid for any algorithm, they are, in particular, valid for the best possible algorithm. The lower bounds are then useful for two main reasons. First, they inform us of the overall complexity of an MDP instance. Second, if we can find an upper bound on the regret or the sample complexity that matches the lower bound, we can conclude that such an algorithm has an optimal theoretical performance.

Lower bounds for finite-horizon MDP

In the case of Finite-Horizon MDP, we have the following lower bound [10] on the regret¹; we rewrite their result below for completeness:

Theorem 2.2.5. *For any algorithm $\mathfrak{A}$, there exists an MDP M with S states, A actions and horizon H , such that the expected regret of $\mathfrak{A}$ after K episodes is:*

$$\mathbb{E}[\mathfrak{R}(M, \mathfrak{A}, K)] = \sqrt{HSAT}, \quad (2.2)$$

where $T = KH$ is the total number of interactions with the environment.

A complete proof of this result is provided in [101].

Similarly, [29] provides a lower bound on the sample complexity in the finite-horizon setting; we rewrite their result below for completeness.

Theorem 2.2.6. *There exist constants c , δ_0 , ε_0 such that for any $\delta \in (0, \delta_0)$ and $\varepsilon \in (0, \varepsilon_0)$ and for any algorithm $\mathfrak{A}$ that satisfy a PAC guarantee (δ, ε) (see def. 2.2.4) and outputs a*

¹We can directly adapt the result found in [10] to the Finite-Horizon MDP setting, realising that the diameter of a finite-horizon MDP is its horizon, H .

deterministic policy, there exist a fixed horizon MDP M_{hard} with:

$$\mathbb{E}[N] = \Omega\left(\frac{SAH^2}{\epsilon^2} \ln \frac{1}{\delta + c}\right) \quad (2.3)$$

where N is the number of episodes until the policy is (ϵ, δ) -accurate.

Note that the expectation $\mathbb{E}[N]$ is taken over all potential sources of randomness, including the initial state distribution, the stochastic transition function, and the algorithm's potential random behaviour. A complete proof of this result is provided in [29].

Connections between the metrics

While each of the above metrics monitors a slightly different aspect of model performance, they are all related and have deep connections. For an in-depth discussion on the relationship between PAC guarantee and regret, we refer the reader to [30]. In the context of this thesis, the important relationships to keep in mind are the following:

Regret and Bayesian regret: The (frequentist) regret is a stronger requirement than the Bayesian regret. The frequentist regret must be valid independently in every MDP instance, including the worst-case scenario. On the contrary, an upper-bound on the Bayesian regret needs to upper-bound the error made on average on a set of plausible (according to the priors) MDPs, which is not as strong as a guarantee as the performance on the worst possible MDP. It is possible to compute an upper bound on the regret knowing the Bayesian regret. The following theorem (presented in [97]) explains this.

Theorem 2.2.7. *Let $\mathcal{M}$ be any set of MDPs such that $P(M^* \in \mathcal{M}) > 0$. Let $\mathfrak{A}$ be any learning algorithm with the following Bayesian regret:*

$$\mathbb{E}[\mathfrak{R}(M^*, \mathfrak{A}, K)] \leq C\sqrt{T}, \quad (2.4)$$

for some $C > 0$ and all $T > 0$, with $T = KH$. Then for any $\varepsilon > 0$ and $\alpha > \frac{1}{2}$,

$$P\left(\frac{\mathfrak{R}(M^*, \mathfrak{A}, K)}{T^\alpha} > \varepsilon | M^* \in \mathcal{M}\right) \rightarrow 0 \text{ as } T \rightarrow \infty \quad (2.5)$$

Proof. Since the regret is always positive, $\mathfrak{R}(M^*, \mathfrak{A}, K) > 0$, we can use Markov's inequality to bound the following equation:

$$\begin{aligned} \frac{\mathbb{E}[\mathfrak{R}(M^*, \mathfrak{A}, K)]}{T^\alpha} &\geq \frac{\mathbb{E}[\mathfrak{R}(M^*, \mathfrak{A}, K) | M^* \in \mathcal{M}]}{T^\alpha} P(M^* \in \mathcal{M}) \\ &\geq \varepsilon P\left(\frac{\mathfrak{R}(M^*, \mathfrak{A}, K)}{T^\alpha} > \varepsilon | M^* \in \mathcal{M}\right) P(M^* \in \mathcal{M}), \end{aligned}$$

where $\varepsilon > 0$ and $T = KH$.

Rearranging the above terms gives us the desired result,

$$P\left(\frac{\mathfrak{R}(M^*, \mathfrak{A}, K)}{T^\alpha} > \varepsilon \mid M^* \in \mathcal{M}\right) \leq \frac{\mathbb{E}[\mathfrak{R}(M^*, \mathfrak{A}, K)]}{\varepsilon P(M^* \in \mathcal{M}) T^\alpha} \rightarrow 0 \text{ as } T \rightarrow \infty. \quad (2.6)$$

□

In a nutshell, Theorem 2.2.7 shows that an upper bound on the Bayesian regret implies an upper bound on the frequentist regret for any MDP M^* supported by the prior distribution (i.e. $P(M^* \in \mathcal{M}) > 0$).

Regret and sample complexity Extensive discussions on this relationship can be found in [97]; we highlight below that it is possible to derive Bayesian regret bounds from PAC bounds and inversely.

Proposition 2.2.8. (*Bayesian regret bounds imply PAC bounds*)

Let $\mathfrak{A}$ be an algorithm, for any $C > 0$ and $\alpha > 0$, if the Bayesian Regret after K episodes is:

$$\mathbb{E}[\mathfrak{R}(M^*, \mathfrak{A}, K)] \leq CK^{1/\alpha}, \quad (2.7)$$

where M^* is the true MDP.

Then algorithm $\mathfrak{A}$ satisfy a PAC guarantee $K > \left(\frac{C}{\varepsilon}\right)^{\frac{\alpha}{\alpha-1}}$ for any $\varepsilon > 0$.

Proof. If we consider the average Bayesian Regret per step, for $K' > \left(\frac{C}{\varepsilon}\right)^{\frac{\alpha}{\alpha-1}}$ steps, we get:

$$\begin{aligned} \mathbb{E}\left[\frac{\mathfrak{R}(M^*, \mathfrak{A}, K')}{K'}\right] &\leq \frac{C\left(\left(\frac{C}{\varepsilon}\right)^{\frac{\alpha}{\alpha-1}}\right)^{1/\alpha}}{\left(\frac{C}{\varepsilon}\right)^{\frac{\alpha}{\alpha-1}}} \\ &= \frac{C\left(\frac{C}{\varepsilon}\right)^{\frac{1}{\alpha-1}}}{\left(\frac{C}{\varepsilon}\right)^{\frac{\alpha}{\alpha-1}}} \\ &= \frac{C^{\frac{\alpha}{\alpha-1}} \varepsilon^{\frac{\alpha}{\alpha-1}}}{C^{\frac{\alpha}{\alpha-1}} \varepsilon^{\frac{1}{\alpha-1}}} \\ &= \varepsilon \end{aligned}$$

□

Proposition 2.2.9. (PAC bounds imply regret bounds)

Let $\alpha > 0$ and $C > 0$ be such that algorithm $\mathfrak{A}$ satisfy the following PAC condition with $\varepsilon > 0$:

$$\mathbb{E}\left[\sum_{k=1}^{\infty} \mathbb{I}\{\Delta_k > \varepsilon\}\right] < C\varepsilon^{-\alpha} \quad (2.8)$$

Then the regret suffered by algorithm $\mathfrak{A}$ can be bounded by:

$$\mathfrak{R}(M, \mathfrak{A}, K) \leq (\alpha + 1) \alpha^{\frac{-\alpha}{\alpha+1}} C^{\frac{1}{\alpha+1}} H^{\frac{1-\alpha}{1+\alpha}} T^{\frac{\alpha}{\alpha+1}} \quad (2.9)$$

Proof. For any number of episode $K > 0$ and interactions $T = KH$ the regret suffered by algorithm $\mathfrak{A}$ can be upper bounded by:

$$F(\varepsilon) := C\varepsilon^{-\alpha}H + \varepsilon \frac{T}{H}. \quad (2.10)$$

The first term stipulates that during the first $C\varepsilon^{-\alpha}$, the regret is upper bounded by H , which is the largest regret the algorithm can suffer during a single episode. We then add ε of regret for all episodes, which will upper bound the number of episodes during which the algorithm plays an epsilon optimal policy. To obtain the result in (2.9), one needs to compute the maximum value the function $F(\varepsilon)$ can take. Leading to $\varepsilon^* = \left(\frac{T}{CH^2\alpha}\right)^{\frac{-1}{\alpha+1}}$ and evaluating $F(\varepsilon^*)$ leads to the bound in (2.9). To find the ε maximizing F , we first compute the derivative of F .

$$\frac{d}{d\varepsilon}F(\varepsilon) = -\alpha C\varepsilon^{-(\alpha+1)}H + \frac{T}{H} \quad (2.11)$$

We now aim to find ε^* such that $\frac{d}{d\varepsilon}F(\varepsilon^*) = 0$, that is,

$$\begin{aligned} -\alpha C(\varepsilon^*)^{-(\alpha+1)}H + \frac{T}{H} &= 0 \\ \Leftrightarrow \frac{T}{H} &= \alpha C(\varepsilon^*)^{-(\alpha+1)}H \\ \Leftrightarrow \left(\frac{T}{H^2C\alpha}\right)^{-\frac{1}{\alpha+1}} &= \varepsilon^* \end{aligned}$$

Then evaluating $F(\varepsilon^*)$ yield the expected result:

$$F(\varepsilon^*) = HC\left(\frac{T}{H^2C\alpha}\right)^{\frac{\alpha}{\alpha+1}} + \frac{T}{H}\left(\frac{T}{H^2C\alpha}\right)^{-\frac{1}{\alpha+1}}$$

Rearranging terms yield the stated upper bound. $\square$

Note that the above equivalences hold in the finite-horizon MDP setting and are generally not true in other settings.

2.3 Algorithms

In this thesis, we consider both model-based and model-free algorithms to compute an optimal policy in an unknown finite-horizon MDP. The term model-based means that the algorithm

explicitly maintains a model of the environment that it refines as more data is observed. In contrast, the term model-free algorithms emphasizes that this family of algorithms do not have an explicit representation of the environment and aims to directly learn the Q function, for example.

The algorithms we will present next are important building blocks for the methods developed in this thesis. They are also quite diverse in nature, covering a large spectrum of the methods used in tabular RL, including Thompson sampling-based methods [128, 99, 103, 102, 14, 98, 51], optimistic methods [13, 12, 10, 46, 24, 59] and model-free methods [143, 134, 144, 129, 88].

Specifically, we present Upper Confidence Bound Value Iteration (UCBVI) [13], a provably efficient algorithm used in chapter 4 to build an efficient HRL framework. Then, we present Posterior Sampling Reinforcement Learning (PSRL) [128, 99], a Thompson sampling-based algorithm that inspires the development of the algorithm presented in chapter 5 that allows performing RL on a graph. Then we present Q-learning [143], and Q-UCB [62] two model-free algorithms that also inspire the algorithms presented in chapter 3 and chapter 4, respectively. Lastly, we present Upper Confidence Fixed-Horizon episodic reinforcement learning (UCFH) [29], an algorithm that was first proposed to minimize the sample efficiency Def. 2.2.4, which contrasts with UCBVI, which minimizes the regret Def. 2.2.2.

2.3.1 Upper Confidence Bound Value Iteration (UCBVI)

The first algorithm we present, UCBVI [13], is a model-based algorithm, which was the first provably efficient algorithm; that is, its upper bound $\tilde{O}(\sqrt{HSAT})$ matches the lower bound for finite-horizon RL $\Omega(\sqrt{HSAT})$ up to some log factors. Note that there exist many attempts in the literature to build efficient RL algorithms such as REGAL [17] UCRL [12], UCRL2 [10], R-Max [24] and E^3 [65].

In RL, we often assume the reward function, r , to be known but the transition function, P , to be unknown. This assumption mostly simplifies the algorithms' analysis and generally does not impact the reported performance because the reward function is significantly simpler to learn than the transition function. Indeed, for each state-action pair, the transition probability consists of a distribution over the entire state space $\Delta(\mathcal{S})$, while the reward function consists of a single bounded value. Still, UCBVI must jointly learn the optimal policy and the unknown transition functions, P . To do so, UCBVI relies on a principle called *optimism in the face of uncertainty* [73, 24, 65, 10]. This principle proposes that the algorithm behaves optimally in the best statistically plausible MDP.

The algorithm maintains a dataset of transition tuples (s, a, s') observed up to episode k , which are stored in the dataset $\mathcal{H}$. In practice, the algorithm computes an optimal policy for a modified value function, called an optimistic value function, that adds a bonus term to the value function of a state inversely proportional to how well that state is understood (i.e. the number of visits). So, more precisely, the algorithm maintains three counters. (i) $N_k(s, a, y) = \sum_{(s', a', y') \in \mathcal{H}} \mathbb{I}\{s' = s, a' = a, y' = y\}$, which counts until the end of episode k the number of times the agent selected an action $a \in \mathcal{A}$ while in state $s \in \mathcal{S}$ which lead to state $y \in \mathcal{S}$. (ii) $N_k(s, a) = \sum_{y \in \mathcal{S}} N_k(s, a, y)$, which simply counts the number of time the action $a \in \mathcal{A}$ was selected while in state $s \in \mathcal{S}$. (iii) $N'_{h,k}(s, a) = \sum_{(s_{i,h}, a_{i,h}, s_{i,h+1}) \in \mathcal{H}} \mathbb{I}\{s_{i,h} = s, a_{i,h} = a\}$, counts the number of time the algorithm selected a specific action $a \in \mathcal{A}$ while in state $s \in \mathcal{S}$ during time step $h \in [H]$. When we would like to make explicit that a state-action pair was observed at time step h of episode k we use the following subscripts: $s_{k,h}$ and $a_{k,h}$. These counters allow us to build an empirical estimation of the transition function. Specifically, at the beginning of episode k , the empirical estimate of the transition function can be updated as follows:

$$\hat{P}_k(y|s, a) = \frac{N_k(s, a, y)}{N_k(s, a)}. \quad (\text{for all } s, y \in \mathcal{S} \text{ and } a \in \mathcal{A})$$

The empirical estimate of the transition function $\hat{P}_k(y|s, a)$ is used to compute the bonus term in line 25 of Algorithm 2.

Then, to encourage exploration, the algorithm adds a bonus term $b_{k,h}(s, a)$ (the exact definition of the bonus term is given in Algorithm 2 on line 25) to the values function $V_{k,h}(s)$, the larger the bonus or the value function, the more likely the corresponding state action pair will be visited. This procedure is detailed in Algorithm 2.

The key technicality that confers this improved efficiency to UCBVI is that, in general, to derive the bonus, previous algorithms usually upper bounded the quantity $\hat{P}_k^{\pi_k} - P^{\pi_k} V_{k,h+1}$ with $\|\hat{P}_k^{\pi_k} - P^{\pi_k}\|_1 \|V_{k,h+1}\|_\infty$ (see [10], for an example). The specificity of UCBVI type bonuses directly upper bound the value $\hat{P}_k^{\pi_k} - P^{\pi_k} V_{k,h+1}$ which lead to the following important result.

Theorem 2.3.1. *Given $\delta > 0$; UCBVI (see Alg. 2) regret is bounded, with probability $1 - \delta$, by*

$$\mathfrak{R}(M, \text{UCBVI}, K) = \tilde{O}(\sqrt{HSAT}) \quad (2.12)$$

with $T = KH$.

More details about UCBVI and a complete proof of Theorem 2.3.1 are available in [13].

Algorithm 2: Upper Confidence Bounds Value Iteration (UCBVI)

```

1 Function UCBVI ( $K$ ):
   Initialisation:  $\mathcal{H} = \emptyset$ 
2   for  $k = 1, 2, \dots, K$  do
3      $Q_{k,h} = \text{UCB-Q-Values}(\mathcal{H}, \text{bonus})$ 
4     for  $h = 1, 2, \dots, H$  do
5       Take action  $a_{k,h} = \arg\max_a Q_{k,h}(s_{k,h}, a)$ 
6        $\mathcal{H} = \mathcal{H} \cup (s_{k,h}, a_{k,h}, s_{k,h+1})$ 
   Output:  $V, \pi$ 

7 Function UCB-Q-Values( $\mathcal{H}, \text{bonus}$ ):
8   for  $(s, a, y) \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}$  do
9      $N_k(s, a, y) = \sum_{(s', a', y') \in \mathcal{H}} \mathbb{I}\{s' = s, a' = a, y' = y\}$ 
10     $N_k(s, a) = \sum_{y \in \mathcal{S}} N_k(s, a, y)$ 
11     $N'_{k,h}(s, a) = \sum_{(s_{i,h}, a_{i,h}, s_{i,h+1}) \in \mathcal{H}} \mathbb{I}\{s_{i,h} = s, a_{i,h} = a\} \quad \forall i \in [k]$ 
12     $\mathcal{K} = \{(s, a) \in \mathcal{S} \times \mathcal{A}, N_k(s, a) > 0\}$ 
13    Estimate  $\hat{P}_k(y|s, a) = \frac{N_k(s, a, y)}{N_k(s, a)} \quad \forall (s, a) \in \mathcal{K}$ 
14    Initialize  $V_{k,H+1}(s) = 0 \quad \forall s \in \mathcal{S}$ 
15    for  $h = H, H-1, \dots, 1$  do
16      for  $(s, a) \in \mathcal{S} \times \mathcal{A}$  do
17        if  $(s, a) \in \mathcal{K}$  then
18           $b_{k,h}(s, a) = \text{bonus}(\hat{P}_k(s, a), V_{k,h+1}, N_k, N'_{k,h})$ 
19           $Q_{k,h}(s, a) =$ 
20             $\min(Q_{k-1,h}(s, a), H, R(s, a) + (\hat{P}_k V_{k,h+1})(s, a) + b_{k,h}(s, a))$ 
21        else
22           $Q_{k,h}(s, a) = H$ 
23         $V_{k,h}(s) = \max_{a \in \mathcal{A}} Q_{k,h}(s, a)$ 
24    return  $Q_{k,h}$ 

24 Function  $\text{bonus}(\hat{P}_k(s, a), V_{k,h+1}, N_k, N'_{k,h})$ :
25    $b(s, a) = \sqrt{\frac{8L \text{Var}_{Y \sim \hat{P}_k(\cdot|s,a)} V_{k,h+1}(Y)}{N_k(s, a)}} + \frac{14HL}{3N_k(s, a)} + \sqrt{\frac{8 \sum_{y \in \mathcal{S}} \hat{P}_k(y|s, a) [\min(\frac{100^2 H^3 S^2 AL^2}{N'_{k,h+1}(y)}, H^2)]}{N_k(s, a)}}$ 
26   where  $L = \ln(5SAT/\delta)$ 
27   return  $b$ 

```

2.3.2 Posterior Sampling for Reinforcement Learning (PSRL)

This section presents PSRL, another model-based reinforcement learning that relies on Thompson sampling [128, 99]. In contrast to UCBVI, the algorithm maintains a posterior distribution over the set of all possible MDPs. The algorithm maintains a dataset of its experiences, which inform its posterior distribution, and at the beginning of each episode, it samples an MDP from its posterior distribution and performs planning on that randomly sampled MDP. This approach to RL was first introduced by [128].

Distributions over MDPs: This approach considers that we have access to a prior distribution over the set of possible MDPs, $\mathcal{M}$, which encodes any prior knowledge about the problem at hand. Then, as the algorithm collects data and learns about the MDP it interacts with, $M \in \mathcal{M}$, it adjusts its belief through a posterior update. The resulting distribution over $\mathcal{M}$ is called the posterior distribution, and it is the most likely distribution over $\mathcal{M}$, given our prior and the observation made so far. In particular, the posterior distribution quantifies the agent's uncertainty about its current estimate of the transition and reward functions. The definition of the distributions over MDPs largely relies on the following set of assumptions. First, the MDP has the Markov property, so every transition function and reward function are independent of the sequence of states and actions that lead the agent to its current state. Second, the transition and the reward functions are independent of each other.

With these assumptions in mind, the posterior distribution can be represented independently for each state-action pair (s, a) as a distribution over the parameters of the immediate reward distribution and the distribution over the parameters that govern the stochastic transitions.

For every state-action pair (s, a) , the transition function is map, $P_h : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$, to a distribution over the state space $\Delta(\mathcal{S})$. Typically, each of these distributions can be represented as a Categorical distribution with $P_h(s'|s, a) = \theta_{s'}^{(s, a)}$; with $\theta_{s'}^{(s, a)} \geq 0$ for all $(s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ and $\sum_{s' \in \mathcal{S}} \theta^{(s, a)}_{s'} = 1$.

As explained above, each θ is a random variable; typically, they are assumed to be drawn from a Dirichlet prior:

$$f_t(\theta_1, \dots, \theta_S; \alpha_1, \dots, \alpha_S) = \frac{\Gamma(\sum_{i=1}^S \alpha_i)}{\prod_{i=1}^S \Gamma(\alpha_i)} \prod_{i=1}^S \theta_i^{\alpha_i - 1} \quad (2.13)$$

with parameters $\alpha_i > 0$ for all $i \in [S]$ and with $\Gamma(\cdot)$ the gamma function.

It is usual to assume that the initial parameters are drawn from a flat Dirichlet distribution, that is, $\alpha = [1, \dots, 1]$. The prior distribution can be seen to regularize the learning problem.

Now, each time the algorithm collects a transition tuples (s, a, r, s') , it updates its belief. The Dirichlet and the Categorical distribution are conjugate distributions, so the resulting posterior distribution is also a Dirichlet distribution with the following parameter: if it transitions to a state s' after executing action a while in state s during time step h , then the new parameter is incremented by one, otherwise it remains unchanged:

$$\alpha'_i = \begin{cases} \alpha_i + 1 & \text{if } s' = i \\ \alpha_i & \text{otherwise} \end{cases} \quad (2.14)$$

If the reward has to be learned as well, it can be treated using the same approach; typically, the reward for an arbitrary state-action pair (s, a) can be modelled by a Normal distribution $r(s, a) \sim N(\mu_{s,a}, \sigma_{s,a})$. Where the parameters $\mu_{s,a}$ and $\sigma_{s,a}$ are governed by a Normal-inverse gamma distribution. A detailed description of PSRL is given in Algorithm 3.

Algorithm 3: Posterior Sampling for Reinforcement Learning (PSRL)

Input: f_t prior over transition, f_r prior over rewards

```

1  $\mathcal{D} = \emptyset$ 
2 for episode  $k = 1, \dots, K$  do
3   Sample MDP  $M_k$ , with  $f_t(\cdot|\mathcal{D})$  and  $f_r(\cdot|\mathcal{D})$ 
4   Compute  $\pi^k = \operatorname{argmax}_{\pi} V^{M_k}(s_1)$ 
5   for  $t = 1, \dots, H$  do
6     Execute  $a_t = \pi_t^k(s_t)$ 
7     Observe  $r_t$  and  $s_{t+1}$ 
8     Append to dataset  $\mathcal{D} = \mathcal{D} \cup \{t, s_t, a_t, r_t, s_{t+1}\}$ 

```

Output: V, π

2.3.3 Q-learning

Initially proposed by Watkins and Dayan [143, 142], Q-learning is a model-free algorithm to solve MDP. As we previously said, the specificity of this class of algorithms is that it does not maintain an explicit model of the environment. In the case of Q-learning, the algorithm directly models the Q-function. It is common in model-free settings to consider the transition functions P and the reward functions R as unknown. At each episode of learning, the algorithm does the following steps:

- it observes its current state $s_h \in \mathcal{S}$.
- it selects and execute an action $a_h \in \mathcal{A}$
- it observes the next state $s_{h+1} \in \mathcal{S}$ and receives the immediate reward r_h .
- it updates its current estimate of the Q-function. Whenever a transition tuple (s, a, r, s') at time step h is observed the Q-function is updated as follow:

$Q_h(s, a) = (1 - \alpha)Q_h(s, a) + \alpha[r_h + \max_a Q_{h+1}(s', a)]$. Where (s, a, r, s') is the observed tuple at time step h .

Algorithm 4 provides all the details of the implementation of ε -greedy Q-learning. The simple action selection strategy, called ε -greedy, is shown in line 5 of Algorithm 4. This

strategy is very simple, with probability $(1 - \varepsilon)$, we exploit, that is, we select the current optimal action $a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q_h(s, a)$. With probability ε , we explore, that is, we select uniformly at random an action ($\mathcal{U}(\mathcal{A})$ denotes the uniform distribution over the set $\mathcal{A}$). On line 7 of Algorithm 4, the Q-function is updated using the Bellman equation (Eq. 2.1). Note that the Q-function is gradually corrected using standard stochastic approximation through a learning rate $\alpha_k \in (0, 1]$. Note that it is common to decrease the value of the learning rate α_k over time; this explains why it is indexed by the current episode k . The learning rate is required to satisfy the Robins-Monro conditions [118]. For example, as depicted in Algorithm 4, the learning rate decays according to running average schedule $\alpha_k = \alpha_0 / (1 + k)$.

Algorithm 4: Q-learning

Input: $\alpha_0 \in (0, 1]$, $\varepsilon > 0$, K

Initialisation: $Q_h(s, a) = 0 \ \forall (s, a, h) \in \mathcal{S} \times \mathcal{A} \times \{1, \dots, H\}$

1 **for** $k = 1, 2, \dots, K$ **do**

2 $s_1 \sim \rho$

3 $\alpha_k = \frac{\alpha_0}{1+k}$

4 **for** $h = 1, 2, \dots, H$ **do**

5 Select action a_h : $\begin{cases} a_h = \operatorname{argmax}_{a \in \mathcal{A}} Q_h(s_h, a), & \text{w. p. } 1 - \varepsilon \\ a_h \sim \mathcal{U}(\mathcal{A}), & \text{w. p. } \varepsilon \end{cases}$

6 Execute a_h and observe s_{h+1} and r_h

7 $Q_h(s_h, a_h) = (1 - \alpha_k)Q_h(s_h, a_h) + \alpha_k[r_h + \max_a Q_{h+1}(s_{h+1}, a)]$

Output: Q

The following theorem guarantees the convergence of Q-learning; a complete proof can be found in [143].

Theorem 2.3.2. *Given bounded rewards $|r_h| \leq C$, learning rates $0 \leq \alpha_n < 1$, and the learning rate has to satisfy the Robins-Monro [118] conditions:*

$$\sum_{i=1}^{\infty} \alpha_{n_i(s,a)} = \infty, \quad \sum_{i=1}^{\infty} \alpha_{n_i(s,a)}^2 < \infty \quad (\forall (s, a) \in \mathcal{S} \times \mathcal{A})$$

then the Q -function learned by Algorithm 4 will converge to the optimal Q -function $Q_h^*(s, a)$ as $K \rightarrow \infty$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ and $h \in \{1, \dots, H\}$.

Note that in ε -greedy Q -learning, it is up to the user to set the exploration rate ε (in the next section, we will see an extension of the Q -learning algorithm where this parameter is computed). This parameter controls the rate at which the algorithm selects a sub-optimal action according to its current belief. As the Q -values are initialised to 0, they are not very informative at the beginning. It is, in practice, useful to use a scheduler for ε since it is usually more efficient to start with a large ε and gradually trust more and more the algorithm's Q -function and decrease the value of ε_k . Note that while Q -learning usually works well in practice, in theory, the ε -greedy strategy might lead the algorithm to suffer exponential regret [62]. The *combination lock* [62, 68] is an example of a particularly hard MDP, for which ε -greedy Q -learning suffers exponential regret. Consider an MDP with a special state $s^* \in \mathcal{S}$; the initial state distribution is a singleton that always selects $s_1 = s^*$. For the first half of the steps $h = \{1, \dots, H/2\}$ there is one special action $a^* \in \mathcal{A}$ such that the next state is always s^* , that is $P_h(s^*|s^*, a^*) = 1$. Then, for any other state $s \in \mathcal{S} \setminus \{s^*\}$ and any other action $a \in \mathcal{A} \setminus \{a^*\}$ the transition distribution $P_h(\cdot|s, a)$ is uniform over $\mathcal{S} \setminus \{s^*\}$. In the second half of the episode, that is when $h = H/2 + 1, \dots, H$, all states become absorbing states, that is, $P_h(s|s, a) = 1$ for all actions $a \in \mathcal{A}$. The agent is only rewarded if it is in state s^* in the second half of the episode, that is $r_h(s^*, a) = 1$ for any action $a \in \mathcal{A}$ and for step $h > H/2$. For any other state-action-step combination, the reward is null, $r_h(s, a) = 0$. This MDP is particularly challenging for Q -learning as to observe its first reward; the algorithm has to produce the sequence of $H/2$ actions a^* . Since the Q -values are initially set to 0 and remain set to 0 as long as no reward is observed, the probability of observing the first reward is at most $A^{-H/2}$, so the algorithm will suffer a regret of $H/2$ per round for the first $A^{H/2}$ episodes.

2.3.4 Q-UCB

Lastly, Q-UCB [62] is a sample-efficient model-free algorithm. With a carefully designed exploration strategy, Q-UCB avoids the potential exponential complexity of Q-learning. Q-UCB works similarly to Q-learning; it directly models the Q-function and collects data through interactions with the environment. The difference is that Q-UCB does not select action with an ε -greedy strategy. In contrast, it greedily selects action according to an optimistic Q function. The optimistic Q-function adds a bonus term for each state-action pair inversely proportional to the number of visits of this specific state-action pair. This bonus term encourages, to some extent, the exploration of unknown regions of the state-action space. The efficiency of this exploration strategy is proven in the following theorem.

Theorem 2.3.3. *There exists a constant $c > 0$ such that, for any $p \in (0, 1)$, if we choose the bonus term $b_t = c\sqrt{H^3\iota/t}$, then with probability $1 - p$, the total regret of Q-learning with UCB-Hoeffding is at most $\mathcal{O}(\sqrt{H^4 SAT \iota})$ with $\iota := \log(SAT/p)$.*

A complete proof of this result is provided in [62].

Algorithm 5: Q-learning with UCB-Hoeffding (Q-UCB)

Initialisation: $Q_h(s, a) = H$ and $N_h(s, a) = 0 \forall (s, a, h) \in \mathcal{S} \times \mathcal{A} \times \{1, \dots, H\}$

1 $\iota = \log(SAT/p)$

2 **for** $k = 1, 2, \dots, K$ **do**

3 $s_1 \sim \rho$

4 **for** $h = 1, 2, \dots, H$ **do**

5 Select action $a_h = \operatorname{argmax}_{a \in \mathcal{A}} Q_h(s_h, a)$

6 Execute a_h and observe s_{h+1} and r_h

7 $t = N_h(s_h, a_h); N_h(s_h, a_h) = N_h(s_h, a_h) + 1$

8 $b_t = c\sqrt{H^3\iota/t}; \alpha_t := \frac{H+1}{H+t}$

9 $Q_h(s_h, a_h) = (1 - \alpha_t)Q_h(s_h, a_h) + \alpha_t[r_h + V_{h+1}(s_{h+1}) + b_t]$

10 $V_h(s_h) = \min\{H, \max_{a \in \mathcal{A}} Q_h(s_h, a)\}$

Output: Q

2.3.5 Upper Confidence Fixed-Horizon Reinforcement Learning (UCFH)

The Upper Confidence Fixed-Horizon episodic reinforcement learning algorithm (UCFH) is an example of a sample-efficient algorithm. With an upper bound on the number of episode $\mathcal{O}(\frac{|\mathcal{S}|^2|\mathcal{A}|H^2}{\epsilon^2} \ln(\frac{1}{\delta}))$ which matches the lower bound $\mathcal{O}(\frac{|\mathcal{S}||\mathcal{A}|H^2}{\epsilon^2} \ln(\frac{1}{\delta}))$ up to some log terms and a linear dependency on the state space size $|\mathcal{S}|$. Similar to UCBVI, the algorithm assumes that the reward function is known and aims to learn the transition function p and the optimal policy π^* .

UCFH uses optimism in the face of uncertainty. The algorithm works in phases consisting of optimistic planning, policy execution, and model updating. Each phase is indexed by k , and as the agent interacts with its environment, it collects (s, a, r, s') tuples. Critically, UCFH maintains a confidence set over statistically plausible transition functions, and this is the particular set of conditions used to build this confidence set that grants UCFH superior sample efficiency.

Similarly to UCBVI (Alg. 2), UCFH maintains a set of four counters; $n(s, a)$ and $n(s, a, s')$ counts the number of times the tuples (s, a) and (s, a, s') are observed since the beginning of training. They are updated in the model updating phase on line 9 of algorithm 6. The two other counters $v(s, a)$, $v(s, a, s')$ are used to count the number of visits during the sampling procedure and are set to zero at the end of the model updating phase.

These counters are needed to compute an empirical estimate of the transition function $\hat{p}(s'|s, a) = \frac{N(s, a, s')}{N(s, a)}$ for all $(s, a) \in \mathcal{S} \times \mathcal{A}$ and $s' \in \mathcal{S}$. For each transition tuple (s, a, s') the empirical estimate of the transition function $\hat{p}$ and the associated number of visits n are used to compute a confidence set (see line 22 in Alg. 6)

$$\mathcal{P} = \left\{ p' \in [0, 1] : \text{if } n > 1 : |p'(1 - p') - p(1 - p)| \geq \frac{2 \ln(6/\delta_1)}{n - 1}, \right. \\ \left. |p - p'| \leq \min \left(\sqrt{\frac{\ln(6/\delta_1)}{2n}}, \sqrt{\frac{2p(1 - p)}{n} \ln(6/\delta_1)} + \frac{7}{3(n - 1)} \ln \frac{6}{\delta_1} \right) \right\}$$

which differentiates itself from usual confidence set bounds as it adds a condition on the standard deviation of the transition $\sqrt{p(1-p)}$.

This confidence set is then used for the optimistic planning phase. Given the set of statistically plausible MDP $\mathcal{M}_k$ at phase k , the algorithm computes the optimistic policy π^k . This computation of the optimistic policy is detailed in FixedHorizonEVI (Alg. 7), which receives as input a set of MDPs $\mathcal{M}$ and returns the pair $\tilde{\pi}, \tilde{M}$ which maximizes the value function $V_{\tilde{M}}^{\tilde{\pi}} = \max_{M \in \mathcal{M}, \pi} V_M^{\pi}$. Note that to simplify the planning problem, the algorithm considers the convex hull of the confidence set $\mathcal{M}$, which then allows to find the optimal policy via dynamic programming, using the following recursive equation:

$$\tilde{Q}_i^*(s, a) = \begin{cases} r_i(s), & \text{if } i = H \\ r_i(s, a) + \max_{\tilde{p}_i \in \mathcal{P}_{s,a}} \left\{ \sum_{s' \in \mathcal{S}} \tilde{p}_i \max_{b \in \mathcal{A}} \tilde{Q}_{i+1}^*(s', b) \right\}, & \text{if } i \in [1, \dots, H-1]. \end{cases}$$

Where

$$\mathcal{P}_{s,a} := \{p \in [0, 1]^{|\mathcal{S}|} \mid \|p\|_1 = 1, \forall s' \in \mathcal{S} : p(s'|s, a) \in \text{conv}(\text{ConfidenceSet}(\hat{p}(s'|s, a), N(s, a)))\},$$

with $\text{conv}(\mathcal{X})$ represents the convex hull of a set $\mathcal{X}$.

The inner max that represents the policy can be directly solved by enumeration, and the outer maximization problem can be easily solved as it consists of putting as much weight as possible on the successor state with the highest value, as depicted on line 13 in Algorithm 7.

Lastly, the policy execution phase consists of the execution of the optimistic policy at phase k , π^k , until the condition on line 8 of Algorithm 6 is reached.

We rewrite the results proven in [29] for completeness. First, lemma 2.3.4 ensures the convergence of FixedHorizonEVI (Alg. 7). Second, Theorem 2.3.5 provides an upper bound on the sample complexity of UCFH.

The validity of *FixedHorizonEVI* is proven in [29]. For completeness, we rewrite the resulting lemma below.

Lemma 2.3.4. *Given a confidence set over MDPs $\mathcal{M}$, *FixedHorizonEVI* returns $\tilde{M}, \tilde{\pi} = \operatorname{argmax}_{M \in \mathcal{M}', \pi} V_M^\pi(s_0)$. And since $\mathcal{M} \subseteq \mathcal{M}'$ it holds that $V_{\tilde{M}}^{\tilde{\pi}}(s_0) \geq \max_{M \in \mathcal{M}, \pi} V_M^\pi(s_0)$.*

The following theorem bounds the number of episodes required by UCFH to learn an ε -optimal policy.

Theorem 2.3.5. *Given any $\varepsilon < 0$, $\delta \geq 1$ it holds that with probability $1 - \delta$, UCFH generates a sequence of policies π^k with a most*

$$\tilde{O}\left(\frac{H^2 |\mathcal{S}|^2 \mathcal{S}}{\varepsilon^2} \ln\left(\frac{1}{\delta}\right)\right)$$

episodes with $V_1^*(s_0) - V_1^{\pi^k}(s_0) > \varepsilon$.

A detailed proof of this result is available in [29].

Algorithm 6: Upper Confidence Fixed-Horizon episodic RL (UCFH)**Input:** desired accuracy $\varepsilon \in (0, 1]$, failure tolerance $\delta \in (0, 1]$, MDP M **Output:** with probability $1 - \delta$: ε -optimal policy**Initialisation:** $k := 1$, $\omega_{\min} := \frac{\varepsilon}{4H|\mathcal{S}|}$, $\delta_1 := \frac{\delta}{2U_{\max}C}$, $U_{\max} := |\mathcal{S} \times \mathcal{A}| \log_2 \frac{|\mathcal{S}|H}{\omega_{\min}}$ $m := 512(\log_2 \log_2 H)^2 \frac{CH^2}{\varepsilon^2} \log^2 \left(\frac{8H^2|\mathcal{S}|^2}{\varepsilon} \right) \ln \frac{6|\mathcal{S} \times \mathcal{A}|C \log_2^2(4|\mathcal{S}|^2H^2/\varepsilon)}{\delta}$ $N(s, a) = v(s, a) = N(s, a, s') = v(s, a, s') := 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{S}$ **1 while do** */* Optimistic planning */***2** $\hat{p}(s'|s, a) := N(s, a, s')/N(s, a)$ for all (s, a) with $N(s, a) > 0$ **3** $\mathcal{M}_k := \left\{ \tilde{M} \in \mathcal{M}_{\text{nonst.}} : \forall (s, a) \in \mathcal{S} \times \mathcal{A}, t = 1, \dots, H, s' \in \mathcal{S} \right.$
4 $\left. \tilde{p}_t(s'|s, a) \in \text{ConfidenceSet}(\hat{p}(s'|s, a), N(s, a)) \right\}$ **5** $\tilde{M}_k, \pi^k := \text{FixedHorizonEVI}(\mathcal{M}_k)$ **6 repeat****7** | $\text{SampleEpisode}(\pi^k)$ */* sample from M using π^k */***8 until** $\exists (s, a) \in \mathcal{S} \times \mathcal{A}$ with $v(s, a) \geq \max\{m\omega_{\min}, N(s, a)\}$ and $N(s, a) \leq |\mathcal{S}|mH$ */* Update the statistics of the above mentioned (s, a) pair */***9** $N(s, a) := N(s, a) + v(s, a)$ **10** $N(s, a, s') := N(s, a, s') + v(s, a, s') \quad \forall s' \in \mathcal{S}$ **11** $v(s, a) := v(s, a, s') := 0 \quad s' \in \mathcal{S}$ **12** $k = k + 1$ **13 Function** *SampleEpisode* (π):**14** $s_0 \sim \rho$ **15 for** $t = 1, \dots, H - 1$ **do****16** | $a_t := \pi_{t+1}(s_t)$ **17** | $s_{t+1} \sim p(\cdot | s_t, a_t)$ **18** | $v(s_t, a_t) := v(s_t, a_t) + 1$ **19** | $v(s_t, a_t, s_{t+1}) := v(s_t, a_t, s_{t+1}) + 1$ **20 Function** *ConfidenceSet* (p, n):**21** $\mathcal{P} := \left\{ p' \in [0, 1] : \text{if } n > 1 : |p'(1 - p') - p(1 - p)| \leq \frac{2\ln(6/\delta_1)}{n-1}, \right.$ **22** $\left. |p - p'| \leq \min \left(\sqrt{\frac{\ln(6/\delta_1)}{n-1}}, \sqrt{\frac{2p(1-p)}{n} \ln(6/\delta_1)} + \frac{2}{3n} \ln \frac{6}{\delta_1} \right) \right\}$ **23** **return** $\mathcal{P}$

Algorithm 7: Fixed-horizon Extended Value Iteration (FixedHorizonEVI)

Input: A set of MDPs $\mathcal{M}$

```

1  $\tilde{Q}_{H:H}^*(s, a) = r_H(s) \forall s, a \in \mathcal{S} \times \mathcal{A}$ 
2 for  $t = H - 1, \dots, 1$  do
3    $\pi_{t+1}(s) := \operatorname{argmax}_{a \in \mathcal{A}} \tilde{Q}_{t+1:H}^*(s, a) \forall s \in \mathcal{S}$ 
4   sort states such that:
5    $\tilde{Q}_{t+1:H}^*(s^{(i)}, \pi_{t+1}(s^{(i)})) \geq \tilde{Q}_{t+1:H}^*(s^{(i+1)}, \pi_{t+1}(s^{(i+1)}))$ 
6   for  $s, a \in \mathcal{S} \times \mathcal{A}$  do
7      $\tilde{p}_t(s'|s, a) := \min \operatorname{ConfidenceSet}(\hat{p}(s'|s, a), n(s, a)) \quad \forall s \in \mathcal{S}$ 
8      $\Delta := 1 - \sum_{s' \in \mathcal{S}} \tilde{p}_t(s'|s, a)$ 
9      $i := 1$ 
10    while  $\Delta > 0$  do
11       $s' := s^{(i)}$ 
12       $\Delta' := \min\{\Delta, \max \operatorname{ConfidenceSet}(\hat{p}(s'|s, a), n(s, a)) - \tilde{p}_t(s'|s, a)\}$ 
13       $\tilde{p}_t(s'|s, a) := \tilde{p}_t(s'|s, a) + \Delta'$ 
14       $\Delta := \Delta - \Delta'$ 
15       $i := i + 1$ 
16     $\tilde{Q}_{t:H}^*(s, a) = \sum_{s' \in \mathcal{S}} \tilde{p}_t(s'|s, a) \tilde{Q}_{t+1:H}^*(s', \pi_{t+1}(s'))$ 
17  $\pi_1(s) := \operatorname{argmax}_{a \in \mathcal{A}} \tilde{Q}_{1:H}^*(s, a) \quad \forall s \in \mathcal{S}$ 
18 return MDP with transition probabilities  $\tilde{p}_t$  and optimal policy  $\pi$ 

```

Chapter 3

A Lower Bound on HRL Efficiency

Hierarchical Reinforcement Learning (HRL) algorithms have the ability to plan and make decisions across multiple levels of abstraction, which is a key feature distinguishing them from traditional reinforcement learning methods. Empirical studies have demonstrated that employing state or temporal abstractions can lead to substantial improvements in the sample efficiency of Reinforcement Learning algorithms. However, despite these observed benefits, a comprehensive understanding of the underlying reasons for such efficiency gains remains elusive, and no well-established, theoretically grounded design principles currently exist to guide the development of HRL systems.

In this chapter, we address this gap by deriving a lower bound on the sample complexity for a specific class of goal-conditioned HRL algorithms. This theoretical lower bound enables us to precisely quantify the advantages offered by hierarchical decomposition. Building on this insight, we propose a simple Q-learning-type algorithm that effectively leverages the hierarchical structure of tasks.

To verify our theoretical contributions, we conduct empirical studies that evaluate the sample complexity of the proposed hierarchical algorithm across a variety of environments, including hierarchical n -rooms tasks and the well-known Gymnasium’s Taxi domain. The

hierarchical n -rooms tasks are designed to allow for adjustable complexity, spanning multiple orders of magnitude, providing a robust testing ground for our algorithm.

The combination of theoretical analysis and empirical validation marks a step toward addressing the fundamental question of how much hierarchical decomposition can enhance the performance of reinforcement learning compared to monolithic approaches. Our findings offer a deeper understanding of HRL and suggest promising directions for designing more efficient learning algorithms.

The work presented in this chapter was previously published as:

Robert, A., Pike-Burke, C., & Faisal, A. A. (2024). Sample complexity of goal-conditioned hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

3.1 Motivation

Hierarchical Reinforcement Learning (HRL) [132, 33, 37, 21] exploits the natural hierarchical structure of tasks by decomposing problems into simpler sub-problems, thereby creating more sample-efficient algorithms. Numerous empirical studies support the idea that hierarchical methods can significantly boost the sample efficiency of traditional Reinforcement Learning (RL) algorithms [94, 95, 139, 33]. However, there are instances where the introduction of temporal abstractions has led to a worsening of sample complexity [63]. This raises a critical question: under what circumstances does HRL improve sample complexity?

Theoretical research on sample complexity bounds in Machine Learning has played a fundamental role in advancing the field. Understanding sample complexity is particularly valuable in RL because, unlike supervised learning, RL tends to be far more resource-intensive, sample-inefficient, and reliant on costly data from interactions with the environment. Moreover, theoretical results [29, 77, 11, 62, 131] have often revealed insights that shape

the development of new algorithms. For example, the analysis of the Q-learning algorithm in [62] deepened our understanding of exploration strategies in model-free RL, and the policy gradient theorem [131] laid the foundation for a host of new RL methods. In contrast, theoretical work on hierarchical RL remains sparse, with many important contributions being empirical, focusing on hierarchies of states [33, 38], time [113], or actions [138, 109, 3].

In an effort to bridge this theoretical gap, we focus on a tabular setting of goal-conditioned HRL [94, 21]. By analyzing the induced Markov Decision Process (MDP) decomposition, we derive a lower bound on the sample complexity for this HRL framework. This lower bound provides insight into when hierarchical decomposition is advantageous and serves as the foundation for a novel hierarchical Q-learning algorithm that exploits this structure to enhance sample efficiency. In the goal-conditioned HRL paradigm, both a high-level policy and a low-level policy are jointly optimized to achieve a primary objective. Specifically, the high-level policy selects sub-goals, while the low-level policy executes primitive actions to achieve these sub-goals (see Fig.3.1, left diagram). This hierarchical breakdown of tasks is highly general and can be applied to a wide range of problem domains, effectively transforming the original MDP into two interrelated sub-MDPs (discussed in Sec.3.2.2).

This chapter advances our understanding of HRL through the following key contributions:

- A theoretical lower bound on sample-complexity: We establish a lower bound on the sample complexity associated with hierarchical decomposition (see Sec. 3.3). This bound equips practitioners with a quantitative measure of the potential efficiency gains from task decomposition.
- A new Q-learning algorithm: We introduce a simple yet novel Q-learning algorithm tailored to goal-conditioned hierarchical RL, leveraging the hierarchical structure for improved sample efficiency (see Sec. 3.5).
- Empirical validation: We validate our theoretical findings through a series of experiments using a synthetic task with adjustable hierarchical complexity (see Sec.3.6).

These results demonstrate that our derived bound can reliably identify cases where hierarchical decomposition yields significant benefits (see Sec.3.6).

Together, these contributions enhance our theoretical and practical understanding of when and how HRL can be effectively utilized to achieve improved sample complexity over monolithic RL approaches.

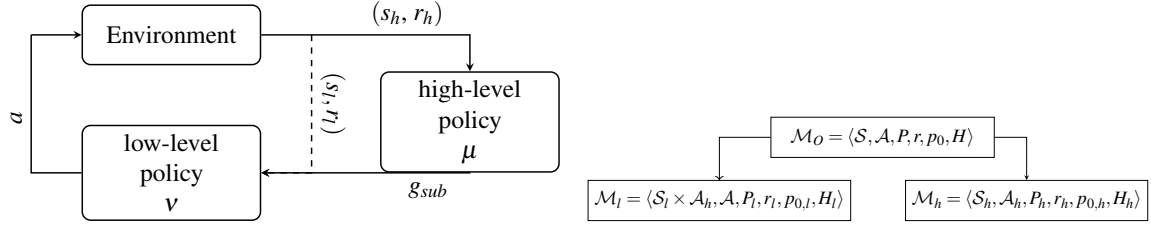


Fig. 3.1 In the left block diagram, the agent's architecture is broken down into two key components: a high-level policy μ and a low-level policy ν , which work together to solve a given task. The high-level policy, μ , receives inputs consisting of the high-level state and reward, represented as the pair (s_h, r_h) . Based on this information, the high-level policy determines and outputs a sub-goal, g_{sub} , which is then passed to the low-level policy, ν . The low-level policy observes the subgoal g_{sub} as well as the state-reward pair, (s_l, r_l) , where s_l is the low-level state and r_l is the low-level reward. The low-level policy ν interacts directly with the environment by executing primitive actions, a , to solve the instructed sub-goal. The right diagram provides a visual representation of the hierarchical decomposition induced by the hierarchical framework. The diagram illustrates how the original MDP, $\mathcal{M}_O$, is decomposed into two distinct sub-MDPs: the low-level MDP, $\mathcal{M}_l$, which is managed by the low-level policy, and the high-level MDP, $\mathcal{M}_h$, which is governed by the high-level policy. A detailed description of this decomposition is provided in Section 3.2.2.

3.2 General Formulation of Goal-Conditioned HRL

In Chapter 2, we presented a non-stationary finite-horizon MDP (see Definition 2.1.1), but in this work, we consider a stationary finite-horizon MDP setting. The main difference is that the reward and transition dynamics do not depend on the time step. But for the sake of completeness, we start by formally defining stationary finite-horizon MDPs.

3.2.1 Stationary Finite-Horizon Markov Decision Process

An episodic finite-horizon MDP is defined by the following tuple: $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, r, P, \rho, H \rangle$. Where $\mathcal{S}$ is a finite state space of size $|\mathcal{S}|$ and $\mathcal{A}$ is a finite action space of size $|\mathcal{A}|$. The goal of the task is encoded in a terminal state $g \in \mathcal{S}$. We assume the reward function $r(s, g) \in [-a, b]$ (for $a, b \geq 0$) is known $\forall s \in \mathcal{S}, g \in \mathcal{S}$, the reward function penalises each step with a negative reward of at most $-a$ and reward the completion of the task with a positive reward of b . The initial state distribution p_0 is a distribution over states that is used to determine in which

state an episode starts. The learner interacts with the MDP in episodes of at most H time steps. The episode's starting state $s_0 \sim p_0$ is drawn from the initial state distribution. In each time step $t = 1, \dots, H$, the learner observes a state s_t and chooses an action a_t . Given a state action pair (s_t, a_t) the next state $s_{t+1} \sim P(\cdot | s_t, a_t)$ is drawn from the transition kernel. Eventually, the episode ends because the agent reaches the terminal state or has interacted with the environment for H time-steps.

The agent aims to select actions that maximise the expected return throughout an episode. We typically assume actions are chosen according to a policy, $a_t \sim \pi(s_t)$, where π is a function that maps each state and time step pair to a distribution over actions $\pi : \mathcal{S} \times [H] \rightarrow \Delta_{\mathcal{A}}$, and $\Delta_{\mathcal{A}}$ is the set of all probability distributions over $\mathcal{A}$ and $[H]$ is the set of natural numbers up to H . The agent aims to select a policy π to maximise the sum of expected rewards, $\mathbb{E}[\sum_{t=1}^H r_t | s_1 \sim \rho, a_t \sim \pi(s_t), s_{t+1} \sim P(s_t, a_t)]$, where the expectation is over the initial state distribution, the policy and the stochastic transitions. Note that it is usually the case for finite-horizon MDPs that the policy also depends on the current time step. However, to simplify notation, we do not make this relation explicit.

For a given policy π , we define at time step $\tau \in [H]$ the value function, $V_{\tau}^{\pi}(s)$:

$$V_{\tau}^{\pi}(s) = \mathbb{E}_P \left[\sum_{t=\tau}^H r_t | s_{\tau} = s, a_{\tau:H} \sim \pi \right],$$

and the Q-function, $Q_{\tau}^{\pi}(s, a)$:

$$Q_{\tau}^{\pi}(s, a) = \mathbb{E}_P \left[\sum_{t=\tau}^H r_t | s_{\tau} = s, a_{\tau} = a, a_{\tau+1:H} \sim \pi \right],$$

where $s \in \mathcal{S}$ denotes the state, $a \in \mathcal{A}$ is the action and the notation $a_{\tau:H} \sim \pi$ is used to specify that actions between time step τ and time step $H - 1$ were selected using π . Whenever $\tau > 1$, the expectations only depend on the stochastic policy and transition function. The optimal policy π^* is the policy with the highest value function for every time step and every state,

$V_\tau^{\pi^*}(s) = V_\tau^*(s) = \max_{\pi} V_\tau^\pi(s) \forall \tau \in [H], \forall s \in \mathcal{S}$. There is always a deterministic Markov policy that maximises the total expected reward in a finite-horizon MDP [114].

In this chapter, we assess the quality of a policy by its expected value at the beginning of an episode. To lighten the notation, we define $V^\pi = \mathbb{E}_{s \sim \rho} [V_0^\pi(s)]$ to be the expected value from the beginning of an episode, where the expectation is taken over initial states.

3.2.2 Episodic Finite-Horizon Hierarchical MDP

Consider an episodic, finite-horizon MDP $\mathcal{M}_o$, which can be hierarchically decomposed into two MDPs: a low-level MDP, $\mathcal{M}_l$, and a high-level MDP, $\mathcal{M}_h$. This decomposition is illustrated in the right diagram of Fig. 3.1. To ensure clarity throughout, we introduce the following notation: the subscript o is used to denote the original MDP, while subscripts l and h refer to the low-level and high-level MDPs, respectively.

The decomposition into low-level and high-level MDPs can be formally represented by the following tuples: $\mathcal{M}_l = \langle \mathcal{S}_l \times \mathcal{A}_h, \mathcal{A}, r_l, P_l, \rho_l, H_l \rangle$ for the low-level MDP, and $\mathcal{M}_h = \langle \mathcal{S}_h, \mathcal{A}_h, r_h, P_h, \rho_h, H_h \rangle$ for the high-level MDP. For this hierarchical decomposition to be valid, the following conditions must hold:

Action spaces: The action space for the low-level MDP consists of the primitive actions available to the agent for interacting with the environment. This space is identical to the action space of the original MDP, denoted by $\mathcal{A}$. On the other hand, the high-level action space, $\mathcal{A}_h$, is composed of sub-goals that the high-level agent can assign to the low-level agent. The set of sub-goals in $\mathcal{A}_h$ is assumed to be sufficient to enable the agent to solve any given task, regardless of the state. Note that the set of available sub-goals usually depends on the current high-level state, s_h , $\mathcal{A}_h(s_h) \subset \mathcal{A}_h$. To simplify the notation we do not make this relation explicit.

State spaces: Both the low-level state s_l and the high-level state s_h must contain enough information to reconstruct the full state, s , from the original MDP, $\mathcal{M}_o$. Typically, states $s \in \mathcal{S} \subset \mathbb{R}^d$ are represented as multi-dimensional vectors, with each dimension encoding a specific feature or characteristic. This allows for the state to be factored into a tuple $(s_l, s_h) \in \mathcal{S}_l \times \mathcal{S}_h$, where one part of the state belongs to the low-level MDP, and the other belongs to the high-level MDP. Thus, any state $s \in \mathcal{S}_o$ in the original MDP can be represented as a combination of a low-level state $s_l \in \mathcal{S}_l$ and a high-level state $s_h \in \mathcal{S}_h$. Additionally, since the low-level policy is goal-conditioned, its state space also includes the sub-goal $a_h \in \mathcal{A}_h$, leading to a complete low-level state space of $\mathcal{S}_l \times \mathcal{A}_h$, where the sub-goal is part of the state description.

Initial state distributions: The initial state distribution for the high-level MDP, ρ_h , is a restriction of the initial state distribution of the original MDP ρ onto the high-level state space, $\mathcal{S}_h$. For any state in the original MDP $s \in \mathcal{S}_o$ and its decomposition (s_l, s_h) , with $s_l \in \mathcal{S}_l$ and $s_h \in \mathcal{S}_h$ we have $\rho_h(s_h) = \sum_{s: (s_l, s_h) \in \mathcal{S}_o} \rho(s)$. The initial state distribution for the low-level MDP, $\rho_l(\cdot | s_{h,0})$, is conditioned on the high-level initial state, $s_{h,0}$, and spans the low-level state space $\mathcal{S}_l$, ensuring that the original state distribution ρ can be reconstructed from the low-level and high-level initial state distributions, that is, $\rho(s) = \rho_h(s_h) \rho_l(s_l | s_h)$, where $s_l \in \mathcal{S}_l$ and $s_h \in \mathcal{S}_h$ correspond to the hierarchical decomposition of $s \in \mathcal{S}_o$.

Transition functions: The low-level transition function P_l is a restriction of the original transition function P on the low-level state description and sub-goals $\mathcal{S}_l \times \mathcal{A}_h$. A key challenge in HRL is that the high-level transition function, P_h , is influenced by the low-level policy $\mathbf{v}$. The success of the low-level policy in achieving its sub-goal directly affects the probability of transitioning to a new high-level state. Consequently, the high-level transition function, $P_h(s'_h | s_h, a_h, \mathbf{v})$, depends not only on the high-level state s_h and the sub-goal a_h but

also on the low-level policy v . This dependence makes the high-level transition function non-stationary, increasing the complexity of the learning process.

Reward functions: The goal states of the original MDP belong to the original state space $\mathcal{S}$. Similarly, the low-level states associated with completing a sub-goal belong to $\mathcal{S}_l$. The low-level reward function can then be derived from the original reward function, as $r_l(s_l, g_{sub}) = 2r(s, g)$, where s and g represent the reconstructed low-level state and sub-goal in the original MDP. The high-level reward function is computed as the sum of rewards accumulated by the low-level policy over the course of a sub-episode. Specifically, the reward for a high-level action, a_h , consists of the sum of the low-level reward accumulated by the low-level policy while trying to solve this particular sub-goal $r_h(s_h, a_h) = \sum_{t=1}^{H_l} r_l(s_{l,t}, a_h)$.

Horizons: The total number of steps in an episode for the original MDP is constrained by its horizon, H . As a result, the horizons of the high-level MDP, H_h , and the low-level MDP, H_l , must satisfy the condition $H = H_h \cdot H_l$.

In practice, it is always possible to find a hierarchical decomposition that satisfies these conditions. For instance, a trivial decomposition could involve a high-level agent whose sole action encodes the end goal of the task, while the low-level agent is given access to complete state information without abstraction. However, while such a decomposition is technically valid, it may not offer any real advantages. The challenge is identifying decompositions that lead to meaningful improvements, particularly in sample efficiency.

The policies governing the high-level MDP, $\mathcal{M}_h$, and low-level MDP, $\mathcal{M}_l$, are denoted as μ and v , respectively. In a goal-conditioned HRL framework, the low-level policy v maps a combination of the low-level state and the sub-goal to an action, i.e., $v : \mathcal{S}_l \times \mathcal{A}_h \rightarrow \mathcal{A}_l$, while the high-level policy, μ , selects sub-goals based on the high-level state, $\mu : \mathcal{S}_h \rightarrow \mathcal{A}_h$. The performance of these policies can be evaluated using their respective value functions, V_l^v

for the low-level MDP and V_h^μ for the high-level MDP. Similar to the non-hierarchical case, we can define optimal high-level and low-level policies as $v^* = \operatorname{argmax}_v V_l^v$ for the low-level policy and $\mu^* = \operatorname{argmax}_\mu V_h^\mu$ for the high-level policy. Any pair of low- and high-level policies (v, μ) can be combined to produce a policy π that interacts with the original MDP $\mathcal{M}_o$. This is formally defined as:

Definition 3.2.1. A hierarchical policy consists of a pair (v, μ) that can be mapped to a policy π in the original MDP $\mathcal{M}_o$ as follows:

$$\pi(a|s) = \pi(a|s_l, s_h) = \sum_{a_h \in \mathcal{A}_h} \mu(a_h|s_h) v(a|a_h, s_l). \quad (3.1)$$

The optimal hierarchical policy is obtained by combining the optimal high-level policy μ^* and low-level policy v^* . However, not every policy in the original MDP can be expressed as a decomposition of (v, μ) . Specifically, there is no guarantee that the optimal policy for the original MDP π^* has an equivalent hierarchical decomposition.

Our goal is to determine when a hierarchical decomposition of the original MDP leads to faster learning of a near-optimal policy. Therefore, we are interested in evaluating the performance of the combination of v and μ while they interact with the original MDP $\mathcal{M}_o$. To quantify the performance of the pair of policies (v, μ) we consider the corresponding policy in the original MDP (see Definition 3.2.1), π , and we evaluate it in the original MDP $\mathcal{M}_o$. Formally, we have:

$$V_o^{v, \mu} = \mathbb{E}_{s \sim \rho} [V_{o,0}^\pi(s)],$$

where the subscript o is a reminder that we are evaluating a policy on the original MDP $\mathcal{M}_o$.

When learning in this hierarchical setting, the agent is tasked with learning both a high-level policy μ and a low-level policy v . The learning is done in an episodic setting, with episodes unfolding as follows. The agent observes the initial state and uses the high-level policy to select a sub-goal. The low-level policy then attempts to achieve this sub-goal over

the course of the next H_l steps. Once H_l steps have passed or the sub-goal is achieved, the high-level agent observes a new high-level state and selects a new sub-goal. This process repeats until the entire task is completed or the horizon H is reached. Through this interaction, each agent seeks to maximise their respective value functions, $\max_v V_l^v$ and $\max_\mu V_h^\mu$, within their corresponding MDPs. After the completion of an episode (i.e. H steps), the algorithm can update its low- and high-level policies.

3.2.3 Probably-Approximately Correct RL

In this section, we recall the definition of the Probably-Approximately Correct (PAC) guarantee defined in Chapter 2 and adapt it to the hierarchical setting we consider in this work. We aim to find, in as few episodes as possible, a pair of policies (v, μ) with a near-optimal value. To formalise this, we use the PAC RL notion defined in Chapter 2 (see Definition 2.2.4). We denote by Δ_k the sub-optimality gap, that is the difference between the optimal (non-hierarchical) policy π^* and the current hierarchical policy (v^k, μ^k) : $\Delta_k := V_o^* - V_o^{v^k, \mu^k}$. Note that both policies are evaluated on the original MDP $\mathcal{M}_o$.

Definition 3.2.2. An algorithm satisfies a PAC bound N if, for a given input $\epsilon, \delta > 0$, it satisfies the following condition for any episodic fixed-horizon MDP: with probability at least $1 - \delta$, the algorithm plays policies that are at least ϵ -optimal after at most N episodes. That is, with probability at least $1 - \delta$, $\max\{k \in \mathbb{N} : \Delta_k > \epsilon\} \leq N$, where N is a polynomial that can depend on the properties of the problem instance.

In Section 3.3, we will bound the sample complexity of HRL algorithms. In this context, the sample complexity refers to the number of episodes, N , in the original MDP, during which the algorithm may not follow a policy that is at least ϵ -optimal with probability at least $1 - \delta$.

3.2.4 Running Example

We introduce the following illustrative example to clarify the hierarchical decomposition of an MDP. Consider a grid-world environment where the original MDP models a maze-solving task. In this environment, the agent's state is represented by a tuple (R, C) , where R indicates the current room the agent occupies, and C represents the specific cell within that room. The agent's objective is to navigate the maze to reach an absorbing goal state. The reward function penalises the agent with a small cost, $-a$, at each time step unless the agent reaches the goal state. Upon reaching the goal, the agent no longer suffers penalties and receives a reward of 0 for all remaining time steps. Mathematically, this reward function is expressed as $r(s) = -a\mathbb{I}\{s \neq g\}$, where $g \in \mathcal{S}$ denotes the goal state and $\mathbb{I}$ is the indicator function.

To decompose this problem hierarchically, we define a high-level MDP and a low-level MDP as follows. The high-level MDP abstracts the task by focusing on the maze at the level of rooms, ignoring the finer details of individual cells. In this high-level representation, the agent's state is reduced to the current room, R . The goal of the high-level agent is to determine the sequence of rooms that leads to the goal. At each high-level time step, the agent identifies the most valuable exit through which the low-level agent should navigate to progress toward the goal. As discussed in Section 3.2.2, the reward for the high-level agent is the sum of the rewards accumulated by the low-level agent during the sub-episode. Meanwhile, the low-level agent operates within a single room and is unaware of the larger maze layout. Its task is to move from cell to cell within the room, aiming to reach a specific exit designated by the high-level agent. The low-level agent faces a penalty of $-2a$ for each action it takes until it reaches the sub-goal. Upon reaching the sub-goal, the penalty ceases. The low-level reward function can be written as $r(s) = -2a\mathbb{I}\{s \neq g_{sub}\}$, where g_{sub} denotes the current sub-goal.

One such example is robotics, where the low-level agent would be tasked with controlling the joints of the robot to produce movements selected by the high-level policy, whose goal

is to perform tasks that require a sequence of distinct movements (i.e. navigational tasks, manipulation tasks or a combination of both).

We will revisit this example throughout the chapter to illustrate key concepts, but it is important to note that the hierarchical reinforcement learning (HRL) framework we describe is general and applicable to a wide variety of tasks beyond grid-world navigation. For instance, in robotics applications, the low-level agent might be responsible for fine-grained control over a robot’s joints, executing movements directed by the high-level policy. The high-level policy, in turn, would focus on higher-level objectives such as completing navigational or manipulation tasks or even combining both. In this case, the high-level agent could direct the robot to perform complex sequences of movements, such as reaching for and grasping objects or navigating through an environment, while the low-level agent handles the detailed control of the robot’s limbs to carry out those instructions. This hierarchical structure enables efficient decision-making in complex, multi-step tasks by breaking them down into manageable sub-problems.

3.3 Lower Bound on the Sample Complexity of HRL

It has been established in [29] that, for any RL algorithm, the number of sample episodes required to achieve an (ϵ, δ) -accurate policy (in the original MDP) is lower bounded by:

$$\mathbb{E}[N] = \Omega\left(\frac{|\mathcal{S}||\mathcal{A}|H^2}{\epsilon^2} \ln\left(\frac{1}{\delta + c}\right)\right), \quad (3.2)$$

where c is a positive constant, ϵ denotes the desired accuracy of the policy and δ is the probability of failure.

We now extend the previous result to hierarchical Markov Decision Processes (MDPs). Before proceeding, it is important to recognise a key limitation of hierarchical policies. Even the best hierarchical policy, as defined in Eq. (3.1), may still be sub-optimal in comparison

to the true optimal policy π^* for the original MDP $\mathcal{M}_o$. This is a direct consequence of the hierarchical goal-conditioned architecture. Let's consider that while executing a sub-episode, the agent realises that another sub-goal becomes more valuable. The proposed architecture does not allow interruptions. Consequently, the agent will first have to complete the current sub-episode before adapting to the new circumstances. In contrast, a monolithic policy will never suffer from such a constraint, and it will always be able to select the most valuable action.

Formally, let $V_o^{v^*, \mu^*}$ denote the value of the optimal hierarchical policy in the original MDP. The sub-optimality gap, Δ_k , denotes the gap between the value of the current policy pair (v^k, μ^k) and the value of the optimal policy π^* :

$$\Delta_k = V_o^* - V_o^{v^k, \mu^k}.$$

This sub-optimality gap is necessarily larger than the gap between the value of current hierarchical policy (v^k, μ^k) and the value of the optimal hierarchical policy (v^*, μ^*) :

$$\Delta_k \geq V_o^{v^*, \mu^*} - V_o^{v^k, \mu^k}.$$

Thus, if for a given number of episodes N , the hierarchical policy (v^k, μ^k) is such that the difference between the value of the optimal hierarchical policy (v^*, μ^*) and the value of the current hierarchical policy exceeds ϵ , that is, $V_o^{v^*, \mu^*} - V_o^{v^k, \mu^k} \geq \epsilon$, it must also hold that the overall sub-optimality gap $\Delta_k \geq \epsilon$ for at least N episodes. Consequently, this implies that N provides a lower bound on the number of episodes during which the algorithm plays a sub-optimal policy.

In the following theorem, we derive a lower bound on the number of episodes, N , required to learn a pair of hierarchical policies (v, μ) that are ϵ -optimal relative to the optimal hierarchical policy (v^*, μ^*) . Importantly, based on the argument outlined above, this

lower bound on the sample complexity for learning an ε -optimal hierarchical policy also serves as a lower bound for learning an ε -optimal policy with respect to the optimal policy in the original MDP, π^* . This theorem highlights the fundamental relationship between hierarchical decomposition and sample complexity in reinforcement learning.

Theorem 3.3.1. *There exist positive constants c_l , c_h and δ_0 such that for every $\delta \in (0, \delta_0)$ and for every algorithm A that satisfies a PAC guarantee for (ε, δ) and outputs a deterministic policy, there is a fixed horizon MDP such that A must interact for*

$$\mathbb{E}[N] = \Omega\left(\max\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right), \frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)\right) \quad (3.3)$$

episodes, in the original MDP, until the policy is (ε, δ) -accurate.

A complete proof of this theorem is given in Section 3.4. For the moment, we just highlight the main steps of the proof.

Sketch of the proof: The following condition holds for any ε -optimal pair of policies, $|V_o^{v^*, \mu^*} - V_o^{v, \mu}| \leq \varepsilon$. To derive a lower bound on the number of episodes N required to obtain an ε -optimal pair of policies (v, μ) we used the following steps:

- (i) The objective is decomposed using the triangle inequality, $|V_o^{v^*, \mu^*} - V_o^{v^*, \mu}| + |V_o^{v^*, \mu} - V_o^{v, \mu}| \leq \varepsilon$.
- (ii) We show that the number of samples required to guarantee $|V_o^{v^*, \mu^*} - V_o^{v^*, \mu}| \leq \varepsilon/2$ is bounded by $\Omega\left(\frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)$
- (iii) We show that the number of samples required to guarantee $|V_o^{v^*, \mu} - V_o^{v, \mu}| \leq \varepsilon/2$ is bounded by $\Omega\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right)\right)$

These three steps combined lead us to the desired result in Theorem 3.3.1; see Section 3.4 for more details.

3.3.1 Interpretation of the Sample Complexity Bound:

Comparing the lower bound 3.3¹ to the lower bound derived for the original MDP 3.2, allow us to identify the specific characteristics of the hierarchical decomposition that may lead to improved sample efficiency. In most cases, only one of the two MDP characteristics—either the high-level or low-level component—will dominate the overall sample complexity due to the presence of the *max* operator in the bound (Eq. 3.3). To minimize the impact of the *max* term and achieve better sample efficiency, the task complexity should be distributed as evenly as possible between the high- and low-level MDPs. Below, we explore several key insights derived from this analysis:

State abstraction: In the hierarchical framework, only one of the state space sizes (either for the high-level or low-level MDP) will dominate the bound in Eq. 3.3. This observation implies that an efficient hierarchical decomposition involves partitioning the original state space in a balanced manner between the high and low levels of the hierarchy. A well-balanced decomposition ensures that neither the high-level nor the low-level state space dominates, which helps to reduce the overall sample complexity.

Another critical factor is the concept of low-level policy re-usability. Due to state abstraction, the low-level agent may be able to reuse the policy it learns across multiple states. For instance, different states $s_1, s_2 \in \mathcal{S}$ might share the same low-level component $s_l \in \mathcal{S}_l$, allowing the low-level agent to generalize and reuse its learned behaviour across these states.

To capture this phenomenon, we can rewrite the lower bound in Eq. 3.3 using the *re-usability index*, $\kappa = \frac{|\mathcal{S}|}{|\mathcal{S}_l|}$, which quantifies the degree to which the low-level policy can be reused. The lower bound then becomes:

¹It is important to note that this is a lower bound, meaning we cannot yet confirm the existence of algorithms that achieve it.

$$\mathbb{E}[N] = \Omega\left(\max\left(\frac{|\mathcal{S} \times \mathcal{A}_h|}{\kappa} |\mathcal{A}| H_l^2 \ln\left(\frac{1}{\delta + c_l}\right), \frac{|\mathcal{S}_h| |\mathcal{A}_h| H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)\right). \quad (3.4)$$

Eq. 3.4 highlights that increasing the re-usability index κ can directly improve sample efficiency by reducing the complexity of the low-level MDP.

Temporal abstraction: Similarly, only one of the two time horizons, either the high-level horizon H_h or the low-level horizon H_l , will dominate the sample complexity bound. This suggests that an effective temporal abstraction, where the task duration is reasonably distributed across the high- and low-level policies, can contribute to improved sample efficiency. Temporal abstraction also simplifies the credit assignment problem by breaking down the original task with time horizon H into a sequence of sub-tasks with a shorter horizon H_l . Consequently, the low-level agent receives more frequent feedback by completing sub-tasks that are significantly shorter than the original task. At the same time, the high-level agent trajectory consists of fewer decision points, as it operates over sub-goals rather than primitive actions, further reducing the complexity of the original MDP.

High-level action space: The high-level action space plays a unique role, as it appears in both terms of the $\max(\cdot, \cdot)$ in Eq. 3.3. This underscores the importance of maintaining a compact and efficient representation of sub-goals.

When considering a hierarchical decomposition, it is interesting to note the contrast between the state space decomposition and the design of the high-level action space. An efficient state decomposition requires distributing the information available at each level as evenly as possible. In contrast, both the low-level MDP $\mathcal{M}_l$ and the high-level MDP $\mathcal{M}_h$ directly benefit from a compact sub-goal space.

The insights discussed above reveal several properties of hierarchical decomposition that can improve the sample complexity of reinforcement learning algorithms. However, it is important to note that our lower bound also demonstrates that hierarchical decomposition does not always guarantee improved sample efficiency. In fact, there are scenarios where a poorly designed hierarchical structure could fail to offer any advantage in terms of sample complexity.

Our lower bound thus serves as a valuable "sanity check", allowing practitioners to assess whether a given hierarchical decomposition has the potential to enhance sample efficiency. However, even when a promising decomposition is identified, finding an algorithm that can fully leverage this decomposition to achieve optimal sample complexity remains a non-trivial challenge.

That being said, the proposed Q-learning-based hierarchical algorithm presented in this work demonstrates empirically that it can effectively exploit the benefits of hierarchical decomposition. In Section 3.6, we explore several different MDP decompositions and validate that when our theoretical bound indicates that the decomposition should be beneficial, the algorithm (see Sec. 3.5) leverages this structure to achieve reduced sample complexity compared to non-hierarchical approaches.

In Chapter 4, we focus on a subset of hierarchical problems, and for this family of MDP, we provide a class of algorithms with theoretical guarantees in terms of an upper bound on the regret suffered.

3.4 Proof of the Lower Bound

Theorem 3.3.1 establishes that there exist positive constants c_l , c_h and δ_0 such that for every $\delta \in (0, \delta_0)$ and for any algorithm A that satisfies a PAC guarantee for (ϵ, δ) and outputs a

deterministic policy, there is a fixed horizon MDP such that A must collect

$$\mathbb{E}[N_e] = \Omega\left(\max\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right), \frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)\right) \quad (3.5)$$

episodes until its policy is (ε, δ) -accurate.

Proof. An ε -optimal pair of policies $(\mathbf{v}, \mu)$ satisfies

$|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon$. Using the triangle inequality, we get $|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}^*, \mu}| + |V_o^{\mathbf{v}^*, \mu} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon$, then we will have $|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon$. As mentioned in Section 3.3 we focus on showing the following:

- (i) the number of samples needed to guarantee $|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}^*, \mu}| \leq \varepsilon/2$ is bounded by $\Omega\left(\frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)$
- (ii) the number of samples needed to guarantee $|V_o^{\mathbf{v}^*, \mu} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon/2$ is bounded by $\Omega\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right)\right)$

Combining the results of (i) and (ii) implies that after

$$\Omega\left(\max\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right), \frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right)\right)$$

episodes, we will have $|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}^*, \mu}| + |V_o^{\mathbf{v}^*, \mu} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon$ and so $|V_o^{\mathbf{v}^*, \mu^*} - V_o^{\mathbf{v}, \mu}| \leq \varepsilon$; which is the result we want to show.

Part (i) Learning the high-level policy μ when the low-level policy $\mathbf{v}^*$ is optimal is equivalent to learning an ε -optimal high-level policy in a stationary high-level MDP $\mathcal{M}_h$. The high-level transition function does not change over time as the low-level behaviour is fixed to follow $\mathbf{v}^*$. We can lower bound the number of episodes N_h needed to have: $|V_h^* - V_h^{\mathbf{v}^*, \mu}| \leq \varepsilon$, by directly applying Eq. (3.2) to the high-level MDP characteristics to have

$$\mathbb{E}[N_h] = \Omega\left(\frac{|\mathcal{S}_h||\mathcal{A}_h|H_h^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_h}\right)\right).$$

To be able to use this result to construct the bound of interest, we need to make sure these results can be translated into the original MDP: $|V_o^{V^*, \mu^*} - V_o^{V^*, \mu}| \leq \varepsilon$. In particular, the reward functions are not the same for $\mathcal{M}_o$ and $\mathcal{M}_h$. We defined the original MDP's reward function to penalize each step unless it reaches the end goal, in which case it rewards the agent with a bonus. This reward function was initially defined to take values in the interval $[-a, b]$. Without loss of generality, we shift the reward function so that the rewards are now drawn from the interval $[-a - b, 0]$. In this new setting, a successful completion of the task is rewarded with a bonus of 0. Similarly, the low-level reward function penalizes each step unless it reaches the instructed sub-goal. Lastly, the high-level reward function $r_h(s_h, a_h) = \sum_{t=1}^{H_l} r_l(s_{l,t}, a_h)$, consists of the sum of the low-level rewards accumulated during each sub-episode. A difference of scale in value function arises because high-level rewards include the bonus the low-level agent receives for completing each sub-goal, whereas the value functions in the original MDP do not include this bonus. To compensate for this difference, the low-level reward is re-scaled with a penalty twice as large; the low-level reward r_l now takes values in $[-2a - 2b, 0]$, and, under this specific reward function, the completion of a sub-goal is rewarded by 0 as well. The re-scaling ensures that the accumulated penalties are larger in the high-level MDPs, even for trajectories with a single intermediary step. This guarantees that $|V_o^{V^*, \mu^*} - V_o^{V^*, \mu}| \leq |V_h^* - V_h^{V^*, \mu}|$. Hence after $\mathbb{E}[N_h]$ episodes, we have $|V_o^* - V_o^{V^*, \mu}| \leq \varepsilon$.

Part (ii) By a similar argument to Part (i), we can bound the number of episodes in the low-level MDP required to obtain an ε -optimal low-level policy for a fixed high-level policy μ . In particular, a lower bound on the number of episodes N_l required to have $|V_l^{V^*, \mu} - V_l^{V, \mu}| \leq \varepsilon$ can directly be obtained from Eq. (3.2):

$$\mathbb{E}[N_l] = \Omega\left(\frac{|\mathcal{S}_l||\mathcal{A}_h||\mathcal{A}|H_l^2}{\varepsilon^2} \ln\left(\frac{1}{\delta + c_l}\right)\right).$$

Ultimately, we are interested in a lower bound on the number of episodes in the original MDP N_o . Since a single episode in the original MDP corresponds to several episodes in the low-level MDP, we can divide the number of episodes by a factor H_h : $\mathbb{E}[N_o] \geq \frac{\mathbb{E}[N_l]}{H_h}$. However, the difference in episode length between the two MDPs also induces a difference of scale in their value functions. To ensure that the learned policies are ε -optimal in the original MDP we need to ensure that $|V_l^{v^*,\mu} - V_l^{v,\mu}| \leq \frac{\varepsilon}{H_h}$, which requires at least $H_h^2 \mathbb{E}[N_l]$ low-level episodes. Combining the two arguments, we get that $\mathbb{E}[N_o] \geq \frac{\mathbb{E}[N_l] H_h^2}{H_h} = \mathbb{E}[N_l] H_h$. Since we are computing a lower bound on the number of episodes in the original MDP required to learn a near-optimal policy and recognising that $H_h \geq 1$, we can conclude that $\mathbb{E}[N_o] \geq \mathbb{E}[N_l]$. This leads us to a lower bound on the number of episodes needed to obtain an ε -accurate pair of policies as the one stated in the theorem. $\square$

3.5 Stationary Hierarchical Q-Learning (SHQL)

Once we know that we have an MDP with a hierarchical decomposition that could lead to improved sample complexity, the next challenge is to design an algorithm that can successfully leverage this hierarchical structure. This section proposes a model-free hierarchical algorithm for this purpose, *Stationary Hierarchical Q-learning* (SHQL). Probably the most challenging aspect of jointly learning the high-level and low-level policies is the non-stationarity induced by the low-level policy on the high-level transition dynamics, P_h . We already mentioned (in Sec. 3.2.2) that the high-level transition function, P_h , changes over time as the low-level policy changes (i.e. as we learn the low-level policy). Indeed, the next high-level state depends on the current behaviour of the low-level policy, v , and if it can reach the instructed sub-goal. To circumvent this issue, we leverage the fact that the algorithm knows beforehand what a successful sub-episode is, i.e. it knows if the low-level agent managed to reach the instructed sub-goal. Similar tricks are used in off-policy methods. This is typically leveraged with hindsight correction [76]. Hindsight correction modifies the trajectories used to perform the algorithm update; specifically, the sub-goal is corrected to be the state in which the trajectory ends. However, in the considered on-policy setting, we propose that the algorithm only performs a high-level update when the low-level policy successfully reaches the goal, i.e. we ignore all low-level trajectories where the low-level policy failed to reach the instructed sub-goal. This ensures that the algorithm filters out all bad examples from the training set and that the behaviour of P_h is more stable. Note, however, that the reward function of the high-level agent remains non-stationary. At first, sub-goals will not be solved optimally, incurring a small reward to the high-level agent. However, as we train the low-level policy, it becomes more efficient at solving the instructed sub-goal, and the associated reward will increase. The function *LowLevelUpdate* in Algorithm 8 highlights that the low-level agent uses Q-learning-type updates on the observed low-level transitions and rewards. Similarly, the high-level agent also performs Q-learning-type updates, but, as explained,

only on successful transitions, as emphasized by the condition at line 15 of Algorithm 8.

Algorithm 8: Stationary Hierarchical Q-learning (SHQL)

Input: $Q_{:, :, :}^l = 0, Q_{:, :, :}^h = 0, \text{done}_h = \text{False}, t = k = 0$

```

1 while not  $\text{done}_h$  and  $k < K$  do
2   Observe  $s_h, s_l$ 
3    $g_{sub} = \pi_h(s_h)$ 
4    $\text{done}_l = \text{False}$  /* True if sub-goal solved */
5
6   while not  $\text{done}_l$  and  $t < T$  do
7      $a_l = \pi_l(s_l)$ 
8     Observe  $s'_l, r_l, \text{done}_l$ 
9      $\text{LowLevelUpdate}((s_l, a_l, r_l, s'_l, g_{sub}))$ 
10     $s_l = s'_l$ 
11     $t = t + 1$ 
12  Observe  $s'_h, r_h, \text{done}_h$ 
13  if  $\text{done}_l$  then
14     $Q_{nxt}^h = \max_g Q_{s'_h, g}^h$ 
15     $Q_{s_h, g_{sub}}^h = Q_{s_h, g_{sub}}^h + \alpha * (r_h + Q_{nxt}^h)$ 
16     $s_h = s'_h$ 
17     $k = k + 1$ 
18 Function  $\text{LowLevelUpdate}(s, a, r, s', g_{sub})$ :
19    $Q_{nxt}^l = \max_{a'} Q_{g_{sub}, s', a'}^l$ 
20    $Q_{g_{sub}, s, a}^l = Q_{g_{sub}, s, a}^l + \alpha * (r + Q_{nxt}^l)$ 
21   return  $Q^l$ 

```

3.6 Experiments

We now empirically validate² if the lower bound found in Section 3.3 correctly predicts the impact of the hierarchical decomposition on a collection of MDPs. We do so by comparing the performance of SHQL and standard Q-learning on a series of hierarchical tasks. To satisfy the assumption of hierarchical structure, we constructed a class of environments that extends the well-known *four-room* problem to an arbitrary number of rooms; we called this class of problems the *N*-rooms problems. We also constructed a hierarchical version of the Gymnasium’s Taxi environment.

3.6.1 N-Rooms Environment

The maze environment is constructed by arranging a customizable number of rooms within a grid layout, allowing for flexible scaling of the task’s complexity. The overarching high-level objective for the agent is to learn the optimal sequence of rooms that will guide the agent from the starting point, located in the top-left room, to the goal in the bottom-right room. The low-level task is to learn to navigate within each room and to reach the instructed hallway successfully. The hallways act as sub-goals, and the high-level policy uses them to guide the agent.

We introduce variability in the room profiles to modulate further the task’s difficulty beyond simply adjusting the maze size (i.e., increasing the number of rooms). Each room can have a distinct layout and structure, impacting the complexity of the low-level navigation task. These different room configurations, visualized in the rightmost plot of Fig. 3.3, add an additional layer of challenge and impact the hierarchical decomposition.

The set of MDPs generated by these environments are the following:

The original MDP: This is a standard grid-world MDP, where the state space indicates the

²Experiments were run on a 12th Gen Intel Core i7 with 16GB of RAM, to train the agents on the largest maze considered takes ~ 7 minutes.

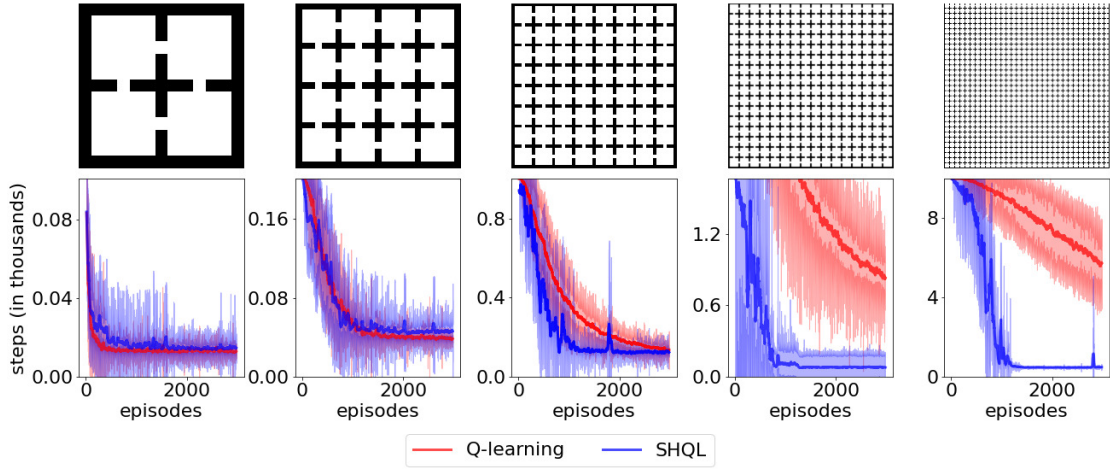


Fig. 3.2 The grid of plots depicts, on the top row, the mazes whose size ranges from 4 rooms to 1024 rooms. The bottom row shows the steps required for SHQL (in blue) and Q-learning (in red) to complete the maze. The standard deviation is obtained by running ten different seeds.

current cell where the agent is located, and the action space allows the agent to move one cell in any cardinal direction (North, South, East, West). To obtain stochastic environments, each action has a success probability of $p_{success} = 4/5$. In case of failure, the executed action will be chosen uniformly at random.

The high-level MDP: The high-level state space is restricted to the room where the agent is currently located, and the exact position of the agent within that room is abstracted away. The high-level actions instruct the low-level to reach one of the available hallways. Note that not all rooms have access to the four hallways. The action executed by the high-level depends on the current high-level state $\mathcal{A}_h(s_h) \subseteq \mathcal{A}_h$ ensuring that all sub-goals instructed are reachable by the low-level policy v within H_l steps.

The low-level MDP: The low-level agent only observes the agent's current location within a room and the sub-goal instructed by the high-level agent (one of the reachable hallways). It then uses its policy v to select a sequence of primitive action space $a \in \mathcal{A}$ (the four cardinal directions) to reach the desired hallway. All required code to reproduce the experiments is made available online [1].

Identical Rooms

We first present the N -rooms environment in its simplest form. In this case, the environments considered are mazes built by assembling identical rooms without obstacles (i.e. the top room profile in Fig. 3.3). We measure the performance of our algorithm as the number of steps it needed to reach the goal room. Figure 3.2 visualises the empirical performance of the proposed algorithm, SHQL, with a blue learning curve. We compare the performance of SHQL with Q-learning (the red learning curves) in the original MDP. Note that Q-learning doesn't have access to the additional structure in the MDP. The lower bound in Equation 3.2 suggests that the hierarchical machinery comes with a cost, and for simple mazes (e.g. with 4 or 16 rooms), the hierarchical decomposition does not provide sufficient improvement to compensate for that initial cost. However, as the problem grows in complexity, empirical evaluations indicate significant improvements in sample efficiency. Note that these results are also supported by our lower bound on the sample complexity (yellow curve on the rightmost plot of Fig. 3.3), which highlights that the efficiency gains of HRL are mostly visible in complex MDPs (i.e. MDPs with large state and action spaces).

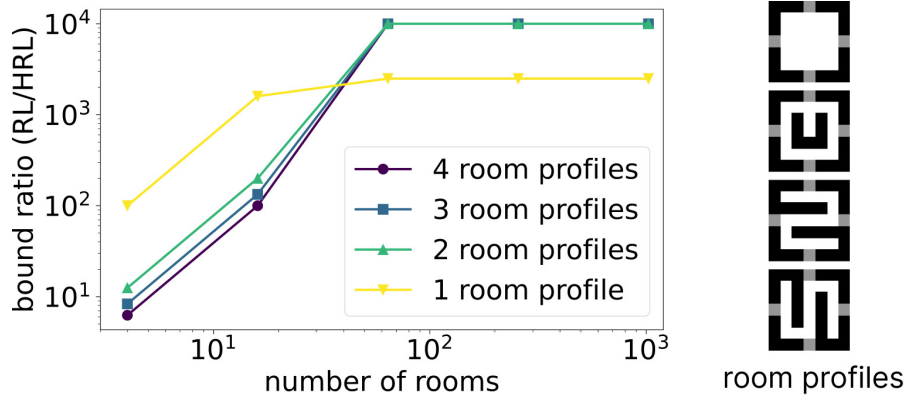


Fig. 3.3 The left-hand plot shows the evolution of the ratio between the RL bound Eq. (3.2) and the HRL bound Eq. (3.3) for various mazes and different room profiles. The plateau is obtained when the high-level MDP dominates the bound, leading to the following ratio: $\frac{|S||A|H}{|S_h||A_h|H_h}$. The curves are colour-coded such that a darker curve indicates more room profiles were considered. The right-hand side of the plot shows the different room profiles available to build the mazes.

Multiple Room Profiles

We now consider more challenging tasks, and we increase the number of room profiles used to construct the mazes. The rightmost plot of Figure 3.3 illustrates the four different room profiles we considered. Essentially, each room profile has a different obstacle in the room. The low-level agent must now learn to navigate multiple types of rooms to reach the sub-goal sent by the high-level agent. We first present the most complicated task considered, that is, when we consider four different room profiles. The performance of our algorithm in this setting is shown in Figure 3.6. The introduction of different room profiles allows us to modulate the complexity of the low-level MDP, in contrast to varying the number of rooms, which only affects the complexity of the high-level MDP. This additional complexity directly impacts the low-level MDP as it results in a larger state space $\mathcal{S}_l$ and may also lead to a longer low-level horizon H_l as the optimal trajectory might require more time to navigate around obstacles to reach the desired hallway successfully. Even in this more challenging setting, it becomes evident that, as the number of rooms increases, the hierarchy's benefits become more significant. Nevertheless, a direct comparison of Figure 3.2 and Figure 3.6 highlights that the introduction of various room layouts has little effect on the Q-learning curve (in red), this behaviour is expected since the monolithic description of the problem already encodes each cell of the maze individually, regardless of the room it is in. In contrast, it makes the task slightly more challenging for the SHQL, the blue learning curve, especially when the number of rooms is small since it suffers from the increased complexity of the low-level MDP. But, when the number of rooms is sufficiently large for the high-level MDP complexity to dominate the bound, the benefit of hierarchical decomposition becomes evident. For completeness, we include in Figure 3.7 (in Fig. 3.4 and Fig. 3.5) the learning curve obtained when mazes are built from two or three different room layouts. These learning curves highlight the same behaviour that we previously discussed. With simple mazes, the low-level

MDP strongly dominates the bound, but as we increase the high-level MDP complexity (i.e. the number of rooms), the benefit of the hierarchical machinery becomes evident.

Interestingly, these empirical results are supported by our lower bound. In Figure 3.3, we consider the evolution of the ratio between the HRL lower bound and the monolithic RL lower bound (HRL/RL) for the set of n -rooms MDPs discussed above. It is important to remember that in these experiments, the low-level MDP complexity remains constant for a given set of room profiles. Indeed, regardless of the number of rooms in the maze, the low-level task is to learn how to navigate the room profiles present. This is important to properly understand the plateau of the curves in the rightmost plot of Figure 3.3. Essentially, when we add rooms to the maze, we increase the complexity of the original MDP and the complexity of the high-level MDP. The curves plateau when the high-level MDP dominates the overall sample complexity. At this point, the complexity of the original MDP and the complexity of the high-level MDP grow at the same pace. Formally, the height of the plateau consists of the ratio between the RL bound (Eq. 3.2) and the high-level part of the HRL bound (Eq. 3.3), $\frac{|\mathcal{S}||\mathcal{A}|H}{|\mathcal{S}_h||\mathcal{A}_h|H_h}$.

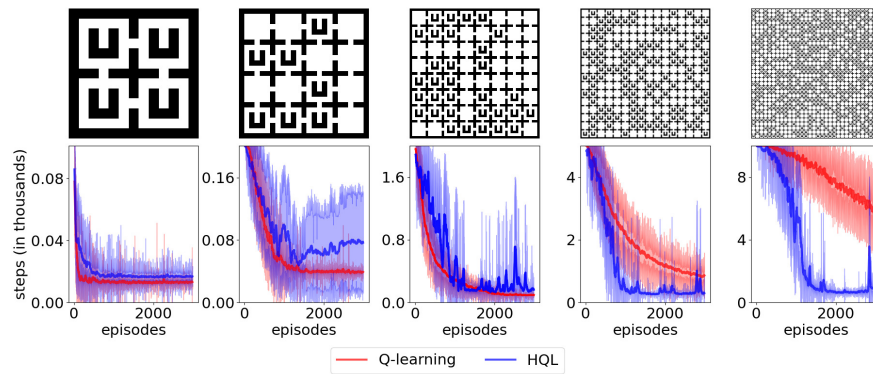
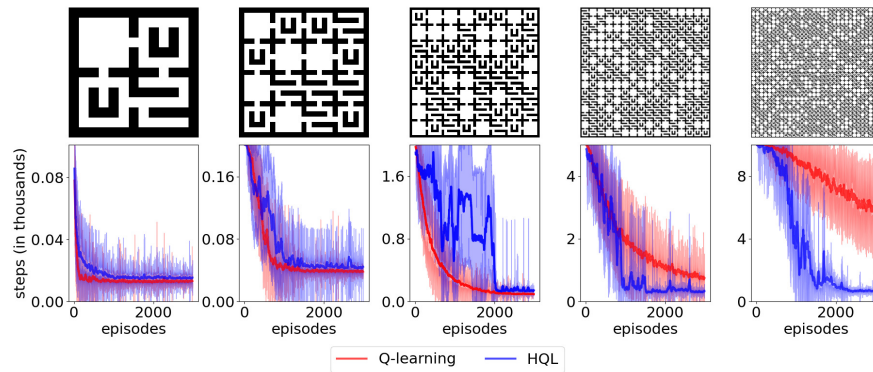
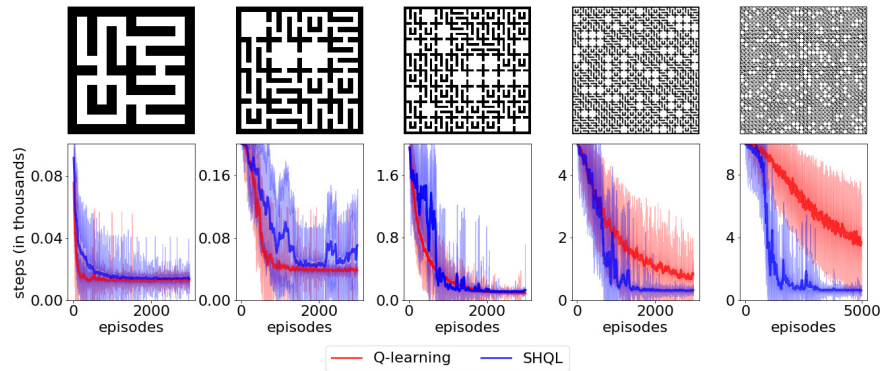
Fig. 3.4 Performance of SHQL in the N -room problem with two room profilesFig. 3.5 Performance of SHQL in the N -room problem with three room profilesFig. 3.6 Performance of SHQL in the N -room problem with four room profiles

Fig. 3.7 This collection of plots shows learning curves on various maze sizes. Each row of the plot considers a different number of room instances, where rooms can have one, two or three types of obstacles. For each experiment, the agent's performance is measured by the number of steps required to solve the task.

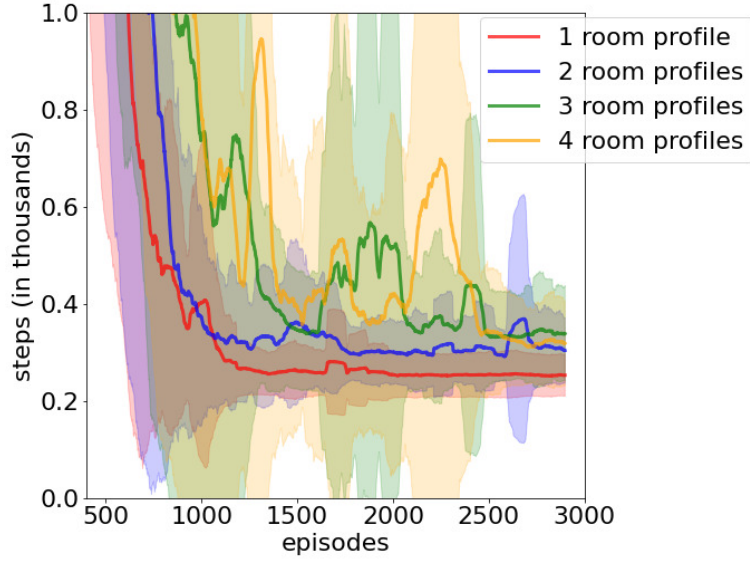


Fig. 3.8 This figure illustrates how the added difficulty of learning new room profiles impacts the performance of SHQL. In particular, we observe, as one would expect, that as we increase the number of room profiles, the number of episodes necessary to learn an optimal policy increases, as well as the learning curve variance.

It is important to note that adding room profiles also impacts the hierarchical algorithm as it increases the overall complexity of the problem. Figure 3.8 shows the performance of the hierarchical algorithm, SHQL, on a maze with 256 rooms and up to four different room profiles. It clearly shows that adding room profiles makes the problem more challenging. In particular, we can see that the algorithm requires more training episodes to converge and suffers from greater variance in performance.

3.6.2 Gymnasium Taxi Task

Another task we consider is the taxi environment [37]. In this environment, the agent is tasked to pick up a passenger and bring him/her to a specific location. As illustrated in Fig. 3.9 the taxi environment is a grid world composed of 25 cells (5×5 grid). At the beginning of an episode, the passenger is assigned a random initial location, which can only be in one of the four taxi stations depicted by the letters R, G, Y and B in Fig. 3.9 right plot, and a final destination which is any of the three remaining taxi stations. When visualising the

environment, the passenger's location is denoted by the letter in blue and the final destination by the purple letter. The initial position of the taxi is also randomly sampled, and to complete the task, the taxi will have first to pick up the passenger and then deliver him to his final destination.

This problem can be formalized by a MDP. The state space $\mathcal{S}$ encodes the 404 reachable states. The twenty-five possible taxi locations, the four possible destinations and the five potential passenger locations suggest a total of five hundred states. Several combinations are considered infeasible. In particular, the states where the passenger is at the destination but the taxi is somewhere else are not possible and there are $24 * 4 = 96$ such configurations. The action space $\mathcal{A}$ consists of the six possible actions (move up, down, left, right, pickup, dropoff). The default horizon H considered for this environment is 200 steps. The initial state distribution uniformly chooses the cell where the taxi starts the episode and the passenger's initial location and destination. The reward function assigns a positive bonus (+20) for successfully delivering the passenger, a penalty for illegal action such as an inappropriate *dropoff* and *pickup* action (−10) and a small penalty (−1) for each remaining step. The characteristics of the original MDP for this problem are the following $|\mathcal{S}| = 404$, $|\mathcal{A}| = 6$ and $H = 200$.

To apply SHQL, the following decomposition was considered:

- (i) **High-level MDP:** The high-level state space only encodes the current passenger location and the final desired destination, hence $|\mathcal{S}_h| = 20$. The high-level action space comprises eight possible sub-goals $|\mathcal{A}_h| = 8$. Each sub-goal encodes one of the four special locations and whether the passenger should be picked up or dropped off. Since all possible initial configurations can be completed with only two sub-goals, first a pick-up instruction and then a drop-off instruction, the horizon of the high-level MDP is $H_h = 2$.
- (ii) **Low-level MDP:** The low-level agent needs to be able to perform all primitive actions so $\mathcal{A}_l = \mathcal{A}$. Then, the low-level state space encodes the current location of the taxi as well as

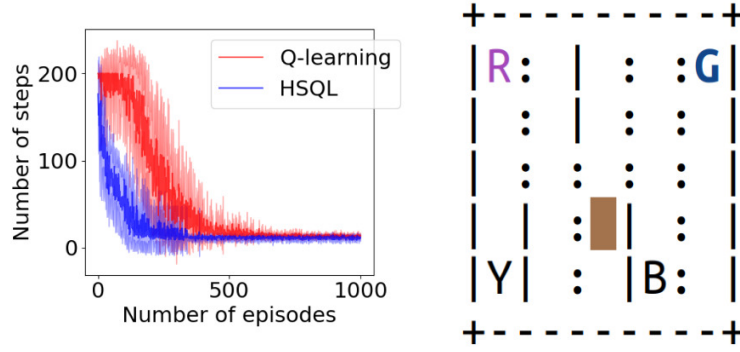


Fig. 3.9 **Left plot** shows the learning curves of SHQL and Q-learning in the taxi environment. **Right plot** illustrates the taxi environment; the agent (yellow square) needs to navigate a grid world to reach the pickup location (encoded by the blue letter) and drop off the passenger at its final destination (encoded as the purple letter). The remaining letters are the possible pickup and drop-off locations. Note that the taxi needs to learn how to navigate the grid world in order to avoid the walls represented by solid lines.

the instructed sub-goal, yielding to a state space of size $|\mathcal{S}_l| = 200$. Since the trip is evenly divided into two separate instructions, the horizon of the low-level agent is $H_l = 100$.

By Theorem 3.3.1, we can observe that the low-level MDP is likely to dominate the bound and to drive the overall complexity of the problem. Ignoring the constant terms in the bound, our theory suggests that small improvement should still be possible. This is confirmed by our empirical results in Fig. 3.9 (left plot), which shows a significantly steeper learning curve for SHQL compared to Q-learning on the taxi task. In order to make the comparison between the two algorithms as fair as possible, we considered a range of possible hyperparameter values and for each algorithm, we only report the best-performing setting. The hyperparameters considered are the initial exploration rate $\epsilon \in [0.1, 0.3, \dots, 0.7, 0.9]$ and the decay rate $\delta \in [1 - 10^{-3}, 1 - 10^{-4}, \dots, 1 - 10^{-7}]$. Note that we only considered the following decay function $\epsilon_{k+1} = \epsilon_k * \delta^{k+1}$, where k denotes the current episode number. The error bars are obtained by running ten different seeds.

3.7 Related work

We consider a system where an agent needs to make a sequence of decisions in an uncertain environment to maximise the sum of cumulated rewards. Such problems are modelled as Markov Decision Processes (MDPs) and can be solved by RL algorithms [130]. When a task is too complex, the number of interactions required to learn a near-optimal policy becomes prohibitive. The task complexity typically depends on the difficulty of temporal credit assignment (which is directly related to the episode length) and the size of the state and action spaces [87]. To address this complexity, HRL leverages temporal abstractions [132] and state abstractions [33] to improve sample efficiency when learning an optimal policy. There exists a wide range of HRL frameworks; see [60] for a survey. In this chapter, we focus on the goal-conditioned HRL framework [94, 21]. Of the other HRL frameworks, only the options framework [132] and the resulting semi-Markov Decision Process [50, 146, 25, 49] benefit from some theoretical understanding. However, in practice, the goal-conditioned hierarchical framework presented in Fig. 3.1 is often preferred. Unlike the options framework, the goal-conditioned HRL framework requires no prior knowledge about the task [60], and its ability to generalize over the goal space when function approximation is used leads to significant performance gains in benchmark tasks [139, 94, 56]. Existing theoretical work [50, 49] on the options framework does not consider the case where all the hierarchy levels are jointly learned. The regret analysis proposed by [146] focuses on the benefit of leveraging repeating sub-structures in hierarchical MDPs but does not provide a lower bound on the regret. Other regret analyses [50, 49] highlight the efficiency gain of learning with temporally extended actions (such as options). However, they always assume the set of options is known, and the intra-option policies are not learned. Options are composed of an intra-option policy, which governs the agent's behaviour while the corresponding option is executed, making intra-option policies very similar to the low-level policy we consider. However, the goal-conditioned HRL setting considered in this chapter quantifies the benefits

of state abstraction, action abstraction, and time abstraction while jointly learning all levels of the hierarchy (low-level and high-level policy) through interaction with the environment. A detailed description of the option framework and its connection to HRL is available in Section 3.7.1.

3.7.1 The Options Framework and Goal-Conditioned HRL

As most of the existing theory on HRL [146, 50, 49] has been developed in the options framework, we recall below the definition of that framework and the connection it has with the goal-condition HRL framework considered in this chapter. An immediate difference is that those previous works consider an infinite horizon setting while we focus on a finite horizon setting. For the remainder of this section, we describe some other significant differences.

Let's first recall that the tuple $M = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ describe a finite MDP, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $p(s'|s, a)$ is the probability of transitioning to the state s' given that in state s action a was executed, $r(s, a)$ is the reward distribution associated with the state action pair (s, a) and, lastly, γ is the discount factor. A policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ maps each state to an action. An option is a tuple $o = \{\mathcal{I}_o, \beta_o, \pi_o\}$, where $\mathcal{I}_o \subset \mathcal{S}$ is the initiation set, i.e. the set of states where the option o can be started, $\beta_o : \mathcal{S} \rightarrow [0, 1]$ is the probability distribution to terminate in option in a given state and finally π_o is the intra-option policy, i.e. the policy followed until the option ends.

Whenever the set of primitive actions $\mathcal{A}$ is replaced by a set of options $\mathcal{O}$ the original MDP M , becomes a semi-MDP $M_{\mathcal{O}} = \langle \mathcal{S}_{\mathcal{O}}, \mathcal{O}, p_{\mathcal{O}}, r_{\mathcal{O}}, \tau_{\mathcal{O}} \rangle$. The state space $\mathcal{S}_{\mathcal{O}} \subseteq \mathcal{S}$ is the set of all initial and terminal states. $\mathcal{O}$ denotes the set of options and $p_{\mathcal{O}}(s', o, s)$ is the probability to end option o in state s' given that it was started in state s :

$$p_{\mathcal{O}}(s', o, s) = \sum_{k=1}^{\infty} P(s_k = s' | s, \pi_o) \beta_o(s'),$$

where $P(s_k = s' | s, \pi_o)$ is the probability to reach state s' in exactly k steps from state s under policy π_o . $r_o(s, o)$ is the distribution of reward cumulated when starting policy π_o in state s . Lastly, τ_o is the distribution of holding time, i.e. the number of primitive steps executed to reach s' from s under policy π_o .

It is possible to map the goal-conditioned HRL framework onto the options framework. Sub-goals would correspond to options, the initiation set of a given option o includes all states where the corresponding sub-goal g_{sub} is available, the option will terminate with probability one if the goal state has been reached or the H_l time steps have been executed and terminate with probability 0 otherwise. Lastly, the intra-option policy π_o corresponds to the low-level policy $v(\cdot | g_{sub})$ conditioned on the corresponding sub-goal.

From a theoretical standpoint, the key difference between the results in [49, 50] and the ones presented in this chapter are that they assume the intra-option policies π_o to be known. If we do not assume strong prior knowledge about the task, the number of potential intra-option policies will be prohibitive. In this chapter, we investigate algorithms that do not make assumptions about the low-level policy and jointly learn the low- and high-level policies through interactions with the environment, thus going beyond what has been done for the options framework.

3.8 Conclusion

This chapter provides, to the best of our knowledge, the first result, which analyses the decomposition induced by goal-conditioned HRL. In particular, our lower bound offers a valuable tool for practitioners that could help them decide whether a hierarchical decomposition of their problem could be beneficial. Additionally, we also designed a novel hierarchical algorithm that can leverage the hierarchy to improve its sample efficiency. We further validate the performance of this algorithm on a set of hierarchical tasks. These experimental results further emphasize the usefulness of the proposed bound since our theoretical findings support efficiency gains observed experimentally.

Chapter 4

An Upper Bound on HRL Regret

The work done in this chapter further investigates the goal-conditioned Hierarchical Reinforcement Learning (HRL) setting, where a high-level agent instructs sub-goals to a low-level agent. Under the additional assumption of a sparse reward function, we can propose a new algorithm to learn optimal hierarchical policies. The proposed algorithm takes a low-level policy as input and is flexible enough to work with a wide range of low-level policies. We show that when the low-level algorithm is optimistic and provably efficient, our HRL algorithm enjoys a regret bound, which represents a significant improvement compared to previous results for HRL. We support our theoretical findings with experiments that underscore that the proposed method consistently outperforms algorithms that ignore the hierarchical structure.

The content of this chapter was submitted¹ to AISTATS as:

Robert, A^{*}., Drappo, G^{*}., Metelli, A., Faisal, A. A., Restelli, M., & Pike-Burke, C. (2025) Efficient Exploitation of the Hierarchical Structure in Sparse Reward Reinforcement Learning.

¹See Section 1.4 for details on the shared contributions.

4.1 Introduction

This work focuses on finding efficient solutions to a family of finite-horizon Markov Decision Processes (MDPs) with a sparse reward function, which only rewards the agent when it successfully reaches the pre-specified goal state. While the sparse reward assumption may appear limiting, it appears in a wide variety of problems. Sparse reward functions are commonly used in practice for tasks like learning robot skills and motion planning, which are commonly goal-based (e.g. Mülling et al. [92], Andrychowicz et al. [6], Qureshi et al. [116]). Indeed, in such scenarios, the learner is only rewarded when the goal is reached. Additionally, such reward functions are easy to design and implement, which is of great practical interest. We refer to this class of MDP as finite-horizon MDP with sparse rewards.

To tackle this problem, we propose a goal-conditioned Hierarchical Reinforcement Learning (HRL) approach (Pateria et al. [106]) that leverages the structured decomposition of the task into a sequence of sub-tasks. This is typically done by considering two policies: a high-level policy instructing sub-goals and a low-level policy aiming to solve each instructed sub-goal. The defined structure naturally induces a decomposition of the original MDP into a low-level MDP and a high-level MDP. Our algorithm directly leverages the known hierarchical decomposition and the sparse reward structure. In particular, it exploits that for the sparse reward setting, the value function of a policy is exactly the probability with which the policy will reach the goal state. Our method uses this property to compute an optimistic approximation of the high-level MDP, which encourages exploration at the sub-goal level. This temporally extended type of exploration is a key feature behind the success of HRL in practice (Nachum et al. [95]). Our algorithm works with a broad range of low-level algorithms. It only requires that the algorithm that computes the low-level policy is optimistic; this is instrumental to the construction of the optimistic high-level MDP.

Our main result is to derive an upper bound on the regret (see Theorem 4.4.2) suffered by the proposed algorithm, whose exact regret will depend on the choice of the low-level

algorithm. If we consider a provably efficient algorithm, such as UCBVI [13], as the low-level algorithm, our regret bound (see Corollary 4.4.3) is an improvement over the regret proposed in Drappo et al. [40] as their regret depends on the learned policy’s coverage, which can be arbitrarily large. However, our analysis is kept general and is valid for any optimistic low-level regret minimiser. Moreover, our upper bound on the frequentist regret provides a stronger theoretical guarantee than the upper bound on the Bayesian regret of Wen et al. [146]. Additionally, we empirically validate our findings in a corridor environment and a maze, where the different rooms present different profiles and, thus, model the different low-level MDPs.

4.2 Related work

Despite a growing literature of provably efficient algorithms for standard RL (also referred to as *monolithic* approaches due to the lack of a hierarchical structure) (e.g. Auer et al. [10], Dann et al. [29], Azar et al. [13]), theoretical guarantees for HRL are still lacking.

Most existing theoretical work focuses on a related but different framework, the *options* framework. Fruit et al. [49, 50] studied the impact of *pre-trained options* (i.e. low-level policies) in Average Reward MDPs [114], showing a benefit in term of more efficient exploration. Fruit et al. [49] proposed an algorithm based on UCRL2 [10] for semi-Markov Decision Processes [132] and compared its upper bound to the regret obtained with UCRL2 with a monolithic approach. Fruit et al. [50] remove the assumption of having particular knowledge of the distributions of cumulative reward and duration of each option. The most important distinction between this work and ours is that we consider a more challenging setting where the high-level and low-level policies (or options) are jointly learned.

Drappo et al. [41, 40] tried to develop approaches that jointly learn the low-level policy (*options*) and the high-level policy. The main challenge when jointly learning both levels of the hierarchy is the non-stationarity experienced by the high level, which makes decisions

based on the execution of a policy which is evolving during the learning. Drappo et al. [41] proposed a naive approach to mitigate this problem. Their ‘explore-then-commit’ algorithm first learns each sub-task policy and then exploits them to learn a higher-level policy in the same fashion as [49]. However, it is well known that Explore-then-Commit (Lattimore et al. [74]) approaches suffer sub-optimal regret. Therefore, the later work of Drappo et al. [40] provided an alternative and more refined solution, which, instead of separating the learning process into two phases, divided it into multiple phases. Then, the learning is alternated between the high-level problem of learning the meta-policy selecting among options and the low-level one consisting of the options’ policies learning. In this way, the authors provide a tighter upper bound. However, this upper bound still suffers from the influence of constants dependent on the covering that each learned policy in the process has over the optimal one. In this work, conversely, we consider a more specific family of problems, goal-based with sparse reward, to show that when there is this particular hierarchical structure, the regret isn’t affected by any external and learning-dependent factor, providing an improved regret guarantee.

The lower bound on the sample efficiency of goal-based HRL presented in Chapter 3 does not necessarily apply to the setting considered in this chapter since we assume the reward is sparse and that we have access to a deterministic function, ψ that maps each possible high-level state and sub-goal pair to the next high-level state in the case of successful completion of the sub-goal. In Chapter 3, the reward function is any bounded reward, and the high-level transition after the successful completion of a sub-goal is not necessarily known (i.e. the map that structures the MDP is unknown and not necessarily deterministic). Interestingly, the lower bound of Chapter 3 highlights that jointly learning both levels of the hierarchy is at least as difficult as the most difficult of the two problems. Since, in our setting, we consider we can compute the high-level dynamics (i.e. we do not need to learn them),

the most complex part of our problem is necessarily learning the low-level dynamics. We provide more details about the connection between the two settings in Section 4.7.1.

4.3 Problem Setting

This section introduces the problem setting, necessary notations and key assumptions. Ultimately, we are interested in solving goal-based finite-horizon MDP; we first recall the definition of this specific framework (Sec. 4.3.1). Then, we define the hierarchical decomposition considered in this work (Sec. 4.3.2), showing how we can reconstruct the original MDP from the decomposition (Sec. 4.3.2). We conclude by stating the learning objective and the assumptions made (Sec. 4.3.3).

4.3.1 Goal-based Finite-Horizon Markov Decision Process with Sparse Reward

A goal-based finite-horizon MDP is a tuple $M_o = \langle \mathcal{S}_o, \mathcal{A}_o, R_o, P_o, H_o, s_{o,1}, g \rangle$. We refer to this MDP as the original MDP, hence we index each of its components with the subscript o . The state space $\mathcal{S}_o$ and action space $\mathcal{A}_o$ are finite. The state space also contains the goal state $g \in \mathcal{S}_o$, which is an absorbing state. The goal state is used to characterise the reward function $R_o : \mathcal{S}_o \times [H] \rightarrow [0, 1]$, which depends on the state and the time step and is equal to 0 except if in the last time step H the current state is a goal state, in which case the reward is equal to 1, that is $R_o(s, h) = \mathbb{I}\{s = g \cap h = H\}$. The transition model $P_o : \mathcal{S}_o \times \mathcal{A}_o \times \mathcal{S}_o \rightarrow [0, 1]$ provides the probability of reaching a specific state s' after executing an action a in state s , $P_o(s'|s, a)$. The horizon H_o determines the number of interactions with the environment the agent can perform within a single episode. Each episode starts from an initial state $s_{o,1} \in \mathcal{S}_o$; then, for the following H_o timesteps, the agent interacts with the MDP by choosing an action a depending on the observed state $s \in \mathcal{S}_o$. The result of this action is immediately observed

as a new state $s' \sim P_o(\cdot|s, a)$ and an immediate reward $R_o(s', h)$, where h denotes the current timestep. After K episodes, the algorithm has interacted with the environment for $T = KH_o$ times.

Reinforcement learning algorithms aim to find policies $\pi : \mathcal{S}_o \rightarrow \mathcal{A}_o$ that maximise the cumulative reward over an episode. We measure this in terms of the value function for any $s \in \mathcal{S}_o$ and $t_o \in [H]$,

$$V_{o,t_o}^\pi(s) = \mathbb{E} \left[\sum_{i=t_o}^{H_o} R_o(s_i, i) \middle| a_i = \pi(s_i), s_{t_o} = s, s_{i+1} \sim P_o(\cdot|s_i, a_i) \right],$$

where the expectation is computed with respect to the stochastic transition function, P_o . Specifically, we evaluate the quality of a policy π with respect to the value function of the initial state $s_{o,1}$, $V_{o,1}^\pi(s_{o,1})$. The aim of RL algorithms is to find an optimal policy π^* within the set of Markov deterministic policies Π , which maximises the value function,

$$\pi^* = \operatorname{argmax}_{\pi \in \Pi} V_{o,1}^\pi(s_{o,1}). \quad (4.1)$$

4.3.2 Hierarchical MDP

We now introduce the concepts of hierarchical decomposition of an MDP, the high- and low-level MDPs and the associated notation. The hierarchical decomposition of an original finite horizon MDP M_o is expressed as $(\{M_{l,i}\}_{i=1}^L, M_h)$, where $\{M_{l,i}\}_{i=1}^L$ is the set of L distinct low-level MDPs and M_h is the high-level MDP. It is known that such a decomposition can always be constructed as discussed in Chapter 3. The algorithm and the theoretical guarantees presented below assume that the decomposition is known. Having an algorithm that learns such a decomposition is of great interest, but it is beyond the scope of this article. We now define the low- and high-level MDPs and then show the connection between the decomposition and the original MDP. Highlighting that executing a hierarchical policy in the decomposed MDP corresponds to executing the corresponding policy in the original MDP.

Low-level MDPs

The low-level MDPs are a set of L unique MDPs $\{M_{l,i}\}_{i=1}^L$, that share the same state space, sub-goal space, transition function and reward function. This definition is similar to the concept of *sub-MDP* introduced in [146]. One low-level MDP is formally defined as $M_{l,i} = \langle \mathcal{S}_{l,i} \cup \mathcal{E}_i, \mathcal{A}_o, R_{l,i}, P_{l,i}, H_l, \rho_{l,i} \rangle$. Here, $\mathcal{S}_{l,i}$ is the low-level state space of the i^{th} low-level MDP. $\mathcal{E}_i$ is the set of reachable sub-goals; these sub-goals are represented as additional artificial absorbing states (note that the goal state g is the only sub-goal that is not artificial). We use the notation $\mathcal{S}_{l,i}^+$ to represent the extended state space, $\mathcal{S}_{l,i}^+ = \mathcal{S}_{l,i} \cup \mathcal{E}_i$. Note that one of the main benefits of hierarchical decomposition is its ability to capture repetitive patterns in the original MDP. In particular, the decomposition is useful if the size of all the low-level state spaces is smaller than the original state space, $\sum_{i=1}^L |\mathcal{S}_{l,i}| < |\mathcal{S}|$. Another benefit of the decomposition is that it breaks down the length of the episodes. A single low-level episode lasts for H_l steps, with the low-level horizon $H_l < H$. Note that the low-level action space consists of the original action space $\mathcal{A}_o$. The low-level reward function is an artificial construction, $R_{l,i} : \mathcal{S}_{l,i}^+ \times \mathcal{E}_i \times [H_l] \rightarrow [0, 1]$, which is defined as,

$$R_{l,i}(s, e, t_l) = \begin{cases} 1, & \text{if } s = e \quad \text{and} \quad t_l = H_l \\ 0, & \text{otherwise,} \end{cases} \quad (4.2)$$

for state $s \in \mathcal{S}_{l,i}^+$ and the goal $e \in \mathcal{E}_i$ at time step $t_l \in [H_l]$. The low-level transition function, $P_{l,i} : \mathcal{S}_{l,i}^+ \times \mathcal{A}_o \rightarrow \mathcal{S}_{l,i}^+$ encodes the dynamics of P_o on the subset of states whose low-level description belong to $\mathcal{S}_{l,i}$. Given two low-level MDP $M_{l,i}$ and $M_{l,j}$, with $j \neq i$, let's consider that $M_{l,i}$ is connected to $M_{l,j}$ via the sub-goal $e \in \mathcal{E}_i$. The probability of transitioning to the artificial sub-goals state e from a connected low-level state $s_l \in \mathcal{S}_{l,i}$ corresponds to the transition probability in the original MDP to go from the state $s \in \mathcal{S}_o$, the original state associated to s_l , to s' , the state corresponding to the initial state in the next MDP $s'_l = \rho_{l,j}(e)$.

So we have $P_{l,i}(e|s_l, a) = P_o(s'|s, a)$ for all $a \in \mathcal{A}_o$. Because those states are absorbing, we define the associated deterministic transition to be $P_{l,i}(e|e, a) = 1$ for any $e \in \mathcal{E}_i, a \in \mathcal{A}_o$ and 0 otherwise. Note that when actions are selected by a low-level policy $\mathbf{v} : \cup_{i=1}^L \mathcal{S}_{l,i} \rightarrow \mathcal{A}_o$ we compactly write the transition function vector as $P_{l,i}^{\mathbf{v}}(\cdot|s_l)$, which contains the probabilities $P_{l,i}(s'_l|s_l, \mathbf{v}(s_l))$ for all $s'_l, s_l \in \mathcal{S}_{l,i}^+$. The learning agent interacts in each low-level MDP for H_l steps using a deterministic policy $\mathbf{v} : \mathcal{S}_{l,i}^+ \rightarrow \mathcal{A} \forall i \in [L]$. Let e be the sub-goal state visited at the end of the previous low-level episode. The initial state in the low-level MDP $M_{l,i}$ is determined by the map $\rho_{l,i}(e) = s_1$, with $s_1 \in \mathcal{S}_{l,i}$. We define the low-level value function of the policy $\mathbf{v}$, in low-level state $s \in \mathcal{S}_{l,i}$, when tasked to solve the sub-goal $e \in \mathcal{E}_i$ in the low-level MDP $M_{l,i}$, $V_{l,i,t_l}^{\mathbf{v}}(s, e)$, at low-level time step $t_l \in \{1, \dots, H_l\}$, as:

$$V_{l,i,t_l}^{\mathbf{v}}(s, e) = R_{l,i}(s, e, t_l) + P_{l,i}^{\mathbf{v}}(\cdot|s)^T V_{l,i,t_l+1}^{\mathbf{v}}(\cdot, e) \quad (4.3)$$

for all low-level MDPs $i \in [L]$, low-level state $s \in \mathcal{S}_{l,i}$ and sub-goal $e \in \mathcal{E}_i$ and with $V_{l,i,H_l}^{\mathbf{v}}(s, \cdot) = 0$ for all $s \in \mathcal{S}_{l,i}^+$. Essentially, the low-level value function consists of the expected total low-level reward. The low-level state-action value function (Q-function for short) is defined as $Q_{l,i,t_l}^{\mathbf{v}}(s, a, e) = R_{l,i}(s, a, e) + \sum_{s' \in \mathcal{S}_{l,i}} P_{l,i}(s'|s, a) V_{l,i,t_l+1}^{\mathbf{v}}(s')$. The optimal low-level policy $\mathbf{v}^*$ is the policy that maximises the value function:

$$\mathbf{v}^*(s, e) = \operatorname{argmax}_{\mathbf{v} \in \mathcal{N}} V_{l,i,1}^{\mathbf{v}}(s, e), \quad (4.4)$$

for all $i \in [L]$, $e \in [\mathcal{E}_i]$, and $s \in \mathcal{S}_{l,i}$ and with $\mathcal{N}$ being the set of deterministic low-level policies.

Recall that the low-level reward function only rewards the agent if it reaches the sub-goal within the H_l steps. In this setting, low-level policies with the highest value function have the highest probability of reaching the sub-goal. The value function of a low-level policy can

be directly related to its probability of reaching the specific sub-goal. For this, we define an event describing a successful sub-goal completion by a policy ν .

Definition 4.3.1. We define $\text{end}_\nu(i, e)$ as the event that the policy ν successfully reached sub-goal e in low-level MDP $M_{l,i}$, that is $\text{end}_\nu(i, e) = \mathbb{I}\{s_{H_l} = e\}$

The low-level value function can then be written as $V_{l,i,1}^\nu(s, e) = \mathbb{P}(\text{end}_\nu(i, e))$, for any low-level MDP $M_{l,i}$ and sub-goal e .

The High-Level MDP

The high-level MDP is defined by the following tuple $M_h = \langle \mathcal{S}_h, \mathcal{A}_h, R_h, P_h^\nu, H_h, s_{h,1}, \phi, \psi \rangle$. The high-level MDP operates at a higher temporal resolution and essentially navigates between low-level MDPs. The hierarchical structure suggests that there is an associated low-level MDP for each high-level state. The relationship between the high-level state $s_h \in \mathcal{S}_h$ and its corresponding low-level MDP $M_{l,i}$ is encoded by the map $\phi : \mathcal{S}_h \rightarrow [L]$. If $\phi(s) = i$, the low-level MDP corresponding to the high-level state $s \in \mathcal{S}_h$ is $M_{l,i}$. The high-level MDP actions are sub-goals that are instructed to the low-level policy. Specifically, the high-level action space in the high-level state s_h comprises the sub-goals of the current low-level MDP $M_{\phi(s_h),l}$, denoted by $\mathcal{E}_{\phi(s_h)}$. The complete high-level action space $\mathcal{A}_h = \cup_{i=1}^L \mathcal{E}_i$ consists of the union of the sub-goals $\mathcal{E}_i$ in the unique sub-MDPs $M_{l,i}$. The hierarchical decomposition also includes a structure map, ψ , that associates to each high-level state and sub-goal pair the next high-level state in case of successful completion of the subgoal $\psi(s_h, e) = s'_h$, for all $s_h \in \mathcal{S}_h, e \in \mathcal{E}_{\phi(s_h)}$. We assume this structure map is known; in the case this map is unknown, it would have a very limited impact on the algorithmic complexity as a single successful completion of every sub-goal would be sufficient to learn the map.

As each high-level state corresponds to a low-level MDP, there are $|\mathcal{S}_h|$ low-level MDPs, with $L \leq |\mathcal{S}_h|$ unique low-level MDPs. The size of the original state space is the sum of the size of all the low-level state spaces $\mathcal{S}_o = \sum_{s \in \mathcal{S}_h} \mathcal{S}_{\phi(s),l}$.

Each high-level episode starts in a unique starting state $s_{h,1}$, which corresponds to the high-level description of the original MDP initial state, $s_{o,1}$, and lasts H_h steps. Since the high-level state abstraction prevents the high-level agent from knowing whether the end goal in the original MDP is reached, we construct a unique, artificial and absorbing high-level state s_h^g . Note that the low-level MDP associated with s_h^g , $M_{l,\phi(s_h^g)}$ is trivial and contains a single absorbing state, which is the goal state, that is, $\{g\} = \mathcal{S}_{l,\phi(s_h^g)}$. Also note that s_h^g is the only high-level state corresponding to the low-level MDP $M_{l,\phi(s_h^g)}$, which is one of the L low-level MDP classes. With this in mind, we define the high-level reward function:

$$R_h(s_h, t_h) = \begin{cases} 1, & \text{if } s_h = s_h^g \text{ and } t_h = H_h \\ 0, & \text{otherwise,} \end{cases} \quad (4.5)$$

for all high-level states $s_h \in \mathcal{S}_h$ and high-level time steps $t_h \in [H_h]$. There is only one high-level goal state. The high-level reward function R_h is also sparse and assigns 0 to all high-level states except for the high-level state associated with the unique low-level MDP that contains the goal state.

The high-level transition function P_h^v encodes the probability of reaching a high-level state $s'_h \in \mathcal{S}_h$ given that in high-level state $s_h \in \mathcal{S}_h$ the low-level policy v was instructed to reach the sub-goal $e \in \mathcal{E}_{\phi(s_h)}$, $P_h^v(s'_h | s_h, e)$. It is generally challenging to learn P_h^v as its dependence on the low-level policy, v , makes it non-stationary.

The high-level policy is defined as $\mu : \mathcal{S}_h \rightarrow \mathcal{A}_h$. When sub-goals are selected by a policy μ and a low-level policy v is used to solve each sub-goal; we compactly write the transition function as $P_h^{(v,\mu)}(s' | s) := P_h^{(v,\mu)}(s' | s, \mu(s))$ for any $s, s' \in \mathcal{S}_h$. We define the high-level value function, $V_{h,t_h}^{(v,\mu)}$, of the policy μ , at high-level time step $t_h \in \{1, \dots, H_h\}$ as:

$$V_{h,t_h}^{(v,\mu)}(s) = R_h(s, t_h) + \sum_{s' \in \mathcal{S}_h} P_h^v(s' | s) V_{h,t_h+1}^{(v,\mu)}(s'), \quad (4.6)$$

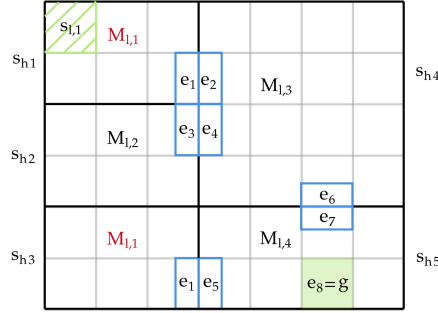


Fig. 4.1 An illustration of a grid world and its associated hierarchical decomposition. The grid world comprises different rooms (depicted with solid black lines); inside each room, the low-level states are illustrated with grey squares. One can observe that the low-level MDP $M_{l,i}$ is used twice (in the top and bottom left rooms). The blue rectangle represents the artificial exit states that connect two low-level MDPs. The overall goal $g \in \mathcal{S}_l$ is illustrated with a green cell in the bottom right room, and it is the only sub-goal state that is not an artificial construction and that belongs to the low-level state space $\mathcal{S}_{l,i}$.

for all $s \in \mathcal{S}_h$ and with $V_{h,H_h}^{(v,\mu)}$. It also depends on the low-level policy v since v selects primitive actions (i.e. in $\mathcal{A}_o$). Finally, $Q_{h,t_h}^{(v,\mu)}(s,a) = R_h(s,t_h) + \sum_{s' \in \mathcal{S}_h} P_h^v(s'|s,a)V_{h,t_h+1}^{(v,\mu)}(s')$ is the state-action value function in the high-level MDP, which corresponds to the value of the hierarchical policy in the original MDP.

Equivalence between the original and the hierarchical MDP

The hierarchical algorithm we consider will learn an optimal policy in the decomposed MDP $(\{M_{l,i}\}_{i=1}^L, M_h)$. This section shows that the optimal hierarchical policy has a corresponding policy in the original MDP M_o . We illustrate this for a small grid world example in Figure 4.1. In what follows, we show the equivalence between each element of the decomposed MDP and the original MDP.

State Space: A state in the original MDP $s_o \in \mathcal{S}_o$ can be factorized into a pair of states $(s_{l,i}, s_h)$ where $s_{l,i} \in \mathcal{S}_{l,i}$ is a state in the i^{th} low-level MDP and $s_h \in \mathcal{S}_h$ is a state in the high-level MDP. Specifically, the low-level state $s_{l,i}$ and the high-level state s_h contains all the necessary information to reconstruct the corresponding state, s_o , in the original MDP.

Action Space: Interactions with the environment are only possible through the original action space $\mathcal{A}_o$; hence the action space of the low-level MDPs is exactly $\mathcal{A}_o$. The high-level action space must encode all the sub-goals the high-level agent wants to instruct; this corresponds to a set of *artificial* low-level states and we assume that this set is given to us by the decomposition. In each high-level state $s \in \mathcal{S}_h$, the agent can instruct the low-level agent to reach one sub-goal $e \in \mathcal{E}_{\phi(s)}$. Then the low-level agent uses actions in $\mathcal{A}_o$ to reach e .

Transition: Most of the complexity of the transition dynamics is encoded in the low-level MDP dynamics $P_{l,i}$. Essentially, the low-level and the original transition functions are almost identical for all the states within a sub-MDP $\mathcal{S}_{l,i}$. The only difference is that the original transition function operates over the full state description $\mathcal{S}_o$. The low-level transition function operates over $\{\mathcal{S}_{l,i}^+\}_{i=1}^L$, which considers only the low-level description of the states and the transition to and from the artificial sub-goal states $\mathcal{E}_i$. To reconstruct the original transition function from the decomposed transition function, we consider two different scenarios.

(i) First, transitions from s_o to s'_o where the low-level description of both states lies in the same low-level MDP $s_l, s'_l \in \mathcal{S}_{l,i}$. Then, the original transition function is equal to the corresponding low-level transition $P_o(s'_o|s_o, a) = P_{l,i}(s'_l|s_l, a)$.

(ii) Second, transitions from s_o to s'_o where the low-level description of the two states do not belong to the same low-level MDP, $s_l \in \mathcal{S}_{l,i}$ and $s'_l \in \mathcal{S}_{j,l}$ for $j \neq i$. We consider all sub-goal states, $e \in \mathcal{E}_i$, supporting such a transition. Formally, we consider the set $Z = \{e : \exists v \in \Pi \text{ s.t. } P_h^v(s'_h|s_h, e) > 0 \text{ and } \rho_{j,l}(e) = s'_l\}$, where $e \in \mathcal{E}_i$ are sub-goals, $s_h, s'_h \in \mathcal{S}_h$ are the high-level state description of the current and the next state and $\rho_{j,l}(e)$ denotes the initial state in low-level MDP $M_{j,l}$ when the low-level comes from the previous sub-goal state $e \in \mathcal{E}_i$. The original transition is the probability of going via any of the sub-goal states in Z ,

$$P_o(s'_o|s_o, a) = \sum_{e \in Z} P_l(e|s_l, a),$$

where $s_l \in \mathcal{S}_{l,i}$ is the low-level state description associated with the state in the original MDP $s_o \in \mathcal{S}_o$.

Reward: There is a direct equivalence between the original reward function and the reward functions in the decomposed MDPs. Since the reward function is the sparse reward that only rewards 1 in the absorbing goal state, it ignores all the bonuses that we grant when the low-level policy reaches a sub-goal $e \in \mathcal{A}_h \setminus g$. Hence, the only reward signal that is preserved is when that agent reaches the goal state g .

Horizon: The two levels of the hierarchy operate at two different time scales, with the low-level episode $t_l \in \{1, \dots, H_l\}$ happening completely within one high-level time step. Hence, $H_o = H_l H_h$. Note that the control returns to the high-level policy exactly after H_l steps; even if the low-level policy reaches the sub-goal in fewer steps, it waits in the absorbing low-level MDP sub-goal state. Each time the high-level recovers the control, it observes the current high-level state s_h and selects the next sub-goal.

4.3.3 Learning in a Hierarchical MDP

The hierarchical decomposition described in section 4.3.2 contains additional information about the structure of the problem, like the structure map ψ and the sub-goal spaces $\{\mathcal{E}_i\}_{i=1}^L$. We assume that this additional information is known to the learner. We now focus on developing an algorithm to solve such hierarchical problems efficiently, assuming that this additional information is known. We evaluate the efficiency of our algorithm using the notion of regret (e.g. Auer et al. [10], Azar et al. [13], Dann et al. [30]).

Regret: The algorithm proceeds in episodes $k = 1, \dots, K$; at the beginning of each episode, the algorithm computes a hierarchical pair of policies $(\nu, \mu)_k$. The regret is defined as the cumulative sum of differences between the value of the optimal pair of hierarchical policies (ν^*, μ^*) and the policy played in each episode. Formally the regret suffered by a hierarchical algorithm $\mathfrak{A}$, which computes in each episode, k , the pair of policies $(\nu, \mu)_k$,

over K episodes is:

$$R_{\mathfrak{A}}((\{M_{l,i}\}_{i=1}^L, M_h), K) := \sum_{k=1}^K V_{h,1}^{(\mathbf{v}^*, \mu^*)}(s_1) - V_{h,1}^{(\mathbf{v}, \mu)^k}(s_1). \quad (4.7)$$

with $s_1 \in \mathcal{S}_h$ the initial state in the high-level MDP.

The goal is to minimize the regret in (4.7). Before formally presenting our algorithm in Section 4.4, we conclude this section by highlighting some important assumptions that we make.

Assumption 1: Every low-level MDP is communicating. This ensures that for every low-level MDP $M_{l,i}$, with $i \in [L]$ and any subgoal $e \in \mathcal{E}_i$, there exists a low-level policy $\mathbf{v}$ able to reach the sub-goal state e within H_l steps.

Assumption 2: The structure map ψ allows the construction of a sequence of sub-goals to reach any high-level state from the initial high-level state.

These assumptions ensure that for any high-level state $s_h \in \mathcal{S}_h$, there exists a sequence of less than H_h sub-goals from the starting high-level state $s_{h,1}$ to s_h and a low-level policy $\mathbf{v}$ to reach every sub-goals. Note also that Assumptions 1 and 2 combined also guarantee that the original MDP M_o is communicating.

Assumption 3: The original MDP M_o is time-homogeneous.

This is a common assumption in RL as it simplifies the setting and the notation. In general, it is reasonable to assume that the user might have access to information about the problem's hierarchical decomposition. Such information could be considered expert knowledge about the problem at hand. In the navigation task presented in Figure 5.1, all the necessary information could be inferred from the building's floor plan.

It is also helpful to consider the regret suffered by the low-level algorithm $\mathfrak{A}$ which in each episode, k , selects the policy $\mathbf{v}_k$ to solve sub-goal $e \in \mathcal{E}_i$ in MDP $M_{l,i}$.

$$R_{\mathfrak{A}}((M_{l,i}, e), K) := \sum_{k=1}^K V_{i,l,1}^{v^*}(s_1, e) - V_{i,l,1}^{v_k}(s_1, e) \quad (4.8)$$

Where $V_{i,l,1}^{v^*}(s_1, e)$ and $V_{i,l,1}^{v_k}(s_1, e)$ are the value functions of the optimal policy, v^* , and the current policy, v_k , when tasked solve sub-goal e in MDP $M_{l,i}$.

4.4 Goal-conditioned Hierarchical Reinforcement Learning Algorithm (GHRL)

We propose a class of HRL algorithms that directly leverage the known hierarchical decomposition to efficiently solve the original MDP. In this section, we first describe the general framework and define the set of low-level algorithms that can be used. We then state the main theorem, which provides a general upper bound on the worst-case regret suffered dependent on the choice of the low-level algorithm. We conclude the section with a corollary bounding the total regret when the low-level algorithm is UCBVI [13] (see Section 2.3.1 for more details).

We present the Goal-conditioned Hierarchical Reinforcement Learning algorithm (GHRL) in Algorithm 9. GHRL takes as input the number of high-level episodes to run, K , the high-level structure map ψ , the high-level reward function R_h , and the choice of the low-level regret minimiser $\mathfrak{A}$. To highlight that a specific low-level algorithm choice $\mathfrak{A}$ was made, we call the resulting hierarchical algorithm $\text{GHRL}_{\mathfrak{A}}$. The theoretical guarantee that we provide expects that $\text{GHRL}_{\mathfrak{A}}$ receives as input a low-level algorithm that belongs to the set optimistic algorithms, $\mathfrak{A} \in \Xi$, where we define an optimistic algorithm as follow.

Definition 4.4.1. We say that an algorithm is optimistic if, in each round k , it computes the optimal policy and its associated value function $\tilde{V}_{l,i,1}^{v_k}$. The computed value function is

optimistic if with probability $1 - \delta$ it upper bounds the value function of the optimal policy on the true MDP: $\tilde{V}_{l,i,1}^{v_k} > V_{l,i,1}^*$.

For example algorithms like UCBVI [13], UCRL2 [10] or Q-UCB [62] belong to Ξ . The optimism of the low-level algorithm is required to ensure sufficient exploration for the high-level policy, which guarantees all low-level MDP sub-goal pairs are visited sufficiently often.

The algorithm maintains a dataset D_k^l containing all the low-level information collected up to the k^{th} episode. Then, at the beginning of each episode, it uses D_k^l to compute a low-level policy, v_k . In each episode k , the optimistic low-level algorithm $\mathfrak{A} \in \Xi$ computes a low-level policy v_k , and an optimistic value function $V_{l,i}^{v_k}$ for each low-level MDP, $M_{l,i}$ and sub-goal $e \in \mathcal{E}_i$ pair.

The algorithm's key innovation is to compute an optimistic approximation of the high-level transition function $P_h^{v_k}$ under the low-level policy v_k . For this, we first recall that the structure map ψ indicates the intended high-level next state s'_h , given that in high-level state s_h , the instructed sub-goal is e , $\psi(s_h, e) = s'_h$. Additionally, in the sparse reward setting, the value function of the low-level policy, $V_{l,i,1}^{v_k}(s, e)$ is equivalent to the probability that policy v_k reaches the sub-goal e while starting in state $s \in \mathcal{S}_{l,i}$. Consequently, the optimistic value function $\tilde{V}_{l,i,1}^{v_k}(s, e)$ upper bounds the probability that the current low-level policy reaches the instructed goal. Then, an optimistic approximation of the high-level transition function can be computed as follows:

$$\tilde{P}_h^{v_k}(s'_h | s_h, e) = \begin{cases} \min(\tilde{V}_{l,i,1}^{v_k}(s_l, e), 1), & \text{if } s'_h = \psi(s_h, e) \\ 0, & \text{otherwise,} \end{cases} \quad (4.9)$$

where $i = \phi(s_h)$ is the low-level MDP associated with the current high-level state. Note that $\tilde{P}_h^{v_k}$ is an approximation of a transition function as it does not necessarily sum to 1, that is, $\sum_{s' \in \mathcal{S}_h} \tilde{P}_h^{v_k}(s' | s, e) \leq 1$.

For each episode k , the algorithm computes the high-level policy μ_k as the optimal policy of the optimistic high-level MDP $\tilde{M}_h = \langle \mathcal{S}_h, \mathcal{A}_h, R_h, \tilde{P}_h^{\nu_k}, H_h, s_{h,1}, \phi, \psi \rangle$, where the high-level transition function $\tilde{P}_h^{\nu}$ is the optimistic approximation of the true transition function P_h^{ν} under the current low-level policy ν . The optimistic high-level Q -function, $\tilde{Q}_h^{(\nu, \mu)_k}$, is computed via backward induction (see Alg. 1) using the optimistic approximate transition function $\tilde{P}_h^{\nu}$ and the high-level reward R_h .

High-level transitions resulting from a failed sub-goal can be ignored since we focus on evaluating the value of high-level *plans*. A high-level plan consists of a sequence of sub-goals selected by the high-level policy μ_k . The whole sequence of sub-goals can be computed at the beginning of each episode as the map ψ allows the construction of the path, and the optimal actions are given by the optimistic high-level Q -function $\tilde{Q}_h^{(\nu, \mu)_k}$.

The current hierarchical policy pair $(\nu, \mu)_k$ is used to collect the low-level samples that are stored in the dataset $\mathcal{D}_l^k$. The high-level policy selects a sub-goal e in the initial state. The low-level policy ν_k is executed in the MDP $M_{l,i}$ with $i = \phi(s_h)$. The low-level agent aims to solve the sub-goals instructed by μ_k in the H_l steps. Whenever a low-level episode ends, either the sub-goal is reached, and the next sub-goal in the high-level plan is instructed, or the sub-goal is not reached. In cases where the sub-goal is not reached, the algorithm stops collecting samples for that episode.

Our main result is an upper bound on the regret suffered by GHRL (alg. 9), which depends on the choice of the low-level algorithm considered.

Theorem 4.4.2. *With probability $1 - \delta$, for any hierarchical decomposition $(\{M_{l,i}\}_{i=1}^L, M_h)$, and for any optimistic low-level regret minimiser $\mathfrak{A} \in \Xi$, the regret of $\text{GHRL}_{\mathfrak{A}}$ is bounded by:*

$$R_{\text{GHRL}}((\{M_{l,i}\}_{i=1}^L, M_h), K) \leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \left(R_{\mathfrak{A}}((M_{l,i}, e), N(i, e)) + 2N(i, e) \mathbb{P}(\neg \text{end}_{\nu^*}(i, e)) \right) \quad (4.10)$$

where $N(i, e)$ is the number of original MDP episodes in which the pair (i, e) has been visited at least once and $R_{\mathfrak{A}}((M_{l,i}, e), N(i, e))$ is the regret suffered by $\mathfrak{A}$ while learning to solve goal e in low-level MDP $M_{l,i}$ for $N(i, e)$ episodes.

The proof of Theorem 4.4.2 follows from combining properties from the sparse reward function and the optimism of the low-level algorithm to upper bound the regret of GHRL by counting the number of failed trajectories. A detailed proof of Theorem 4.4.2 is in Section 4.5.3.

Algorithm 9: Goal-conditioned Hierarchical Reinforcement Learning (GHRL)

Input: K episodes, high-level structure map Ψ , high-level reward function R_h , $\mathfrak{A}$ and failure tolerance $\delta \in (0, 1]$.

```

1 Arbitrarily initialize  $\mu_0$  and  $v_0$ , and  $\mathcal{D}_0^l \leftarrow \{\}$ 
2 for  $k = 1, \dots, K$  do
3    $v_{i,k}, \tilde{V}_{l,i}^{v_k} = \mathfrak{A}(\mathcal{D}_k^l, M_{i,l}, e, \theta) \quad \forall i \in [L], e \in \mathcal{E}_i$ 
4    $\tilde{V}_l^{v_k} = [\tilde{V}_{l,i}^{v_k}, \dots, \tilde{V}_{l,L}^{v_k}]$ 
5    $\tilde{P}_h^{v_k}(\psi(s_h, e)|s_h, e) = \min(\tilde{V}_{l,i,1}^{v_k}(s_l, e), 1) \quad \forall s_h \in \mathcal{S}_h, e \in \mathcal{E}_{\phi(s_h)}$ 
6    $\tilde{P}_h^{v_k} = \Psi \otimes \tilde{V}_l^{v_k}$ 
7    $\tilde{Q}_k^{\mu,h} = \text{BackwardInduction}(\mathcal{S}_h, \mathcal{A}_h, R_h, \tilde{P}_h^{v_k})$ 
8    $\mu_k(s_h) = \operatorname{argmax}_{a_h} \tilde{Q}_k^{\mu,h}(s_h, a_h, t_h) \quad \forall s_h \in \mathcal{S}_h$ 
9    $s = s_{o,1} = (s_{i,l,1}, s_{h,1})$  with  $i = \phi(s_{h,1})$ 
10   $s_h = s_{h,1}, s_{i,l} = s_{i,l,1}$ 
11  for  $t_h = 1, \dots, H_h$  do
12     $e = \mu_k(s_h)$ 
13    for  $t_l = 1, \dots, H_l$  do
14      play action  $a = v_k(s_l, e)$ , in the MDP  $M_{i,l}$  with  $i = \phi(s_h)$ 
15      observe  $s'_{i,l}$  and update  $\mathcal{D}^L \leftarrow \mathcal{D}^L \cup \{(s_{i,l}, a, s'_{i,l})\}$ 
16       $s_{i,l} = s'_{i,l}$ 
17    if  $s_{i,l} \neq e$  then
18       $s_h = s_{h,1}, s_{i,l} = s_{i,l,1}$ 
19    else
20       $s_h = \psi(s_h, e)$ 

```

Output: with probability $1 - \delta$ an optimal hierarchical policy.

It is important to note that the algorithm incurs regret in the low-level MDPs selected by the high-level policy. The high-level policy's exploration strategy is guided by the value of the sub-goal and the uncertainty of the low-level. This exploration strategy is achieved by constructing an optimistic high-level MDP. In Section 4.5.2 we guarantee that the explorative behaviour of the high-level policy is guided by the optimism of the low-level policy, which

ensures that the high-level policy explores low-level MDPs for which it has not collected enough samples.

GHRL's theoretical and computational guarantees are tightly linked to $\mathfrak{A}$'s guarantees. Additionally, this regret bound highlights the impact of the hierarchical structure $(\{M_{l,i}\}_{i=1}^L, M_h)$ on the algorithm's performance. The regret consists of the sum of the regret suffered by the low-level policy at each possible low-level MDP sub-goal pair. Importantly, Theorem 4.4.2 suggests that the performance improves in problems with highly frequent patterns (i.e., $L \ll S_h$).

We obtain the following upper bound if we choose UCBVI [13] as the low-level algorithm.

Corollary 4.4.3. *With probability $1 - \delta$ it holds that for any hierarchical decomposition $(\{M_{l,i}\}_{i=1}^L, M_h)$ the regret suffered by $GHRL_{UCBVI}$ for K high-level episodes is bounded by:*

$$R_{GHRL}((\{M_{l,i}\}_{i=1}^L, M_h), K) \leq \sqrt{A_o A_h \max_i \{S_{l,i}^+\} H_l K} + \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 2N(i, e) \mathbb{P}(\neg \text{end}_{v^*}(i, e)) \quad (4.11)$$

A proof of Corollary 4.4.3 is in Section 4.5.4

We stated in section 4.3.2 that given a decomposition $(\{M_{l,i}\}_{i=1}^L, M_h)$ we are able to reconstruct the original MDP problem M_o . Suggesting that the learned hierarchical policy (v, μ) could be used to solve the original MDP. However, note that in the regret formulated in 4.4.2 we compare the performance of the learned hierarchical policy to the best hierarchical policy pair (v, μ) in the high-level MDP and not with the optimal policy in the original MDP.

4.5 Upper Bounds on the Regret of GHRL

This section provides the proofs for Theorem 4.4.2 and Corollary 4.4.3. We first define useful concepts in the Section 4.5.1. The proofs rely on a particular notion of optimism of the high-level MDP we clarify this notion in Section 4.5.2.

4.5.1 High-level Plans and Failure Events

First of all, let's define the high-level plan. At the beginning of each episode, k , the high-level computes the exact trajectory it would like to complete. We call such a trajectory the high-level plan Plan_k .

Definition 4.5.1. A plan, Plan_k , is the sequence of high-level states the high-level policy μ_k would like to follow in episode k , in accordance with the map ψ . It consists of the pairs of high-level states encoding the expected trajectory:

$$[s_1, s_2, s_3, \dots, s_{n-1}, s_n] = \text{Plan}_k \quad (4.12)$$

describes a plan consisting of n high-level states, where $s_i \in \mathcal{S}_h$ for all $i \in [n]$, $s_{i+1} = \psi(s_i, \mu_k(s_i))$, and $n \leq H_h$. Note that given all the samples collected up to episode k , the plan, Plan_k , is deterministic.

We now state a proposition that helps to relate the regret of the low-level learning problem with the regret of the high-level learning problem.

The low-level MDP reward is a denser version of the original reward. It is enriched with a value of 1 given when the agent reaches the sub-goal state. This reward model allows us to state the following proposition:

Definition 4.5.2. Consider the low-level MDP $M_{l,i}$ with a reward that returns one when the agent reaches the sub-goal state. For this reward model, the value function $V_{l,i,1}^{v_k}$ of the policy v_k , conditioned on the sub-goal e , relates to the probability of successful completion of the sub-goal (see definition 4.3.1):

$$\mathbb{E}[\mathbb{I}\{\text{end}_{v_k}(i, e)\}] = \mathbb{P}(\text{end}_{v_k}(i, e)) = V_{l,i,1}^{v_k}(s_1, e) \quad s_1 \in \mathcal{S}_{l,i}, \text{ and, } s_1 = \rho_{l,i}(e). \quad (4.13)$$

This proposition is valid for any low-level policy and, in particular, for the optimal low-level policy v^* :

$$\mathbb{E} [\mathbb{I}\{\text{end}_{v^*}(i, e)\}] = \mathbb{P}(\text{end}_{v^*}(i, e)) = V_{l,i,1}^{v^*}(s_1, e) \quad s_1 \in \mathcal{S}_{l,i}, \text{ and, } s_1 = \rho_{l,i}(e). \quad (4.14)$$

4.5.2 Optimism

This section presents the high-level optimism necessary to prove Theorem 4.4.2. In particular, this section shows that the optimistic low-level regret minimised propagates optimism to the high-level.

Since our algorithm only considers trajectories that follow the instructed high-level plan and stops collecting samples after the first missed sub-goal, it has an inherent sub-optimality gap as the optimal hierarchical policy (v^*, μ^*) does not have this limitation. The following lemma guarantees that the plan computed by the high-level policy is optimistic up to this specific sub-optimality gap; that is, the value of the plan in the optimistic MDP plus the sub-optimality gap is greater or equal to the value of the optimal policy pair in the true high-level MDP.

Lemma 4.5.3. *Let (v^*, μ^*) be the optimal policies on M_h , and $(v, \mu)_k$ be the policy pair compute by Algorithm 9 at the beginning of any episode k . Given optimistic low-level value functions, we guarantee that the high-level value function constructed in our algorithm and the sub-optimality gap upper bound the value function of the optimal hierarchical policy:*

$$\tilde{V}_{h,t_h}^{(v,\mu)_k}(s) + \sum_{s' \in \text{Plan}_k[t_h:H_h]} \mathbb{P}(\neg \text{end}_{v^*}(\phi(s'), \mu_k(s'))) \geq V_{h,t_h}^{(v^*, \mu^*)}(s) \quad \forall s \in \mathcal{S}_h. \quad (4.15)$$

where $\mathbb{P}(\neg \text{end}_{v^*}(\phi(s'), \mu_k(s')))$ is the probability for the optimal low-level policy to fail to reach sub-goal $e \in \mathcal{E}_{\phi(s)}$ in state $s \in \mathcal{S}_h$ (see Eq. 4.14).

The sub-optimality gap consists of this failure probability for the remaining states of the plan computed by the high-level policy. At time step t_h , we denote the remaining part of the plan, Plan_k , using vector notation $\text{Plan}_k[t_h : H_h]$.

Proof. We need the following observations to fully understand the construction of the optimistic high-level MDP.

First, we note that the regret minimiser $\mathfrak{A}$ used by $\text{GHRL}_{\mathfrak{A}}$ computes at every episode k an optimistic value function for every low-level MDP $M_{l,i}$ and sub-goal $e \in \mathcal{E}_i$ pair.

Second, we notice that in this goal-conditioned setting, with sparse reward, the low-level value function $V_{l,i,1}^v(s, e)$ of a policy v for any state $s \in \mathcal{S}_{l,i}$ and sub-goal $e \in \mathcal{E}_i$ is equal to the probability that the policy v reaches the goal e while starting the low-level episode in state s , $V_{l,i,1}^v(s, e) = \mathbb{P}(\text{end}_v(i, e))$, as explained in Proposition 4.5.2. From the high-level perspective, this corresponds to the probability under the low-level policy v to transition to the instructed next high-level state $\psi(s, e)$, that is, $P_h^v(\psi(s, e)|s, e) = V_{l,\phi(s),1}^v(s, e)$. Combining these observations with the map ψ , we obtain an approximate optimistic high-level transition function $\tilde{P}_h^v$ presented in Equation (4.9).

Now, let's define the optimistic high-level Q -function as the expected return the pair of policies $(v, \mu)_k$ can collect after executing sub-goal e from high-level state s ,

$$\tilde{Q}_{h,t_h}^{(v,\mu)_k}(s, e) = R_h(s, t_h) + \sum_{s' \in \mathcal{S}_h} \tilde{P}_h^{v_k}(s'|s, e) \tilde{V}_{h,t_h}^{(v,\mu)_k}(s'). \quad (4.16)$$

We will use this in our proof.

We note that Lemma 4.5.3 trivially holds for the last step of the high-level MDP H_h as

$$\tilde{V}_{h,H_h}^{(v,\mu)_k}(s) = V_{h,H_h}^{(v^*, \mu^*)}(s) \quad \forall s \in \mathcal{S}_h.$$

Let's now consider any step $t_h < H_h$ and prove that the lemma still holds. We proceed with a proof by induction. We use the inductive hypothesis that the lemma holds for the step

$t_h + 1$, that is,

$$\tilde{V}_{h,t_h+1}^{(v,\mu)_k}(s) + \sum_{s' \in \text{Plan}_k[t_h+1:H_h]} \mathbb{P}(-\text{end}_{v^*}(\phi(s'), \mu(s'))) \geq V_{h,t_h+1}^{(v^*,\mu^*)}(s). \quad (4.17)$$

with $s \in \mathcal{S}_h$ being the high-level state in step $t_h + 1$ of Plan_k .

We now show that the equation holds for time step t_h for an arbitrary state $s \in \mathcal{S}_h$ and sub-goal $e \in \mathcal{E}_{\phi(s)}$. Note that the plan, Plan_k , computed by μ_k from state $\psi(s, e)$ is the sequence of high-level states with the largest probability to reach the goal state, according to the high-level transition $\tilde{P}_h^{v_k}$.

We write the gap between the optimistic high-level Q-function $\tilde{Q}_{h,t_h}^{(v,\mu)_k}$ and the optimal high-level Q-function $Q_{h,t_h}^{(v^*,\mu^*)}$, denoting the instructed next high-level state with $s_\psi = \psi(s, e)$ to lighten the notation.

$$\tilde{Q}_{h,t_h}^{(\mathbf{v},\mu)_k}(s,e) - Q_{h,t_h}^{(\mathbf{v}^*,\mu^*)}(s,e) \quad (4.18)$$

$$= R_h(s, t_h) - R_h(s, t_h) + \sum_{s' \in \mathcal{S}_h} \left(\tilde{P}_h^{\mathbf{v}_k}(s'|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s') - P_h^*(s'|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s') \right) \quad (4.19)$$

$$= \sum_{s' \in \mathcal{S}_h} \left(\tilde{P}_h^{\mathbf{v}_k}(s'|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s') - P_h^*(s'|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s') \right) \quad (4.20)$$

$$= \tilde{P}_h^{\mathbf{v}_k}(s_\psi|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s_\psi) - P_h^*(s_\psi|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) + \sum_{s' \in \mathcal{S}_h \setminus \{s_\psi\}} \left(\tilde{P}_h^{\mathbf{v}_k}(s'|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s') - P_h^*(s'|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s') \right) \quad (4.21)$$

$$= \tilde{P}_h^{\mathbf{v}_k}(s_\psi|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s_\psi) - P_h^*(s_\psi|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) - \sum_{s' \in \mathcal{S}_h \setminus \{s_\psi\}} P_h^*(s'|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s') \quad (4.22)$$

$$\geq \tilde{P}_h^{\mathbf{v}_k}(s_\psi|s, e) \tilde{V}_{h,t_h+1}^{(\mathbf{v},\mu)_k}(s_\psi) - P_h^*(s_\psi|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) - \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s, e)) \quad (4.23)$$

$$\geq (\tilde{P}_h^{\mathbf{v}_k} - P_h^*)(s_\psi|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) - \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s, e)) - \tilde{P}_h^{\mathbf{v}_k}(s_\psi|s, e) \sum_{s' \in \text{Plan}_k[t_h+1:H_h]} \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s', \mu_k(s'))) \quad (4.24)$$

$$\geq (\tilde{P}_h^{\mathbf{v}_k} - P_h^*)(s_\psi|s, e) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) - \sum_{s' \in \text{Plan}_k[t_h:H_h]} \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s', \mu_k(s'))) \quad (4.25)$$

$$= (\tilde{V}_{l,\phi(s),1}^{\mathbf{v}_k}(s_\psi) - V_{l,\phi(s),1}^*(s_\psi)) V_{h,t_h+1}^{(\mathbf{v}^*,\mu^*)}(s_\psi) - \sum_{s' \in \text{Plan}_k[t_h:H_h]} \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s', \mu_k(s'))) \quad (4.26)$$

$$\geq - \sum_{s' \in \text{Plan}_k[t_h:H_h]} \mathbb{P}(\neg \text{end}_{\mathbf{v}^*}(s', \mu_k(s'))) \quad (4.27)$$

Equation (4.19) is simply the definition of the optimistic Q -function and the optimal Q -function. Equation (4.20) follows from the fact that the two high-level rewards are identical. Equation 4.21 divides the summation over all possible next states in two; the first term considers the next state defined by the map $\psi(s, e)$ and the other term considers all the other

possible next states. Equation 4.22 holds since $\tilde{P}_h^{V_k}$ is zero for all the potential next high-level states not supported by the map $s' \neq \psi(s, e)$ (as defined in Eq. 4.9). Equation 4.23 uses the fact the $V_{h,t_h+1}^{(V^*, \mu^*)}(s)$ can be upper bounded by one for all $s \in \mathcal{S}_h$ and the sum of the probabilities to reach any other states than s_ψ , is the failure event $\mathbb{P}(\neg \text{end}_{V^*}(s, e))$. Equation (4.24) follows from the induction hypothesis defined in Equation 4.17, note that the induction hypothesis is applied to $\tilde{P}_h^{V_k}(s_\psi|s, e)\tilde{V}_{h,t_h+1}^{(V, \mu)_k}(s_\psi)$. Equation (4.25) upper bounds $\tilde{P}_h^{V_k}(s_\psi|s, e)$ with 1. As the failure event $\mathbb{P}(\neg \text{end}_{V^*}(s, e))$ consists of failing the the t_h^{th} step of the plan, $s = \text{Plan}_k[t_h]$ and $\psi(s, e) = \text{Plan}_k[t_h + 1]$. We then extend the sum over the sub-plan $\text{Plan}_k[t_h : H_h]$ to account for this additional failure event. Equation (4.26) follows directly from the equivalence between the high-level transition function and the low-level value function (see Eq. 4.9). Finally, Equation (4.27) is obtained by the optimistic guarantee of the value function computed by the low-level algorithm $\mathfrak{A} \in \Xi$ and because the optimal value function is necessarily positive, $V^{(V^*, \mu^*)} \geq 0$. $\square$

4.5.3 Proof of Theorem 4.4.2

We first prove Theorem 4.4.2 that we restate below for clarity.

Theorem. *With probability $1 - \delta$, for any hierarchical decomposition $(\{M_{l,i}\}_{i=1}^L, M_h)$, and for any optimistic low-level regret minimiser $\mathfrak{A} \in \Xi$, the regret of $\text{GHRL}_{\mathfrak{A}}$ is bounded by:*

$$R_{\text{GHRL}}((\{M_{l,i}\}_{i=1}^L, M_h), K) \leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \left(R_{\mathfrak{A}}((M_{l,i}, e), N(i, e)) + 2N(i, e)\mathbb{P}(\neg \text{end}_{V^*}(i, e)) \right)$$

where $N(i, e)$ is the number of original MDP episodes in which the pair (i, e) has been visited at least once and $R_{\mathfrak{A}}((M_{l,i}, e), N(i, e))$ is the regret suffered by $\mathfrak{A}$ while learning to solve goal e in low-level MDP $M_{l,i}$ for $N(i, e)$ episodes.

Proof. The total regret suffered consists of the regret suffered while learning policies to solve every sub-goal of each low-level MDP. Let's first recall how the GHRL (algorithm 9) works.

Every episode k , GHRL updates the low-level policy with an optimistic regret minimiser $\mathfrak{A}$ using the samples collected in the low-level MDPs. Note that the collected samples are modified with an artificial reward, i.e. the low-level reward function R_l , which gives an additional reward of 1 when the agent fulfils the sub-goal. Because the low-level reward function is sparse and only rewards the agent when it completes the low-level episode in the sub-goal state, the low-level optimistic value function (computed by $\mathfrak{A}$) can be used to approximate the transition function of the high-level MDP $\tilde{P}_h^{v_k}$ (defined in Eq. 4.9). Equipped with this approximation of the high-level transition function and the known high-level reward function, we compute the high-level policy, μ_k , using backward induction in the high-level MDP. The algorithm collects new samples using v_k to interact with the environment when trying to complete sub-goals selected by μ_k . Note that each time the low-level policy fails to achieve its sub-goal, the algorithm stops collecting samples until the end of the episode.

The regret of the algorithm GHRL, on the hierarchical structure $(\{M_{l,i}\}_{i \in [L]}, M_h)$, is defined as the sum over the K episode of the performance difference between the value function of the optimal hierarchical policy (v^*, μ^*) and the learned policies at time step k , $(v, \mu)_k$:

$$R_{\text{GHRL}}((\{M_{l,i}\}_{i \in [L]}, M_h), K) = \sum_{k=1}^K V_{h,1}^{(v^*, \mu^*)}(s_1) - V_{h,1}^{(v, \mu)_k}(s_1) \quad (4.28)$$

$$\leq \underbrace{\sum_{k=1}^K \tilde{V}_{h,1}^{(v, \mu)_k}(s) - V_{h,1}^{(v, \mu)_k}(s)}_{(a)} + \underbrace{\sum_{k=1}^K \sum_{s \in \text{Plan}_k} \mathbb{P}(\neg \text{end}_{v^*}(s, \mu(s)))}_{(b)} \quad (4.29)$$

Equation (4.29) is obtained through the optimism guarantee of Lemma 4.5.3 and decomposes the regret into two terms (a) and (b) .

Let's first analyse the (a) term:

$$(a) \leq \sum_{k=1}^K \sum_{j=1}^{H_h} \left[(R_h(s_j^k, j) - R_h(s_j^k, j)) + \sum_{s' \in \mathcal{S}_h} (\tilde{P}_h^{v_k}(s'|s_j^k, \mu_k(s_j^k)) - P_h^{v_k}(s'|s_j^k, \mu_k(s_j^k))) \tilde{V}_{h,j+1}^{(v, \mu)_k}(s') \right] \quad (4.30)$$

$$\leq \sum_{k=1}^K \sum_{s_j^k \in \text{Plan}_k} \left[\left(\tilde{P}_h^{v_k}(s_{j+1}^k | s_j^k, \mu_k(s_j^k)) - P_h^{v_k}(s_{j+1}^k | s_j^k, \mu_k(s_j^k)) \right) \tilde{V}_{h,j+1}^{(v, \mu)_k}(s_{j+1}^k) \right] \quad (4.31)$$

$$\leq \sum_{k=1}^K \sum_{s_j^k \in \text{Plan}_k} \left[\left(\tilde{P}_h^{v_k}(s_{j+1}^k | s_j^k, \mu_k(s_j^k)) - P_h^{v_k}(s_{j+1}^k | s_j^k, \mu_k(s_j^k)) \right) \right] \quad (4.32)$$

$$\leq \sum_{k=1}^K \sum_{s_j^k \in \text{Plan}_k} (1 - P_h^{v_k}(s_{j+1}^k | s_j^k, \mu_k(s_j^k))) \quad (4.33)$$

$$= \sum_{k=1}^K \sum_{s_j^k \in \text{Plan}_k} \mathbb{P}(\neg \text{end}_{v_k}(\phi(s_j^k), \mu_k(s_j^k))) \quad (4.34)$$

$$= \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \sum_{n=1}^{N(i,e)} \mathbb{P}(\neg \text{end}_{v_n}(i, e)) \quad (4.35)$$

$$(4.36)$$

Equation 4.30 uses the performance difference lemma (see Lemma E.15 of Dann et al. [30]). Then, Equation 4.31 reformulates the sum over the entire episode of the difference of the transition model by the sum over a defined trajectory. This trajectory corresponds to the high-level plan, Plan_k , which is deterministically constructed with the policy μ_k and the map ψ (see Definition 4.5.1). The state s_{j+1}^k corresponds to the next state considered in the plan, Plan_k . Then considering only the steps anticipated by Plan_k is possible since the optimistic high-level transition 0 for transitions to a high-level state do not correspond to the instructed subgoal, that is, $\tilde{P}_h^{v_k}(s'|s, e) = 0$ for all $s' \neq \psi(s, e)$. Since Equation 4.31 is an upper bound, we include the regret suffered in each step of the plan, ignoring the possibility that the low-level policy v_k might fail to reach a sub-goal and to pursue the high-level plan. We obtain Equation (4.32) by upper bounding the optimistic value function $\tilde{V}_h^{(v, \mu)_k}$ with 1,

this yields a valid upper bound as the optimism property of the low-level algorithm guarantee $\tilde{P}_h^{v_k} \geq P_h^{v_k}$ with high-probability. Then in Equation (4.33) we upper bound the optimistic transition function $\tilde{P}_h^{v_k}$ with 1. Finally, we rewrite Equation (4.33) in terms of the probability that the current low-level policy v_k does not reach the instructed sub-goal $\mu_k(s_j)$ to obtain Equation (4.34). Equation (4.35) is obtained by taking the sum over $N(i, e)$ instead of K , with $N(i, e)$ being a counter of the number of original MDP episodes in which the pair (i, e) has been visited at least once. This allows us to analyse the pairs (i, e) separately, explicitly indicating the regret suffered in the single problem using the fact that the sum of failing probability can be related to the low-level regret.

To bound $\sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \sum_{n=1}^{N(i, e)} \mathbb{P}(\neg \text{end}_{v_n}(i, e))$, we use the following relationship between the low-level regret of an optimistic algorithm and its failure probability. The regret suffered by any optimistic algorithm $\mathfrak{A} \in \Xi$ to learn an optimal policy to solve a sub-goal e in a low-level MDP $M_{l,i}$:

$$\begin{aligned} R_{\mathfrak{A}}((M_{l,i}, e), K) &= \sum_{k=1}^K V_{l,i,1}^*(s_1, e) - V_{l,i,1}^{v_k}(s_1, e) \\ &= \sum_{k=1}^K V_{l,i,1}^*(s_1, e) - 1 + 1 - V_{l,i,1}^{v_k}(s_1, e) \\ &= \sum_{k=1}^K \mathbb{P}(\text{end}_{v^*}(i, e)) - 1 + \mathbb{P}(\neg \text{end}_{v_k}(i, e)), \end{aligned} \quad (4.37)$$

where we use the notation $(M_{l,i}, e)$ to explicitly mention that we compute the regret suffered on MDP $M_{l,i}$ while trying to solve the sub-goal e .

Equation 4.37 directly relates the expected probability of failing of a policy v_k to the regret paid to learn the sub-goal e . Rearranging, we obtain:

$$\sum_{k=1}^K \mathbb{P}(\neg \text{end}_{v_k}(i, e)) = R_{\mathfrak{A}}((M_{l,i}, e), K) + K \mathbb{P}(\neg \text{end}_{v^*}(i, e)). \quad (4.38)$$

Hence, we use this in (4.35) to show that,

$$(a) \leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \sum_{n=1}^{N(i,e)} \mathbb{P}(\neg \text{end}_{v_n}(i, e)) \quad (4.39)$$

$$\leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \left(R_{\mathfrak{A}}((M_{l,i}, e), N(i, e)) + N(i, e) \mathbb{P}(\neg \text{end}_{v^*}(i, e)) \right). \quad (4.40)$$

It is important to notice that the algorithm $\mathfrak{A}$ can be any optimistic algorithm in Ξ . Optimism is required to guarantee that the high-level policy considers the uncertainty of the low-level transition model while planning; refer to section 4.5.2 for further details.

Let's now focus on analysing the term (b). With similar algebraic modifications, we can relate it to:

$$(b) = \sum_{k=1}^K \sum_{s \in \text{Plan}_k} \mathbb{P}(\neg \text{end}_{v^*}(s, \mu(s))) \quad (4.41)$$

$$= \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \sum_{n=1}^{N(i,e)} \mathbb{P}(\neg \text{end}_{v^*}(i, e)) \quad (4.42)$$

$$= \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} N(i, e) \mathbb{P}(\neg \text{end}_{v^*}(i, e)) \quad (4.43)$$

In equation 4.42, we sum over all possible low-level MDP, sub-goal pair (i, e) and the corresponding number of visits $N(i, e)$. As the probability that the optimal policy v^* fails is independent of the number of visits, we directly obtain Equation 4.43.

Then, by summing (a) and (b), we see that the regret can be upper bounded by:

$$R(\text{GHRL}, (\{M_{l,i}\}_{i \in [L]}, M_h), K) \leq (a) + (b) \quad (4.44)$$

$$\leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \left(R_{\mathfrak{A}}((M_{l,i}, e), N(i, e)) + 2N(i, e) \mathbb{P}(\neg \text{end}_{v^*}(i, e)) \right). \quad (4.45)$$

Lastly, it is important to observe that the above theorem only holds if the optimism holds for every low-level MDP, sub-goal pair (i, e) , and episode k , which is formally defined as the

following event:

$$A = \cap_{k=1}^K \cap_{i=1}^L \cap_{e \in \mathcal{E}_i} \mathbb{I}\{\tilde{V}_{l,i,1}^{V_k}(s, e) \geq V_{l,i,1}^*(s, e)\} \quad (4.46)$$

where the low-level optimism, i.e. $\mathbb{I}\{\tilde{V}_{l,i,1}^{V_k}(s, e) \geq V_{l,i,1}^*(s, e)\}$, is guarantee to hold with probability $1 - \beta$.

Theorem 4.4.2 requires that the event A holds with high probability, $\mathbb{P}(A) > 1 - \delta$. To ensure this, we need to set the low-level optimism failure probability to $\beta = \frac{\delta}{A_h K}$. This can be obtained by observing that we need to guarantee $\mathbb{P}(A^c) \leq \delta$. Where A^c is the complement of A :

$$A^c = \cup_{k=1}^K \cup_{i=1}^L \cup_{e \in \mathcal{E}_i} \mathbb{I}\{\tilde{V}_{l,i,1}^{V_k}(s, e) < V_{l,i,1}^*(s, e)\} \quad (4.47)$$

which implies that,

$$\delta \leq \sum_{k=1}^K \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \beta = K A_h \beta \quad (\text{where we recall that } \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 1 = A_h)$$

which implies that $\beta = \frac{\delta}{K A_h}$ and concludes the proof. $\square$

4.5.4 Proof of Corollary 4.4.3

We now focus on proving Corollary 4.4.3. For clarity, we first restate the result.

Corollary. *With probability $1 - \delta$ it holds that for any hierarchical decomposition $(\{M_{l,i}\}_{i=1}^L, M_h)$ the regret suffered by $\text{GHRL}_{\text{UCBVI}}$ for K high-level episodes is bounded by:*

$$R_{\text{GHRL}}((\{M_{l,i}\}_{i=1}^L, M_h), K) \leq \sqrt{A_o A_h \max_i \{S_{l,i}^+\} H_l K} + \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 2N(i, e) \mathbb{P}(\neg \text{end}_{V^*}(i, e)) \quad (4.48)$$

Note that the additional term in the regret (Eq. 4.48) is inherited by the high-level policy's inability to adapt to the low-level algorithm's failures. In particular, if the failure probability

of the optimal low-level policy $\mathbf{v}^*$ is zero. That is $\mathbb{P}(\text{end}_{\mathbf{v}^*}(i, e)) = 0 \forall i \in [L]$ and $\forall e \in \mathcal{E}_i$ then the sub-optimality gap disappear.

Proof. Let's now consider the first term in Theorem 4.4.2 (i.e. Equation (4.10)) with UCBVI as a low-level regret minimiser.

$$\begin{aligned} \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} R_{\text{UCBVI}}((M_{l,i}, e), N(i, e)) &\leq \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} \sqrt{A_o H_l S_{l,i} N(i, e)} \\ &\leq \sqrt{A_o H_l} \sqrt{\sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 1} \sqrt{\sum_{i=1}^L \sum_{e \in \mathcal{E}_i} S_{l,i} N(i, e)} \quad (4.49) \end{aligned}$$

$$\leq \sqrt{A_o H_l A_h \max_i \{S_{l,i}\} K}, \quad (4.50)$$

Equation (4.49) follows by applying the Cauchy-Schwarz inequality, while the last inequality follows from:

$$\sum_{i=1}^L \sum_{e \in \mathcal{E}_i} S_{l,i} N(i, e) \leq \max_i \{S_{l,i}\} \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} N(i, e) = \max_i \{S_{l,i}^+\} K. \quad (4.51)$$

Where $\mathcal{A}_h$ is the set of all possible sub-goals the high-level can instruct in all low-level MDPs, that is $\sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 1 = |\mathcal{A}_h|$. Because of the sparse reward setting, the infinity norm of the value function is bounded by one, $\|V^k\|_\infty \leq 1$.

□

4.6 Experiments

As previously explained, the GHRL framework presented in Algorithm 9 is flexible enough to accommodate a wide range of regret minimiser choices for the low-level algorithm.

To empirically validate our framework, we consider the Q-UCB algorithm [62] (see Section 2.3.4 for more details) to compute the low-level policy. We denote the resulting hierarchical algorithm as $\text{GHRL}_{\text{Q-UCB}}$ in that context. We propose to evaluate $\text{GHRL}_{\text{Q-UCB}}$

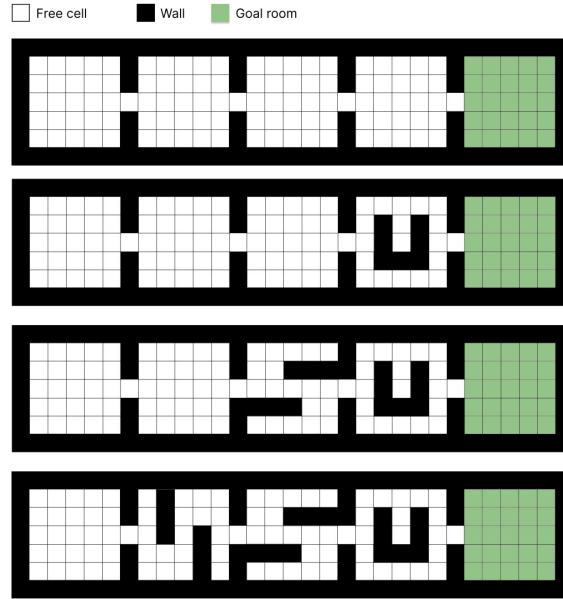


Fig. 4.2 An illustration of the four path-like MDPs considered. All path MDPs shown in this figure are composed of five rooms. The plot at the top shows an MDP with identical rooms. In each row, we change the room profile of a room; that is, we add a unique obstacle inside the room. We repeat this until all rooms have a unique room profile, as shown in the bottom plot.

on a path-like grid world composed of several rooms and compare its performance to the standard Q-UCB, which does not leverage the hierarchical properties of the environment. An illustration of the environment considered is available in Figure 4.2.

Like the N -rooms environment presented in Chapter 4, the path-like grid-world comprises multiple rooms arranged to form a path. The agent’s goal is to go through the entire path from the leftmost room to the rightmost room. The agent can only move from one maze cell to a neighbouring cell. Figure-4.2 depicts some of the path environments considered in our experiments. Each row shows a path of five rooms, the top row shows a path with a single room profile (no obstacle in the room), and each subsequent path introduces a new type of room profile. Note that the composition of rooms used to build the path induces a hierarchical structure. The high-level agent navigates from room to room, and the low-level agent navigates within a room. Figure 4.3 only considers paths of increasing length, with a single room profile similar to the path depicted in the first row of Figure 4.2. In particular,

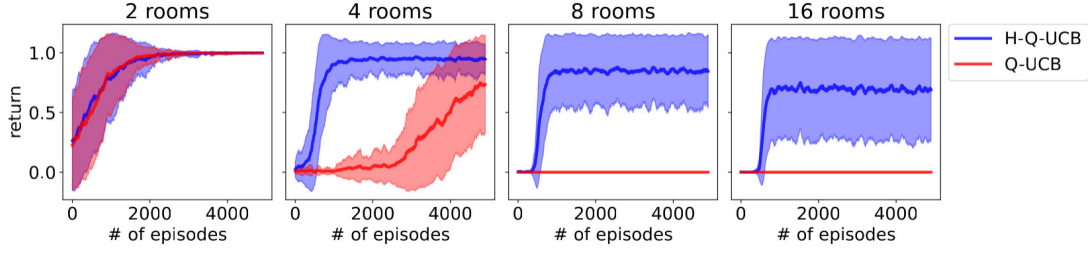


Fig. 4.3 Learning curve of $\text{GHRL}_{\text{Q-UCB}}$ (in blue) and standard Q-UCB (in red). We compare the performance of the two algorithms on a navigation environment that describes a path composed of several rooms. As we increase the number of identical rooms, the efficiency gain of $\text{GHRL}_{\text{Q-UCB}}$ becomes more evident.

Figure 4.3 shows that the performance of $\text{GHRL}_{\text{Q-UCB}}$ and Q-UCB are similar if there are only two rooms, that is, one room to navigate and the goal room, as the hierarchy cannot be leveraged. However, as we increase the number of rooms, the benefit of the hierarchical framework becomes more visible. Note that if there are more than four rooms, Q-UCB cannot find a path to the goal within the allowed number of interactions.

When the environment has a single room profile, i.e., all the rooms are identical, the hierarchical algorithm has a considerable advantage because it can repeatedly re-use the same learned policy. To evaluate how the algorithm behaves in an environment with more room profiles, we consider all the paths depicted in Figure 4.2 and evaluate the performance of $\text{GHRL}_{\text{Q-UCB}}$ on each environment. Figure 4.4, we show the learning curve of our algorithm as we increase the number of room profiles. Increasing the number of room profiles creates a more challenging problem for the low-level policy, as it has to learn to navigate more than a single low-level MDP. As expected, increasing the number of room profiles makes the problem more challenging. This behaviour is supported by the theoretical results such as Corollary 4.4.3, which anticipate that the regret scales with $\sqrt{A_h} \geq \sqrt{L}$, where L in this scenario denotes the number of room profiles.

Finally, we evaluate $\text{GHRL}_{\text{Q-UCB}}$ on the N -rooms environment presented in Chapter 3. Figure 4.5 illustrates a version of the environment with four rooms; in the experiment, we

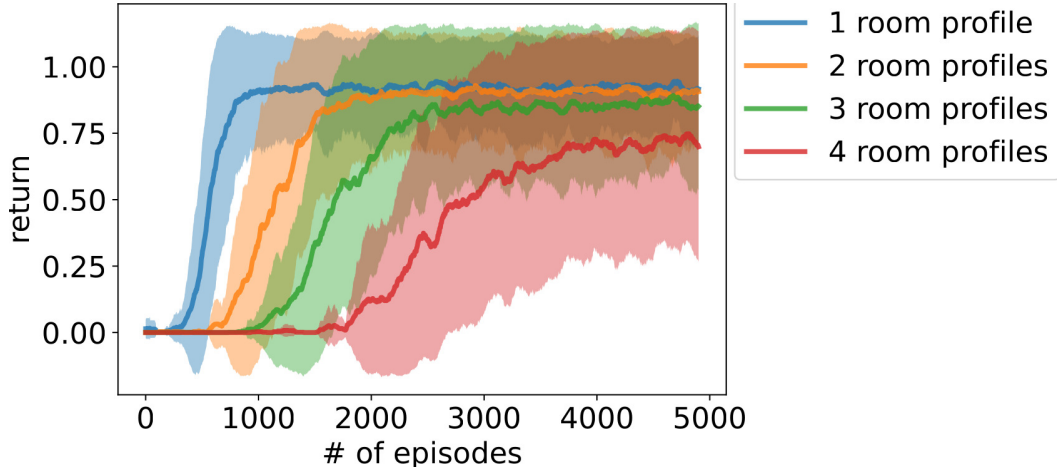


Fig. 4.4 This plot shows the impact of an increasing number of low-level MDPs on the learning curve. In blue, we show the performance of $\text{GHRL}_{\text{Q-UCB}}$ on a 5-room path with a single-room profile. Each colour encodes the learning curve of $\text{GHRL}_{\text{Q-UCB}}$ with a different number of room profiles. We can clearly see that performance deteriorates as we increase the number of room profiles. All environments considered in this experiment are depicted in Figure 4.2).

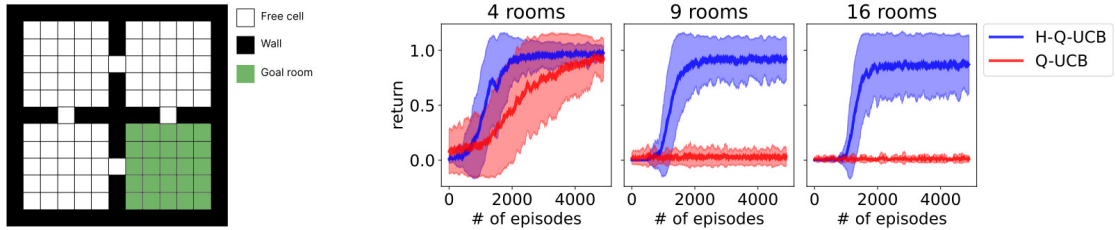


Fig. 4.5 An illustration of the results obtained on the n -room environment. The leftmost plot shows an example of a maze composed of 4 rooms, while the plots on the right show the performance obtained by $\text{GHRL}_{\text{Q-UCB}}$ (in blue) against Q-UCB (in red). As expected, as we increase the number of rooms, the efficiency gains of our method become more evident.

consider as well mazes with nine and sixteen rooms. The performance of $\text{GHRL}_{\text{Q-UCB}}$ and Q-UCB on these tasks is depicted on the rightmost plot of Figure 4.5, where the blue learning curves are generated by $\text{GHRL}_{\text{Q-UCB}}$ outperform the red learning curves generated by Q-UCB.

4.7 Discussion

The result obtained in Corollary 4.4.3 clearly shows an improvement over a monolithic approach. Osband and Van Roy [101] provide a lower bound on the regret paid by any algorithm $\mathfrak{A}$ in the original MDP M_o .

$$R_{\mathfrak{A}}(M_o, K) = \Omega(\sqrt{\|V\|_{\infty} A_o S T}) \quad (4.52)$$

$$= \Omega(\sqrt{A_o S H_o K}) \quad (4.53)$$

with T being the total number of interactions with the environment, in this case, $T = K H_o$ and the infinity norm of the value function $\|V\|_{\infty} = 1$ in the sparse reward setting. This lower bound quantifies the regret paid by the best possible monolithic algorithm in the original MDP M_o .

If we now consider the ratio between the regret of GHRL, which leverages the hierarchical decomposition, and the lower bound in Equation 4.52, we can provide insight on when to prefer GHRL over a standard monolithic approach. The ratio up to constants can be written as:

$$\begin{aligned} \frac{R_{\text{GHRL}}((\{M_{l,i}\}_{i=1,L}, M_h), K)}{R_{\mathfrak{A}}(M_o, K)} &\leq \frac{A_o A_h \max_i (S_{l,i}^+) H_l K + \Delta_{opt}}{A_o S_o H_o K} \\ &= \frac{A_h \max_i (S_{l,i}^+) H_l + \Delta_{opt}}{S H_o} \end{aligned} \quad (4.54)$$

where we use Δ_{opt} to compactly write the sub-optimality gap

$$\Delta_{opt} = \sum_{i=1}^L \sum_{e \in \mathcal{E}_i} 2N(i, e) \mathbb{P}(\neg \text{end}_{V^*}(i, e)).$$

In general, $\text{GHRL}_{\text{UCBVI}}$ is more efficient than any regret minimiser operating in the original MDP when the ratio in Eq. (4.54) is smaller than 1. This is guaranteed when the number of

sub-goals times the size of the largest low-level MDP, multiplied by the low-level episode length, is smaller than the original MDP state space times the original MDP episode's length, $A_h \max_i (S_{l,i}^+) H_l < S H_o$. This highlights the dependency of hierarchical algorithms on repetitive patterns inside the MDP and the ability to solve problems that present a shorter planning horizon, H_l .

Our results complement existing work on HRL efficiency (Wen et al. [146]). The regret guarantee of Wen et al. [146] holds in the Bayesian regret setting, representing a weaker guarantee than the frequentist regret provided in Theorem 4.4.2. In addition, the Bayesian regret of Wen et al. [146] holds only for the posterior sampling-based algorithm for hierarchical reinforcement learning (PSHRL). Theorem 4.4.2 gives a general guarantee that holds for a class of optimistic low-level algorithms Ξ . It is interesting to compare the regret of $\text{GHRL}_{\text{UCBVI}}$ to the Bayesian regret term obtained in Wen et al [146] (adapted to the sparse reward setting):

$$\mathbb{E}[R_{\text{PSHRL}}(M_o, K)] = \Delta_{sg} A_h K + \tilde{O}(H_o \max_i S_{l,i}^+ \sqrt{L A_o K}) \quad (4.55)$$

An immediate difference between the two regret bounds is the nature of their sub-optimality gap. The term Δ_{sg} in Equation 4.55 depends on the algorithm capacity to properly quantify the relative value of the different sub-goals. We completely avoid this cost because we can value each sub-goal with Backward Induction in the optimistic high-level MDP. In turn, Equation 4.11 does need to account for the failure probability of the optimal policy, Δ_{opt} , as the considered algorithm does not allow the high-level policy to continue to select sub-goal after an unsuccessful low-level episode. Additionally the bound provided in Corollary 4.4.3 improves the regret of Equation 4.55 by a factor $\sqrt{H \max_i |S_{l,i}^+|}$.

Finally, our analysis highlights how the low-level optimism is used in our algorithm to guide the exploration of the high-level algorithm, which represents an interesting mechanism that was not studied in previous work.

4.7.1 Connection with Chapter 3

The goal-conditioned HRL setting considered in this chapter makes several key assumptions that prevent a direct comparison to the lower bound derived in Chapter 3. In particular, in Chapter 3, we aimed to formulate the problem in the most general way possible. The setting tackled by GHRL makes several additional assumptions. First, Chapter 3 considered a general bounded reward, while this chapter focuses on sparse reward functions. Second, the setting considered in Chapter 3 makes no particular assumptions about the high-level transition function. In the setting of this chapter, we assume that the algorithm has access to a deterministic map that indicates which high-level state we observe next if the sub-goal is successfully completed. Combined, these two assumptions enable us to construct an approximate high-level transition function, reducing the problem to learning only the low-level MDPs. This is an important distinction to the setting considered in Chapter 3 where none of the above assumptions about the structure of the transition function and the sparse reward function are made, and the two levels of the hierarchies have to be learned.

Given the nature of the problem we have to solve in this particular setting, a conservative lower bound on the sample complexity would consist of the number of samples required to learn each of the individual low-level MDPs, accounting for the fact that we could run up to H_h low-level episode in a single episode in the original MDP.

Formally, we could state the lower bound as follows:

Theorem 4.7.1. *There exist positive constants c and δ_0 such that for every $\delta \in (0, \delta_0)$ and for every algorithm A that satisfies a PAC guarantee for (ϵ, δ) and outputs a deterministic policy, there is a fixed horizon MDP such that A must interact for*

$$\mathbb{E}[N] = \Omega\left(\sum_{i \in [L]} \frac{|S_{l,i}^+| |\mathcal{A}| H_l^2}{H_h \epsilon^2} \ln\left(\frac{1}{\delta + c}\right)\right) \quad (4.56)$$

episodes, in the original MDP, until the policy is (ϵ, δ) -accurate.

This lower bound highlights that in the setting considered, we do not have to consider the cost incurred by learning the high-level transitions, and we only account for the low-level dynamics.

4.8 Conclusion

This work studies the efficiency of a Hierarchical Reinforcement Learning method when applied to goal-based RL problems with sparse rewards. We propose a regret minimisation algorithm and study how its regret relates to the lower bound on monolithic RL approaches that do not explicitly exploit the hierarchical structure. Our regret bound underscores the nature of the efficiency gains of the hierarchical decomposition, suggesting that hierarchical methods can leverage patterns (low-level MDPs) that are repeated to construct the original problem. The importance of this repeated structure was already highlighted in previous work such as Wen et al. [146]. In contrast to Wen et al. [146] Theorem 4.4.2 provides a frequentist regret, which is a stronger guarantee than the Bayesian regret guarantee provided in [146]. Our result also improves upon the only frequentist regret upper bound on HRL by Drappo et al. [40] since the bound in [40] depends on the behaviour of the policy during the learning. Overall, our analysis of HRL with a sparse reward highlights key aspects of the nature of the collaboration between the high-level agent and the low-level agent. More specifically, it underscores how low-level optimism can be used to guide high-level exploration. Additionally, we provide an improved guarantee on the regret of HRL algorithms. The sparse reward setting allowed us to highlight that problems exist for which an improved regret bound is achievable. Nevertheless, extending our results to more general reward functions is an interesting perspective for future work.

Chapter 5

Posterior Sampling for RL on Graphs

Graphs are a natural representation that encodes the relations between connected entities. RL algorithms which leverage this latent graphical structure have emerged as a promising alternative to traditional methods for discovering better decision-making strategies. However, we still have an incomplete understanding concerning the sources of these efficiency gains. This chapter focuses on a sub-class of MDPs with a latent graphical structure. We propose and analyse a posterior sampling-based algorithm to efficiently leverage this structure. We provide theoretical and empirical validation of the proposed algorithm. The content of this chapter consists of the work done in:

Robert, A., Faisal, A. A., & Pike-Burke, C. (2024). Posterior Sampling for Reinforcement Learning on Graphs.

Which has been submitted to *Transactions on Machine Learning Research*.

5.1 Introduction

In this chapter, we consider a sub-class of Markov Decision Processes where the environment’s dependency structure is encoded in a directed acyclic graph. In such a setting, the state and the action can be decomposed into states and actions at each node of the graph, and the reward and transition functions depend on the connectivity of each node. In particular, the state at each node of the graph will depend on the state and action taken at all its parent nodes.

This structure is present in a lot of real-world settings. For example, such a structure can be found in wind farm optimisation tasks, e.g. [2], where a grid of rotating wind turbines creates a stream of turbulent wind that impacts the yield of downstream wind turbines. The turbulence at downstream wind turbines depends on the orientation of the upstream wind turbine. Finding the optimal orientation of all wind turbines in the wind farm becomes an interesting control problem. Since the turbines’ locations are known, the interaction pattern between all turbines composing the farm is also known. This interaction pattern can be encoded as a directed acyclic graph, where each node represents a wind turbine and each edge indicates a potential interaction. We dedicate Chapter 6 to empirically demonstrate that such a latent structure exists in the wind farm yield optimisation problem.

Such problems are typically modelled as directed acyclic graphs where the state at each node depends on the action taken at the parent nodes, and we observe the outcome of the actions at the parent nodes before taking actions at the child nodes. The setting we consider encompasses a broad range of problems that can be formulated as maximum flow problems. This includes, for instance, network routing challenges [85] and supply chain management scenarios [119], both of which can be effectively modelled using this framework. Typically, maximum flow problems are represented as directed acyclic graphs (DAGs), where the state of a given node is determined by the actions executed at its parent nodes. Crucially, the

outcomes of actions taken at parent nodes are observed before any decisions are made at the child nodes.

This chapter demonstrates that exploiting the underlying latent graphical structure can lead to significant performance improvements. These gains in performance are made possible through a careful analysis of the proposed algorithm that breaks down the state and action spaces' combinatorial structure and reuses recurring patterns within the graph. It is important to highlight that the DAMDP framework we explore shares similarities with the Factored MDP (FMDP) framework [23]. However, there are notable differences between the two frameworks that call for further innovation. As discussed in Section 5.8, the main difference is that FMDP framework considers a time-homogeneous setting, where the factorisation of the state space remains the same for every time step. Alternatively, in the DAMDP framework, we consider a time-inhomogeneous setting where the factorisation structure might evolve over time. This flexibility allows us to model more complex graph structures, thereby capturing a broader range of real-world scenarios and making the framework more applicable to practical problems with varying dependencies over time.

Contribution: To illustrate the efficiency gains obtained from leveraging the underlying latent graphical structure, we begin by formally defining the subclass of MDPs that our approach targets. In scenarios where the system dynamics are known, we show that a dynamic programming approach can be applied to find an optimal policy. For scenarios with unknown rewards and transition dynamics, we introduce a novel posterior sampling-based algorithm. We establish an upper bound on the Bayesian regret of the proposed algorithm. This upper bound represents an improvement compared to methods that ignore the latent graphical structure, such as [10, 99]. Additionally, we provide empirical evidence to support the effectiveness of our approach. In particular, we validate our algorithm on a maximum flow problem and show that it outperforms an algorithm that does not leverage the

latent graphical structure. Chapter 6 complements this empirical evaluation by providing an empirical validation of this method on a wind farm optimization problem.

To summarize, this chapter introduces, analyses and evaluates a novel posterior sampling algorithm specifically designed to exploit the graphical structure present in many real-world problems.

5.2 Time Inhomogeneous MDP and Directed Acyclic Graphs

We first recall the foundations of time-inhomogeneous Markov Decision Processes and Directed Acyclic Graphs. In Chapter 2, we formally introduced the time-homogeneous MDPs (see Definition 2.1.1). In the time-inhomogeneous setting, we consider that all the components describing the MDP can change over time, and in particular, the state and the action spaces depend on time. Let's first recall the foundations of time-inhomogeneous Markov Decision Processes and some important notions about Directed Acyclic Graphs.

5.2.1 Time-Inhomogeneous Fixed-Horizon Markov Decision Process

A fixed-horizon time-inhomogeneous MDP is a tuple

$$M = \langle \{\mathcal{S}_t\}_{t=1}^H, \{\mathcal{A}_t\}_{t=1}^H, \{R_t\}_{t=1}^H, \{P_t\}_{t=1}^H, H, \rho \rangle.$$

We consider a finite, time-dependent setting where the state and action spaces, denoted by $\mathcal{S}_t$ and $\mathcal{A}_t$, respectively, depend on the time step $t \in \{1, \dots, H\}$. At each time step $t \in [H]$, the mean reward function $R_t : \mathcal{S}_t \times \mathcal{A}_t \rightarrow [0, 1]$ output the expected reward for a specific state-action pair. Specifically, when the agent is in state s_t , and performs action a_t , the agent receives a reward $R_t(s_t, a_t) + \eta_t$, where η_t are independent identically distributed sub-Gaussian noise ($\sigma = 1$). At each time step t , the environment's dynamics are governed by a transition probability $P_t(s_{t+1}|s_t, a_t)$, which defines the probability of reaching a specific state

$s_{t+1} \in \mathcal{S}_{t+1}$ when action $a_t \in \mathcal{A}_t$ was performed in state $s_t \in \mathcal{S}_t$. The agent interacts with the environment over episodes of length H . At the beginning of each episode, the agent starts in an initial state $s_1 \sim \rho$, drawn from the initial state distribution ρ . Over the subsequent H time steps, $t \in \{1, \dots, H\}$, the agent observes a state $s_t \in \mathcal{S}_t$ and selects the action $a_t \in \mathcal{A}_t$ it wishes to perform. As a result of this action, the environment transitions to a new state $s_{t+1} \sim P_t(\cdot | s_t, a_t)$ and issues an immediate reward $R_t(s_t, a_t) + \eta_t$. This process continues for the duration of the episode, with the agent making sequential decisions at each step based on the state of the system and the observed rewards. As we know, RL algorithms aim to find policies $\mu_t : \mathcal{S}_t \rightarrow \mathcal{A}_t$ for all $t \in \{1, \dots, H\}$ that maximise the cumulative reward over an episode. We measure this in terms of the value function:

$$V_{\mu,t}^M(s_t) = \mathbb{E} \left[\sum_{h=t}^H R_h(s_h, a_h) \middle| a_h = \mu_h(s_h), s_{h+1} \sim P_h(\cdot | s_h, a_h), s_t = s \right], \quad (5.1)$$

where the expectation is taken over the starting state distribution $s_1 \sim \rho$ and the stochastic transition dynamics $s_{h+1} \sim P_h(\cdot | s_h, a_h)$. Often, we evaluate the quality of a policy μ in terms of the value function of the initial state s_1 , $V_{\mu,1}^M(s_1)$, where the superscript M indicates that we compute the value in the MDP M . The aim is to find an optimal policy μ^* within the set of Markov deterministic policies Π , which maximizes the value function,

$$\mu^* \in \operatorname{argmax}_{\mu \in \Pi} V_{\mu,1}^M(s_1). \quad (5.2)$$

It is useful to define the Q -function, which represents the expected reward a policy μ can obtain given that it first executes an action a and then selects actions according to μ . Specifically, the Q -function at time t of a policy μ after executing the action $a_t \in \mathcal{A}_t$ while in state $s_t \in \mathcal{S}_t$ is:

$$Q_{\mu,t}(s_t, a_t) = r_t(s_t, a_t) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} P_t(s_{t+1} | s_t, a_t) V_{\mu,t+1}(s_{t+1}). \quad (5.3)$$

5.2.2 Directed Acyclic Graphs

Definition 5.2.1 (Directed Acyclic Graph). A directed acyclic graph (DAG) $G = (V, E)$ is defined by a set of vertices V and a set of edges E . Each edge has a direction associated with it. The graph is said to be acyclic if there is no node for which there exists a path (with one or more edges) that leads to itself.

For any DAG, it is possible to establish a corresponding topological ordering of the nodes, as outlined in [16, Sec. 2.3.2]. This ordering assigns a specific layer to each node, and the layer of a node is defined as the length of the longest path from the root to the current node. In cases where the DAG does not have a unique root, we can trivially add an artificial node that serves as a root. In this work, we consider this form of *layered directed acyclic graphs*, whose edges only connect nodes from adjacent layers.

Definition 5.2.2 (Layered Directed Acyclic Graphs). A Layered, Directed Acyclic Graph (LDAG) is a DAG with a specific topology. It has a unique root node and a unique leaf node. All nodes are organised in layers, and all edges go from one layer to the next. Edges between nodes of the same layer are not allowed, as well as edges that connect nodes that are two or more layers away.

Under this topological ordering, all edges in the graph are constrained to connect nodes from a layer l to nodes in the next adjacent layer $l + 1$ with $l \in \{1, \dots, H - 1\}$. Larger steps, backward steps, and transversal steps (within the same layer) are not allowed. Importantly, the assumption of LDAG does not lose any generality. We demonstrate in Section 5.3.3 that we can construct an LDAG for any DAG. As this transformation preserves the connectivity patterns of the original DAG, one could argue that it has a limited impact on the complexity measure. We illustrate and discuss this point further in Section 5.3.3.

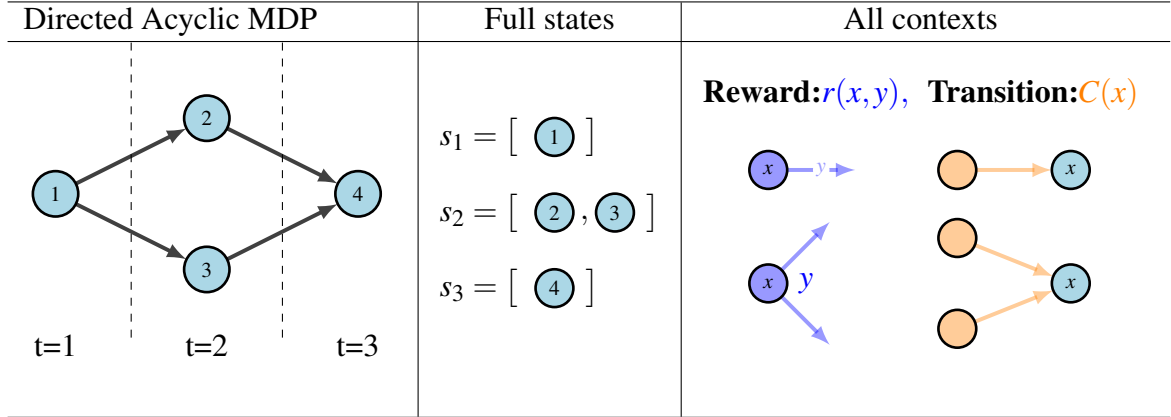


Fig. 5.1 Illustration of the important components of DAMDPs. Inside the leftmost box is an illustration of a simple DAMDP, M_G . The underlying graph has four nodes, which are visited within three time steps, $t = \{1, 2, 3\}$. The box in the middle illustrates the *full state* representation of M_G . At time step t , the *full state* s_t consists of the concatenation of the observations at each node in layer t . The rightmost box shows in the first column the two atomic reward contexts that arise in M_G , which is composed of the *atomic state* x and the *atomic action* y of the current node. The second column shows the two atomic transition contexts that arise in M_G , which consists of the *atomic states* and *atomic actions* observed at the current node, x 's, parents.

5.3 Directed Acyclic Markov Decision Process

We consider a particular subclass of MDPs that can be rolled out on an LDAG. To formalise this, we introduce Directed Acyclic Markov Decision Processes (DAMDPs), which can be defined by the following tuple:

$$M_G = \langle \mathcal{X}, \{\mathcal{Y}_j\}_{j=1}^{U_r}, \{r^j\}_{j=1}^{U_r}, \{p^i\}_{i=1}^{U_\tau}, H, \{\rho_A^v\}_{v=1}^{n_1}, G \rangle. \quad (5.4)$$

A visual representation of a simple DAMDP is provided in the leftmost plot of Figure 5.1. In addition to the MDP components (described below), a DAMDP M_G is structured by an LDAG, $G = (V, E)$. In this context, the horizon H of the DAMDP corresponds to the number of layers in the LDAG G . Since we sequentially observe the graph's layers and all edges $e \in E$ connect a node from a layer l to a node in layer $l + 1$, for $l \in \{1, \dots, H - 1\}$. At each time step, a layer of nodes of G describes the current state of the DAMDP; that is, in layer t ,

the n_t nodes that compose this layer describe the full state at time step t . For each layer t , we consider an arbitrary ordering of its nodes $\{1, \dots, n_t\}$, which remains fixed for the entirety of the training. The edges of G encode the dependence between different nodes.

In the DAMDP context, each node of G is associated with an *atomic state* x , where x belongs to the *atomic state* space $\mathcal{X}$. Similarly, at each node, an *atomic action* $y \in \mathcal{Y}_j$ can be performed, where $\{\mathcal{Y}_j\}_{j=1}^{U_r}$ denote the U_r different *atomic action* spaces. In some applications, like the one considered in our experiments (see Section 5.7), the *atomic action* description can be tightly linked to the number of outgoing edges at each node. Hence, it is natural to define several action spaces depending on the connectivity pattern of the graph G . When the agent performs an *atomic action* $y_t^v \in \mathcal{Y}_j$ at a node which is in *atomic state* x_t^v , it receives a reward $r^j(x_t^v, y_t^v) + \eta_{t,v}$, where $r^j(x_t^v, y_t^v)$ is the mean atomic reward function associated with the j^{th} atomic action space, $\eta_{t,v}$ is i.i.d. sub-gaussian noise, and $t \in \{1, \dots, H\}$ denotes the current layer. Here, and throughout, $v \in \{1, \dots, n_t\}$ denotes the position of the current node in its layer, and we use the notation x_t^v and y_t^v to refer to the v^{th} atomic state and action in the t^{th} layer. If two nodes have the same number of parents and if their parents have the same *atomic action* spaces, then these two nodes obey the same dynamics and are said to belong to the same transition equivalence class $[\tau_i]$, with $i \in \{1, \dots, U_\tau\}$ and U_τ the number of equivalence classes. Formally, this implies that the dynamics of nodes that belong to the same equivalence class $[\tau_i]$ are governed by the same atomic transition distribution $p^i(\cdot|c)$, with c the transition context, which consists of the atomic state and actions at all parent nodes. Concretely, let's consider the v^{th} node of layer t and assume that this node belongs to the equivalence class $[\tau_{i_{t,v}}]$, where here we use the notation $i_{t,v}$ to denote the index of the equivalence class that node v in layer t belongs to. The atomic state at this node is drawn from the atomic transition function $x_t^v \sim p^{i_{t,v}}(\cdot|c_t^v)$. The transition context $c_t^v \in \mathfrak{C}_{i_{t,v}}$ consists of the concatenation of all parent's nodes *atomic states* and *atomic actions*. $\mathfrak{C}_{i_{t,v}}$ is the space of all possible transition contexts and is the same for all nodes in the same equivalence class.

We denote by $\mathcal{V}_i$ the set containing the parent nodes of the node of interest, then the space of transition contexts for the i^{th} equivalence class is, $\mathfrak{C}_i = \bigotimes_{v \in \mathcal{V}_i} \mathcal{X} \times \mathcal{Y}_{j_v}$, the Cartesian product of all possible configuration of the parent nodes *atomic states* and *atomic actions*. Finally, the initial *atomic state* components are sampled from initial atomic state distribution $x_1^v \sim \rho_A^v$ for all nodes of the initial layer of G , $v \in \{1, \dots, n_1\}$.

5.3.1 Relationship Between DAMDPs and MDPs

A DAMDP $M_G = \langle \mathcal{X}, \{\mathcal{Y}_j\}_{j=1}^{U_r}, \{r^j\}_{j=1}^{U_r}, \{p^i\}_{i=1}^{U_\tau}, H, \rho_A, G \rangle$ represents a special case of a time-inhomogeneous finite horizon MDP. It would be particularly challenging to learn an optimal policy in this MDP as the state and action spaces are time-dependent, and the reward and transition functions are non-stationary. However, from the DAMDP perspective, the atomic state and action spaces do not depend on the time step, and the atomic reward and transition function are stationary. Before considering algorithms that can learn optimal policies in this setting, we first discuss the relationship between the DAMDP M_G and its corresponding MDP M constructed from the atomic components and the graph G .

Time step and horizon: The execution of an episode in a DAMDP is intricately connected to the structure of the graph G . Given that the graph G is layered, directed and acyclic, each layer l corresponds to a time step t in the corresponding MDP M . Consequently, the depth H of the graph G corresponds to the MDP's horizon (i.e. the maximum number of steps within a single episode). After executing K episodes, the total number of time steps is $T = HK$.

State space: At any given time step t , the *full state* $s_t \in \mathcal{S}_t$ consists of the atomic state observed at each node within the t^{th} layer of the graph G . More precisely, the atomic

representation of the full state $s_t \in \mathcal{S}_t$ can be written as follow:

$$s_t = [x_t^1, \dots, x_t^{n_t}],$$

where x_t^v is the observation collected in the v^{th} node of layer t (see Fig. 5.1, middle box) and n_t denotes the number of nodes in layer t . For simplicity, we assume that every node of G shares the same atomic state space, $\mathcal{X}$. Consequently, the full state space at time step t is the Cartesian product of the atomic state space of all nodes in layer t , $\mathcal{S}_t = \bigotimes_{v=1}^{n_t} \mathcal{X}$. In this context, the term *full state* refers to the concatenation of the observed values across all nodes of a layer, i.e. $s_t = [x_t^1, \dots, x_t^{n_t}]$. The *full state* is the state observed in the MDP M . In contrast, an *atomic state* is the value observed at a specific node, i.e. x_t^v for some $v \in \{1, \dots, n_t\}$ in the DAMDP M_G .

Action space: We similarly define an *atomic action* space as the set of actions available at a particular node. In contrast to the *atomic state*, all nodes do not necessarily share the same action space; $\{\mathcal{Y}_j\}_{j=1}^{U_r}$ represents the U_r distinct *atomic action* spaces available. Let $y_t^v \in \mathcal{Y}_{j_{t,v}}$ be the *atomic action* of the v^{th} node of layer t , and $\mathcal{Y}_{j_{t,v}}$ be the *atomic action* space associated with this specific node, with $j_{t,v} \in \{1, \dots, U_r\}$ indicating which equivalence class node v belongs to. The *full action* space at time step t can be expressed as the Cartesian product of the *atomic action* space of each node composing layer t , $\mathcal{A}_t = \bigotimes_{v=1}^{n_t} \mathcal{Y}_{j_{t,v}}$. The *full action* at time step t , $a_t \in \mathcal{A}_t$, is the concatenation of the *atomic actions* selected at each node of layer t :

$$a_t = [y_t^1, \dots, y_t^{n_t}]. \quad (5.5)$$

In Figure 5.1 (rightmost plot), we show examples where the dimension of the *atomic action* depends on the number of exiting edges.

Policies: Let $\mu = \{\mu_t\}_{t=1}^H$ denote a collection of time-dependent policies that maps a *full state* s_t to a *full action*: $\mu_t : \mathcal{S}_t \rightarrow \mathcal{A}_t$ for all $t \in \{1, \dots, H\}$. To identify the optimal *full action* at each layer, the policy needs to jointly select all *atomic actions* simultaneously. Since nodes in the same layer can have common successors, all *atomic actions* in a given layer must be jointly selected; ignoring these common dependencies and selecting *atomic actions* independently could lead to sub-optimal policies. Hence, the policies we consider still operate on the *full state* and *full action* spaces. Note that sometimes, we are interested in the action selected at a specific node. Let $a_t = \mu_t(s_t)$ be the *full action*, we denote the *atomic action* corresponding to the v^{th} node of layer t as $\mu_t(x_t^v | s_t)$, to make it explicit that the policy requires knowledge about the *full state* to select *atomic actions*.

Transition function: We assume that the known latent graphical structure encodes conditional independence between nodes, with the state at a node depending only on the atomic state and actions taken at the parent nodes. We split nodes into equivalence classes $[\tau_i]$ based on their connectivity patterns, with all nodes in the same equivalence class having the same number of parents with the same atomic action spaces. The DAMDP is then governed by a set of atomic transition functions $\{p^i\}_{i=1}^{U_\tau}$. In particular, the probability of observing an atomic state $x_t^v \in \mathcal{X}$ at the v^{th} node of layer t is $p^{i,v}(x_t^v | c_t^v)$, where $c_t^v \in \mathcal{C}_{i,v}$ denotes the *atomic state* and the *atomic actions* at all parent nodes and $i_{t,v}$ denotes the equivalence class of the node. Figure 5.1 (rightmost plot) shows all the transition contexts that arise in a simple DAMDP. The atomic transition function offers two immediate benefits. First, it reduces the number of dependent variables, focusing only on the subset of variables that influences the next atomic state. Second, it is possible that a layer contains more than a single node belonging to the equivalence class $[\tau_i]$; in that case, within a single step, the algorithm will observe several samples from the same atomic transition function, p^i , offering more opportunities to collect data and learn the true unknown dynamics. We sometimes want the transition

context to reflect that a specific *full action* $a \in \mathcal{A}_{t-1}$ was executed in layer $t - 1$. To clarify this relation, we write the context of node $v \in \{1, \dots, n_t\}$ in layer t as $c_{t,a}^v$. The context of any node in layer t can then be constructed using the atomic description of a , $a = [y_{t-1}^1, \dots, y_{t-1}^{n_{t-1}}]$. Similarly, if actions are selected by a given policy μ , we make this relation explicit in our notation with $c_{t,\mu}^v$. Then, the probability of observing a new *full state* s_{t+1} , given that the *full action* a_t has been executed in *full state* s_t is given by $P_t(s_{t+1}|s_t, a_t) = \prod_{v=1}^{n_{t+1}} p^{j_{t,v}}(x_{t+1}^v | c_{t,a}^v)$, where $s_t = [x_t^1, \dots, x_t^{n_t}]$, $s_{t+1} = [x_{t+1}^1, \dots, x_{t+1}^{n_{t+1}}]$ and $c_{t,a}^v$ denotes the context of node v in layer t given that full action a was selected. Note that for the first transition, at $t = 1$, we define the context to be the empty set, that is, $c_1^v = \emptyset$.

Reward function: We assume that the expected reward at a time step t in the full MDP is the sum of the atomic expected reward observed at each node composing the t^{th} layer, $R_t(s_t, a_t) = \sum_{v=1}^{n_t} r^{j_{t,v}}(x_t^v, y_t^v)$, where x_t^v and y_t^v are the *atomic states* and *atomic actions* at node v , $j_{t,v}$ denotes the action space available at the v^{th} node of layer t , and $r^{j_{t,v}}(x, y)$ is the mean of the atomic reward distribution associated with atomic action set $\mathcal{Y}_{j_{t,v}}$.

5.3.2 Learning in a DAMDP

As depicted in Figure 5.1 (rightmost plot), G can be partitioned into layers, each representing a time step, $t \in \{1, \dots, H\}$, of the DAMDP. At each time step within an episode, t , the agent observes the state of each node within the t^{th} layer of the graph. Note that the number of nodes, n_t , will change depending on the time step t . Hence, the dimensionality of the *full state* and the observation also depends on the time step t . After observing each node's *atomic state*, the agent chooses the *full action* it wishes to perform (recall that this corresponds to an atomic action being taken at each node). Again, the dimensionality of the *full action* will depend on the current time step. Once the *full action* is taken, the agent observes the reward obtained at each node in the current layer and the new *atomic states* in each node of the next

layer. This process repeats until the algorithm goes through all the H layers in the graph, which completes an episode.

Value function: Similar to the value function of a policy in an MDP, the value of a policy μ in a DAMDP consists of the expected reward accumulated at each node; when ambiguous, we use the subscript M_G to indicate that this value function is computed under the DAMDP M_G , namely $\forall t \in \{1, \dots, H\}$ and $\forall s \in \mathcal{S}_t$,

$$V_{\mu,t}^{M_G}(s) = \mathbb{E} \left[\sum_{h=t}^H \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, y_h^v) \middle| y_h^v = \mu_h(x_h^v | s_h), s_{h+1} \sim P_h(\cdot | s_h, a_h), s_t = [x_t^1, \dots, x_t^{n_t}] \right]. \quad (5.6)$$

When $t > 1$, the expectation is only computed over the stochastic transition dynamics.

Among all the Markov and deterministic policies $\mu \in \Pi$ we aim to find the optimal policy, μ^* , i.e. the policy with the largest value function:

$$\mu^* = \operatorname{argmax}_{\mu \in \Pi} V_{\mu,1}^{M_G}(s_1). \quad (5.7)$$

We consider a sequential learning setting where the learner interacts with the DAMPD over K episodes. After each episode, the learner can use the information gathered to improve their policy in the next episode. To measure the learner's performance on this task, we define the regret, which compares, at each point in time, the performance of the current policy against the optimal policy μ^* .

Definition 5.3.1 (Regret). A reinforcement learning algorithm $\mathfrak{A}$ chooses for each episode $k \in \{1, \dots, K\}$ the policy μ_k that interacts with the environment. The regret of algorithm $\mathfrak{A}$ over K episodes is defined as,

$$\text{Regret}(K, \mathfrak{A}) = \sum_{k=1}^K \Delta_k, \quad (5.8)$$

where Δ_k denotes the difference of value function in the true DAMDP, M_G^* , between the optimal policy μ^* and the current policy μ_k :

$$\Delta_k = \sum_{s \in \mathcal{S}_1} \rho(s) \left(V_{\mu^*,1}^{M_G^*}(s) - V_{\mu_k,1}^{M_G^*}(s) \right). \quad (5.9)$$

where $\rho(s) = \prod_{v=1}^{n_1} \rho_A^v(x_1^v)$ with $s = [x_1^1, \dots, x_1^{n_1}]$.

Note that the regret is a stochastic quantity that depends on the random behaviour of M_G^* and the algorithm's sampling procedure. In the remainder of this chapter, we consider the Bayesian regret, where we also take an expectation over all possible DAMDPs M_G that are drawn from a prior over MDPs f ,

$$\mathbb{E}[\text{Regret}(K, \mathcal{A})] = \sum_{k=1}^K \mathbb{E}[\Delta_k].$$

Note that, in general, we use t_k to denote the starting time of the k^{th} episode, so $t_k = (k-1)H + 1$, and we denote with $t_k + i$ the i^{th} time step of episode k .

5.3.3 Additional Properties of DAMDP

Remark 5.3.2. Any DAMDP with an arbitrary DAG, G , has an equivalent DAMDP with a layered directed acyclic graph, G' .

We now show how a LDAG, G' , can be constructed from a DAG, G . In particular, we focus on two possible transformations that are illustrated in Figure 5.2, that is, when an edge spans more than one layer (first row of Fig. 5.2) or when we have a transversal edge, i.e. an edge that connects two nodes that belong to the same layer, (second row of Fig. 5.2).

Note that both illustrations Figure 5.2 suggest that to transform a DAG into LDAG, we need to add artificial nodes (represented by a red square) and artificial edges (coloured in blue and orange). The artificial node does not directly increase the complexity of the problem, as

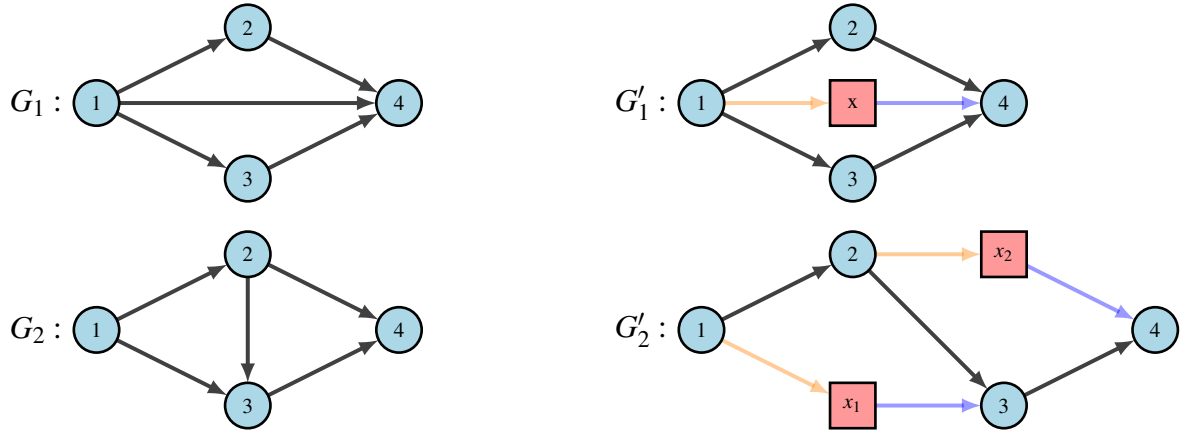


Fig. 5.2 Illustration of how to transform a directed acyclic graph (on the left) into a layered directed acyclic graph (on the right)

both its atomic state and atomic action will be identical to the atomic state and atomic action of its unique parent node. The additional edges do not increase the problem complexity as when we remove an "illegal" edge with a single orange edge and a single blue edge. In particular, it leaves the number of equivalence classes U_τ unchanged since all incoming illegal edges were replaced by a single blue edge, and it leaves the number of action spaces unchanged as well as all outgoing illegal edges are replaced by an orange edge. Since those transformations are always possible and leave the DAMDP characteristics unchanged, they have no impact on the algorithm complexity measure presented in Theorem 5.5.1.

While this argument is reasonable in problems of the type of "the leaky maximum flow" problem, i.e., when we have a single type of edge, it will not necessarily hold for more complex problems where we would like to consider different types of edges. In Chapter 6, we consider a problem where we want to optimise the yield of a wind farm, and we use a graph to model potential interaction. The nature of the interaction will likely depend on the relative distance between two graph nodes.

5.4 Dynamic Programming on Graphs

We begin by considering how to learn an optimal policy in a known DAMDP. This will form an important building block of the reinforcement learning algorithm for DAMDPs presented in Section 5.5.

The planning algorithm 10 computes an optimal policy via backwards recursion. Dynamic programming algorithms rely on the self-consistency property of the Bellman operator. We show that there exists an atomic Bellman operator for any time step $t \in \{1, \dots, H\}$, $\mathcal{T}_{\mu,t}^{M_G}$, that performs a one-step rollout according to the policy μ , using only the atomic description of the DAMDP M_G ,

$$(\mathcal{T}_{\mu,t}^{M_G} V_{\mu,t+1}^{M_G})(s_t) := \sum_{v=1}^{n_t} r^{j_t,v}(x_t^v, \mu_t(x_t^v | s_t)) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \prod_{v=1}^{n_{t+1}} p^{i_t,v}(x_{t+1}^v | c_{t+1,\mu}^v) V_{\mu,t+1}^M(s_{t+1}), \quad (5.10)$$

for all $t \in \{1, \dots, H\}$ and for all $s_t \in \mathcal{S}_t$. Where $s_t = [x_t^1, \dots, x_t^{n_t}]$ and $c_{t,\mu}^v$ denotes the transition context of the v^{th} node of layer t when actions are selected by μ .

The following lemma guarantees the self-consistency of the atomic Bellman operator in equation 5.10.

Lemma 5.4.1 (Consistency of atomic Bellman operator). *For any DAMDP*

$M_G = \langle \mathcal{X}, \{\mathcal{Y}_j\}_{j=1}^{U_r}, \{r^j\}_{j=1}^{U_r}, \{p^i\}_{i=1}^{U_\tau}, H, \rho_A, G \rangle$ *and policy* μ , *the value function* $V_{\mu,t}^{M_G}$ *satisfies:*

$$V_{\mu,t}^{M_G}(s_t) = \mathcal{T}_{\mu,t}^{M_G} V_{\mu,t+1}^{M_G}(s_t), \quad (5.11)$$

for $t \in \{1, \dots, H-1\}$ *and* $s_t \in \mathcal{S}_t$, *with* $V_{\mu,H}^{M_G}(s_H) = \max_{a \in \mathcal{A}_H} R_H(s_H, a)$ *for all* $s_H \in \mathcal{S}_H$.

This property is directly inherited from the MDP assumption; for the sake of completeness, we prove this lemma.

Proof.

$$\begin{aligned}
\mathcal{T}_{\mu,t}^{M_G} V_{\mu,t+1}^{M_G}(s_t) &= \sum_{v=1}^{n_t} r^{j_{t,v}}(x_t^v, \mu(x_t^v | s_t)) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \prod_{v=1}^{n_{t+1}} p^{i_{t,v}}(x_{t+1}^v | c_{t+1,\mu}^v) V_{\mu,t+1}^{M_G}(s_{t+1}) \\
&= \sum_{v=1}^{n_t} r^{j_{t,v}}(x_t^v, \mu(x_t^v | s_t)) + \\
&\quad \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \prod_{v=1}^{n_{t+1}} \left(p^{i_{t,v}}(x_{t+1}^v | c_{t+1,\mu}^v) \mathbb{E}_P \left[\sum_{h=t+1}^H \sum_{v=1}^{n_h} r^{j_{h+1,v}}(x_{h+1}^v, \mu(x_{h+1}^v | s_{h+1})) \right] \right) \\
&= \mathbb{E}_P \left[\sum_{h=t}^H \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, \mu(x_h^v | s_t)) \right] \\
&= V_{\mu,t}^M(s_t)
\end{aligned}$$

To obtain the first equality, we used the definition of the Bellman operator, and to obtain the second equality, we used the definition of $V_{\mu,t+1}^{M_G}$. Observing that we consider all possible next states s_{t+1} weighted by their probability of appearing, we can directly include this computation into the expectation, leading to the third equality. $\square$

Before developing a reinforcement learning algorithm for DAMDPs in Section 5.5, we need to verify that there exists a planning algorithm that can compute the optimal policy for a given atomic transition distribution p and atomic reward function r . We show in Theorem 5.4.2 that this can be done via dynamic programming as outlined in Algorithm 10. The algorithm leverages the one-step Bellman equation for DAMDP in equation 5.10. It starts by finding the optimal *full action* for each *full state* at the last layer, H . This is trivial as the last *full action* will not impact future rewards. For each time step $t \in \{H-1, \dots, 1\}$ and each *full state* $s_t \in \mathcal{S}_t$ it uses the one-step Bellman equation for DAMDP equation 5.10 to find the corresponding value $u_t(s)$ and the optimal *full action* a^* . The following theorem guarantees that Algorithm 10 retrieves the optimal policy μ^* and its associated value function $V_{\mu^*,1}^{M_G}$ for any known DAMDP M_G . The policy μ^* can be directly used to find the optimal *atomic*

Algorithm 10: Planning on a DAMDP**Input:** $\{\hat{p}^i\}_{i=1}^{U_\tau}, \{\hat{r}^j\}_{j=1}^{U_r}, G$ 1 $u_H(s_H) = \max_{a \in \mathcal{A}_H} \sum_{i=1}^{n_H} \hat{r}^{j_{H,i}}(s_H^i, a^i) \quad \forall s_H \in \mathcal{S}_H$ 2 **for** $t = H - 1, \dots, 1$ **do**3 **for** $s_t \in \mathcal{S}_t$ **do**4 Evaluate $u_t(s_t)$ according to:

5

$$u_t(s_t) = \max_{a_t \in \mathcal{A}_t} \left\{ \sum_{v=1}^{n_t} \hat{r}^{j_{t,v}}(x_t^v, y_t^v) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \prod_{v=1}^{n_{t+1}} \hat{p}^{i_{t,v}}(x_{t+1}^v | c_{t+1, a_t}^v) u_{t+1}(s_{t+1}) \right\},$$

6 Set

7

$$\pi(s) = \operatorname{argmax}_{a_t \in \mathcal{A}_t} \left\{ \sum_{v=1}^{n_t} \hat{r}^{j_{t,v}}(x_t^v, y_t^v) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \prod_{v=1}^{n_{t+1}} \hat{p}^{i_{t,v}}(x_{t+1}^v | c_{t+1, a_t}^v) u_{t+1}(s_{t+1}) \right\}$$

8

9 here $s_t = [x_t^1, \dots, x_t^{n_t}]$ and $a_t = [y_t^1, \dots, y_t^{n_t}]$.**Output:** π and $u_1(s_1) \forall s_1 \in \mathcal{S}_1$

actions, since the optimal *full action*, $a_t^* = [y_t^{1,*}, \dots, y_t^{n_t,*}]$, directly encodes the optimal *atomic action*, $y_t^{i,*}$ for each node $i \in \{1, \dots, n_t\}$ of the t^{th} layer of G .

Theorem 5.4.2. *If Algorithm 10 receives as input $\{p^i\}_{i=1}^{U_\tau}$ and $\{r_j\}_{j=1}^{U_r}$, the atomic reward and transition functions of a known DAMDP M_G , then it returns $u_t(s_t)$ and π , the optimal value function and policy, i.e.*

$$u_1(s_1) = V_{\mu^*, 1}^M(s_1) \forall s_1 \in \mathcal{S}_1 \text{ and } \pi = \mu^*. \quad (5.12)$$

The proof of Theorem 5.4.2 follows the one presented in [114, ch. 4] for completeness; it is stated below.

Proof. The proof follows the one presented in [114, ch. 4] and consists of two parts.

Part i: Show that the learned value function $u_t(s_t) \geq V_{\pi,t}^{MG}(s_t)$ for all $\pi \in \Pi$ and for all $(t, s_t) \in \{1, \dots, H\} \times \mathcal{S}_t$, where Π is the class of all deterministic and Markov policies.

Part ii: Show that the algorithm's output satisfies $u_t(s_t) = V_{\pi^*,t}^{MG}(s_t)$ for all t and all $s \in \mathcal{S}_t$.

Proof of i: Let $\pi \in \Pi$ be any policy in the class of deterministic Markov policies. Since at the last time step H the goal is only to maximise the immediate reward, the suggested result holds at time step H , $u_H(s_H) \geq V_{\pi,H}^{MG}(s_H)$, for all $s_H \in \mathcal{S}_H$. Let's now assume that $u_t(s_t) \geq V_{\pi,t}^{MG}$ for $t = h+1, \dots, H$. We prove by induction that it also holds for $t = h$,

$$\begin{aligned}
 u_h(s_h) &= \max_{a \in \mathcal{A}_h} \left\{ \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, y_h^v) + \sum_{s_{h+1} \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,a}^v) u_{h+1}(s_{h+1}) \right\} \\
 &\geq \max_{a \in \mathcal{A}_h} \left\{ \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, y_h^v) + \sum_{s_{h+1} \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,a}^v) V_{\pi,h+1}^{MG}(s_{h+1}) \right\} \\
 &\geq \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, \pi(x_h^v | s_h)) + \sum_{s_{h+1} \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,\pi}^v) V_{\pi,h+1}^{MG}(s_{h+1}) \\
 &= \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, \pi(x_h^v | s_h)) + \mathbb{E}_{P,\pi} \left[\sum_{t=h+1}^H \sum_{v=1}^{n_t} r^{j_{t,v}}(x_t^v, y_t^v) \right] \\
 &\quad \text{(by def. of } V^{MG} \text{ in equation 5.6)} \\
 &= V_{\pi,h}^{MG}(s)
 \end{aligned}$$

where the notation $\mathbb{E}_{P,\pi}$ is used to explicitly mention that the expectation is computed over the stochastic transition dynamics and that actions are selected according to the policy π .

Proof of ii: The second part is also shown by induction. We start by observing that the last step focuses only on maximising the immediate reward. So at time step H it trivially holds that $u_H(s) = V_{*,H}^{MG}(s)$ for all $s \in \mathcal{S}_H$. We then assume that $u_t(s) = V_{*,t}^{MG}(s)$ for $t = h+1, \dots, H$

and for all $s \in \mathcal{S}_t$. Then,

$$\begin{aligned}
u_h(s) &= \max_{a \in \mathcal{A}_h} \left\{ \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, y_h^v) + \sum_{s' \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,a}^v) u_{h+1}(s') \right\} \\
&= \max_{a \in \mathcal{A}_h} \left\{ \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, y_h^v) + \sum_{s' \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,a}^v) V_{*,h+1}^{M_G}(s') \right\} \\
&= \sum_{v=1}^{n_h} r^{j_{h,v}}(x_h^v, \mu^*(x_h^v | s_h)) + \sum_{s' \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p^{i_{h,v}}(x_{h+1}^v | c_{h+1,*}^v) V_{*,h+1}^{M_G}(s') \\
&= V_{*,h}^{M_G}(s)
\end{aligned}$$

Where the third line is obtained by the hypothesis that $u_t(s) = V_{*,t}^{M_G}(s)$ for $t \geq h+1$ and from the definition of μ^* .

□

Remark 5.4.3. Using the atomic description of the environment does not impact the computational complexity of the planning algorithm, which remains $\mathcal{O}(H|\mathcal{A}||\bar{\mathcal{S}}|^2)$, where $\bar{\mathcal{S}}$ denotes the largest *full state* space.

5.5 Posterior Sampling RL on Graphs (PSGRL)

We now present the Posterior Sampling on Graphs for Reinforcement Learning (PSGRL) algorithm, which leverages the rich underlying structure in the DAMDP setting. The fact that both the transition and the reward functions can be written in terms of their atomic counterparts represents an opportunity to increase the algorithm's efficiency. Indeed, the size of the combinatorial spaces $\mathcal{S}_t$ and $\mathcal{A}_t$ can present a challenge when a learner is tasked to learn the full transition distribution P and the full reward function R directly. The inputs of the atomic transition functions and the atomic reward functions lie in spaces much smaller than their non-atomic counterparts, presenting an opportunity to speed up learning. The input space for the atomic transition function i depends on the size of the context space, $|\mathcal{C}_i|$, for

$i \in \{1, \dots, U_\tau\}$, where U_τ is the number of equivalence classes. Critically, $|\mathcal{C}_i|$ includes states and actions for only a subset of the nodes at the previous layer. This space is smaller than the *full state-action* space, which is the Cartesian product of the atomic state-action pairs of all nodes in the previous layer. Similarly, the atomic reward only depends on the atomic state $x \in \mathcal{X}$ and the atomic action $y \in \mathcal{Y}_j$, for $j \in \{1, \dots, U_r\}$, which is also smaller than the *full state-action* spaces.

We can exploit the structure to achieve further efficiency gains by observing that the input of the atomic transition and reward function can be observed more than once per time step. If two nodes in the same layer have the same number of incoming edges, then they are learning the same atomic transition function. Similarly, if they have the same *atomic action* space $\mathcal{Y}_j$, then they are learning the same atomic reward function. This benefits the algorithm as we observe more samples corresponding to the atomic functions than their non-atomic counterparts. However, in the analysis, we need to account for the fact that we can observe a reward or a transition input more than once per time step, which deviates from the classic analysis of RL frameworks that assume that a single sample is collected at each time step.

We present our posterior sampling algorithm for DAMDPs in Algorithm 11. The algorithm proceeds in episodes. At the beginning of each episode k , the algorithm samples a DAMDP $M_{G,k} \sim f(\cdot | D_k)$, where $f(\cdot | D_k)$ represents the current belief about the true DAMDP M_G^* , given the observations collected up to episode k , D_k . Note that in the context of DAMDP, to sample $M_{G,k}$ the algorithm essentially samples a set of *atomic* transition functions $\{p^i\}_{i=1}^{U_\tau}$ and *atomic* reward functions $\{r^j\}_{j=1}^{U_r}$. The algorithm then computes the optimal policy $\mu^{M_{G,k}}$ on $M_{G,k}$ and runs the policy for an episode. The trajectory $(s_t, a_t, r_t, s_{t+1})_{t=1}^H$ is added to our dataset D_k . We then use D_k to update our belief about the distribution of the true DAMDP M_G^* . We assume that each transition function is a categorical distribution governed by a set of parameters $\theta_{p^i} \in \mathbb{R}^X$, for all $i \in \{1, \dots, U_\tau\}$. The reward function has a Gaussian distribution with mean μ_{r^j} and standard deviation σ_{r^j} , for $j \in \{1, \dots, U_r\}$. The prior f maintains a

Algorithm 11: Posterior sampling on graph MDPs (PSGRL)**Input:** Prior f encoding $\mathcal{G}$

```

1 for episodes  $k = 1, 2, \dots$  do
2   Sample  $M_{G,k} \sim f(\cdot | D_{t_k})$ ;
3   Compute the optimal policy of  $M_{G,k}$ ,  $\mu^k$ , using Algorithm 10;
4   for time steps  $t = 1, \dots, H$  do
5     Select and execute  $a_t = \mu^k(s_t, t)$ ;
6     Observe  $r_t = [r_t^1, \dots, r_t^{n_t}]$  and  $s_{t+1} = [x_{t+1}^1, \dots, x_{t+1}^{n_{t+1}}]$ ;
7     Append  $(s_t, a_t, r_t, s_{t+1})$  to  $D_{t_{k+1}}$ ;

```

Dirichlet distribution over the space of parameters θ_{p^i} and a Normal-inverse gamma prior over the space of parameters μ_{r^j} and σ_{r^j} . This induces a prior distribution over DAMDPs.

The regret suffered by PSGRL (Alg. 11) is bounded by the following theorem.

Theorem 5.5.1. *If f is the distribution of M_G^* then the regret suffered by PSGRL (Alg. 11) over K episodes is upper bounded by,*

$$\mathbb{E}[\text{Regret}(K, \text{PSGRL})] = \tilde{O}\left(\sum_{j=1}^{U_r} H \sqrt{XY_j m_r^j K} + \sum_{i=1}^{U_\tau} H \sqrt{(X\bar{Y})^{d_i} m_\tau^i X K}\right) \quad (5.13)$$

where $X = |\mathcal{X}|$ is the size of the atomic state space, $\bar{Y} = \max_{j \in \{1, \dots, U_r\}} Y_j$ is the size of the largest atomic action space. For nodes that belong to the i^{th} equivalence class $[\tau_i]$, d_i denotes the number of parent nodes and m_τ^i denotes the number nodes in G that share the same equivalence class $[\tau_i]$. Finally, m_r^j is the number of nodes in G with the same atomic action space $\mathcal{Y}_j$.

As the proof of Theorem 5.5.1 is quite long, we defer to Section 5.9. It follows the proof in [99], with the key difference being that the confidence sets are built around the estimates of the atomic transition and reward functions. This approach has two immediate benefits. First, we might be able to collect more than a single sample per time step. If we consider the case where we have k nodes in the same layer, t , that belong to the same equivalence class,

$[\tau_i]$, then at time step t , we collect k samples from p^i . Second, the input space of the atomic reward and transition function can be significantly smaller than the input space in the original MDP (i.e. the transition and reward functions over the full state and action spaces). This leads to a significant gain in performance since the atomic transition and reward function are easier to estimate than their full state-action counterparts.

The bound presented in Theorem 5.5.1 is an improvement over the lower bound that can be obtained by running any RL algorithm in the full MDP and ignoring the latent graphical structure:

$$\Omega \left(\sum_{h=1}^H \sqrt{S_t A_t T} \right) = \Omega \left(\sum_{h=1}^H \sqrt{(XY)^{n_t} T} \right). \quad (5.14)$$

Where $\underline{Y} = \min_{j \in \{1, \dots, U_r\}} Y_j$ is the size of the smallest action space in the DAMDP. The comparison of PSGRL upper bound equation 5.13 and the lower bound equation 5.14 suggests an improved efficiency for PSGRL compared to RL algorithms that ignore the latent graphical structure. In particular, note that the number of incoming edges for any node in a given layer t is always bounded by the number of nodes at the previous layer $d_i \leq n_{t-1}$ where i denotes the equivalence class of any node in layer t . In conclusion, the regret suffered by PSGRL, as described in equation 5.13, is smaller than the regret any RL algorithm that ignores the graphical structure G could suffer as described in equation 5.14.

$$\sum_{j=1}^{U_r} H \sqrt{XY_j m_r^j K} + \sum_{i=1}^{U_\tau} H \sqrt{(X\bar{Y})^{d_i} m_\tau^i X K} \leq \sum_{h=1}^H \sqrt{(XY)^{n_t} T}$$

5.6 Regret of PSRL on the *full* MDP

A natural baseline to our approach is to consider the Posterior Sampling for Reinforcement Learning [99] algorithm. This algorithm can solve any DAMDP by considering the MDP

built from the DAMDP's atomic components. In that case, the algorithm will directly learn the *full state* transition distribution P and the reward function R .

To simplify the analysis, we upper bound the state space at each time step t , $\mathcal{S}_t \subseteq \mathcal{X}^{N_{max}}$, with the largest state space in the MDP, note that N_{max} denotes the number of nodes in the largest layer. Similarly, we upper bound the *full action* space at each time step t , $\mathcal{A} \subseteq \bar{\mathcal{Y}}^{N_{max}}$.

If we apply the results of [99] to the proposed MDP, we get a regret of

$$O\left(H\mathcal{X}^{N_{max}}\sqrt{\bar{\mathcal{Y}}^{N_{max}}T\log((\mathcal{X}\bar{\mathcal{Y}})^{N_{max}}T)}\right) \quad (5.15)$$

5.7 Maximum Leaky Flow of a Graph

We now illustrate that the performance gains suggested by Theorem 5.5.1 manifest themselves empirically. The maximum leaky flow problem is defined as a directed acyclic graph $G = (V, E)$, with a single root node, the source node $v_1 \in V$, and a single leaf node, called the sink node $v_N \in V$. Using the edges to transport flow from one node to the next, the goal is to transport as much flow as possible from the source node to the sink node. Because G is directed, it is only possible to transport flow in the edge's direction. The environment is governed by two parameters: edges have a given capacity, $c > 0$, which denotes the maximum amount of flow that can be transported along that edge, and an unknown failure probability, $p \in [0, 1]$, which dictates how likely the edge is to fail to transport the flow assigned to it.

The above problem is similar to the graph maximum flow problem [48]. However, the leaky version of the problem is more challenging as the probability of failure is unknown to the learner. In the leaky version of the problem, the learner is tasked to jointly learn the unknown dynamics and the paths that maximise the expected flow.

We consider two algorithms for this problem. The proposed PSGRL, which has access to the graphical structure and an extension of the PSRL algorithm [99] that works in time-inhomogeneous settings, which does not know the graphical structure.

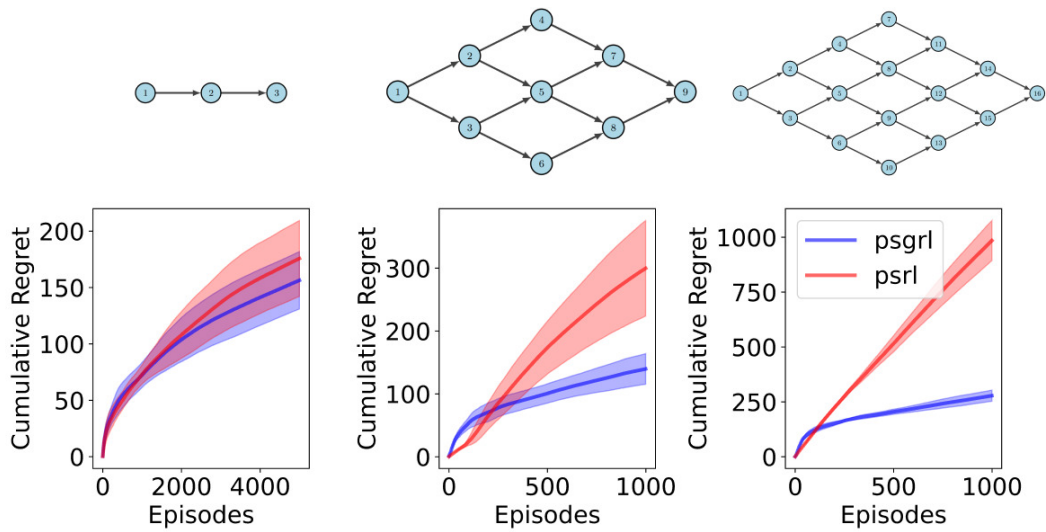


Fig. 5.3 The first row depicts the graph that governs the DAMDP. The second row shows the learning curve for both algorithms considered. PSRL ignores the latent graphical structure, and PSGRL leverages the graphical structure. The left-most plot shows the performance obtained on a simple chain graph. As expected, the performance for both algorithms is similar. Looking at the remaining plots, where we consider larger diamond-shaped graphs, it becomes clear that as we increase the complexity of the graph, the benefit of PSGRL becomes evident. To monitor the evolution of the regret, we ran ten different seeds; the solid line represents the mean regret while the shaded area covers $\pm$ one standard deviation.

PSGRL representation: At each time step t , we observe the set of *atomic states* $\{x_t^v\}_{v=1}^{n_t}$, which represent the amount of flow into node v of layer t . For each node, the agent decides how to distribute the current flow along its edges. For each node, the *atomic action* space consists of the possible allocation of flow along its edges; the action space is discrete and represents a possible fraction of the total flow. The atomic reward is then the amount of flow a specific node sends to the next layer of the graph.

PSRL representation: The *full state* representation at time step t , s_t is a vector representing the amount of flow in each node in layer t . The *full action* representation is the flow assignment for all edges connecting nodes from layer t to layer $t + 1$. The reward at time step t is the total amount of flow the agent managed to move from layer t to layer $t + 1$.

Figure 5.3 compares the performances of PSGRL and PSRL on the leaky maximum flow problem. The first row of Figure 5.3 depicts the DAMDP considered, while the second row shows the cumulated regret incurred by PSGRL (in blue) and PSRL (in red). The uncertainty estimates around the curve are obtained by running the experiment with ten seeds. We present in Figure 5.4 another series of experiments that highlight similar performance results on a different family of graphs.

The leftmost DAMDP is a simple chain graph. For this specific DAMDP instance, the *atomic* representation and the *full state* representation are equivalent. This explains the similar performance of PSGRL and PSRL. In contrast, considering more complex DAMDP instances, the benefit of leveraging the latent graphical structure becomes more apparent. Breaking down the combinatorial nature of the *full state* space and *full action* space using the DAG structure is beneficial. This is even more obvious on the rightmost plot, which shows the most complex leaky maximum flow instance considered.

Remark 5.7.1. Even for a simple example like the leaky maximum flow problem, it is interesting to analyse how the Bayesian regret bound for PSGRL equation 5.13 compares to the RL lower bound equation 5.14. The diamond-shaped graph with 16 nodes in the rightmost

part of Figure 5.3 implies that the DAMDP is parameterized by the following quantities. The number of distinct atomic action spaces is $U_r = 2$ as some nodes can distribute their flow in a single edge while others can distribute it in two different edges. We observe that there are 6 nodes with a single outgoing edge, hence $m_r^1 = 6$ and 9 nodes with two outgoing edges, $m_r^2 = 9$. The number of transition equivalence classes is $U_\tau = 2$ as some nodes have two incoming edges while others have a single incoming edge. We observe that there are 6 nodes with a single incoming edge (i.e. $m_\tau^1 = 6$) and 9 with two incoming edges (i.e. $m_\tau^2 = 9$). We also note that the maximum number of nodes in a layer is $N_{max} = 4$, and the largest number of edges between two adjacent layers is $M_{max} = 6$. In this specific context, the regret upper bound of PSGRL can be expressed as follows:

$$\mathbb{E}[\text{Regret}(K, \text{PSGRL})] = \tilde{O}\left(\sum_{j=1}^{U_r} H \sqrt{2XY_j m_r^j K} + \sum_{i=1}^{U_\tau} H \sqrt{2(X\bar{Y})^{d_i} m_\tau^i X K}\right) \quad (5.16)$$

$$= \tilde{O}\left(H \sqrt{12XY_1 K} + H \sqrt{18XY_2 K} + H \sqrt{12X^2\bar{Y} K} + H \sqrt{19X^3\bar{Y}^2 K}\right). \quad (5.17)$$

In orange, we have the regret associated with learning the transition function, and in blue, the regret associated with learning the reward function. Similarly, using the characteristics of this specific problem instance for any algorithm that does not leverage the latent graphical structure, such as PSRL, we obtain the following lower bound:

$$\mathbb{E}[\text{Regret}(K, \text{PSRL})] = \Omega\left(\sum_{h=1}^H \sqrt{S_t A_t T}\right) = \Omega\left(\sum_{h=1}^H \sqrt{(X\bar{Y})^{n_t} T}\right) \quad (5.18)$$

$$= 2\sqrt{(X\bar{Y})^1 T} + 2\sqrt{(X\bar{Y})^2 T} + 2\sqrt{(X\bar{Y})^3 T} + \sqrt{(X\bar{Y})^4 T}, \quad (5.19)$$

where $\bar{Y}$ denotes the smallest atomic action space.

In this simple example, the efficiency gain provided by PSGRL will be significant as long as $\sqrt{X^3\bar{Y}^2} < \sqrt{(X\bar{Y})^4}$ which is in general true, unless the discrepancy between the largest

and smallest action space becomes too large, that is, $\underline{Y} \ll \bar{Y}$. Additionally, this efficiency gain is reinforced as we increase the size of the graph. Considering that the number of nodes increases while the diamond-like architecture on Figure 5.3 is preserved, PSGRL's regret equation 5.16 scales linearly with the number of nodes, through m_τ^i and m_r^j . On the other hand, PSRL's lower bound on the regret scales exponentially with the number of nodes through n_t , the number of nodes at each layer of the graph.

In Figure 5.4, we present an additional set of leaky maximum flow experiments. We consider an alternative graph structure presented in the first row of Figure 5.4. Observing the respective performance of PSRL and PSGRL, we can see that as the graph complexity increases, the benefit of PSGRL increases as well. These results are similar to the experiment presented in Figure 5.3; we just revisit this experiment considering a different network architecture. In the leftmost case, we see no benefit in using PSGRL as we have a single node per layer. However, as we increase the central grid of nodes, creating multiple nodes that belong to the same equivalence class, the benefit of PSGRL becomes more evident.

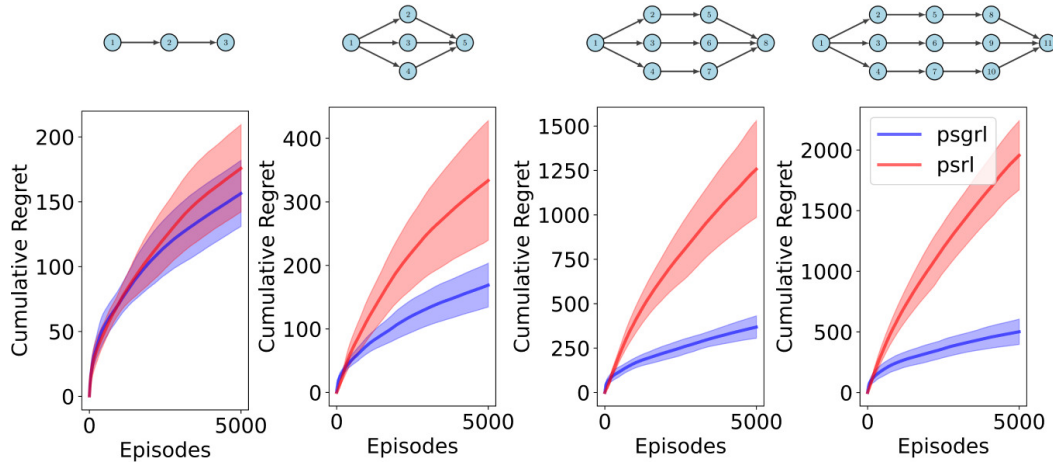


Fig. 5.4 The first row depicts the graph that governs the DAMDP. The second row shows the learning curve for both algorithms considered. PSRL, which ignores the latent graphical structure and PSGRL, which leverages the graphical structure. The left-most plot shows the performance obtained on a simple chain graph. As expected, the performance for both algorithms is similar. Looking at the remaining plots, it becomes clear that as we increase the complexity of the graph, the benefit of PSGRL becomes self-evident.

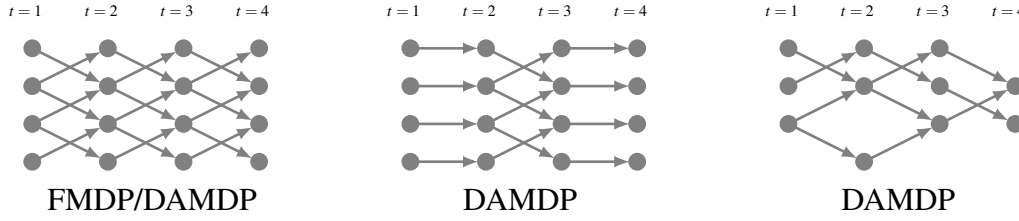


Fig. 5.5 This figure shows 3 examples of DAMDPs; the leftmost one has a constant connectivity pattern. Each layer has the same number of nodes, and the connectivity pattern from one layer to the next remains the same. MDPs that exhibit such a structure could be modelled by an FMDP or a DAMDP. On the contrary, the middle and the rightmost graphs could only be modelled by DAMDPs. The example depicted in the middle graph shows interactions that vary from one time step to the next, which cannot be modelled by an FMDP. The rightmost graph shows a graph where the number of nodes varies from time step to time step, which also cannot be modelled as an FMDP

5.8 Related work

The framework we consider in this chapter, the DAMDP model, is related to the factored MDP (FMDP) framework. While the first works on FMDP considered only a factorisation of the state space [64, 23, 55], the FMDP framework was later extended to consider factorisation over the state-action space [100, 28] - which in spirit is more similar to DAMDP. Indeed, FMDPs and DAMDPs share similar assumptions; they both consider the factorisation of the state and action spaces, the additivity of the reward function and the factorisation of the transition function. However, an important difference between the two frameworks is that FMDPs were initially proposed in a time-homogeneous setting, while DAMDPs allow the structure to vary as dictated by the graph and its connectivity.

In particular, DAMDP models a larger class of problems, as depicted in Figure 5.5; the encoding of the entire episode in a graph allows the representation of richer dynamics. The leftmost plot of Figure 5.5 shows a graph that exhibits the same connection patterns across all layers; such a problem could be modelled with FMDPs or DAMDPs. The remaining examples in the middle and on the right side of Figure 5.5 represent problems that cannot be directly modelled with FMDPs. The example in the middle of Figure 5.5 shows a scenario

where the connection dynamics change from one layer to the next, while the rightmost example shows a graph with a varying number of nodes and edges in each layer. Since DAMDPs are rolled out on an arbitrary directed acyclic graph, we can model each of the three graphs in Figure 5.5 with a DAMDP.

On the subset of DAMDP problems that can be modelled as FMDP, the regret obtained for the algorithm proposed in [100] is equivalent to the regret of our algorithm, PSGRL.

In general DAMDPs, a naive extension of the regret proposed [100] to the time inhomogeneous setting, is worse than the result obtained in Theorem 5.5.1. A naive extension of the FMDP framework to the time inhomogeneous setting, where the factored dynamics are encoded in graph G , would produce a regret bound driven by the number of nodes in the graph G . To be more specific, using our notation, the regret paid by the posterior sampling-based algorithm proposed in [100] to estimate the transition distribution of an FMDP is $\sum_{v=1}^{|V|} \mathcal{O}(H(X\bar{Y})^{d_v} + \sqrt{(X\bar{Y})^{d_v}XT})$, where $|V|$ represents the number of nodes in G and d_v is the number of incoming edges in node v . For the case of DAMDPs, the bound we propose in Theorem 5.5.1 is tighter as in equation 5.9 the size of the transition context $(X\bar{Y})^{d_i}$ is under the square root and, in general, $\sum_{v=1}^{|V|} (XY)^{d_v} \geq \sum_{i=1}^{U_\tau} \sqrt{(X\bar{Y})^{d_i}}$, as the first sum is over all nodes in the graph while the second is only over the U_τ equivalence classes. Additionally, this difference could be reinforced if we consider increasing the number of nodes $|V|$ while preserving a relatively small number of equivalence classes U_τ . For example, in the case of the diamond-shaped graph, the graph's architecture remains constant, and the only term that grows is m_t , the largest number of nodes in a single layer that has the same transition context.

In this work, we have assumed that the graphical structure is known. An interesting direction for future work would consist of learning the latent graphical structure. Structure discovery has been studied in the case of time-homogeneous FMDP [35, 127, 93], but these methods have not been extended to the time-inhomogeneous setting. Approaches like the

one considered in [93], where the proposed algorithm, $C - PSRL$, can learn the factored representation, could be extended to the time-inhomogeneous setting. Such an extension is not trivial as it would require adding another layer of inference in our proposed algorithm and treating the latent graph G as a random variable.

5.9 Proof of PSGRL Upper Bound on Regret

We follow the proof structure of [99], where the analysis focuses on a modified regret term that removes the dependency on μ^* and is equivalent to the original regret in expectation. For any episode k , we can write the modified regret as follows:

$$\tilde{\Delta}_k = \sum_{s_1 \in \mathcal{S}_1} \rho(s_1) (V_{\mu_{k,1}}^{M_{G,k}}(s_1) - V_{\mu_{k,1}}^{M_G^*}(s_1)). \quad (5.20)$$

where M_G^* is the true DAMDP and $M_{G,k}$ is the DAMDP sampled at episode k .

The equivalence in expectation between the original regret Δ_k and the modified regret $\tilde{\Delta}_k$ is possible thanks to the following lemma, which is first presented in [99, Lemma 1] and restated below in the DAMDP setting:

Lemma 5.9.1. *If f is the distribution of M_G^* and $M_{G,k} \sim f$ then, for any $\sigma(D_k)$ -measurable function g ,*

$$\mathbb{E}[g(M_G^*)|D_k] = \mathbb{E}[g(M_{G,k})|D_k]. \quad (5.21)$$

Where $\sigma(D_k)$ is the σ -algebra generated by all the data accumulated up to episode k , D_k . This allows the following equivalence between the two different regret terms,

$$\mathbb{E} \left[\sum_{k=1}^K \Delta_k \right] = \mathbb{E} \left[\sum_{k=1}^K \tilde{\Delta}_k \right].$$

Indeed, since $\Delta_k - \tilde{\Delta}_k = \sum_{s_1 \in \mathcal{S}_1} \rho(s_1)(V_{\mu_k,1}^{M_k}(s_1) - V_{\mu^*,1}^{M^*}(s_1))$, from Lemma 5.9.1, we get that $\mathbb{E}[\Delta_k - \tilde{\Delta}_k | D_k] = 0$. Finally, by the tower rule $\mathbb{E}[\Delta_k - \tilde{\Delta}_k] = \mathbb{E}[\mathbb{E}[\Delta_k - \tilde{\Delta}_k | D_k]] = 0$.

To lighten the notation we now write $V_{\mu_k,i}^k$ for $V_{\mu_k,i}^{M_k}$. Working with the modified regret $\tilde{\Delta}$ (presented in equation 5.20) allows us to rewrite the modified regret in terms of Bellman error,

$$\mathbb{E}[\tilde{\Delta}_k | M_G^*, M_{G,k}] = \mathbb{E} \left[\sum_{i=1}^H (\mathcal{T}_{\mu,i}^{M_{G,k}} - \mathcal{T}_{\mu,i}^{M_G^*}) V_{\mu,i+1}^k(s_{t_k+i}) \middle| M_G^*, M_{G,k} \right]. \quad (5.22)$$

To prove that equation 5.22 holds, we apply the Dynamic programming equation 5.10 inductively (note that we denote by $p_*^i(x|c)$ the atomic true transition distribution for nodes that belong to the equivalence class $[\tau_i]$):

$$\begin{aligned} (V_{\mu_k,1}^k - V_{\mu_k,1}^*)(s_{t_k+1}) &= (\mathcal{T}_{\mu_k,1}^k V_{\mu_k,2}^k - \mathcal{T}_{\mu_k,1}^* V_{\mu_k,2}^*)(s_{t_k+1}) \\ &= (\mathcal{T}_{\mu_k,1}^k - \mathcal{T}_{\mu_k,1}^*) V_{\mu_k,2}^k(s_{t_k+1}) + \sum_{s' \in \mathcal{S}_2} \prod_{v=1}^{n_2} p_*^{i_2,v}(x_2^v | c_2^v)(V_{\mu_k,2}^k - V_{\mu_k,2}^*)(s') \\ &= (\mathcal{T}_{\mu_k,1}^k - \mathcal{T}_{\mu_k,1}^*) V_{\mu_k,2}^k(s_{t_k+1}) + (V_{\mu_k,2}^k - V_{\mu_k,2}^*)(s_{t_k+2}) + d_{t_k+1} \\ &= \dots \\ &= \sum_{h=1}^H (\mathcal{T}_{\mu_k,h}^k - \mathcal{T}_{\mu_k,h}^*) V_{\mu_k,h+1}^k(s_{t_k+h}) + \sum_{i=1}^{\tau} d_{t_k+i} \end{aligned}$$

for

$$d_{t_k+h} = \sum_{s' \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} p_*^{i_{h,v}}(x_{h+1}^v | c_{h+1}^v)(V_{\mu_k,h+1}^k - V_{\mu_k,h+1}^*)(s') - (V_{\mu_k,h+1}^k - V_{\mu_k,h+1}^*)(s_{t_k+h+1}).$$

Since,

$$\mathbb{E}[(V_{\mu_k,h+1}^k - V_{\mu_k,h+1}^*)(s_{t_k+h+1}) | M_G^*, M_{G,k}] = \sum_{s' \in \mathcal{S}_{h+1}} \prod_{v=1}^{n_{h+1}} (p_*^{i_{h,v}}(x_{h+1}^v | c_{h+1}^v, \mu_k)(V_{\mu_k,h+1}^k - V_{\mu_k,h+1}^*)(s'))$$

the terms d_{t_k+h} disappear in expectation (i.e. $\mathbb{E}[d_{t_k+h} | M_G^*, M_{G,k}] = 0$ for all k and $h \in \{1, \dots, H\}$) and so we obtain equation 5.22.

Since the regret has been rewritten as the sum of one-step Bellman errors (see equation 5.22), the next step is to show that as interactions with the environment are observed, the sampled DAMDPs, $M_{G,k}$, concentrate around the true DAMDP M_G^* . This is done in the following subsection.

5.9.1 Confidence Sets

For the remainder of this section, the notation is a bit more involved. The below list introduces or recalls the concepts and the notation used.

1. $G = (V, E)$ is an LDAG that encodes the underlying structure in the execution of a DAMDP episode. We denote the i^{th} node of layer t by v_t^i . We recall that each layer $t = 1, \dots, H$ has n_t nodes. The graph has a total of $|V| = \sum_{t=1}^H n_t = n$ nodes.
2. If two nodes $v_{t_1}^{i_1}$ and $v_{t_2}^{i_2}$, with $(i_1, t_1) \neq (i_2, t_2)$ have the same number of parents and their parents have the same atomic action space, they belong to the same transition equivalence class $[\tau_i]$. We denote by d_i the number of parent (or incoming edges) of a node that belongs to equivalence class $[\tau_i]$. All nodes in G belong to one of the U_τ equivalence classes, $[\tau_i]_{i=1}^{U_\tau}$. The number of nodes in G that belong to the same equivalence class is denoted by m_τ^i , and $\sum_{i=1}^{U_\tau} m_\tau^i = n$. For example, in Fig. 5.1, there are two different equivalence classes, $U_\tau = 2$, because there exist nodes with two parent nodes or a single parent node.
3. Each equivalence class $[\tau_i]$ with $i \in \{1, \dots, U_\tau\}$, has a corresponding state-action space $\mathfrak{C}_i = \bigotimes_{k=1}^{d_i} (\mathcal{X} \times \mathcal{Y}_k)$ that contains all the possible values the transition context (of an element of the i^{th} equivalence class $[\tau_i]$) can take. Recall that the atomic action space might depend on the node, so two nodes belong to the same equivalence class if they have the same number of parent nodes with the same atomic action spaces. This space consists of the atomic states and the atomic action values of each parent of the node.

We can upper-bound the size of this context space by $(X\bar{Y})^{d_i}$, where d_i is the number of parent nodes in the i^{th} equivalence class, and $\bar{Y}$ is the size of the largest atomic action space.

4. Each equivalence class $[\tau_i]$ has a dedicated transition function $p^i(\cdot|c_i)$ for all $i \in \{1, \dots, U_\tau\}$ and all possible transition context $c_i \in \mathfrak{C}_i$.
5. U_r is the number of distinct atomic action sets. For example, in the case of the leaky maximum flow problem in Section 5.7 $U_r = 2$, the atomic action space is different if a node has a single out-going edge or two out-going edges. In the wind farm optimisation problem $U_r = 1$, regardless of the number of outgoing edges, the atomic action space of a node remains the same. The number of nodes in G that have the atomic action set j is denoted by m_r^j , for all $j \in \{1, \dots, U_r\}$. Then, $\sum_{j=1}^{U_r} m_r^j = n$, where n is the number of nodes in G .
6. For each distinct atomic action set $\{\mathcal{Y}\}_{j=1}^{U_r}$, we call the atomic reward context the associated atomic state-action space $\mathcal{Z}_j = \mathcal{X} \times \mathcal{Y}_j$. We denote by $z_j \in \mathcal{Z}_j$ an element of the reward context j .
7. For each atomic action set $\{\mathcal{Y}_j\}_{j=1}^{U_r}$ we define a corresponding atomic reward function $r^j(z_j)$.
8. Define $N_{t_k}^{\tau,i}(c)$ to count the number of times a node of the equivalence class $[\tau_i]$ has observed the transition context $c \in \mathfrak{C}_i$ during the first t_k time steps, where t_k indicates the time step at which episode k starts (i.e. $t_k = (k-1)*H + 1$). Sometimes, we are interested in monitoring the number of visits more closely. Then, $N_{t_k,v}^{\tau,i}(c)$ counts the number of time the transition context $c \in \mathfrak{C}_i$ given that the agent started episode k and already observed the v first nodes of G , with $v \in \{1, \dots, n\}$. Nodes are indexed by layers, but the indexing within a layer is arbitrary and fixed before the learning

starts. In Figure 5.1 leftmost plot, we show an example of an LDAG with a valid node indexing.

9. Define $N_{t_k}^{r,j}(z)$ to count the number of times a node has observed the reward context $z \in \mathcal{Z}_j$ during the first t_k time steps for all $j \in [U_r]$ and for all $z \in \mathcal{Z}_j$. $N_{t_k,v}^{\tau,i}(z)$ counts the number of times the reward context z was observed until the v^{th} node of the execution of the k^{th} episode, where the node indexing is the same as the one considered for $N_{t_k,v}^{\tau,i}(c)$ (see entry 8 of this list for more details).

The posterior sampling algorithm proceeds by sampling an atomic reward function for each possible action set $\{\mathcal{Y}_j\}_{j=1}^{U_r}$ and a transition function for each equivalence class $[\tau_i]$ for all $i \in \{1, \dots, U_\tau\}$. The atomic transition distributions and the atomic reward distributions sampled at the beginning of episode k (step 3. in Alg. 11) are denoted $\{p_k^i\}_{i=1}^{U_\tau}$ and $\{r_k^j\}_{j=1}^{U_r}$, respectively. The sampled DAMDP is then constructed by combining the atomic transition and reward function as dictated by the graph G . To show that the sampled DAMDPs, $M_{G,k}$, concentrate around the true DAMDP M_G^* , we construct confidence intervals around the empirical atomic transition and reward functions. We define the confidence set of feasible DAMDPs that we may sample at episode k as:

$$\begin{aligned} \mathcal{M}_G^k = \{M : & \|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\|_1 \leq \beta_k^i(c) \forall c \in \mathcal{C}_i, \forall i \in [U_\tau] \text{ \& } \\ & \|\hat{r}_k^j(z) - r^j(z)\| \leq \gamma_k^j(z) \forall z \in \mathcal{Z}_j \forall j \in [U_r]\}. \end{aligned} \quad (5.23)$$

Where $\hat{p}_k^i$ denotes the empirical estimate at time step k of the i^{th} atomic transition function, $\hat{r}_k^j$ denotes the empirical estimate at time step k of the j^{th} reward function, $\beta_k^i(c)$ is the threshold for the error on the i^{th} transition function, and $\gamma_k^j(z)$ is the error threshold for the error on the j^{th} atomic reward function. We propose the following definition for the error

thresholds:

$$\beta_k^i(c) = \sqrt{\frac{2}{N_{t_k}^{\tau,i}(c)} \left(X \log \left(\frac{2}{\delta} \right) + 2d_i \log \left(\frac{X\bar{Y}m_t^i k}{\delta} \right) \right)} \quad \forall i \in \{1, \dots, U_\tau\} \text{ and } \forall c \in \mathfrak{C}_i \quad (5.24)$$

$$\gamma_k^j(z) = \sqrt{\frac{\log \left(\frac{X\bar{Y}m_r^j k}{\delta} \right)}{\max(1, N_{t_k}^{r,j}(z))}} \quad \forall j \in \{1, \dots, U_r\} \text{ and } \forall z \in \mathcal{Z}_j, \quad (5.25)$$

where $N_{t_k}^{r,j}(z)$ and $N_{t_k}^{\tau,i}(c)$ are defined above (elements 8 and 9 in the list).

Analysis of the Confidence Sets

Lemma 5.9.2. *For any $k \geq 1$, the true MDP M_G^* belongs to the confidence set at episode k , $\mathcal{M}_G^k$, defined in equation 5.23 with probability:*

$$\mathbb{P}(M_G^* \notin \mathcal{M}_G^k) \leq \delta/k.$$

Proof. The L_1 deviation between the true atomic transition distribution $p^i(\cdot|c)$ and its empirical estimate $\hat{p}_k^i(\cdot|c)$ is bounded for any ε by [145]:

$$\mathbb{P} \left(\|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\|_1 \geq \varepsilon \right) \leq (2^X - 2) \exp \left(-\frac{n\varepsilon^2}{2} \right) \quad \forall i \in \{1, \dots, U_\tau\} \text{ and } \forall c \in \mathfrak{C}_i, \quad (5.26)$$

where X is the number of distinct outcomes of $p(\cdot|c)$ and n is a fixed number of samples.

We now define

$$\varepsilon_\tau^i = \sqrt{\frac{2}{n} \log \left(\frac{2^X 2U_\tau (X\bar{Y})^{d_i} m_t^i k^2}{\delta} \right)}$$

where

$$\varepsilon_\tau^i \leq \sqrt{\frac{2}{n} \left(X \log \left(\frac{2}{\delta} \right) + 2d_i \log \left(\frac{2U_\tau X\bar{Y}m_t^i k}{\delta} \right) \right)}$$

This upper bound on ϵ_τ^i gives the exact expression of $\beta_k^i(c)$ defined in equation 5.24.

We can, therefore, show that the confidence bound in equation 5.23 holds with high probability. In particular, for the atomic transition probability of a given context $c \in \mathfrak{C}_i$ and fixed number of visits n we have:

$$\begin{aligned}
 & \mathbb{P} \left(\|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\|_1 \geq \beta_k^i(c) \right) \\
 &= \mathbb{P} \left(\|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\|_1 \geq \sqrt{\frac{2}{n} \left(X \log \left(\frac{2}{\delta} \right) + 2d_i \log \left(\frac{2U_\tau X \bar{Y} m_t^i k}{\delta} \right) \right)} \right) \\
 &\leq 2^X \exp \left(-\frac{n}{2} (\epsilon_\tau^i)^2 \right) \\
 &= 2^X \exp \left(-\frac{n}{2} \frac{2}{n} \log \left(\frac{2^X 2U_\tau (X \bar{Y})^{d_i} m_t^i k^2}{\delta} \right) \right) \\
 &= \frac{\delta}{2U_\tau (X \bar{Y})^{d_i} m_t^i k^2}.
 \end{aligned}$$

Now, with a fixed number of samples n , the deviation between the true atomic expected reward and the empirical atomic mean reward is bounded by Hoeffding's inequality for any $\epsilon_r^j > 0$:

$$\mathbb{P} \left(|\hat{r}_k^j(z) - r^j(z)| \geq \epsilon_r^j \right) \leq 2 \exp(-2n(\epsilon_r^j)^2) \quad \forall j \in \{1, \dots, U_r\} \text{ and } \forall z \in \mathcal{Z}_j. \quad (5.27)$$

We define

$$\epsilon_r^j = \sqrt{\frac{1}{2n} \log \left(\frac{4U_r X \bar{Y} m_r^j k^2}{\delta} \right)} \quad \text{and note that} \quad \epsilon_r^j \leq \sqrt{\frac{1}{n} \log \left(\frac{4U_r X \bar{Y} m_r^j k}{\delta} \right)}.$$

Again, this upper bound on ϵ_r^j will be used to determine the exact expression of $\gamma_k^j(z)$.

Combining Hoeffdings inequality with our definition of ε_r^j , we get:

$$\mathbb{P}\left(|\hat{r}_k^j(z) - r^j(z)| \geq \gamma_k^j(z)\right) = \mathbb{P}\left(|\hat{r}_k^j(z) - r^j(z)| \geq \sqrt{\frac{1}{n} \log\left(\frac{4U_r X \bar{Y} m_r^j k}{\delta}\right)}\right) \quad (5.28)$$

$$\leq 2 \exp(-2n(\varepsilon_r^j)^2) \quad (5.29)$$

$$= 2 \exp\left(-\frac{2}{n} \log\left(\frac{4U_r X \bar{Y} m_r^j k^2}{\delta}\right)\right) \quad (5.30)$$

$$= \frac{\delta}{2U_r X \bar{Y} m_r^j k^2} \quad (5.31)$$

To compute the probability of interest $\mathbb{P}(M_G^* \notin \mathcal{M}_G^k)$, we need to compute the union bound by summing over all possible numbers of visits. It is important to note that in the same layer, more than one node might observe a reward for the same *atomic state-action* pair $z = (x, y)$. It is also possible that in the same layer, more than one node observes the same transition context c . As a consequence, the number of observations can increase by more than a unit per time step. If we perform k episodes, the number of visits for a specific reward context j might range between 0 and km_r^j , where m_r^j is the number of occurrences of reward architecture j in G (see element 5 of the above list for a more complete definition). Similarly, there are m_τ^i nodes that belong to the equivalence class $[\tau_i]$, so a given transition context c_i can be observed up to km_τ^i times (see element 2 of the above list for a more complete definition). Now, computing the union bound by summing over all possible numbers of visits, we obtain the following:

$$\mathbb{P}\left(|\hat{r}_k^j(z) - r^j(z)| \geq \sqrt{\frac{\log\left(\frac{4U_r X \bar{Y} m_r^j k}{\delta}\right)}{\max(1, N_{t_k}^{r,j}(z_j))}}\right) \leq \sum_{n=1}^{m_r^j k} \frac{\delta}{2U_r X \bar{Y} m_r^j k^2} < \frac{\delta}{2U_r X \bar{Y} k} \quad (5.32)$$

$$\begin{aligned}
& \mathbb{P}\left(\|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\|_1 \geq \sqrt{\frac{2}{\max(1, N_{t_k}^{\tau,i}(c))} \left(X \log\left(\frac{2}{\delta}\right) + 2d_i \log\left(\frac{2U_\tau X \bar{Y} m_t^i k}{\delta}\right)\right)}\right) \\
& \leq \sum_{n=1}^{m_t^i k} \frac{\delta}{2U_\tau (XY)^{d_i} m_t^i k^2} < \frac{\delta}{2U_\tau (X\bar{Y})^{d_i} k}
\end{aligned}$$

This allows us to define $\beta_k^i(c)$ (introduced in equation 5.23) for all $i \in \{1, \dots, U_\tau\}$ and for all $c \in \mathfrak{C}_i$:

$$\beta_k^i(c) := \sqrt{\frac{2}{\max(1, N_{t_k}^{\tau,i}(c))} \left(X \log\left(\frac{2}{\delta}\right) + 2d_i \log\left(\frac{2U_\tau X \bar{Y} m_t^i k}{\delta}\right)\right)} \quad (5.33)$$

and $\gamma_k^j(z)$ for all $j \in \{1, \dots, U_r\}$ and for all $z \in \mathcal{Z}_i$, as

$$\gamma_k^j(z) := \sqrt{\frac{\log\left(\frac{4U_r X \bar{Y} m_t^j k}{\delta}\right)}{\max(1, N_{t_k}^{r,j}(z))}}. \quad (5.34)$$

Summing the probabilities over all possible reward contexts $z \in \mathcal{Z}_j$ (the number of which is bounded by $X\bar{Y}$) and transition context $c \in \mathfrak{C}_i$ (the number of which is bounded by $(X\bar{Y})^{d_i}$) we get

$$\mathbb{P}(M_G^* \notin \mathcal{M}_G^k) \quad (5.35)$$

$$\begin{aligned}
& = \sum_{j=1}^{U_r} \sum_{z \in \mathcal{Z}_j} \mathbb{P}(|\hat{r}_k^j(z) - r^j(z)| < \gamma_k^j(z)) + \sum_{i=1}^{U_\tau} \sum_{c \in \mathfrak{C}_i} \mathbb{P}(\|\hat{p}_k^i(\cdot|c) - p^i(\cdot|c)\| < \beta_k^i(c)) \\
& \quad (5.36)
\end{aligned}$$

$$\leq \sum_{j=1}^{U_r} \sum_{z \in \mathcal{Z}_j} \frac{\delta}{2U_r X \bar{Y} k} + \sum_{i=1}^{U_\tau} \sum_{c \in \mathfrak{C}_i} \frac{\delta}{2U_\tau (X\bar{Y})^{d_i} k} \leq \frac{\delta}{k} \quad (5.37)$$

as desired. $\square$

5.9.2 Upper Bounding the Regret

The regret can be upper bounded by the sum of the errors made at each node.

Lemma 5.9.3. *If $M_{G,k}$ and M_G^* belong to the confidence set at time step k , $\mathcal{M}_G^k$, then the one-step Bellman error is upper bounded by the size of the confidence interval:*

$$|(\mathcal{T}_{\mu_k,h}^k - \mathcal{T}_{\mu_k,h}^*)V_{\mu_k,h+1}^k(s_{h+1})| \leq H \min\left\{\sum_{v=1}^n \beta_k^{i_v}(c_k^v) + \gamma_k^{j_v}(z_k^v), 1\right\} \quad (5.38)$$

Proof. Error in the atomic Bellman operator has the most impact if the value function is high; we can then directly upper bound the value function with H .

$$|(\mathcal{T}_{\mu_k,h}^k - \mathcal{T}_{\mu_k,h}^*)V_{\mu_k,h+1}^k(s_{h+1})| \leq H |(\mathcal{T}_{\mu_k,h}^k - \mathcal{T}_{\mu_k,h}^*)| \quad (5.39)$$

$$\leq H \left(\left| \sum_{v=1}^{n_h} r_k^{j_v}(x_k^v | c_{h,\mu}^v) + \prod_{v=1}^{n_{h+1}} p_k^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) - \sum_{v=1}^{n_h} r_*^{j_v}(x_k^v | c_{h,\mu}^v) + \prod_{v=1}^{n_{h+1}} p_*^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) \right| \right) \quad (5.40)$$

$$\leq H \left(\left| \sum_{v=1}^{n_h} r_k^{j_v}(x_k^v | c_{h,\mu}^v) - \sum_{v=1}^{n_h} r_*^{j_v}(x_k^v | c_{h,\mu}^v) \right| + \left| \prod_{v=1}^{n_{h+1}} p_k^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) - \prod_{v=1}^{n_{h+1}} p_*^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) \right| \right) \quad (5.41)$$

$$\leq H \left(\left| \sum_{v=1}^{n_h} r_k^{j_v}(x_k^v | c_{h,\mu}^v) - \sum_{v=1}^{n_h} r_*^{j_v}(x_k^v | c_{h,\mu}^v) \right| + \sum_{v=1}^{n_{h+1}} \left| p_k^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) - p_*^{i_v}(x_{h+1}^v | c_{h+1,\mu}^v) \right| \right) \quad (5.42)$$

$$\leq H \min \left\{ \sum_{v=1}^{n_h} \gamma_k^{j_v}(z_{h,\mu}^v) + \beta_k^{i_v}(c_{h+1,\mu}^v), 1 \right\} \quad (5.43)$$

Equation 5.40 is directly obtained using the definition of the atomic Bellman operator equation 5.10. Then equation 5.41 is obtained by grouping together the terms depending on the atomic reward functions and the one depending on the atomic transition functions. To

obtain equation 5.42 we use the following inequality:

$$\prod_{i=1}^n a_i - \prod_{i=1}^n b_i \leq \sum_{i=1}^n |a_i - b_i|, \quad \forall a_i, b_i \leq 1.$$

Finally, in equation 5.43, we upper-bound the error in the transition and reward function at each node with the width of the corresponding confidence interval. Note that since the reward at each time step is bounded in $[0, 1]$, the sum of all the nodes' errors in a given layer is upper-bounded by 1. $\square$

With this upper bound on the Bellman error, we are now equipped to upper-bound the regret; we first decompose it as follows, observing that $\tilde{\Delta}_k \leq H$ for any $k \in \{1, \dots, K\}$. Hence,

$$\sum_{k=1}^K \tilde{\Delta}_k \leq \sum_{k=1}^K \tilde{\Delta}_k \mathbb{I}\{M_G^k, M_G^* \in \mathcal{M}_G^k\} + H \sum_{k=1}^K [\mathbb{I}\{M_G^* \notin \mathcal{M}_G^k\} + \mathbb{I}\{M_G^k \notin \mathcal{M}_G^k\}].$$

By Lemma 5.9.1, $\mathbb{E}[\mathbb{I}\{M_{G,k} \notin \mathcal{M}_G^k\} | D_{t_k}] = \mathbb{E}[\mathbb{I}\{M_G^* \notin \mathcal{M}_G^k\} | D_{t_k}]$. Additionally, setting $\delta = \frac{1}{K}$ in Lemma 5.9.2 shows the true DAMDP M_G^* does not belong to the confidence set at time step k , $\mathcal{M}_G^k$, with probability $\mathbb{P}(M_G^* \notin \mathcal{M}_G^k) < \frac{1}{K}$. Then,

$$\mathbb{E}\left[\sum_{k=1}^K \tilde{\Delta}_k\right] \leq \mathbb{E}\left[\sum_{k=1}^K \tilde{\Delta}_k \mathbb{I}\{M_{G,k}, M_G^* \in \mathcal{M}_G^k\}\right] + 2H \sum_{k=1}^K \mathbb{P}(M^* \notin \mathcal{M}_k) \quad (5.44)$$

$$\leq \mathbb{E}\left[\sum_{k=1}^K \mathbb{E}[\tilde{\Delta}_k | M_G^*, M_{G,k}] \mathbb{I}\{M_{G,k}, M_G^* \in \mathcal{M}_G^k\}\right] + 2H \quad (5.45)$$

$$\leq \mathbb{E}\left[\sum_{k=1}^K \sum_{h=1}^H |(\mathcal{T}_{\mu_k, h}^k - \mathcal{T}_{\mu_k, h}^*) V_{\mu_k, h+1}^k(s_{t_k+1})| \mathbb{I}\{M_{G,k}, M_G^* \in \mathcal{M}_G^k\}\right] + 2H \quad (5.46)$$

$$\leq H \mathbb{E}\left[\sum_{k=1}^K \sum_{h=1}^H \min\left\{\sum_{v=1}^{n_h} \beta_k^{i_v}(c_k^v) + \gamma_k^{i_v}(z_k^v), 1\right\}\right] + 2H, \quad (5.47)$$

where Eq. 5.45 is obtained by applying the tower property and noting that $\mathcal{M}_G^k$ is measurable if $M_{G,k}$ is known. Equation 5.46 is the definition of the modified regret term (see eq. 5.20). Finally, equation 5.47 is a direct consequence of Lemma 5.9.3. We also denote by n the number of nodes in G ; we use i_v to denote the equivalence class of the node v and j_v to denote the atomic action set available at node v . The transition context observed during episode k at node v is denoted by $c_k^v \in \mathfrak{C}_{i_v}$, and the reward context observed at node v during the k^{th} episode is denoted by $z_k^v \in \mathcal{Z}_{j_v}$.

The contribution to the regret incurred by errors in the reward function r_k^j can be upper-bounded by the sum of γ_k^j for every episode and every node $v \in V_j$, where V_j is the set of nodes that exhibit the reward architecture j . Then, for a single reward architecture j we have:

$$\begin{aligned} \sum_{k=1}^K \sum_{v \in V_j} \gamma_k^j(z_k^v) &= \sum_{k=1}^K \sum_{v \in V_j} \gamma_k^j(z_k^v) \mathbb{I}\{N_{t_k}^{r,j}(z_k^v) \leq m_r^j\} + \sum_{k=1}^K \sum_{v \in V_j} \gamma_k^j(z_k^v) \mathbb{I}\{N_{t_k}^{r,j}(z_k^v) > m_r^j\} \\ &\leq \sum_{k=1}^K \sum_{v \in V_j} \mathbb{I}\{N_{t_k}^{r,j}(z_k^v) \leq m_r^j\} + \sum_{k=1}^K \sum_{v \in V_j} \mathbb{I}\{N_{t_k}^{r,j}(z_k^v) > m_r^j\} \sqrt{\frac{\log\left(\frac{4U_r X \bar{Y} m_r^j k}{\delta}\right)}{\max(1, N_{t_k}^{r,j}(z_k^v))}}. \end{aligned}$$

Where $z_k^v \in \mathcal{Z}_j$ is used to denote the reward context observed in the v^{th} node of G during the k^{th} episode.

Consider a fixed $z_k^v = z$. The case where $N_{t_k}^{r,j}(z) \leq m_r^j$ happens less than $2m_r^j$ times for each $z \in \mathcal{Z}_j$ can be upper bounded as follows $\sum_{k=1}^K \sum_{v \in V_j} \mathbb{I}\{N_{t_k}^{r,j}(z) \leq m_r^j\} \leq 2m_r^j X \bar{Y}$. Now, let's suppose that $N_{t_k}^{r,j}(z) > m_r^j$ for a $z \in \mathcal{Z}_j$. In the k^{th} episode, for any node $v \in V_j$, we have $N_{t_k,n}^{r,j}(z) + 1 \leq N_{t_k}^{r,j}(z) + m_r^j \leq 2N_{t_k}^{r,j}(z)$. Note that the number of occurrences of a specific reward architecture in each layer depends on G , but we know that it occurs m_r^j times over an episode.

We can then bound the following ratio:

$$\sum_{k=1}^K \sum_{v \in V_j} \sqrt{\frac{\mathbb{I}\{N_{t_k}^{r,j}(z_{k,v}) > m_r^j\}}{N_{t_k}^{r,j}(z_{k,v})}} \leq \sum_{k=1}^K \sum_{v \in V_j} \sqrt{\frac{2}{N_{t_k,v}^{r,j}(z_{k,v}) + 1}} \quad (5.48)$$

$$= \sqrt{2} \sum_{t=1}^{Km_r^j} (N_t^{r,j}(z_t) + 1)^{-1/2} \quad (5.49)$$

$$\leq \sqrt{2} \sum_{z \in \mathcal{Z}_j} \sum_{b=1}^{N_{T_j}^{r,j}(z)} b^{-1/2} \quad (\text{with } T_j = Km_r^j + 1) \quad (5.50)$$

$$\leq \sqrt{2} \sum_{z \in \mathcal{Z}_j} \int_0^{N_{T_j}^{r,j}(z)} x^{-1/2} dx \quad (5.51)$$

$$\leq \sqrt{2X\bar{Y} \sum_{z \in \mathcal{Z}_j} N_{T_j}^{r,j}(z)} = \sqrt{2X\bar{Y}Km_r^j}. \quad (5.52)$$

Equation 5.49 rewrites the two sums as a single one, where now, each index t in $N_t^{r,j}$ uniquely encodes a pair (t_k, n) . We obtain equation 5.50 by considering all possible reward contexts $z \in \mathcal{Z}_j$ and the total number of times this specific reward context was visited. In equation 5.51 we use the fact that the sum $\sum_{i=n+1}^N x^{-1/2}$ is upper bounded by $\int_n^N x^{-1/2} dx$. Lastly, equation 5.52 is obtained by Cauchy-Schwartz inequality.

With the same approach, we can bound the regret incurred by the error in the i^{th} transition function,

$$\sum_{k=1}^K \sum_{v \in V_i} \beta_k^i(c_k^v) \leq \sum_{k=1}^K \sum_{v \in V_i} \mathbb{I}\{N_{t_k}^{\tau,i}(c_k^v) \leq m_\tau^i\} + \sum_{k=1}^K \sum_{v \in V_i} \mathbb{I}\{N_{t_k}^{\tau,i}(c_k^v) > m_\tau^i\} \beta_k^i(c_k^v),$$

where V_i denotes all the nodes of the equivalence class $[\tau_i]$. For any $c \in \mathcal{C}_i$, the case where $N_{t_k}^{\tau,i}(c) \leq m_\tau^i$ happens less than $2m_\tau^i$ times. Given that $|\mathcal{C}_i| \leq (X\bar{Y})^{d_i}$ we can upper-bound $\sum_{k=1}^K \sum_{v \in V_i} \mathbb{I}\{N_{t_k}^{\tau,i}(c) \leq m_\tau^i\} \leq 2m_\tau^i (X\bar{Y})^{d_i}$. For a given transition context c and episode k when $N_{t_k}^{\tau,i} > m_\tau^i$, we have that for any node $V \in V_i$, $N_{t_k,v}^{\tau,i}(c) + 1 \leq N_{t_k}^{\tau,i}(c) + m_\tau^i \leq 2N_{t_k}^{\tau,i}(c)$.

We can then bound the following ratio:

$$\sum_{k=1}^K \sum_{v \in V_i} \sqrt{\frac{\mathbb{I}\{N_{t_k}^{\tau,i}(c_k^v) > m_\tau^i\}}{N_{t_k}^{\tau,i}(c_k^v)}} \leq \sum_{k=1}^K \sum_{v \in V_i} \sqrt{\frac{2}{N_{t_k,v}^{\tau,i}(c_k^v) + 1}} \quad (5.53)$$

$$= \sqrt{2} \sum_{t=1}^{Km_\tau^i} (N_t^{\tau,i}(c_t) + 1)^{-1/2} \quad (5.54)$$

$$\leq \sqrt{2} \sum_{c \in \mathfrak{C}_i} \sum_{b=1}^{N_{T_i}^{\tau,i}(c)} b^{-1/2} \quad (\text{with } T_i = Km_\tau^i + 1) \quad (5.55)$$

$$\leq \sqrt{2} \sum_{c \in \mathfrak{C}_i} \int_0^{N_{T_i}^{\tau,i}(c)} x^{-1/2} dx \quad (5.56)$$

$$\leq \sqrt{2(X\bar{Y})^{d_i} \sum_{c \in \mathfrak{C}_i} N_{T_i}^{\tau,i}(c)} = \sqrt{2(X\bar{Y})^{d_i} Km_\tau^i}. \quad (5.57)$$

All steps here are obtained following the same reasoning as in equation 5.48 to equation 5.52.

Because the rewards at every time step (or layer) are bounded in $[0, 1]$, the total regret consists of the following:

$$\min \left\{ H \sum_{k=1}^K \sum_{h=1}^H \min \left\{ \sum_{v=1}^{n_h} \gamma_k^{i_v}(z_k^v) + \beta_k^{i_v}(c_k^v), 1 \right\}, T \right\} \quad (5.58)$$

$$\begin{aligned} &\leq \min \left\{ H \sum_{j=1}^{U_r} 2m_r^j X\bar{Y} + \sqrt{2X\bar{Y} Km_r^j \log(4U_r X\bar{Y} m_r^j K)} \right. \\ &\quad \left. + H \sum_{i=1}^{U_\tau} 2m_\tau^i (X\bar{Y})^{d_i} + \sqrt{2(X\bar{Y})^{d_i} Km_\tau^i (X \log(\frac{2}{\delta} + 2d_i \log(2U_\tau X\bar{Y} m_\tau^i)))}, KH \right\} \end{aligned} \quad (5.59)$$

$$\begin{aligned} &\leq \min \left\{ H \sum_{j=1}^{U_r} 2m_r^j X\bar{Y} + \sqrt{2X\bar{Y} Km_r^j \log(4U_r X\bar{Y} m_r^j K)}, KH \right\} \\ &\quad + \min \left\{ H \sum_{i=1}^{U_\tau} 2m_\tau^i (X\bar{Y})^{d_i} + \sqrt{2(X\bar{Y})^{d_i} Km_\tau^i (X \log(\frac{2}{\delta} + 2d_i \log(2U_\tau X\bar{Y} m_\tau^i)))}, KH \right\} \end{aligned} \quad (5.60)$$

To complete the proof, we note that $\min(a + b, c) \leq \sqrt{ac} + b$ holds for $a, b, c > 0$. We then apply this inequality twice in equation 5.60.

$$\begin{aligned}
\text{Equation 5.60} &\leq \sqrt{KH \sum_{j=1}^{U_r} 2m_r^j X \bar{Y} H} + \sum_{j=1}^{U_r} \sqrt{2X \bar{Y} K m_r^j \log(4U_r X \bar{Y} m_r^j K) H} \\
&\quad + \sqrt{KH \sum_{i=1}^{U_\tau} 2m_\tau^i (X \bar{Y})^{d_i} H} \\
&\quad + \sum_{i=1}^{U_\tau} \sqrt{2(X \bar{Y})^{d_i} H K m_\tau^i (X + 2d_i \log(2U_\tau X \bar{Y} m_\tau^i K))} \\
&\leq H \sum_{j=1}^{U_r} \sqrt{2X \bar{Y} K m_r^j \log(4U_r X \bar{Y} m_r^j K)} \\
&\quad + \sum_{i=1}^{U_\tau} H \sqrt{2(X \bar{Y})^{d_i} K m_\tau^i (X + 2d_i \log(2U_\tau X \bar{Y} m_\tau^i K))}
\end{aligned}$$

This gives us the result obtained in equation 5.13.

5.10 Conclusion

This chapter formalises a new subclass of MDPs with an underlying graphical structure, DAMDPs. It also introduces a posterior sampling-based algorithm, PSGRL, that leverages this latent graphical structure to learn an optimal policy efficiently. We also provide evidence that PSGRL outperforms PSRL in theory and practice whenever this graphical structure is present. Although the structure of DAMDPs may appear restrictive, we argue that it does appear in many practical settings. Finding the maximum flow of a graph is a common framework for many real-world problems, such as network routing [85] and supply chain management [119]. Additionally, in the next chapter, we provide strong evidence that this latent graphical representation is present in wind farm yield optimisation. However, immediate steps can be taken to extend our method's applicability further and broaden

the range of problems it can tackle. First, the proposed framework could be extended to accommodate a broader range of real-world problems to allow for continuous state and action. Quantifying the benefit of the graphical factorisation in such a context remains of great interest. Second, we could add edge attributes, for example, if we revisit the maximum leaky flow problem presented in Section 5.7. We could imagine that pipes are not built with the same material; hence, they do not have the same probability of leaking or that the longer the pipe is, the more likely it is to leak; adding additional attributes to the edge would allow us to consider a richer set of transition dynamics and to model real-world problems more realistically. Such an increase in complexity will impact the upper bound on the regret but would offer more flexibility in the modelling. We leave these endeavours for future work as, for the time being, our graph-based approach renders a potentially large subset of real-world MDPs amenable to more efficient training.

Chapter 6

Real-World Example: Wind Farm Yield Optimisation

In this chapter, we present and motivate the problem of optimising the yield of a wind farm. Critically, we aim to validate that a structure similar to the one considered in Chapter 5 manifests itself in this real-world inspired problem and that this structure can be learned. To that end, we consider a prediction task: given a complete description of the farm and the wind conditions, a transformer model predicts the yield of the entire farm. The transformer model proposed for this task not only performs extremely well but also identifies the potential interactions between the wind turbines within the farm, allowing us to establish that the latent graphical structure studied in Chapter 5 appears in this real-world scenario.

The work presented¹ in this chapter was published as Li, S., Robert, A., Faisal, A. A., & Piggott, M. D. (2024). Learning to optimise wind farms with graph transformers. In *Applied Energy*.

¹See Section 1.4 for details on the shared contributions.

6.1 Introduction

Wind energy is a key component in addressing critical environmental, economic, and energy security challenges.

Wind farms comprise clusters of wind turbines strategically positioned within a limited geographical area to maximize energy capture. However, the proximity of turbines to one another introduces a key challenge: the wake effect. This phenomenon occurs when the airflow downstream of a wind turbine is altered by the presence of the turbine's rotor. As the rotor extracts kinetic energy from the wind, it reduces the wind speed in the downstream region, creating a wake. Additionally, interactions between the airflow and the turbine structure, including the rotating blades, generate increased turbulence intensity within the wake [122]. When a downstream turbine operates within the wake of an upstream turbine, whether fully or partially, it is exposed to significantly reduced wind speeds and elevated turbulence. These altered conditions can lead to a substantial reduction in the downstream turbine's energy production. The airflow in the wake does not immediately recover to its undisturbed state; instead, the wind speed remains diminished, and turbulence levels remain high for a considerable distance downstream [152, 112]. This diminished wind resource adversely affects the power output of turbines located within these wake regions, making the spatial layout of turbines a critical factor in wind farm design. Given the importance of mitigating the wake effect, accurately modelling wind turbine wakes has become a central focus in both the optimal design and operational control of wind farms. Detailed wake models are essential for optimizing turbine placement, maximizing overall energy production, and ensuring efficient farm operations. Research in this area has explored various computational and empirical methods for predicting wake behaviour, from simplified engineering models to high-fidelity simulations that account for complex fluid dynamics. For more in-depth insights, one can refer to [110] and the references therein. Accurate wake modelling is not only crucial for increasing the energy yield of existing wind farms but also plays a significant

role in the development of future large-scale farms, where minimizing the negative impact of turbine wakes becomes even more challenging. As wind energy continues to grow as a key renewable energy source, advancements in wake prediction and mitigation strategies will be vital for optimizing the efficiency and profitability of wind farm projects.

One effective method for improving wind farm performance and reducing the negative effects of wake interactions is yaw steering optimisation. This technique involves making controlled adjustments to the yaw angle of each wind turbine, redirecting the wake away from downstream turbines. By doing so, the downstream turbines can benefit from higher wind speeds, increasing their power generation. This improvement comes at the cost of a small reduction in the power output of the upstream turbine that is being yawed. However, the overall power output of the entire wind farm can be enhanced. Yaw steering optimisation must consider the interaction patterns between the turbines, meaning that the optimisation process cannot be conducted independently for each turbine. Instead, it must be formulated as a global optimisation problem, where the yaw angles of all turbines are adjusted in a coordinated manner to maximize the collective energy production of the farm. This holistic approach contrasts with traditional turbine control methods, where turbines are oriented directly into the wind to maximize their individual power output, often at the expense of downstream turbines. Numerous studies have demonstrated the effectiveness of yaw steering optimisation in boosting wind farm power generation. For instance, Adaramola and Krogstad [4] conducted wind tunnel experiments with two lab-scale turbines and observed a 12% increase in power output by adjusting the yaw angle of the upstream turbine. Similarly, Ma et al. [84] applied a cooperative yaw control strategy on a line of five turbines using high-fidelity large eddy simulations (LES), resulting in a 17.5% improvement in power production. In operational settings, Howland et al. [58] reported up to a 13% increase in power generation through wake steering optimisation under specific wind conditions at a commercial wind farm. Overall, yaw steering optimisation represents a strategic approach to maximizing the

total power output of a wind farm by carefully managing wake interactions, outperforming conventional control methods focused solely on individual turbine performance.

Data-driven wake models, especially those based on deep learning and trained on high-quality wind farm datasets, hold significant promise in balancing computational cost and accuracy. Many recent studies integrating deep learning into wind farm power prediction rely on establishing surrogate models for individual wind turbine wakes. For instance, Ti et al. [135] trained artificial neural networks (ANNs) on RANS data and employed wake superposition methods to effectively combine predicted individual turbine wakes and accurately predict the generated power. Zhang et al. [149] introduced an innovative wind turbine wake model capable of delivering 2D predictions by training a convolutional conditional generative adversarial neural network on LES data. Furthermore, Li et al. [78] utilised graph neural networks (GNNs) to learn from RANS simulations using their native multi-scale unstructured mesh-based data, leading to the development of a data-driven wake model that can rapidly generate precise 3D flow field solutions.

When considering the farm as a whole, Park et al. [104] developed a data-driven model that represents wind conditions and the relative positions of wind turbines as a graph. They trained a graph neural network with physics-induced attention (PGNN) to predict both the power output of individual turbines and the overall farm's power with high accuracy.

In a similar vein, Duthé et al. [44] investigated the effects of varying graph connectivity patterns on model performance, offering insight into how different configurations of turbine relationships can influence prediction accuracy. Bentsen et al. [153] further refined the GNN approach by introducing an attention mechanism that is entirely learned from data, enhancing the model's ability to flexibly weigh the influence of neighboring turbines and improve overall predictive performance. Nevertheless, the representation of wind farms as graphs is not without limitations. One potential drawback is the reliance on predefined graph connectivities or adjacency matrices, which are crucial for model performance but can

be hard to determine accurately. However, representing wind farms as graphs introduces certain challenges. A key limitation lies in the reliance on predefined graph connectivities or adjacency matrices, which are critical to the model's success but can be difficult to determine with precision. Additionally, the varying nature of underlying physics across models of different fidelities—such as analytical models, computational fluid dynamics (CFD) simulations, and real-world data—compounds the challenge. GNNs trained on one dataset with pre-established connectivities may struggle to generalize or transfer effectively to datasets of differing fidelities. The work presented in this chapter addresses these limitations by proposing a more versatile and generalizable framework for learning data-driven models at the farm scale. Specifically, it adopts a transformer architecture, which integrates the ability to process graph-structured data, similar to GNNs, while leveraging the scalability and capacity of transformer models [137]. The transformer approach not only enhances scalability but also mitigates common issues associated with deep GNNs, such as over-smoothing [27], which can hamper the effectiveness of deeper network layers. This hybrid approach aims to offer a more flexible, robust solution for accurately modelling wind farms while improving generalization across datasets and optimization tasks.

The primary novelties and contributions of this work can be summarised as follows:

1. A novel farm-scale model based on transformers was developed using training data generated from PyWake. The model can predict wind farm power accurately, averaging an accuracy of 99.8% on previously unseen layouts and wind conditions.
2. The presented data-driven model can precisely identify the sparse wake interactions among wind turbines in the farm, i.e. it can accurately characterise where there is aerodynamic influence between turbines and where there is none. This is achieved by the proposed transformer model while taking as input fully connected graph representations of the wind farm containing information that only includes farm layouts,

turbine yaw angles and wind conditions. This provides transparency into the model's inner workings and aids in model interpretability.

3. To improve the model's generalisability over various layouts and yaw angle configurations, a layout generator was employed to produce random yet realistic wind farm layouts of different sizes. To generate our dataset, we use Genetic Algorithms (GAs) to search over various yaw angle configurations while aiming to optimise overall wind farm power. The selected training data was randomly sampled from GA runs on wind farms of randomly generated layouts and wind conditions, enabling the model to discern favourable and sub-optimal configurations of yaw angle. The trained transformer model accurately identifies optimal yaw angle configurations despite not necessarily encountering them during training.
4. Equipped with strong evidence that the dependencies between the wind turbines can be encoded in a directed acyclic graph. We conduct a simple experiment highlighting the potential of PSGRL (see Chapter 5) for finding optimal policies efficiently by leveraging the latent graphical structure.

The subsequent sections of this chapter are organised as follows: We first briefly summarize the background context for this work, encompassing engineering tools for wind farm simulation, GNNs and transformers presented in Section 6.2. The deep learning frameworks considered are then presented in Section 6.3. We provide a detailed description of the data generation processes in Section 6.4. Section 6.5 provides empirical validation of the proposed approach.

6.2 Background

6.2.1 Wind Farm Simulations with Engineering Wake Models

Engineering wake models are computationally efficient tools for computing the wake interactions between wind turbines across a wide range of steady-state conditions. This work relies on the Bastankhah-Gaussian deficit model [18] to calculate the wake effect's impact on the velocity at each downstream wind turbine, and the Jiménez deflection model [61] to characterise the wake deflection induced by steering the wind turbine's yaw angle. These two models are implemented in the open-source wind farm simulation software PyWake. PyWake simulator has been benchmarked against established high-fidelity simulations and real-world wind farm data [107, 47]. The Bastankhah-Gaussian model assumes that the velocity deficit has a Gaussian shape. It can formally be written as follows:

$$\Delta U = \left(1 - \sqrt{1 - \frac{C_T}{8(\sigma(x)/d_0)^2}}\right) \exp\left(-\frac{x^2}{2\sigma(x)^2}\right) U_\infty, \quad (6.1)$$

where x denotes the downstream distance from the turbine rotor, $\sigma(x)$ is the standard deviation of the Gaussian velocity deficit profile at x (defined in Eq. 6.2), C_T is the thrust coefficient of the turbine, U_∞ is the upstream wind velocity and d_0 represents the diameter of the wind turbine. The standard deviation of the Gaussian velocity deficit, $\sigma(x)$, is commonly defined as follows:

$$\sigma(x) = kx + 0.2\sqrt{\beta}d_0, \quad (6.2)$$

where $k = \frac{\partial \sigma(x)}{\partial x}$ denotes the growth rate of the wake and β is a function of C_T .

The lateral deflection of the wake caused by yaw steering is approximated with the Jiménez model. The effective wind speed U_w at each wind turbine is computed using a

superposition-based approach to combine the lateral deflection and the velocity deficit of the wake. Given the incoming wind speed and wake deficits from upstream turbines. The wind turbine's power P can be computed as follows:

$$P = \int_{A_0} \frac{1}{2} \rho U_w^3 dA, \quad (6.3)$$

where A_0 is the area swept by the wind turbine blades and ρ is the air density.

6.2.2 Graph Neural Networks and Transformers

A wide variety of physical, biological and social systems can be formulated as graphs at a certain level of abstraction. Using graph representations in these systems facilitates the applications of various mathematical and computational tools, including graph neural networks (GNNs), allowing researchers to uncover hidden relationships, make accurate predictions and solve complex problems. While traditional machine learning techniques that are effective across many scenarios often prove inadequate when applied to graph-structured data, GNNs have succeeded in various domains where relationships and dependencies are embedded within network structures. Recent years have witnessed the application of GNNs in predicting material properties of crystals [26], identifying influential users within online social networks [72], recommending meaningful mobile applications to potential users [79] and simulating complex mesh-based numerical models of high-dimensional physical systems [108]. The transformer model [137] has been widely acknowledged as the most powerful neural network for modelling sequential data and has seen great success in natural language processing (NLP), speech analysis [39] and computer vision [81]. In recent years, researchers have found ways to alleviate limitations pertaining to GNNs such as over-smoothing [27], over-squashing [5] by incorporating some of the principles underpinning transformers, but

adapting them to modelling the unique properties of graphs [147, 117]. More details about GNNs and transformers are introduced in Section 6.3.

6.3 Methodology

6.3.1 Graph Representation of Wind Farms

The representation of a wind farm and the wind conditions can be encoded as a directed graph denoted by $\mathcal{G} = (V, E, u)$. Where, $\mathcal{G}$ consists of a tuple of vertex-level features denoted as $V \in \mathbb{R}^{N_v \times d_v}$, edge-level features represented as $E \in \mathbb{R}^{N_e \times d_e}$, and graph-level features expressed as $u \in \mathbb{R}^{d_u}$. In this context, the number of vertices and edges are denoted by N_v and N_e , respectively. The dimensions of vertex, edge, and graph-level features are denoted by d_v , d_e , and d_u , respectively. Each vertex in the graph corresponds to one of the wind turbines composing the farm. Formally, the vertex are labelled as v_i for $i = 1, \dots, N_v$, when the farm is composed of N_v wind turbines. The vertex-level features encode key information about the wind turbine state, including the turbine's position and yaw angle. The edges in the graph encode potential interactions between the wind turbines. Essentially, a directed edge exists from an upstream wind turbine to a downstream wind turbine if the wake of the upstream wind turbine influences the power generated by the downstream wind turbine. Formally, a directed edge, denoted as $e_{ij} \in E$, is established if the wake generated by v_i may influence the power output of v_j . The specific process to construct the wind farm graph, especially determining which edges to include in the graphical representation of the wind farm, is detailed in Section 6.3. Features at the edge level encode factors such as relative positions and distances between turbines. Lastly, graph-level features are used to specify physical parameters shared among all vertices in the graph, including variables such as wind speed and turbulence intensity (TI).

This approach of representing wind farms as graphs is largely inspired by the work of Park et al. [104], building on the idea that the interactions modelled within the graph structure can provide a strong foundation for predicting the energy output of a wind farm. A more comprehensive explanation of how wind farms are represented as graphs is provided in Section 6.3.

One critical aspect to highlight is that the graph representation used in this context is characterized by a high degree of heterophily. This means that connected nodes tend to have dissimilar features. As a result, traditional graph neural network (GNN) models, like the Graph Convolutional Network (GCN) [67], which are designed with the assumption that nodes connected by an edge have similar features (a property known as homophily), may not be suitable for this task. GCN's reliance on feature similarity across edges makes them suboptimal for graphs with pronounced heterophily, such as those representing wind farms.

In contrast, more advanced models, such as attentional networks, which learn to aggregate features from neighbouring nodes using attention mechanisms, or graph neural networks that employ message-passing frameworks, are better suited for this problem.

6.3.2 Message-Passing Graph Neural Networks

The message-passing graph neural network [19] is among the most adaptable and powerful variations of graph neural networks, enabling the exchange of arbitrary message vectors between vertices connected by edges. In this model, the edges define pathways for transmitting information, which is conveyed through messages. The content of these messages is determined by a message function that leverages the features of both the sending and receiving vertices, making the framework highly flexible in capturing complex relationships within the graph.

The graph neural network architecture used in this work closely aligns with the designs detailed in the papers by Battaglia et al. [19] and the work by Park et al. [104]. Given a wind

farm consisting of N turbines, the graph neural network model receives input the graphical description of the wind farm $\mathcal{G} = (V, E, u)$, where $V \in \mathbb{R}^{N_v \times d_v}$, $E \in \mathbb{R}^{N_e \times 2}$ and $u \in \mathbb{R}^2$. In our experiments, we consider the number of wind turbines composing the farm, N_v , to range from 2 to 100. Each vertex encodes three turbine-level features, including span-wise and stream-wise coordinates and turbine yaw angles, $d_v = 3$.

The edge features $E \in \mathbb{R}^{N_e \times 2}$ describe the relative span-wise and stream-wise distances between turbines, while the global features $u \in \mathbb{R}^2$ characterise the farm's upstream wind speed and turbulence intensity. The edge features, $E \in \mathbb{R}^{N_e \times 2}$, encapsulate the spatial relationships between turbines by encoding the relative span-wise and stream-wise distances between them. Additionally, global features $u \in \mathbb{R}^2$ include critical farm-wide environmental factors such as the upstream wind speed and turbulence intensity, essential for capturing the overall dynamics affecting turbine performance.

The GNN architecture consists of three distinct multi-layer perceptron (MLP) encoders, each responsible for transforming the features of vertices, edges, and global properties into more expressive latent spaces. These encoded features are subsequently processed through a series of graph-net blocks. Within each graph-net block, an edge model combines edge features, source and target vertex features, and global features to generate new edge embeddings. A vertex model aggregates incoming edge features with existing vertex and global features to produce new vertex embeddings. Finally, a global model constructs new graph-level embeddings by pooling all vertex and edge-level features concatenated with the global features. A decoder MLP then utilises the updated latent representation computed by the processor to extract meaningful information. In this case, the transformed vertex features are decoded to generate vertex-level predictions for individual turbine powers. An illustration of the message-passing graph neural network is shown in Figure 6.1.

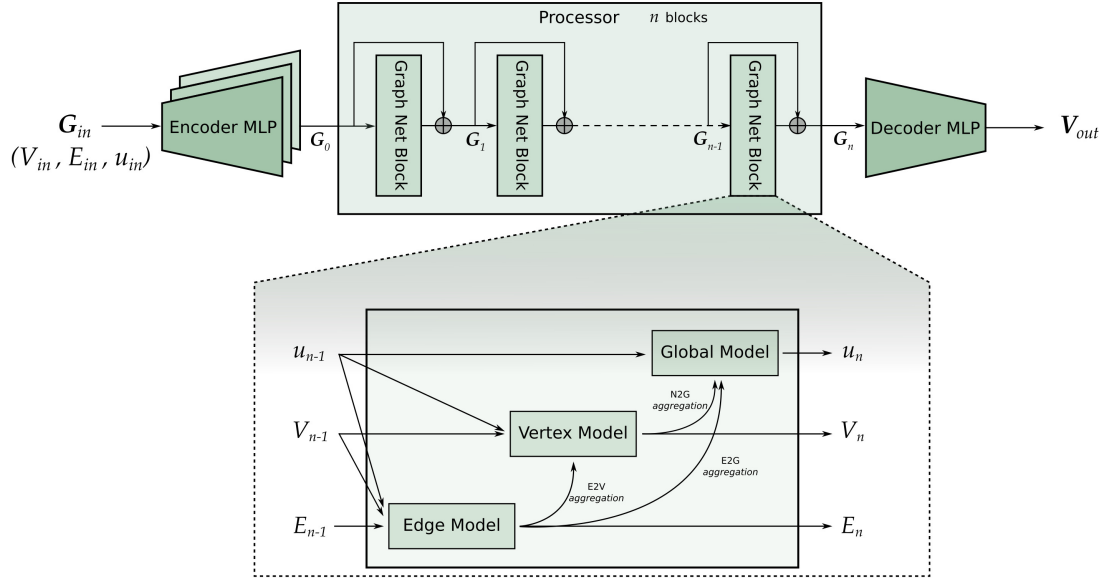


Fig. 6.1 This diagram illustrates the architecture of the message-passing graph network. The edge, vertex and global models in a graph-net block are multi-layer perceptrons (MLPs).

6.3.3 Transformers

The proposed transformer model's architecture follows a similar encoder-processor-decoder structure, where the graph-net blocks are substituted with multi-head attention blocks.

These multi-head attention blocks are inspired by those in the encoder section of the original transformer paper [137] and incorporate elements from the transformer architecture with additional layer normalisation, as described in [123] and [115]. In the context of a wind farm consisting of N turbines, the transformer model is designed to process only vertex-level features, denoted as $X \in \mathbb{R}^{N \times 5}$, where $N \in [2, 100]$ denotes the number of wind turbines. Note that the upper limit of 100 wind turbines in the farm is completely arbitrary and could be extended if needed. The vertex-level features consist of all the graph attributed, including the span-wise and stream-wise coordinates of the wind turbines, turbine yaw angles, and the upstream wind speed and turbulence intensity. In particular, no explicit positional encodings were applied, aligning with the principle that wind farm graph representations

should exhibit permutation invariance while acknowledging that spatial coordinates were directly incorporated into the input features.

The proposed architecture's multi-head attention block serves as a critical mechanism for effectively learning and representing the complex interactions among wind turbines within a wind farm. These interactions are essential for generating precise power predictions and gaining deeper insights into the patterns governing wind farm power generation. Each multi-head attention block comprises two primary components: the multi-head self-attention

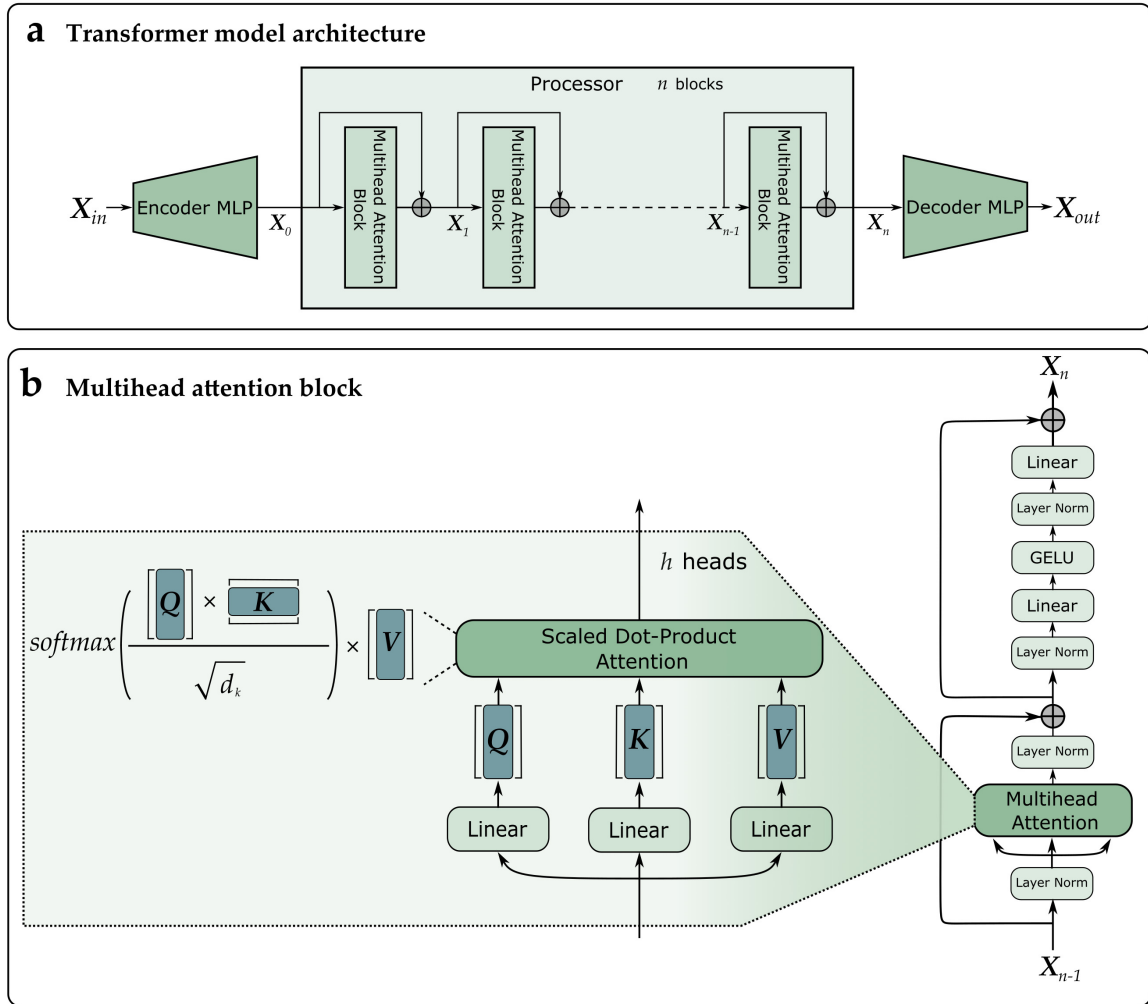


Fig. 6.2 Schematic of the proposed transformer model architecture. **a:** The proposed transformer architecture and its building blocks. **b:** Details of the multi-head attention block utilised here, where GELU refers to the Gaussian Error Linear Unit.

module and the point-wise feed-forward network. The multi-head self-attention module allows the model to assign different weights to vertex-level features for each turbine, which helps understand the subtle dependencies and interactions between them. This attention mechanism is vital in a wind farm setting, where factors such as the relative position of turbines, their orientations, and environmental conditions (e.g., wind speed, turbulence) all influence power generation. By weighing these relationships dynamically, the model can better capture how the configuration and operation of one turbine may affect others, thus improving the accuracy of power output predictions. The point-wise feed-forward network applies independent non-linear transformations to each feature. This step refines the learned representations, allowing the model to better capture the complexities of turbine behaviour by adjusting the feature values more expressively. The feed-forward network effectively transforms the attention-weighted features into more useful latent representations that the model can utilize to make predictions. Incorporating layer-normalisation layers into the architecture improves the model stability and contributes to faster convergence. Figure 6.2 visually depicts the considered model architecture, Figure 6.2a shows how predictions are computed, while Figure 6.2b illustrates the architecture of the multi-head attention block. At the core of the multi-head attention block is the multi-head attention mechanism, where each head h_i computes a different latent representation of the input X . To compute the output of a single attention head h_i , the model first projects the input $X \in \mathbb{R}^{N \times d}$ to a query, $Q_i = XW_i^Q$, a key, $K_i = XW_i^K$ and a value $V_i = XW_i^V$. Where $W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$ and $W_i^V \in \mathbb{R}^{d \times d_v}$ respectively, where d is the hidden dimension with $d_k = d_v = d$. Then, the attention head h_i can be computed as follows:

$$h_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i. \quad (6.4)$$

The multi-head attention block output consists of the concatenation of all attention heads $\{h_i\}_{i=1}^m$ multiplied with a projection matrix $W^O \in \mathbb{R}^{md_v \times d}$, yielding:

$$\text{MultiHeadAttention}(X) = \text{Concat}(\gamma_1 h_1, \dots, \gamma_m h_m) W^O, \quad (6.5)$$

where m is a hyper-parameter that indicates the number of attention heads used by the model, and γ_i are learnable parameters initialised to 1.

As illustrated in Figure 6.2a the input X is first encoded with a neural network MLP_{enc} , $X_0 = \text{MLP}_{enc}(X_{in})$, then it goes through a series of Multi-head Attention Blocks MAB, $X_i = \text{MAB}(X_{i-1}) + X_{i-1}$, for $i = 1, \dots, n$. Then the final latent representation X_n is decoded using a neural network MLP_{dec} , $X_{out} = \text{MLP}_{dec}(X_n)$.

Note that the exact sequence of operations performed inside a multi-head attention block is also detailed in Figure 6.2b. The output of the i^{th} attention block X_i can be written as:

$$X_i = \text{FFN}(\text{LN}(\text{MultiHeadAttention}(\text{LN}(X_{i-1}))) + X_{i-1}), \quad (6.6)$$

where FFN denotes a feed-forward network, while LN indicates layer normalisation. The specific calculations are outlined as follows:

$$\text{FFN}(X) = \text{LN}(\sigma(\text{LN}(X)W_1 + b_1))W_2 + b_2 + X, \quad (6.7)$$

where $\sigma(\cdot)$ denotes the GELU non-linear activation function.

The parameters of each block are learned via back-propagation during training.

6.4 Model Training

6.4.1 Data Generation

Wind farm layouts were randomly generated with the Plant Layout Generator (PLayGen), originally developed by NREL [52]. For each generated layout we carried out several wind farm simulations, using the PyWake [107] software. Note that PyWake is a simulator we use to generate data; the proposed model does not depend on PyWake and could work with other simulators or potentially with real-world data. The simulation model computes the wake effect of a single turbine with the Bastankhah-Gaussian analytical model [18]. Individual turbine wakes are combined using the sum-of-squares superposition method. To simulate yaw angle-based wake steering effects we used the Jiménez wake deflection model [61]. Graph representation of wind farms was built incorporating the farm layouts, simulation parameters and individual turbine powers obtained through PyWake simulations. For GNN training, directed edges were established from an upstream wind turbine to a downstream one if the angle between the two turbines fell within the range of $[-15^\circ, 15^\circ]$. For the transformer training the input, consisted of a fully connected wind farm graph. Figure 6.3 provides visual depictions of various wind farm layouts generated by PLayGen alongside their corresponding graph representations. Notably, the wind turbine parameters employed in all simulation scenarios were calibrated to align with the specifications of the Vestas V80 wind turbine.

Two datasets were generated for training and experimenting with different machine learning-based surrogate models. The “standard” dataset comprises diverse wind farm layouts and physical parameters with randomly initialised wind turbine yaw angles, serving as a validation set for experimenting with various surrogate models. In contrast, the “enhanced” dataset is significantly larger and incorporates a broader range of yaw angle configurations. The standard dataset was utilised for initial model validation and experimentation. In contrast, the enhanced dataset is designed to leverage the transformer models’ capabilities to

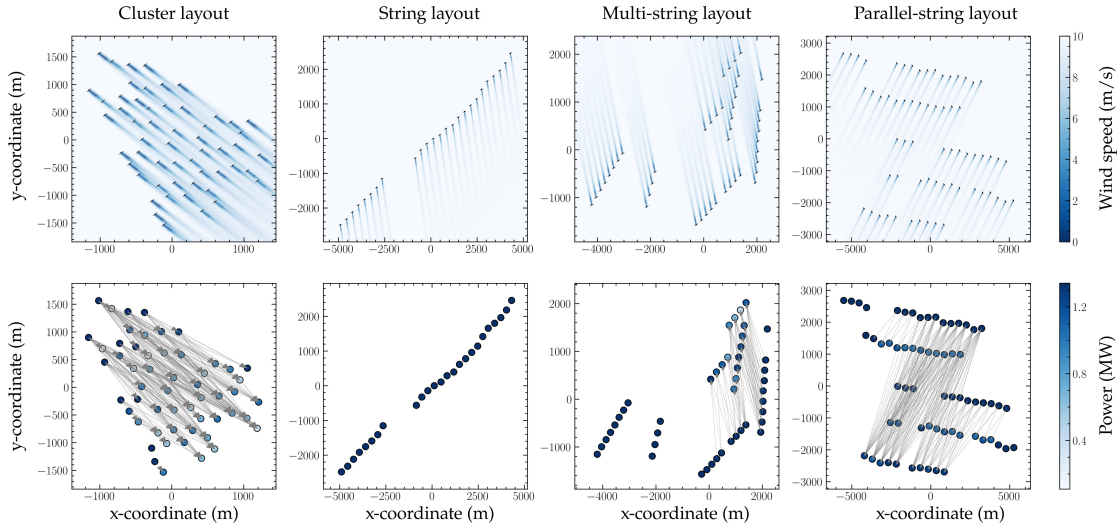


Fig. 6.3 Visualisation of various layout pattern produced by PPlayGen alongside their corresponding graph representations. Wake maps were generated using the Bastankhah-Gaussian analytical model integrated in the PyWake simulator. For the purposes of these visualisations, simulations were conducted with a wind speed of 10 m/s , 5% turbulence intensity, and all turbines facing the wind direction, which in each case was chosen at random. It is important to note that flow field information was neither produced during the data generation process nor included as a component of the features within the dataset. The directed arrows represent edges used in GNN training. Wind turbines were coloured based on their individual power generation.

handle enormous datasets and train the most advanced surrogate model for wind farm power prediction and optimisation.

In the standard dataset, 200k simulation scenarios were generated, each scenario subjected to a random wind farm layout from one of the four layout styles shown in Figure 6.3, with a random number of wind turbines between 2 and 100, wind speed between 8 m/s and 15 m/s , wind direction between 0° and 359° , turbulence intensity (TI) between 5% and 15%. All wind turbine yaw angles were randomly initialised between -30° and 30° . In the enhanced dataset, the training data was randomly sampled from different genetic algorithm runs to improve the model's ability to discern subtle variations in yaw angle configurations. Each GA run was performed using the PyWake model to optimise for total power generation by varying individual wind turbine yaw angles between -30° and 30° . All instances of the genetic algorithm were executed for 50 generations with a population size of 100, a

cross-over probability of 70% and a mutation rate of 50% so that a more diverse range of yaw angle configurations can be introduced. A total of 200,000 scenarios (each comprising a unique wind farm layout and set of physical wind parameters) were randomly generated and optimised using genetic algorithms to produce the dataset. The final dataset included 100 randomly sampled yaw angle configurations from each scenario and was split into 80% for training, 10% for validation and another 10% for testing.

The creation of both datasets was parallelised and executed on Imperial College London's CX1 High-Performance Computing (HPC) cluster. The generation of the standard dataset required approximately eight CPU hours, while the enhanced dataset necessitated over 5,000 CPU hours.

6.4.2 Validation of the Learned Latent Graph

All models were trained to minimise the mean squared error (MSE) of predicted individual turbine powers with the AdamW optimiser [83]. Training experiments and model comparisons were conducted primarily on the standard dataset, with a final larger model being trained on the enhanced dataset. The models were implemented using the open-source deep learning library PyTorch [105], and Pytorch Geometric [45].

Figure 6.4 illustrates the results of training various GNN models with different numbers of graph-net blocks and hidden units per Multi-Layer Perceptron (MLP) layer within the encoder, the decoder, and the graph-net blocks. These are compared to a transformer consisting of five multi-head attention blocks, four attention heads, and a hidden size of 128 neurons. The transformer model featured 16.3 million trainable parameters and reached its lowest validation loss at 8.6×10^{-6} . In contrast, the best-performing GNN model achieved a validation loss of 1.79×10^{-5} at 124 thousand parameters. The results indicate that increasing model capacity by adding more layers and hidden units per layer led to only marginal improvements in GNN performance. On the other hand, the transformer model

demonstrated superior scalability with respect to the number of trainable parameters in the model, and a sufficiently large transformer was able to outperform the GNNs in terms of validation loss.

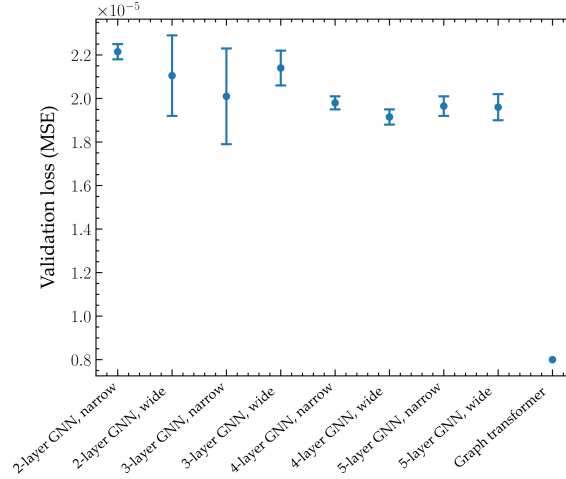


Fig. 6.4 Comparison of lowest validation loss of GNN models of various sizes and the transformer model. The lowest validation loss of each GNN model was computed from training runs with three different random seeds. GNN models with 64 hidden units per MLP layer were denoted as “narrow”, while those with 128 hidden units per MLP layer were referred to as “wide”. The transformer model was trained only once due to the excessive training time required.

The final choice for the surrogate model involved opting for a transformer model with increased width and depth. This model comprised eight multi-head attention blocks, each with four attention heads of 128 hidden dimensions. The encoder MLP, the decoder MLP and the feed-forward layers have 512 hidden dimensions each, resulting in a model with a total of 25.8 million trainable parameters. The model was trained for one million steps on the enhanced dataset with a batch size of 256. A cosine annealing learning rate scheduler [82] was incorporated into the training process, with a maximum learning rate of 4×10^{-4} , which was reached after a linear warm-up period of 50,000 steps. To optimise GPU memory usage and accelerate training, 16-bit automatic mixed precision (AMP) was employed. The entire training process spanned 75 hours and was conducted on 5 NVIDIA A100 GPUs on the Department of Earth Science and Engineering’s “Hivemind” DGX GPU cluster. The trained

model achieved an MSE loss of 3×10^{-4} when evaluated on the previously unseen test data set. This result corresponds to a relative accuracy of 99.8% in predicting farm power.

6.5 Results and Discussion

6.5.1 Assessment of Surrogate Model Efficacy

As mentioned in Section 6.3, a wind farm can be naturally defined as a graph, where wind turbines are represented as vertices of the graph, and the wake effects among turbines can be modelled as (directed) edges between pairs of vertices. Figure 6.5b illustrates graph representations of randomly generated wind farms under different incoming wind directions. The graph representations were created by treating the wind turbines as graph vertices and establishing edges based on wind direction, where a directed edge is established from an upstream turbine to its downstream counterparts. This edge is formed when the angle between the pairs of turbines and the wind direction falls within a range of 15° , considering potential yaw steering effects. It is important to acknowledge that alternative methods for determining edges, such as nearest-neighbour connections or fully connected edges, are equally valid.

In situations where graph learning is applied to problems lacking predefined relationships among vertices, the approach chosen for defining edges, be it based on factors such as wind direction or downstream distance, markedly impacts the effectiveness of the graph learning tasks. As sub-optimal choices of edges could hinder the predictive power and generalisability of the model, one potential alternative is to consider fully connected graphs, which is similar to the transformer approach adopted here, where each vertex is free to interact with every other vertex and the significance of this pair-wise interaction is evaluated through an attention score.

Attention scores computed within a transformer model can enhance model interpretability by offering valuable insights into how the model focuses on different parts of the input data

to make accurate predictions. Graph representations of the attention map output from the surrogate model on previously unseen wind farms are shown in Figure 6.5c. The attention map was extracted from the last multi-head attention block in the transformer model and averaged over all attention heads.

It can be seen that the transformer model is able to demonstrate a notable proficiency in accurately capturing the dependencies of power and wake effects among wind turbines, even when flow field data was not directly provided as model input. Moreover, there is a consistent correspondence between the attention scores learned by the transformer model and the wake patterns deducible from the wind farm's wake map produced (for visualisation purposes) by PyWake.

The assessment of the transformer surrogate model's predictive power and ability to generalise was extended by employing it to predict the power output of wind farms with layouts not encountered during the training phase. This evaluation spans all wind directions ranging from 0° to 359° at intervals of 1° . Three exemplar outcomes are showcased in Figure 6.6, where the farm layouts are presented in Figure 6.6a. Figure 6.6b illustrates the predicted farm power from PyWake, regarded as the ground truth for the purposes of this work, the predictions from the transformer surrogate, and the discrepancy between the two. The first two rows of Figure 6.6 show the evaluation of the model on two randomly generated layouts featuring 45 and 55 wind turbines, respectively. The third row shows an evaluation of the model on a real-world wind farm, Horns Rev 1, a Danish offshore installation of 80 wind turbines in the North Sea. Across all wind directions, the transformer surrogate demonstrates average mean relative accuracies of 99.85%, 99.71%, and 99.69%, along with minimum accuracies of 99.25%, 97.65%, and 97.56%, for the cluster, multi-string, and Horns Rev 1 layouts, respectively. This is consistent with the 99.8% relative accuracy the model achieved in the previously unseen test data set. In general, the slightly reduced accuracies tend to correspond to wind directions associated with more rapid changes

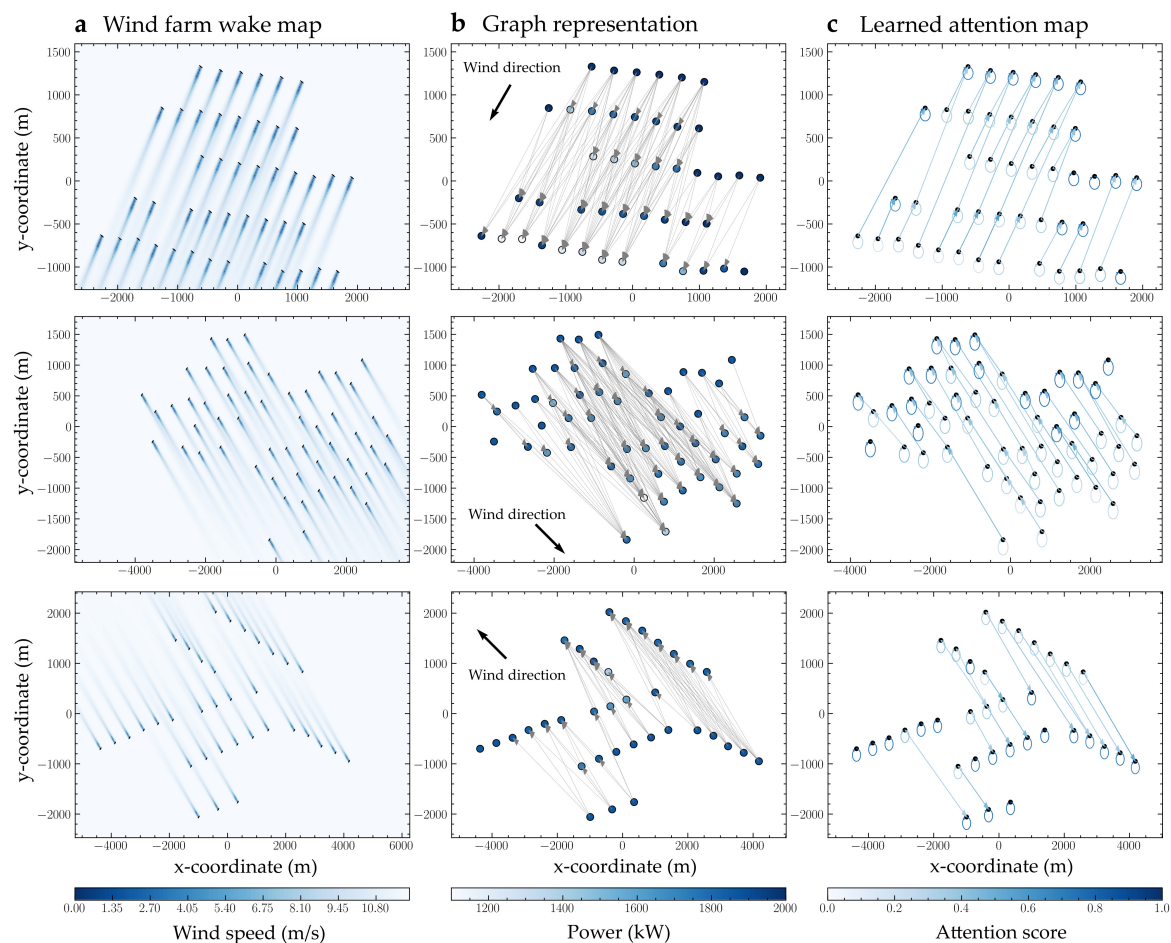


Fig. 6.5 Illustration of different approaches of representing wind farms. The three rows correspond to three randomly generated wind farms previously unseen by the model, subject to similar physical conditions of 12 m/s wind speed and 5% TI at incoming wind directions 45° , 315° and 135° . The three wind farms comprise 45, 55, and 35 wind turbines, respectively. In particular, all turbines in all configurations are orientated to face the direction of the wind. **a**: Wake map produced by PyWake using the Bastankhah-Gaussian analytical model. **b**: Graph representation of a wind farm, where solid dots represent wind turbines (vertices) and grey lines with arrows represent interactions (directed edges). **c**: Graph representation of the attention score output from the trained transformer model. Only attention scores larger than 0.1 were visualised. Circles with arrows at the end denote self-attending tokens.

in total wind farm power. This is explainable in part as relative error values were plotted, and the related fact that higher model errors correspond to those wind directions where wake effects between turbines are strongest. The strong agreement observed between the predictions made by the transformer surrogate and PyWake serves as additional evidence of

the model's generalisability. Remarkably, even without explicit training on identical layouts across different wind directions, the transformer model is able to demonstrate high accuracy when applied to novel layouts under different wind directions.

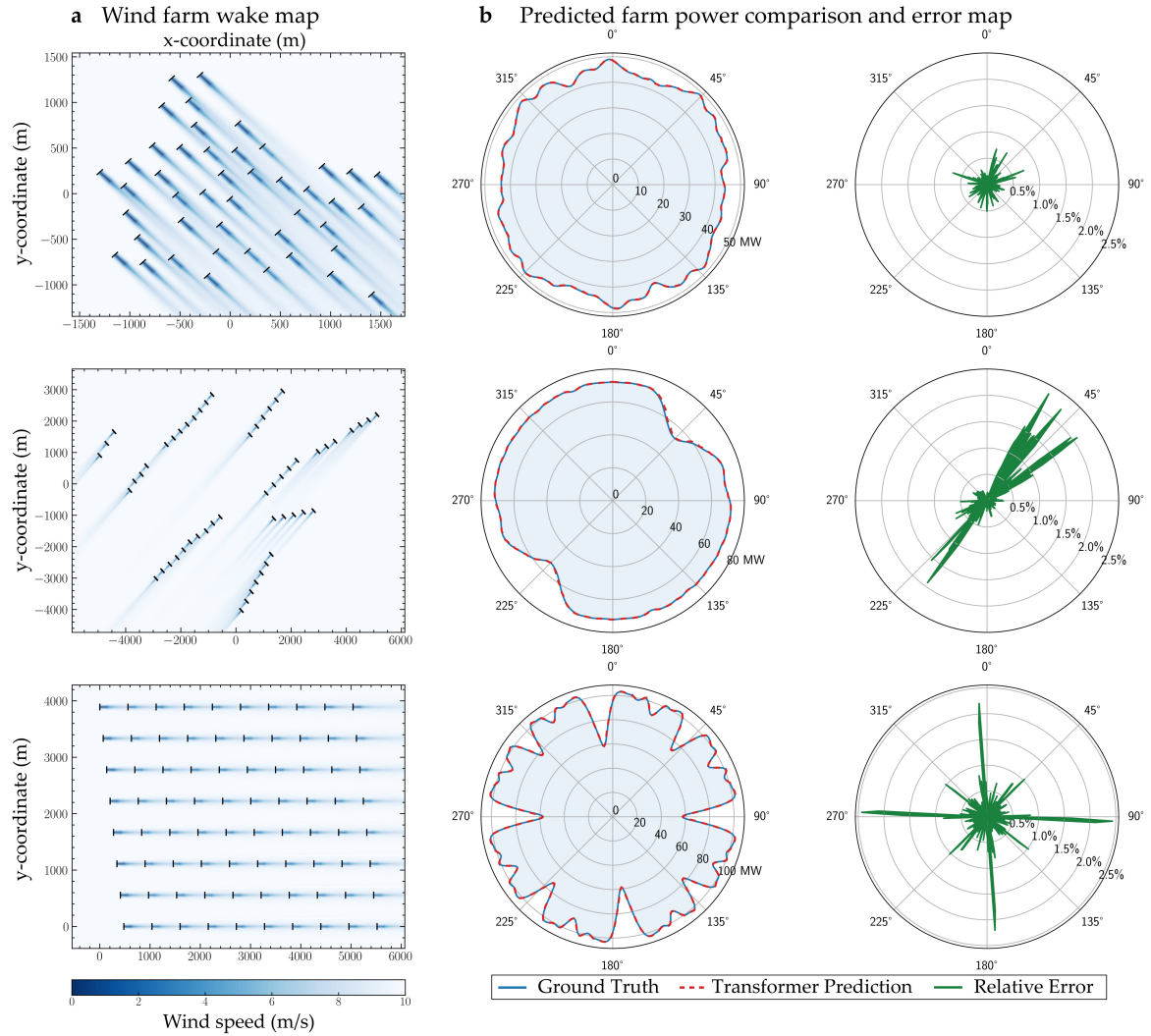


Fig. 6.6 The model was tested across different scenarios consisting of various wind directions using previously unseen farm layouts. The results are presented in three rows, demonstrating outcomes for a random cluster-type layout, a random multi-string-type layout, and the Horns Rev 1 layout, respectively. **a:** Wake map generated by PyWake using the Bastankhah-Gaussian analytical model. The assumed conditions for the three wind farms included wind speeds of 10 m/s, TIs of 5%, and dominant wind directions of 315°, 45°, and 270°, respectively. All wind turbines were oriented to face the incoming wind direction. **b:** Comparison of total farm power computed by PyWake and the transformer surrogate model, as well as relative error at different wind directions.

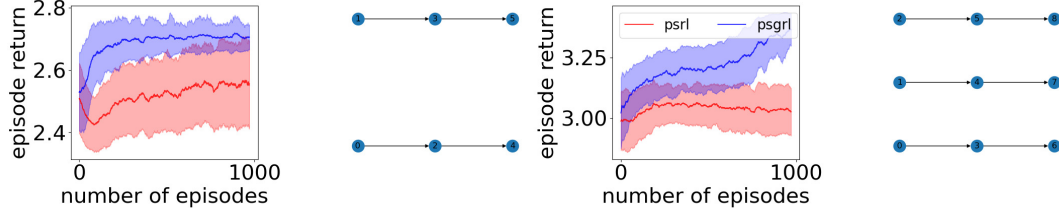


Fig. 6.7 The two leftmost plots summarize the results obtained on a wind farm yield optimisation task with six wind turbines. The first plot shows the performances when PSGRL and PSRL directly optimise the yaw angle. The second plot illustrates the farm layout (each node represents a wind turbine, and each edge contains the wake effect). The two rightmost plots show the same experiment for a larger farm with nine wind turbines.

6.5.2 Wind Farm Yield Optimisation

While the proposed transformer model can successfully predict the wind farm's yield, it cannot be used to identify the yaw angle setting that maximises the wind farm's overall yield. To identify this optimal strategy, we consider two different approaches. (i) We formulate the optimisation problem as an MDP and use PSRL to find an optimal strategy. By doing so, we deliberately ignore the additional structure in the wind farm optimisation problem. (ii) We model the problem as a DAMDP, and we leverage the graph G that encodes the interaction patterns within the farm using PSGRL (see Chapter 5).

To simplify the problem, we consider that the farm layout is organized on a grid. Each column of that grid represents a layer of the directed acyclic graph that encodes the interactions between the wind turbines.

The wind farm's yield optimisation problem can be modelled as a finite horizon time inhomogeneous MDP $M = \langle \{\mathcal{S}_h\}_{h=1}^H, \{\mathcal{A}_h\}_{h=1}^H, \{P_h\}_{h=1}^H, \{R_h\}_{h=1}^H, H, \rho \rangle$. The state $s \in \mathcal{S}_t$ at time step t consists of the wind speeds observed at each wind turbine in the t^{th} layer of the grid. The wind speeds are discretized and include the 40 increments of $0.1m/s$ from $6m/s$ to $10m/s$. The size of the state space at time step t is $|\mathcal{S}_t| = 40^{n_t}$, where n_t is the number of wind turbines in layer t . The action $a \in \mathcal{A}_t$ at time step t consists of the yaw angle applied at each wind turbine in the t^{th} layer of the grid. The action space is discretized, and at each

turbine, the possible actions are either to deflect in one of the two possible directions or face the wind, i.e. $\{30^\circ, 0^\circ - 30^\circ\}$. The size of the action space at time step t is $\mathcal{A}_t = 3^{n_t}$. The transition dynamics $\{P_h\}_{h=1}^H$ are computed with FLORIS [8]² and determine the wind speeds observed at the next layer. The reward function, R_t , which consists of the energy production of all wind turbines in layer t , is also computed with FLORIS.

This problem can also be formalized using a DAMDP

$M_G = \langle \mathcal{X}, \{\mathcal{Y}_j\}_{j=1}^{U_r}, \{r^j\}_{j=1}^{U_r}, \{p^i\}_{i=1}^{U_\tau}, H, \{\rho_A^v\}_{v=1}^{n_1}, G \rangle$ (see Chapter 5). We recall that the DAMDP execution is tightly linked to the graph G . In particular, the usual MDP formulation is broken down into its atomic components, which are described at the node level (more details are given in Chapter 5). In this context, each node of the graph G corresponds to a wind turbine. The atomic state of each vertex, $x \in \mathcal{X}$, describes the wind speed observed at the location of the corresponding wind turbine. Similarly, the atomic action $y \in \mathcal{Y}$ consists of the desired yaw angle for the corresponding wind turbine.

Since the wind speed only depends on the action performed at the wind turbines whose wake effect can reach the current location, atomic transition dynamics use the graph's connectivity and explicitly learn these atomic dynamics.

The performance, measured as the power generated by the wind farm, of the two approaches is shown in Figure 6.7. We consider on the left of Figure 6.7 a grid of 6 wind turbines, and we show in blue the performance of PSGRL and in red the performance of PSRL. The plots on the right of Figure 6.7 consider a farm with nine turbines. In both scenarios, we can see that PSGRL (blue curve) is more efficient in learning the optimal yaw angle setting. This improved performance of PSGRL is additional evidence that leveraging the problem's underlying structure can boost RL algorithms' efficiency.

²The code for the FLORIS simulator is available at the following address: <https://github.com/NREL/floris>

6.6 Conclusions

First, the proposed methodology is a contribution on its own. It facilitates the application of transfer learning to wind farm datasets of varying fidelities or real-world data, overcoming potential limitations faced by graph neural networks utilising pre-determined connectivities. The trained model was shown to generalise well, averaging a relative accuracy of 99.8% over power predictions of wind farms that were not encountered during the training process. Furthermore, the model can discern wake effects between turbines in previously unseen farm layouts and wind conditions, aligning well with the corresponding wind farm wake maps. This encouraging result motivates further research into the potential of surrogate models for wind farm simulations with more complex physics, transfer learning transformers on real-world wind farm data, and data-driven modelling of wind farms with heterogeneous turbines.

Additionally, the proposed methodology highlights that the latent graphical structure considered in Chapter 5 manifests itself in real-world problems such as wind farm yield optimisation. More importantly, this structure can be learned, the proposed computational framework encodes the wind farm into a fully connected graph and processes it through a transformer. The transformer's self-attention mechanism allows for the identification of latent relationships between turbines within wind farms.

Chapter 7

Conclusion

To conclude, we briefly summarize the contributions of the material presented in this thesis and highlight important future work directions.

7.1 Summary

In this work, we have presented algorithms that successfully leverage latent structures in MDPs to improve their learning efficiency. We focused on two important types of latent structure: the hierarchical structure and the graphical structure.

The structures considered in this thesis are of great practical interest as they naturally appear in many real-world scenarios. The hierarchical structure we consider underpins the great success of deep Hierarchical Reinforcement learning [94, 21, 139], and the latent graphical structure of Chapter 5 finds several real-world applications; in particular, the optimisation of a wind farm’s yield as motivated in Chapter 6.

This final chapter first summarises the overall contributions of the material presented in this thesis. Then, in Section 7.2, we highlight potential future research directions for each chapter. We finally conclude this thesis with a broader discussion on future research directions and the relevance of structure discovery in RL.

Our work in Chapters 3 and 4 improves our understanding of goal-conditioned HRL in several ways.

Chapter 3 considers the MDP decomposition induced by goal-conditioned HRL. In particular, we show that the original MDP can be decomposed into a low-level MDP and a high-level MDP and each agent is tasked to find the optimal policy in their respective MDP.

The lower bound on HRL’s sample complexity offers practitioners a valuable tool to help them decide whether to consider a hierarchical decomposition for their problem. The lower bound also clearly highlights the benefit of state and temporal abstraction in HRL.

The novel Hierarchical Q-learning-based algorithm designed in Chapter 3 successfully leverages the hierarchy to improve its sample efficiency. The performance of the proposed algorithm is empirically validated on a set of hierarchical tasks, which includes a set of maze-like environments specially designed for that purpose, as well as more traditional benchmark tasks such as the Gymnasium taxi environment. The performance of the proposed algorithm on these tasks further emphasizes the usefulness of the lower bound since the empirical efficiency gains observed support our theoretical findings.

The implementation of Hierarchical Q-learning proposes an interesting way to cope with the non-stationarity of the high-level transition dynamics unique to the goal-based approach. Our implementation relies on the fact that in the goal-based setting, the intended behaviour of the optimal low-level policy is known in advance: an optimal low-level policy successfully reaches the instructed sub-goal. The non-stationarity is mitigated as we only aim to learn the high-level transition under the optimal low-level policy. This is achieved by performing high-level updates only if the behaviour exhibited is the one we expect from the optimal low-level policy.

Chapter 4 refines our understanding of HRL by providing a provably efficient algorithm in a setting that requires several additional assumptions. Concretely, we assume that the

reward function is sparse and only rewards the agent when the episode ends in the goal state. We also assume that the agent can access a map ψ that describes the hierarchical structure of the decompositions. Specifically, the structure map ψ explicitly encodes the expected next high-level state after successfully completing each sub-goal.

In this particular setting, we propose a regret minimisation algorithm and study how its regret relates to the lower bound on monolithic RL approaches that do not explicitly exploit the hierarchical structure. The proposed algorithm is flexible enough to accommodate a broad choice of low-level algorithms (i.e., any optimistic algorithm) to compute the low-level policy. Our upper bound on the regret is valid for any choice of low-level algorithm. This theoretical result suggests that hierarchical methods' main source of efficiency gains is their ability to leverage repeated patterns (low-level MDPs) in the original problem.

The algorithm proposed in Chapter 4 represents an improvement over existing hierarchical algorithms. Whether it is about the nature of the theoretical guarantee. The posterior sampling-based hierarchical algorithm studied in [146] has an upper bound on its Bayesian regret. We were able in Chapter 4 to derive an upper bound on the frequentist regret of the proposed algorithm, which is a stronger theoretical guarantee. The only existing work deriving a frequentist upper bound on the regret of an HRL algorithm [40] expresses the regret as a function of the decomposition's characteristics and the behaviour (coverage) of the policy during learning. With our algorithm and its upper bound, we remove this dependency on the policies' behaviour and propose an upper bound on the regret that only depends on the decomposition's characteristics.

Overall, our contributions strengthen our understanding of the sources of the efficiency gains offered by HRL algorithms.

We also consider algorithms that could leverage latent graphical structure. Chapter 5 proposes a posterior sampling-based algorithm to efficiently solve problems exhibiting a graphical structure. Chapter 6 demonstrates that the graphical structure considered appears

in real-world problems such as wind farm yield optimisation problems. More precisely, Chapter 5 formalises a new subclass of MDPs with an underlying graphical structure, DAMDPs. It also introduces a posterior sampling-based algorithm, PSGRL, that leverages this latent graphical structure to learn an optimal policy efficiently. In that context, we provide evidence that PSGRL outperforms PSRL in theory and practice whenever this graphical structure is present. In particular, we provide an upper bound on PSGRL's regret and show that it provides efficiency gains compared to traditional methods that ignore the graphical structure (like PSRL) or model it differently (like FMDP). We validate our theoretical findings with a set of experiments on a graph problem, which is a stochastic version of the well-known maximum flow of a graph. Although the structure of DAMDPs may appear restrictive, it does appear in many practical settings. Indeed, our empirical validation shows that our algorithm performs well on a stochastic version of the maximum flow problem. Suggesting that this algorithm could be applied to the large family of real-world problems that could be modelled as finding the maximum flow of a graph, such as network routing [85] and supply chain management [119].

Chapter 6 introduces a data-driven model that can accurately predict the power generation and learn the interaction patterns of all wind turbines in a given wind farm, regardless of the wind farm layout, yaw angle configurations and wind conditions. The proposed model encodes the wind farm into a fully connected graph and processes it through a transformer. Critically, the self-attention mechanism adopted by the transformer model allows for identifying latent relationships between the turbines composing the wind farm. The learned relationship can be encoded in a directed acyclic graph similar to the one considered in Chapter 5. Finally, we show that the PSGRL algorithm presented in Chapter 5 can learn an optimal action profile to optimise the yield of a wind farm more efficiently than PSRL, an algorithm that ignores this latent graphical structure. The model presented to learn the wind farm's interaction pattern is an interesting contribution for several reasons. This

approach facilitates the application of transfer learning to wind farm datasets of varying fidelities or real-world data, overcoming potential limitations faced by graph neural networks utilising pre-determined connectivities. This model could be used as a surrogate model for wind farm simulations. As one trained, the forward pass in the model could be significantly less computationally intensive than the current simulation engine. Additionally, the model generalises well, averaging a relative accuracy of 99.8% when it predicts the power of wind farms that were not encountered during the training process. Furthermore, the model can discern wake effects between turbines in previously unseen farm layouts and wind conditions, aligning well with the corresponding wind farm wake maps.

7.2 Further work

In this section, we consider future directions for research related to the material presented in this thesis. We first discuss each chapter individually and conclude by considering more general extensions and the role of structure discovery in RL.

7.2.1 A Lower Bound on HRL Efficiency

Although the lower bound and the algorithm presented have taken a significant first step in bettering our understanding of the benefits of hierarchical decomposition, there is still scope for further work in this area. Three immediate open questions are particularly interesting.

(i) Could our lower bound be refined by explicitly accounting for the low-level and high-level agent interactions?

The current proof of the lower bound considers the learning of the low-level MDP and high-level MDP as independent problems. While this simplification results in a valid lower bound, it might be refined with a more careful analysis of the effect of the interactions between

the two hierarchy levels. This analysis is more challenging than simply considering the high-level MDP to be non-stationary. It also requires accounting for the fact that exploration of entire low-level MDP regions depends on the behaviour of the high-level policy.

(ii) Could we learn the hierarchical decomposition, particularly the sub-goal space?

Indeed, methods that design efficient sub-goal spaces remain largely unexplored and are a critical aspect of the design of HRL algorithms. We discuss this point more extensively in Section 7.2.5.

(iii) Moreover, the insights we proposed are framed in a tabular setting and do not yet extend to a continuous setting where function approximation could be leveraged to allow the low-level agent to generalise over sub-goals and to consider setting beyond the tabular case described in this Chapter 3.

7.2.2 An Upper Bound on HRL Regret

The algorithm and the theoretical guarantee provided in Chapter 4 is a significant step toward our quest to design a provably efficient algorithm for HRL. While the proposed algorithm seems reasonably efficient, there are some immediate areas for improvement.

(i) Is it possible to design a more efficient hierarchical algorithm?

We identify two sources of inefficiency in our algorithms that might suggest that an algorithm with improved efficiency could be considered. Our algorithm treats all sub-goals of a low-level MDP as independent tasks. However, all these tasks share the same low-level MDP and are governed by the same dynamics. Sharing information about the learned dynamics could boost the algorithm's efficiency. Another drawback of our approach is that the algorithm stops collecting samples whenever it fails a single sub-goal. Using the remaining episode to

collect more information about the environment would probably lead to a gain in efficiency as well.

(ii) Could we extend our result to a more general setting, like the one considered in Chapter 3?

The structure map ψ was instrumental in computing the high-level transition function. If this map is deterministic, it could be trivially learned without affecting the overall complexity of the algorithm, as a single observation would be sufficient to learn the map. However, if the map is stochastic, meaning that a successful sub-goal could lead to different high-level states, we must learn the high-level transition function, which would affect the algorithm's efficiency. However, it seems reasonable to have some additional structure here as there might be only a small number of candidates as potential next high-level states. Formulating the problem for a more general reward setting might be possible. Note that the low-level sparse reward construction could be preserved as these rewards are completely artificial. This approach might be limited as the low-level would ignore the rewards that are not directly encoded as sub-goals.

7.2.3 Posterior Sampling for RL on Graphs

We successfully demonstrated that several problems could be modelled as DAMDPs and then solved with PSGRL. Nevertheless, immediate steps could be taken to extend our method's applicability further and broaden the range of problems it can tackle.

(i) Currently, all edges in the graph considered are identical.

An interesting extension would be considering different types of edges, which would broaden the set of problems that could be modelled with the proposed approach. This could typically be done by adding an attribute to each edge. For example, if we revisit the maximum leaky flow problem presented in Section 3.6. We could imagine that edges have an additional

attribute that encodes the distance between two nodes, and the larger the distance is, the more likely it is for the associated edge to leak. Adding additional attributes to the edge would allow us to consider a richer set of transition dynamics and to model real-world problems more realistically. Such an increase in complexity will impact the upper bound on the regret but would offer more flexibility in the modelling.

(ii) The current approach only focuses on a tabular setting. It would be interesting to extend it to a continuous setting.

A potential solution would require designing a deep RL algorithm that uses GNN to estimate the transition dynamics. The GNN model would be able to leverage the graphical structure while considering continuous input and output spaces. Obtaining any theoretical guarantees for this continuous setting would be difficult, but evaluating the empirical efficiency gain of graphical factorisation in this context remains of great interest.

Modelling the dynamics with a GNN would also be a convenient way to deal with the previously suggested innovation of adding an attribute to each edge. The edge attributes would become part of the GNN input. The network would then be able to account for varying types of edges and even extrapolate and interpolate to previously unseen edge attributes, a feat not approachable by tabular methods.

7.2.4 Real-World Example: Wind Farm Yield Optimisation

As already mentioned in Chapter 6, the model presented in Chapter 6 is a contribution on its own and provides a lot of opportunity for future research direction. The wind farm yield optimisation experiment also highlights the limitations of the PSGRL algorithm. Several immediate open questions could be addressed.

(i) Could the transformer model be used to model higher-fidelity wind farm simulations? High-fidelity wind farm simulations are computationally intensive. Using the model proposed as a surrogate model could be an efficient alternative. While it seems challenging to collect a

dataset of high-fidelity simulations to train a transformer from scratch, we could consider training in phases. The initial training phase would be performed with a dataset collected from a low-fidelity simulator. Typically, the interaction pattern between the wind turbines could be learned with this dataset as well as a crude estimate of the wake. In the second phase, the wake shape and characteristics could be refined with a dataset consisting of high-fidelity simulation data.

(ii) From the RL perspective, while PSGRL proposed a reasonable solution, it worked for a relatively simple layout of wind farms. An immediate next step would be to relax some of the constraints of DAMDPs further and allow for arbitrary directed graphs with edge attributes. We could then encode the relative distance between the wind turbines and consider an even broader set of wind farm layouts.

Also, the excellent performance of deep learning algorithms motivates the deep RL approach for this specific type of problem.

7.2.5 Learning the structure

Until now, we have assumed the structure given. While this is a reasonable assumption in many scenarios, it is not generally true. The structure could also be present but unknown. This motivates the study of algorithms that can learn this unknown latent structure.

An immediate challenge is that learning the structure might be even more difficult than solving the task itself, making it unlikely that any efficiency gains will be realised. For example, learning the correct hierarchical structure induces a search in an exponentially large space of possible hierarchical decompositions. Indeed, in such a scenario, any state could be considered a potential sub-goal. Even learning the right state abstraction, i.e., which state features describe high-level states and which set of features describe low-level states, would

also be challenging.

From an efficiency perspective, it is worth learning the structure if it is universal enough to be used in various tasks. Learning the structure would represent an initial effort that pays off when the agent discovers new tasks with the same shared structure. At this point, the agent could benefit from an efficiency boost associated with the learned structure.

We believe that the graph and hierarchical structure we considered are quite general and could be used in a wide range of tasks.

A natural way to approach such a solution would be to consider the problem of jointly learning the agent policy, the environment dynamics and the latent structure. One way to approach this problem is to compute a set of likely structures given the current beliefs about the dynamics. It would be interesting to consider an exploration strategy that aims to collect interactions to confirm or disprove the validity of the current structure. Alternatively, we could consider an algorithm that samples a given structure from a posterior distribution over plausible structures. This alternative approach raises the problem of planning with wrong or incomplete structure definitions.

When we consider the problem of learning the structure, we also face the issue that there might be more than one valid structure for a given MDP. If we think of the hierarchical structure and the n-rooms problem, we could have a hierarchical decomposition that does not consider a single room but a block of 4 rooms and learn to exit through one of the eight potential exits. However, some structures lead to more significant efficiency improvement than others. While tools like our lower bound could be instrumental in searching the space of valid structure, in practice, it is not immediately clear how to guide the search for the most efficient decompositions.

Another area for improvement is identifying whether an MDP exhibits additional structure. In the case of a hierarchical structure, this could motivate the decomposition of the MDP

into a hierarchy of arbitrary depth. After a first decomposition, the algorithm eventually realised that the first generation of low-level MDPs could be further decomposed, resulting in a deeper hierarchy, and so on. Building a more general structure, like a hierarchy of arbitrary depth, would require specialised algorithms that can adequately leverage the more complex emerging structures.

These questions are vastly unexplored, yet they might be vital to developing agents who can learn new tasks and efficiently leverage their previous experiences.

References

- [1] A. A. Faisal A. Robert, C. Pike-Burke. Code and resources for running Hierarchical Stationary Q-learning: doi.org/10.6084/m9.figshare.24291172.
- [2] Mahdi Abkar, Navid Zehtabiyani-Rezaie, and Alexandros Iosifidis. Reinforcement learning for wind-farm flow control: Current state and future actions. *Theoretical and Applied Mechanics Letters*, page 100475, 2023.
- [3] Ekaterina Abramova, Luke Dickens, Daniel Kuhn, and Aldo Faisal. Hierarchical, heterogeneous control of non-linear dynamical systems using reinforcement learning. In *European Workshop On Reinforcement Learning, at ICML*, volume 2012, 2012.
- [4] M.S. Adaramola and P.-Å. Krogstad. Experimental investigation of wake effects on wind turbine performance. *Renewable Energy*, 36(8):2078–2086, 2011.
- [5] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021.
- [6] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in Neural Information Processing Systems*, 30, 2017.
- [7] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.
- [8] J. Annoni, P. Fleming, A. Scholbrock, J. Roadman, S. Dana, C. Adcock, F. Porte-Agel, S. Raach, F. Haizmann, and D. Schlipf. Analysis of control-oriented wake modeling tools using lidar field results. *Wind Energy Science*, 3(2):819–831, 2018.
- [9] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *International Conference on Machine Learning*, volume 97, pages 12–20, 1997.
- [10] Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. *Advances in Neural Information Processing Systems*, 21, 2008.
- [11] Peter Auer and Ronald Ortner. Online regret bounds for a new reinforcement learning algorithm. In *Austrian Cognitive Vision Workshop*, pages 35–42. Österr. Computer-Ges., 2005.

- [12] Peter Auer and Ronald Ortner. Logarithmic online regret bounds for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 49–56, 2006.
- [13] Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. Minimax regret bounds for reinforcement learning. In *International Conference on Machine Learning*, pages 263–272. PMLR, 2017.
- [14] Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. Efficient exploration through bayesian deep q-networks. In *Information Theory and Applications Workshop*, pages 1–9. IEEE, 2018.
- [15] Pierre-Luc Bacon. *Temporal Representation Learning*. McGill University (Canada), 2018.
- [16] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.
- [17] Peter L Bartlett and Ambuj Tewari. Regal: A regularization based algorithm for reinforcement learning in weakly communicating mdps. *arXiv preprint arXiv:1205.2661*, 2012.
- [18] Majid Bastankhah and Fernando Porté-Agel. A new analytical model for wind-turbine wakes. *Renewable Energy*, 70:116–123, 2014. Special issue on aerodynamics of offshore wind energy systems and wakes.
- [19] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çağlar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew M. Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *Computing Research Repository*, abs/1806.01261, 2018.
- [20] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.
- [21] Benjamin Beyret, Ali Shafti, and A Aldo Faisal. Dot-to-dot: Explainable hierarchical reinforcement learning for robotic manipulation. In *International Conference on Intelligent Robots and Systems*, pages 5014–5019. IEEE, 2019.
- [22] Matthew Botvinick, Sam Ritter, Jane X Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, 23(5):408–422, 2019.
- [23] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

- [24] Ronen I Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3(Oct):213–231, 2002.
- [25] Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International Conference on Machine Learning*, pages 316–324. PMLR, 2014.
- [26] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials*, 31(9):3564–3572, 2019.
- [27] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3438–3445, Apr. 2020.
- [28] Xiaoyu Chen, Jiachen Hu, Lihong Li, and Liwei Wang. Efficient reinforcement learning in factored mdps with application to constrained rl. *arXiv preprint arXiv:2008.13319*, 2020.
- [29] Christoph Dann and Emma Brunskill. Sample complexity of episodic fixed-horizon reinforcement learning. *Advances in Neural Information Processing Systems*, 28, 2015.
- [30] Christoph Dann, Tor Lattimore, and Emma Brunskill. Unifying pac and regret: Uniform pac bounds for episodic reinforcement learning. *Advances in Neural Information Processing Systems*, 30, 2017.
- [31] Victor-Alexandru Darvari, Stephen Hailes, and Mirco Musolesi. Solving graph-based public goods games with tree search and imitation learning. *Advances in Neural Information Processing Systems*, 34:1739–1751, 2021.
- [32] Peter Dayan and Bernard W Balleine. Reward, motivation, and reinforcement learning. *Neuron*, 36(2):285–298, 2002.
- [33] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, 5, 1992.
- [34] Agostino De Marco, Paolo Maria D’Onza, and Sabato Manfredi. A deep reinforcement learning control approach for high-performance aircraft. *Nonlinear Dynamics*, 111(18):17037–17077, 2023.
- [35] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *International Conference on Machine Learning*, pages 257–264, 2006.
- [36] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *International Conference on Machine Learning*, pages 465–472, 2011.

- [37] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.
- [38] Thomas G Dietterich et al. The maxq method for hierarchical reinforcement learning. In *International Conference on Machine Learning*, volume 98, pages 118–126, 1998.
- [39] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5884–5888, 2018.
- [40] Gianluca Drappo, Alberto Metelli, and Marcello Restelli. A provably efficient option-based algorithm for both high-level and low-level learning. In *Reinforcement Learning Conference*, pages 1–21, 2024.
- [41] Gianluca Drappo, Alberto Maria Metelli, and Marcello Restelli. An option-dependent analysis of regret minimization algorithms in finite-horizon semi-MDP. *Transactions on Machine Learning Research*, 2023.
- [42] Xin Du, Jinjian Zhai, and Koupin Lv. Algorithm trading using q-learning and recurrent reinforcement learning. *Positions*, 1(1):1–7, 2016.
- [43] Jingliang Duan, Shengbo Eben Li, Yang Guan, Qi Sun, and Bo Cheng. Hierarchical reinforcement learning for self-driving decision-making without reliance on labelled driving data. *IET Intelligent Transport Systems*, 14(5):297–305, 2020.
- [44] Gregory Duthé, Francisco de Nolasco Santos, Imad Abdallah, Pierre Élouan Réthore, Wout Weijtjens, Eleni Chatzi, and Christof Devriendt. Local flow and loads estimation on wake-affected wind turbines using graph neural networks and pywake. *Journal of Physics: Conference Series*, 2505(1):012014, may 2023.
- [45] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
- [46] Sarah Filippi, Olivier Cappé, and Aurélien Garivier. Optimism in reinforcement learning and kullback-leibler divergence. In *Annual Allerton Conference on Communication, Control, and Computing*, pages 115–122. IEEE, 2010.
- [47] J. Fischereit, K. Schaldemose Hansen, X. G. Larsén, M. P. van der Laan, P.-E. Réthoré, and J. P. Murcia Leon. Comparing and validating intra-farm and farm-to-farm wakes across different mesoscale and high-resolution wake models. *Wind Energy Science*, 7(3):1069–1091, 2022.
- [48] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
- [49] Ronan Fruit and Alessandro Lazaric. Exploration-exploitation in mdps with options. In *Artificial Intelligence and Statistics*, pages 576–584. PMLR, 2017.
- [50] Ronan Fruit, Matteo Pirota, Alessandro Lazaric, and Emma Brunskill. Regret minimization in mdps with options without prior knowledge. *Advances in Neural Information Processing Systems*, 30, 2017.

- [51] Mohammad Ghavamzadeh, Shie Mannor, Joelle Pineau, Aviv Tamar, et al. Bayesian reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 8(5-6):359–483, 2015.
- [52] Andrew Glaws, Dylan Harrison-Atlas, Ryan King, and Eric Lantz. Scaling wind power innovation assessment for rapid energy transition with artificial intelligence. In *39th USAEE/IAEE North American Conference*, 1 2023.
- [53] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *International Conference on Robotics and Automation*, pages 3389–3396. IEEE, 2017.
- [54] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored mdps. *Advances in Neural Information Processing Systems*, 14, 2001.
- [55] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.
- [56] Tuomas Haarnoja, Kristian Hartikainen, Pieter Abbeel, and Sergey Levine. Latent space policies for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 1851–1860. PMLR, 2018.
- [57] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.
- [58] Michael F. Howland, Sanjiva K. Lele, and John O. Dabiri. Wind farm power optimization through wake steering. *Proceedings of the National Academy of Sciences*, 116(29):14495–14500, 2019.
- [59] Jean-Francois Hren and Rémi Munos. Optimistic planning of deterministic systems. In *European Workshop on Reinforcement Learning*, pages 151–164. Springer, 2008.
- [60] Matthias Hutsebaut-Buysse, Kevin Mets, and Steven Latré. Hierarchical reinforcement learning: A survey and open research challenges. *Machine Learning and Knowledge Extraction*, 4(1):172–221, 2022.
- [61] Ángel Jiménez, Antonio Crespo, and Emilio Migoya. Application of a LES technique to characterize the wake deflection of a wind turbine in yaw. *Wind Energy*, 13(6):559–572, 2010.
- [62] Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I Jordan. Is q-learning provably efficient? *Advances in Neural Information Processing Systems*, 31, 2018.
- [63] Nicholas K Jong, Todd Hester, and Peter Stone. The utility of temporal abstraction in reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 299–306, 2008.
- [64] Michael Kearns and Daphne Koller. Efficient reinforcement learning in factored mdps. In *IJCAI*, volume 16, pages 740–747, 1999.

- [65] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49:209–232, 2002.
- [66] H Kim, Michael Jordan, Shankar Sastry, and Andrew Ng. Autonomous helicopter flight via reinforcement learning. *Advances in Neural Information Processing Systems*, 16, 2003.
- [67] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.
- [68] Sven Koenig and Reid G Simmons. Complexity analysis of real-time reinforcement learning. In *Association for the Advancement of Artificial Intelligence*, volume 93, pages 99–105, 1993.
- [69] Matthieu Komorowski, Leo A Celi, Omar Badawi, Anthony C Gordon, and A Aldo Faisal. The artificial intelligence clinician learns optimal treatment strategies for sepsis in intensive care. *Nature Medicine*, 24(11):1716–1720, 2018.
- [70] George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, volume 7, pages 895–900, 2007.
- [71] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in Neural Information Processing Systems*, 29, 2016.
- [72] Sanjay Kumar, Abhishek Mallik, Anavi Khetarpal, and B.S. Panda. Influence maximization in social networks using graph embedding and graph neural network. *Information Sciences*, 607:1617–1636, 2022.
- [73] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.
- [74] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.
- [75] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.
- [76] Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight. *arXiv preprint arXiv:1712.00948*, 2017.
- [77] Gen Li, Laixi Shi, Yuxin Chen, Yuejie Chi, and Yuting Wei. Settling the sample complexity of model-based offline reinforcement learning. *arXiv preprint arXiv:2204.05275*, 2022.
- [78] Siyi Li, Mingrui Zhang, and Matthew D. Piggott. End-to-end wind turbine wake modelling with deep graph representation learning. *Applied Energy*, 339:120928, 2023.

- [79] Tingting Liang, Xuan Sheng, Li Zhou, Youhuizi Li, Honghao Gao, Yuyu Yin, and Liang Chen. Mobile app recommendation via heterogeneous graph neural network in edge computing. *Applied Soft Computing*, 103:107162, 2021.
- [80] TP Lillicrap. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
- [81] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, Furu Wei, and Baining Guo. Swin transformer v2: Scaling up capacity and resolution. In *International Conference on Computer Vision and Pattern Recognition*, 2022.
- [82] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017.
- [83] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.
- [84] Hongliang Ma, Mingwei Ge, Guangxing Wu, Bowen Du, and Yongqian Liu. Formulas of the optimized yaw angles for cooperative control of wind farms with aligned turbines to maximize the power production. *Applied Energy*, 303:117691, 2021.
- [85] Zoubir Mammeri. Reinforcement learning based routing in networks: Review and classification of approaches. *IEEE Access*, 7:55916–55950, 2019.
- [86] Shie Mannor, Yishay Mansour, and Aviv Tamar. Reinforcement learning: Foundations. *Online manuscript*, 2022.
- [87] Amy McGovern, Richard S Sutton, and Andrew H Fagg. Roles of macro-actions in accelerating reinforcement learning. In *Grace Hopper celebration of women in computing*, volume 1317, page 15, 1997.
- [88] Volodymyr Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [89] Volodymyr Mnih. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.
- [90] Brett L Moore, Eric D Sinzinger, Todd M Quasny, and Larry D Pyeatt. Intelligent control of closed-loop sedation in simulated icu patients. pages 109–114, 2004.
- [91] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.
- [92] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.
- [93] Mirco Mutti, Riccardo De Santi, Marcello Restelli, Alexander Marx, and Giorgia Ramponi. Exploiting causal graph priors with posterior sampling for reinforcement learning. *arXiv preprint arXiv:2310.07518*, 2023.

- [94] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018.
- [95] Ofir Nachum, Haoran Tang, Xingyu Lu, Shixiang Gu, Honglak Lee, and Sergey Levine. Why does hierarchy (sometimes) work so well in reinforcement learning? *arXiv preprint arXiv:1909.10618*, 2019.
- [96] Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *International Conference on Machine Learning*, pages 673–680, 2006.
- [97] Ian Osband. *Deep exploration via randomized value functions*. Phd thesis, Stanford University, 2016. Available at <https://searchworks.stanford.edu/view/11891201>.
- [98] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in Neural Information Processing Systems*, 29, 2016.
- [99] Ian Osband, Daniel Russo, and Benjamin Van Roy. (more) efficient reinforcement learning via posterior sampling. *Advances in Neural Information Processing Systems*, 26, 2013.
- [100] Ian Osband and Benjamin Van Roy. Near-optimal reinforcement learning in factored mdps. *Advances in Neural Information Processing Systems*, 27, 2014.
- [101] Ian Osband and Benjamin Van Roy. On lower bounds for regret in reinforcement learning. *arXiv preprint arXiv:1608.02732*, 2016.
- [102] Ian Osband and Benjamin Van Roy. Why is posterior sampling better than optimism for reinforcement learning? In *International Conference on Machine Learning*, pages 2701–2710. PMLR, 2017.
- [103] Ian Osband, Benjamin Van Roy, Daniel J Russo, and Zheng Wen. Deep exploration via randomized value functions. *Journal of Machine Learning Research*, 20(124):1–62, 2019.
- [104] Junyoung Park and Jinkyoo Park. Physics-induced graph neural network: An application to wind-farm power estimation. *Energy*, 187:115883, 2019.
- [105] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. nips autodiff workshop. In *Website <https://openreview.net/forum>*, 2017.
- [106] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

- [107] Mads M. Pedersen, Alexander Meyer Forsting, Paul van der Laan, Riccardo Riva, Leonardo A. Alcayaga Romàn, Javier Criado Risco, Mikkel Friis-Møller, Julian Quick, Jens Peter Schøler Christiansen, Rafael Valotta Rodrigues, Bjarke Tobias Olsen, and Pierre-Elouan Réthoré. PyWake 2.5.0: An open-source wind farm simulation tool. 2023.
- [108] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021.
- [109] Marc Pickett and Andrew G Barto. Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. In *International Conference on Machine Learning*, volume 19, pages 506–513, 2002.
- [110] Matthew D. Piggott, Stephan C. Kramer, Simon W. Funke, David M. Culley, and Athanasios Angeloudis. Optimization of marine renewable energy systems. In *Comprehensive Renewable Energy, 2nd Edition*, volume 8, pages 176–220. Elsevier, 2022.
- [111] Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science Advances*, 4(7):eaap7885, 2018.
- [112] Fernando Porté-Agel, Majid Bastankhah, and Sina Shamsoddin. Wind-turbine and wind-farm flows: A review. *Boundary-Layer Meteorology*, 174(1):1–59, Jan 2020.
- [113] Doina Precup and Richard S Sutton. Multi-time models for temporally abstract planning. *Advances in Neural Information Processing Systems*, 10, 1997.
- [114] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [115] Huilin Qu, Congqiao Li, and Sitian Qian. Particle Transformer for jet tagging. In *International Conference on Machine Learning*, pages 18281–18292, 2022.
- [116] Ahmed H Qureshi, Anthony Simeonov, Mayur J Bency, and Michael C Yip. Motion planning networks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2118–2124. IEEE, 2019.
- [117] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a General, Powerful, Scalable Graph Transformer. *Advances in Neural Information Processing Systems*, 35, 2022.
- [118] Herbert Robbins and Sutton Monroe. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.
- [119] Benjamin Rolf, Ilya Jackson, Marcel Müller, Sebastian Lang, Tobias Reggelin, and Dmitry Ivanov. A review on reinforcement learning algorithms and applications in supply chain management. *International Journal of Production Research*, 61(20):7151–7179, 2023.
- [120] John Schulman. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

- [121] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [122] Nima Sedaghatizadeh, Maziar Arjomandi, Richard Kelso, Benjamin Cazzolato, and Mergen H. Ghayesh. Modelling of wind turbine wake using large eddy simulation. *Renewable Energy*, 115:1166–1176, 2018.
- [123] Sam Shleifer, Jason Weston, and Myle Ott. Normformer: Improved transformer pretraining with extra normalization, 2021.
- [124] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [125] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- [126] Eric D Sinzinger and Brett Moore. Sedation of simulated icu patients using reinforcement learning based control. *International Journal on Artificial Intelligence Tools*, 14(01n02):137–156, 2005.
- [127] Alexander L Strehl, Carlos Diuk, and Michael L Littman. Efficient structure learning in factored-state mdps. In *Association for the Advancement of Artificial Intelligence*, volume 7, pages 645–650, 2007.
- [128] Malcolm Strens. A bayesian framework for reinforcement learning. In *International Conference on Machine Learning*, volume 2000, pages 943–950, 2000.
- [129] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
- [130] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [131] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, 12, 1999.
- [132] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.
- [133] Shengpu Tang, Maggie Makar, Michael Sjoding, Finale Doshi-Velez, and Jenna Wiens. Leveraging factored action spaces for efficient offline reinforcement learning in healthcare. *Advances in Neural Information Processing Systems*, 35:34272–34286, 2022.
- [134] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

- [135] Zilong Ti, Xiao Wei Deng, and Hongxing Yang. Wake modeling of wind turbines using machine learning. *Applied Energy*, 257:114025, 2020.
- [136] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *ACM workshop on Hot Topics in Networks*, pages 185–191, 2017.
- [137] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [138] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. *Advances in Neural Information Processing Systems*, 29, 2016.
- [139] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549. PMLR, 2017.
- [140] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.
- [141] Letian Wang, Jie Liu, Hao Shao, Wenshuo Wang, Ruobing Chen, Yu Liu, and Steven L Waslander. Efficient reinforcement learning for autonomous driving with parameterized skills and priors. *arXiv preprint arXiv:2305.04412*, 2023.
- [142] Christopher J.C.H. Watkins. *Deep exploration via randomized value functions*. PhD thesis, Cambridge University, 1989. Available at https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.
- [143] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.
- [144] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.
- [145] Tsachy Weissman, Erik Ordentlich, Gadiel Seroussi, Sergio Verdu, and Marcelo J Weinberger. Inequalities for the ℓ_1 deviation of the empirical distribution. *Hewlett-Packard Labs, Tech. Rep*, 2003.
- [146] Zheng Wen, Doina Precup, Morteza Ibrahimi, Andre Barreto, Benjamin Van Roy, and Satinder Singh. On efficiency in hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 33:6708–6718, 2020.
- [147] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? In *Advances in Neural Information Processing Systems*, 2021.

- [148] Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. *Advances in Neural Information Processing Systems*, 31, 2018.
- [149] Jincheng Zhang and Xiaowei Zhao. Wind farm wake modeling based on deep convolutional conditional generative adversarial network. *Energy*, 238:121747, 2022.
- [150] Muchen Zhao and Vadim Linetsky. High frequency automated market making algorithms with adverse selection risk control via reinforcement learning. In *International Conference on AI in Finance*, pages 1–9, 2021.
- [151] Yufan Zhao, Michael R Kosorok, and Donglin Zeng. Reinforcement learning design for cancer clinical trials. *Statistics in Medicine*, 28(26):3294–3315, 2009.
- [152] Per Åge Krogstad and Pål Egil Eriksen. “Blind test” calculations of the performance and wake development for a model wind turbine. *Renewable Energy*, 50:325–333, 2013.
- [153] Lars Ødegaard Bentsen, Narada Dilp Warakagoda, Roy Stenbro, and Paal Engelstad. Wind park power prediction: Attention-based graph networks and deep learning to capture wake losses. *Journal of Physics: Conference Series*, 2265(2):022035, may 2022.

Index

- Action space, 14
- Bayesian regret, 22
- Bellman equations, 18
- Bellman optimality equations, 19
- Directed Acyclic Graph (DAG), 124
- Directed Acyclic Markov Decision Process (DAMDP), 125
- Finite Horizon Markov Decision Process, 13
- Hierarchical Markov Decision Process, 49
- Horizon, 14
- Initial state distribution, 14
- Lower bounds, 23
- Posterior Sampling on Graphs for RL (PSGRL), 138
- PSRL, 32
- Q-function, 18
- Q-learning, 34
- Q-UCB, 37
- re-usability index, 58
- Regret, 21
- Return, 21
- Reward function, 14
- Sample complexity, 22
- State space, 14
- Stationary Hierarchical Q-Learning (SHQL), 64
- Time-homogeneous model, 14
- Time-inhomogeneous model, 14
- Transition function, 14
- UCBVI, 28
- Upper bounds, 22
- Value function, 17