Université de Lille
École Doctorale MADIS

# Thèse de Doctorat

Spécialité **Informatique**

présentée par
**Nathan Grinsztajn**

---

# Reinforcement Learning for Combinatorial Optimization: Leveraging Uncertainty, Structure and Priors

---

## Apprentissage par Renforcement pour l'Optimisation Combinatoire: Exploiter l'Incertitude, les Structures et les Connaissances a priori

---

sous la direction de **Philippe Preux**.

---

Soutenue publiquement à **Villeneuve d'Ascq**, le **15 juin 2023** devant le jury composé de

| | | |
|---|---|---|
| Pr. Ludovic **Denoyer** | Ubisoft, Paris | Rapporteur |
| Pr. Sylvain **Lamprier** | Université d'Angers | Rapporteur |
| Pr. Clarisse **Dhaenens** | Université de Lille | Examinatrice |
| Dr. Zhiguang **Cao** | Agency for Science Technology and Research (A*STAR), Singapour | Examinateur |
| Pr. Emmanuel **Rachelson** | Isae-Supaéro, Toulouse | Président |
| Pr. Philippe **Preux** | Université de Lille | Directeur de thèse |

---

Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL),
UMR 9189 Équipe Scool, 59650, Villeneuve d'Ascq, France

# Remerciements

Mes premières lignes vont à mes parents, ainsi qu'à mes frères Léo et Max, pour leur soutien indéfectible. Je n'aurais jamais entrepris une telle aventure sans vos encouragements.

Je tiens à remercier chaleureusement mon directeur de thèse, Philippe Preux, pour son accompagnement, sa patience et son expertise. Aussi, je voudrais remercier toutes les personnes avec qui j'ai eu la chance de collaborer au cours de ces trois années: Olivier Beaumont, Johan Ferret, Matthieu Geist, Emmanuel Jeannot, Toby Johnstone, Louis Leconte, Edouard Oyallon, Olivier Pietquin, Bastien Tagliaro.

I would like to extend my gratitude to the jury members: Ludovic Denoyer and Sylvain Lamprier for their precious reporting on the manuscript, Clarisse Dhaenens, Zhiguang Cao, and Emmanuel Rachelson for their time and expertise in evaluating this thesis.

During this thesis, I also had the chance to intern at Instadeep, in the research team led by Tom Barrett. I would like to thank Tom for his guidance and his support, as well as the whole team for their warm welcome. I would also like to thank my colleagues at Instadeep, with whom I had the opportunity to work closely and learn immensely: Clément Bonnet, Daniel Furelos-Blanco, Shikha Surana, and Felix Chalumeau.

Que serait une thèse sans vie de laboratoire? Ces trois années ont été rythmées par les débats enflammés de la pause café, le club ciné et son fameux tableur compilant probablement toutes les métriques imaginables, les sessions escalades présidées par Mathieu, la retraite à Varengeville-sur-Mer en plein cœur du covid avec Edouard, Ariane, Dorian, Mathieu, et Antoine, les soirées jeux de société, les dégustations de bière lilloises, les déjeuners parfois au soleil... Je voudrais remercier tous les doctorants du laboratoire avec qui j'ai eu la chance de partager ces moments, et qui ont rendu cette thèse si agréable: Dorian, Edouard, Fabien, Hector, Marc, Mathieu, Matheus, Omar, Reda, Sarah, Xuedong, Yannis... Sans oublier Antoine et Toby, qui même s'ils n'étaient à l'époque que stagiaires, méritent définitivement de figurer sur cette liste. Enfin, je voudrais remercier les chercheurs: Emilie, Odalric, Jill-Jênn, et Debabrota pour leur accueil et leurs conseils.

Mais puisqu'une thèse ne se limite pas non plus à la vie de laboratoire, je voudrais remercier tous les amis qui m'ont accompagné pendant ces trois années, et qui ont rendu cette aventure

si riche: Arnaud, Alexandre, Raphael pour leurs golioteries, Louis M., Louis Q., Victor, Rémi, Marie-Caroline, Vincent R., Cyril et Naila, Alexandre M., Paul, Antoine B. Raphael N., Teven et Laeticia, Richard, Matei, Vincent A., Raphael F., Hadrien, Bea, Romain... Et tous les autres que j'oublie.

Enfin, merci Emeraude pour ton indéfectible soutien, ta patience, et ton amour.

# Résumé

Les problèmes d'optimisation combinatoire ont été largement étudiés, notamment en raison de leurs nombreuses applications (planification, logistique, distribution, investissement, production...) et de leur complexité. Coûteux à résoudre de manière exacte, les approches les plus populaires s'appuient sur des heuristiques pour prendre leurs décisions. Cependant, produire des heuristiques efficaces et performantes est un exercice difficile, d'autant plus dans des environnements réalistes avec de l'incertitude ou de la stochasticité.

Dans cette thèse de doctorat, nous étudions comment l'apprentissage par renforcement peut être utilisé en optimisation combinatoire pour automatiser la production de telles heuristiques. Après nous être intéressés à un exemple concret, l'ordonnancement de tâches, nous isolons plusieurs caractéristiques clés de ces problèmes, rencontrées en pratique: une possible incertitude sur les données, les décisions à prendre, voire la définition même du problème, et une structure forte, qui a la particularité d'être souvent connue ou partiellement connue a priori. Nous explorons différentes façons de tenir compte de ces caractéristiques dans le cadre de l'apprentissage par renforcement pour une large gamme de problèmes, sortant parfois du cadre strict de l'optimisation combinatoire.

# Abstract

Combinatorial optimization problems have been extensively studied due to their numerous applications (planning, logistics, distribution, investment, production...) and their complexity. As they are expensive to solve optimally, the most popular approaches rely on heuristics to make their decisions. However, formulating efficient and effective heuristics is challenging, especially in realistic environments with uncertainty and stochasticity.

This Ph.D. thesis studies how reinforcement learning can be used in combinatorial optimization to automate the production of such heuristics. After focusing on a concrete example, task scheduling, I isolate several key features of these problems, encountered in practice: a possible uncertainty on the data, the decisions to be taken, or even the definition of the problem itself, and a strong structure, which has the particularity of being often known or partially known a priori. I explore different ways to take into account these characteristics in reinforcement learning for a wide range of problems, whilst occasionally surpassing the strict boundaries of combinatorial optimization.

# Contents

# List of Acronyms

**A**

| | |
|---|---|
| **A2C** | Advantage Actor-Critic , 23, 31, 47, 55 |
| **ADP** | Approximate Dynamic Programming |
| **ASAP** | As Soon As Possible , 31, 34–37 |

**B**

| | |
|---|---|
| **BLAS** | Basic Linear Algebra Subprograms |

**C**

| | |
|---|---|
| **CNN** | Convolutional Neural Network |
| **CO** | Combinatorial Optimization , 13, 93, 122 |
| **COP** | Combinatorial Optimization Problem , 16, 20, 24, 25, 93, 94, 121–123 |
| **CP** | Critical Path , 29, 37, 175 |
| **CPU** | Central Processing Unit |
| **CVRP** | Capacitated Vehicle Routing Problem , 19, 20, 103, 159 |

**D**

| | |
|---|---|
| **d-MCTS** | data Monte Carlo Tree Search , 108, 109, 111, 116, 171 |
| **DAG** | Directed Acyclic Graph , 24–29, 31, 32, 34–36, 44, 168 |
| **DP** | Dynamic Programming , 16 |
| **DQN** | Deep Q-Network , 10 |

**DRL** Deep Reinforcement Learning , 26

**F**

**FC** Fully-Connected layer , 32

**G**

**GCN** Graph Convolutional Network , 30, 46, 47, 50

**GPU** Graphics Processing Unit

**H**

**HEFT** Heterogeneous Earliest-Finish Time , 43, 44, 46

**HPC** High-Performance Computing , 24, 33, 55, 122

**Hustler** HUman-adversarial Search TaiLored at Exploiting Recurrent mistake , 107, 108, 115, 117–120, 171, 172

**I**

**i.i.d.** independent and identically distributed

**ILP** Integer Linear Programming , 13, 16, 19

**K**

**KP** Knapsack Problem , 13, 16, 20, 75, 160

**L**

**LKH** Lin-Kernighan-Helsgaun , 17, 19

**LP** Linear Programming , 16

**LU** Lower Upper decomposition , 26, 39, 41, 42, 47, 49

**M**

**MAB** Multi-Armed Bandits

**MCT** Minimum Completion Time , 43, 44, 50, 51, 53, 54, 169

**MCTS** Monte-Carlo Tree Search , 102, 103, 107–111, 120

**MDP** Markov Decision Process , 7, 19, 25, 29

**ML** Machine Learning , 18, 94–96, 101, 103, 104, 156

**MLE** Maximum Likelihood Estimation

**MVC** Minimum Vertex Cover

**N**

**NN** Neural Network

**NP** Nondeterministic Polynomial

**O**

**OFU** Optimism in the Face of Uncertainty

**P**

**POMDP** Partially Observable Markov Decision Process

**PPO** Proximal Policy Optimization , 12, 69, 70, 72, 73, 139, 140, 142, 170, 172

**Q**

**QR** QR decomposition

**R**

**READYS** Reinforcement-learning Algorithm for Dynamic Scheduling

**ReLU** Rectified Linear Unit , 32, 139–141, 150

**RL** Reinforcement Learning , 9, 18–20, 23, 24, 26, 34–39, 42, 43, 46, 47, 53, 55, 59–62, 66–68, 71, 73, 93–98, 102–106, 109, 112, 121–123, 170, 175

**S**

**SAT** Boolean Satisfiability problem , 13, 24

**T**

**TRPO** Trust Region Policy Optimization

**TSP** Travelling Salesman Problem , 13–15, 17, 19, 20, 24, 93, 95, 97, 101–105, 123, 156, 157, 160, 162, 164, 172, 173, 175

**U**

**UCT** Upper Confidence bounds applied to Trees , 113

**V**

**VRP** Vehicle Routing Problem , 13, 18, 24

# List of Symbols

### Mathematical notations

| | |
|---|---|
| $\mathbb{N}$ | set of integers |
| $[n]$ | range of integers $\{1, \ldots, n\}$ |
| $\mathbb{R}_+$ | set of positive reals $\{\tau \in \mathbb{R} : \tau \geq 0\}$ |
| $\mathbb{R}$ | set of real numbers |
| $\mathbb{N}_+$ | set of positive integers $\mathbb{N} \cap \mathbb{R}_+$ |
| $\mathcal{L}_\infty$ | the set of all inputs $u$ with the property $\|\|u\|\| < \infty$ |
| $\|x\|$ | Euclidean norm for a vector $x \in \mathbb{R}^n$ |
| $\|z\|$ | absolute value |
| $I_n$ | the identity matrix with dimension $n \times n$ |
| $M^\top$ | transpose of a matrix $M$ |
| $\|A\|_2$ | the induced $L_2$ matrix norm $\max_{i\in[n]} \lambda_i(A^\top A)$ |
| $\|A\|_{\max}$ | the elementwise maximum norm $\|A\|_{max} = \max_{i\in[n],j\in[n]} \|A_{i,j}\|$, it is not sub-multiplicative |
| $\lambda(A)$ | the vector of eigenvalues of a matrix $A \in \mathbb{R}^{n\times n}$ |
| $P \prec 0$ $(P \succ 0)$ | a symmetric matrix $P \in \mathbb{R}^{n\times n}$ is negative (positive) definite |
| $x_1 \leq x_2$ | for two matrices $A_1, A_2 \in \mathbb{R}^{n\times n}$, (including vectors), the relation $A_1 \leq A_2$ is understood elementwise |
| $o(\cdot), \mathcal{O}(\cdot), \Omega(\cdot)$ | Landau notations for positive functions: $f(x) = o(g(x)$ means that $g(x) \neq 0$ and $f(x)/g(x) \to 0$ for $x \to \infty$, $f(x) = \mathcal{O}(g(x))$ means that there exists $x_0, K > 0$ such that $f(x) \leq Kg(x)$ from $x \geq x_0$, and $f(x) = \Omega(g(x))$ means $g(x) = \mathcal{O}(f(x))$ |

| | |
|---|---|
| $\mathbb{E}$ | expectation under a probabilistic model |
| $\mathbb{V}$ | variance under a probabilistic model |
| $\mathcal{H}$ | Shannon entropy $\mathcal{H}$ , 32 |
| $\mathcal{B}$ | Binomial distribution |
| $\mathcal{N}$ | Normal distribution |
| $\mathcal{U}(\mathcal{X})$ | uniform distribution on a measurable space $\mathcal{X}$ |

**Markov Decision Processes**

| | |
|---|---|
| $\mathcal{S}$ | set of states $s \in \mathcal{S}$ , 7 |
| $\mathcal{A}$ | set of actions $a \in \mathcal{A}$ , 7 |
| $R(s, a)$ | reward function $R : s, a \to R(s, a) \in [0, 1]$ , 7 |
| $P(s' \mid s, a)$ | transition distribution $s' \sim P(s' \mid s, a)$ , 7 |
| $T(s, a)$ | determinitstic transition function $s' = T(s, a)$ |
| $\gamma$ | discount factor in $[0, 1)$ , 7 |
| $\pi$ | policy , 8 |
| $\pi^\star$ | optimal policy , 9 |
| $G$ | discounted return for the reward signal , 8 |
| $J$ | objective function , 8 |
| $V$ | state value function ($^\star$ for optimal value, $^\pi$ for policy value) |
| $Q$ | state-action value function ($^\star$ for optimal value, $^\pi$ for policy value) , 9 |
| $\mathcal{T}$ | Bellman operator ($^\star$ for optimality, $^\pi$ for evaluation) , 9 |
| $\mathcal{D}$ | dataset |

# Chapter 1

# Introduction

## 1.1 Introductionary anecdote

A week after the beginning of the thesis, as I joined a group of people from my lab for lunch, I had the following discussion with a professor:

PROFESSOR: So you're the guy working on reinforcement learning for combinatorial optimization?

ME: Hey, yes, I just arrived!

PROFESSOR: It won't work.

I think I mumbled something about chess, but I'm not sure he had the time to develop his point of view, as it was cut off by another colleague who changed the subject. But this comment stayed in my mind for a little while. That is why I would like to start this thesis with a few arguments for why reinforcement learning shouldn't be used to solve combinatorial optimization problems:

1. Combinatorial optimization has been studied for decades. A lot of powerful algorithms have been developed to give exact or approximate solutions to important problems.
2. Combinatorial optimization problems are deterministic, while reinforcement learning is a general framework meant to deal with stochastic problems.
3. Combinatorial optimization problems generally have a structure. It is not exactly clear how to exploit this structure in reinforcement learning.

There are other arguments, which apply to reinforcement learning in general:

- Deep reinforcement learning suffers from poor sample efficiency.

- Deep reinforcement learning algorithms can be sensitive to hyperparameters, which may require extensive tuning for each combinatorial problem instance.
- Deep reinforcement learning is computationally expensive, which may make it impractical for large-scale combinatorial problems.
- Deep reinforcement learning may be stuck in local optima.
- The strategies found using reinforcement learning algorithms are not interpretable.

Leaving aside the last general arguments, where each bullet point is more than a thesis subject by itself, we have tried in this thesis to give some answers to the first three arguments. Apart from the first part of this thesis, which explores a concrete problem, the second part can be seen as an attempt at solving the second argument, while the third part attacks the third argument. We dive more into the details of the thesis' structure in the next section.

## 1.2 Outline and contributions

The general goal of this thesis is to explore the use of reinforcement learning for combinatorial optimization. What are the main challenges? Can they be overcome? What are the advantages of using reinforcement learning for such problems?

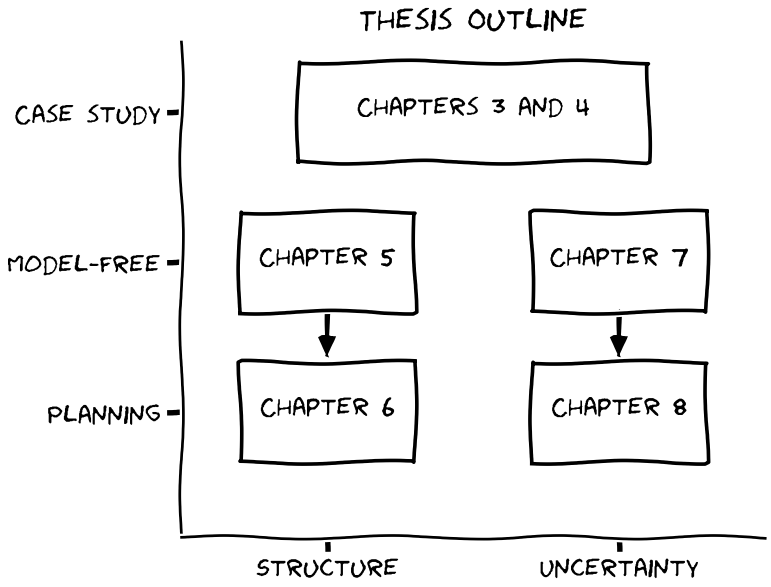



**Figure 1.1** – After a case study, we base the thesis around two divisions: planning *vs.* model-free, and leveraging structure and priors *vs.* dealing with uncertainty.

Part I is devoted to a case study of the problem of scheduling a set of jobs on a set of machines, and aims at giving some intuitions on building heuristics with reinforcement learning,

as well as some insights on the common challenges of using reinforcement learning for combinatorial optimization. It is divided into three chapters. Chapter 2 is a general introduction to reinforcement learning and combinatorial optimization. It is meant to give the reader a general idea of the field, and to provide a common vocabulary for the rest of the thesis. Moreover, after discussing *why reinforcement learning shouldn't be used for solving combinatorial problems* we will provide some motivation for the opposite point of view, namely *why is reinforcement learning a useful tool for solving combinatorial problems?*. Chapter 3 and chapter 4 study the use of reinforcement learning for dynamic scheduling problems. Chapter 3 tackles a simple environment, while Chapter 4 extends the previous work to a richer setting (larger set of scheduling problems, heterogeneous machines, noisy task durations, *etc.*)

The second part is structured around two divisions, as shown in Figure 1.1. The first one is whether or not a model of the environment is used. The second dichotomy revolves around the two challenges evoked in the previous section: *how to leverage the structure of combinatorial problems in reinforcement learning* and *how to deal with uncertainty*. The first part of the thesis is devoted to the first challenge, while the second part is devoted to the second challenge.

Part II tackles the problem of leveraging structure and priors in reinforcement learning. It is divided into two chapters, each one focusing on a different aspect of the problem. Chapter 5 studies the structure of a particular combinatorial problem, Sokoban, and proposes to introduce and exploit the notion of reversibility in the learning process. Chapter 6 studies how to utilize prior structure knowledge in reinforcement learning in a way that benefits exploration. Although both approaches are motivated primarily by tackling combinatorial problems, they tend to be more generally applicable. In both chapters, we additionally illustrate their effectiveness on a variety of non-combinatorial problems.

Part III focuses on the problem of uncertainty. It is divided into two chapters, each one focusing on a different aspect of the problem. Chapter 7 focuses on uncertainty at the decision level. It starts from the idea that combinatorial problems are too hard to be solved reliably in a single shot, and thus good heuristics should maintain a level of uncertainty about their decisions as the training progresses. This idea leads to using a population-based approach to learn a distribution over actions, instead of a single action. We show that this approach provides very good results on a variety of popular combinatorial problems. Chapter 8 focuses on uncertainty at the model level. The goal is to develop a planning approach that can be applied to fixed datasets used as an empirical model of the environment. We explore this idea in the context of chess to find the most efficient policies against opponents of specific levels, leveraging millions of past chess games.

## List of publications

### Publications in international conferences with proceedings

- Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux (2020). Geometric deep reinforcement learning for dynamic DAG scheduling. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning* (*ADPRL*). Virtual: IEEE Press (used in Chapter 4)

- Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux (2021b). Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. In *2021 IEEE International Conference on Cluster Computing* (*CLUSTER*). Virtual: IEEE Press (used in Chapter 4)

- Nathan Grinsztajn, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021a). There Is No Turning Back: A Self-Supervised Approach for Reversibility-Aware Reinforcement Learning. In *Advances in Neural Information Processing Systems* (*Neurips*). Virtual (used in Chapter 6)

### Workshop presentations in international conferences

- Nathan Grinsztajn, Toby Johnstone, Johan Ferret, and Philippe Preux (Dec. 2022). Better state exploration using action sequence equivalence. In *NeurIPS workshop on Deep Reinforcement Learning*. Virtual, United States (used in Chapter 5)

### Under review

- Nathan Grinsztajn, Daniel Furelos-Blanco, and Thomas D. Barrett (2022). *Population-Based Reinforcement Learning for Combinatorial Optimization*. Submitted to ICML 2023, under review. (used in Chapter 7)

- Nathan Grinsztajn and Philippe Preux (2023). *Efficient Search of Human Adversarial Policies in Chess*. Submitted to IJCAI 2023, under review. (used in Chapter 8)

### Collaborations not presented in this thesis

- Manh Hung Nguyen, Lisheng Sun-Hosoya, Nathan Grinsztajn, and Isabelle Guyon (July 2022). Meta-learning from Learning Curves: Challenge Design and Baseline Results. In *IJCNN 2022 - International Joint Conference on Neural Networks*. Padua, Italy: IEEE, pp. 1–8

- Manh Hung Nguyen, Nathan Grinsztajn, Isabelle Guyon, and Lisheng Sun-Hosoya (2021). MetaREVEAL: RL-based Meta-learning from Learning Curves. In *Workshop on Interactive Adaptive Learning co-located with European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (*ECML PKDD 2021*). Virtual, Bilbao
- Nathan Grinsztajn, Philippe Preux, and Edouard Oyallon (May 2021). Low-Rank Projections of GCNs Laplacian. In *ICLR 2021 Workshop GTRL*. Virtual
- Nathan Grinsztajn, Louis Leconte, Philippe Preux, and Edouard Oyallon (2021). Interferometric Graph Transform for Community Labeling. *CoRR* abs/2106.05875

### Statement of Authorship

**There is no Turning Back: A Self-Supervised Approach to Reversibility-Aware Reinforcement Learning** Nathan Grinsztajn, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021a). There Is No Turning Back: A Self-Supervised Approach for Reversibility-Aware Reinforcement Learning. In *Advances in Neural Information Processing Systems* (*Neurips*). Virtual

JF proposed the initial idea and research direction. NG and JF led the research project with OP, PP and MG serving as advisors. JF performed early experiments to validate the idea. NG came up with the formalism and wrote the theoretical analysis. MG proof checked these. NG did the Turf and Cartpole experiments while JF designed the Sokoban experiments and analysis. NG and JF equally contributed to the writing of the article. MG, PP and OP also contributed to the writing

**More Efficient Exploration with Symbolic Priors on Action Sequence Equivalences** Nathan Grinsztajn, Toby Johnstone, Johan Ferret, and Philippe Preux (Dec. 2022). Better state exploration using action sequence equivalence. In *NeurIPS workshop on Deep Reinforcement Learning*. Virtual, United States

NG proposed the research direction and the collaboration. TJ conducted early experiments and designed the first bits of the method. NG coded the part involving the graph construction, while TJ the solving of the convex problem. NG ran the pure exploration experiments, TJ ran Minigrid and Catcher experiments, and JF ran Atari experiments. JF proposed additional experiments, while NG analyzed the results. NG and TJ equally contributed to the writing of the article. JF gave feedback on various drafts of the article. PP supervised the project.

# Chapter 2

# Background

## 2.1 Reinforcement Learning

This section constitutes a brief introduction to reinforcement learning (RL), a subfield of machine learning that focuses on sequential decision-making. For a more in-depth introduction, we refer the reader to Sutton and Barto (2018).

### 2.1.1 Markov Decision Processes

Achieving a complex goal often requires making a series of thoughtful decisions that lead to a particular outcome. This is known as sequential decision-making, and it is a fundamental problem in many areas of artificial intelligence, including robotics, game playing, autonomous driving, and, as we will see, combinatorial optimization. In this setting, a decision maker, or *agent*, takes a sequence of decisions, *actions*, in an *environment*. To decide which action to take, the agent can base itself on the current environment's *state*, which changes each time an action is taken according to a *transition function*, which can be understood as the system dynamics. The goal of the agent is to find a strategy (*policy*) that maximizes some numerical *reward*.

We now introduce some notations. Formally, this type of problem is often formulated as a Markov Decision Process (MDP), which adds the hypothesis that the environment is *Markovian*, meaning that the next state only depends on the current state and action, and not on the whole history.

**Definition 2.1** (Markov decision process)**.** *A* Markov decision process (*MDP*) *is a tuple* $(\mathcal{S}, \mathcal{A}, P, R)$*, where* $\mathcal{S}$ *is a set of states,* $\mathcal{A}$ *is a set of actions,* $P$ *is a transition function,* $R$ *is a reward function, and* $\gamma$ *is a discount factor. At each time step* $t$*, the agent is in a state* $s_t \in \mathcal{S}$*, and it takes an*

*action $a_t \in \mathcal{A}$. The environment then transitions to a new state $s_{t+1} \in \mathcal{S}$, drawn from a conditional distribution $P(s_{t+1} \mid s_t, a_t)$, and receives a bounded reward $R(s_t, a_t)$.*

**Remark 2.2.** *For deterministic environments, we modify our notations slightly and write the transition function as $s_{t+1} = P(s_t, a_t)$.*

A policy $\pi$ maps any state $s \in \mathcal{S}$ into a probability distribution over $\mathcal{A}$. It specifies the agent's strategy for choosing its next action, sampled from $\pi(\cdot \mid s)$.

### 2.1.2 Objective and value functions

The goal of an agent is to find a policy that maximizes the expected cumulative reward over time, which is known as the *objective function*.

**Definition 2.3** (Objective function)**.** *The objective function is defined as:*

$$\begin{aligned} J(\pi) &= \mathop{\mathbb{E}}_{\pi}\left(\sum_t \gamma^t R(s_t, a_t) \mid \delta_0\right), \\ &= \mathop{\mathbb{E}}_{\pi}\left(G^\pi \mid \delta_0\right), \end{aligned}$$

*where $\delta_0$ is the initial state distribution, $\gamma \in [0, 1)$ is the discount factor, a parameter that determines how much importance the agent places on future rewards, and $G^\pi$ is the policy return, which is the random variable $G^\pi \triangleq \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$. In most cases, the horizon $T$ is considered finite, such that $G^\pi = \sum_{t=0}^{T} \gamma^t R(s_t, a_t)$.*

In the same way that the objective function is the expected cumulative reward, the *value function* $V^\pi(s)$ is the expected return starting from state $s$ and following policy $\pi$, while the action-value (or *Q-value*) function $Q^\pi(s, a)$ is the expected return starting from state $s$, taking action $a$, and following policy $\pi$.

**Definition 2.4** (Value functions)**.** *The value function of a policy $\pi$ is defined as:*

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\pi}\left(\sum_t \gamma^t R(s_t, a_t) \mid s_0 = s\right), \\ &= \mathop{\mathbb{E}}_{\pi}\left(G^\pi \mid s_0 = s\right). \end{aligned}$$

*Similarly, the action-value function of a policy $\pi$ is defined as:*

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\pi}\left(\sum_t \gamma^t R(s_t,a_t) \mid s_0 = s, a_0 = a\right),$$
$$= \mathop{\mathbb{E}}_{\pi}\left(G^\pi \mid s_0 = s, a_0 = a\right).$$

The goal of Reinforcement Learning is to find a policy $\pi^\star$.

**Definition 2.5** (Optimality). *A policy $\pi^\star$ is said to be optimal if it maximizes the value functions $V^\pi$ and $Q^\pi$ in every state and action. We call $V^\star \triangleq V^{\pi^\star}$ the* optimal value function *and $Q^\star \triangleq Q^{\pi^\star}$ the* optimal Q function *in any state and action:*

$$\forall s \in \mathcal{S}, \qquad V^\star(s) = \max_\pi V^\pi(s) = V^{\pi^\star}(s),$$
$$\forall (s,a) \in \mathcal{S}\times\mathcal{A}, \qquad Q^\star(s,a) = \max_\pi Q^\pi(s,a) = Q^{\pi^\star}(s,a).$$

### 2.1.3 Learning to act

After introducing the framework of Reinforcement Learning (RL), we now turn to the problem of learning a policy that maximizes the objective function. The first line of approach focuses on learning a value function that approximates the optimal value functions, while the second line of approaches directly optimizes the policy.

#### Value-based methods

**Theorem 2.6** (Bellman Optimality Equation, Bellman, 1957). *The optimal Q function $Q^\star$ is a fixed point of the Bellman optimality operator $\mathcal{T}$:*

$$Q^\star(s,a) = (\mathcal{T}Q^\star)(s,a) \triangleq \mathop{\mathbb{E}}_{s'\sim P(s'|s,a)} \max_{a'\in\mathcal{A}} \left[R(s,a) + \gamma Q^\star(s',a')\right].$$

*It can also be shown that $\mathcal{T}$ is a $\gamma$-contraction for the $\|\cdot\|_\infty$ norm, which implies that $Q^\star$ is the unique fixed point of $\mathcal{T}$.*

An application of this theorem is the Q-learning algorithm (Watkins and Dayan, 1992), which is a value-based method that learns the Q function by iteratively updating randomly initialized values $\hat{Q}$ with the Bellman optimality operator.

**Algorithm 2.1:** Q-learning

Initialize $\hat{Q}(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ ;
**for** $i = 1$ *to* $N$ **do**
  Sample $s \sim \delta_0$ ;
  **for** $t = 1$ *to* $T$ **do**
    Sample $a \sim \pi(\cdot \mid s)$ ; `/* for some policy π */`
    Sample $s' \sim P(s' \mid s, a)$ ;
    $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left( R(s, a) + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right)$ ;
    $s \leftarrow s'$ ;

Explicitly storing the Q function is not always possible, especially when the state space is large. Instead, we can use a neural network to approximate the Q function. The neural network, *Q network*, takes a state as input and produces a probability distribution on the set of possible actions as output, which should reflect their expected performance when performed in the input state. Since it is impossible to exhaustively visit all states within a reasonable amount of time, we take advantage of the ability of a neural network to generalize from its observations. The idea to use a neural network to approximate value function was first proposed by Tesauro (1995) to learn to play backgammon at the level of top human players.

Two decades later, this idea resurfaces as Deep Q-Network (DQN) (Volodymyr Mnih, Kavukcuoglu, et al., 2015) to play Atari games at a superhuman level solely from image pixels. The architecture used in the paper is now widely adopted, with slight modifications. It is composed of two neural networks: a target network $Q_\theta$ and a behavior network $Q_{\theta'}$. The behavior network is updated at each step of the algorithm using the Bellman optimality operator, using the target network to compute the loss:

$$\ell(s_t, a_t, s_{t+1}, \theta) = \left( Q_\theta(s_t, a_t) - \left( r_t + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a') \right) \right)^2 .$$

The target network is updated periodically by copying the parameters of the behavior network. The target network is a delayed and more stable version of the behavior network, which helps to smooth the learning process.

During training, the replay buffer is used to sample a batch of transitions to update the behavior network using the loss function (8). This has the advantage of making the training process more sample-efficient since the same transition can be used for multiple updates. The

replay buffer allows the agent to learn from past experiences and prevent it from forgetting important information.

### 2.1.4 Policy-based methods

#### Policy gradient methods

From now on, we write $\pi_\theta$ to denote a policy parameterized by $\theta$. Maximizing is now equivalent to finding $\theta^\star = \arg\max_\theta J(\pi_\theta)$. It turns out that the gradient of the objective function with respect to the policy parameters $\theta$ can be estimated through Monte Carlo estimation, using the following theorem.

**Theorem 2.7** (Policy gradient theorem, Sutton, McAllester, et al., 1999)**.** *The gradient of the expected return with respect to the policy parameters $\theta$ is proportional to:*

$$\nabla_\theta J(\pi_\theta) \propto \mathop{\mathbb{E}}_{\pi_\theta}\left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\left(\sum_{t'=t}^{T} \gamma^{t'} R(s_{t'}, a_{t'})\right)\right),$$

This theorem gives an explicit way of computing the gradient of the objective function with respect to the policy parameters $\theta$ and is the basis of policy gradient methods, the class of methods that directly optimize the policy parameters $\theta$.

This theorem can be used to derive the REINFORCE algorithm (Williams, 1992a), which is a policy-based method that learns the policy by iteratively updating the policy parameters $\theta$ by estimating the gradient of the expected return. It first gathers a set of trajectories using the current policy $\pi_\theta$, and then updates the policy parameters $\theta$ replacing the expectancy with the average. The algorithm is summarized in Algorithm 2.2.

**Algorithm 2.2:** REINFORCE

Initialize policy parameters $\theta$ arbitrarily ;
**for** *each episode* **do**
  Sample a trajectory $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_T, a_T, r_T)$ using $\pi_\theta$ ;
  **for** $t = 0$ *to* $T$ **do**
    $G_t \leftarrow \sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'})$ ;
    $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ ;

It is an *on-policy* method, which means that the policy is updated using the same policy that is used to collect the data. It is in contrast to the previous Q learning algorithm, where any policy can be used to collect the data, and is therefore *off-policy*.

Interestingly, it is possible to add a *baseline*, which is a function of the state $s$ that is subtracted from the return $\sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'})$ in the theorem. This is done to reduce the variance of the gradient estimator (Weaver and Tao, 2001), without changing its expectation. The intuitive idea is that the absolute return obtained from a state should be assessed relative to the state quality, and to the other returns obtained from the same state. The final gradient formula is thus:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\pi_\theta} \left( \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left( \sum_{t'=t}^{T} \gamma^{t'-t} R(s_{t'}, a_{t'}) - b(s) \right) \right).$$

REINFORCE with baseline is used in Chapter 7. The baseline $b$ can be implemented in various ways, for example as a neural network.

### Actor-critic algorithm

*Actor-critic methods* is a general framework, first proposed in Barto, Sutton, and C. W. Anderson (1983b) to balance a pole on a moving cart. It corresponds to replacing the previous baseline $b$ by an approximation of the value function, generally using a neural network $V_\phi$. $V$ has indeed been shown to be a good baseline for the policy gradient theorem (Greensmith, Bartlett, and Baxter, 2004), although other baselines have been explored (Flet-Berliac et al., 2021b). The neural network $V_\phi$ is called the *critic*, and the policy network $\pi_\theta$ is called the *actor*. This class of algorithm is used in Chapter 3 and Chapter 4.

### Trust region optimization

The actor-critic algorithm has been improved upon through various methods, including the trust region policy optimization (TRPO) algorithm (Schulman, Levine, et al., 2015a). In TRPO, a constraint is introduced on the policy update to prevent excessive changes to the policy parameters $\theta$. This constraint is imposed by limiting the Kullback-Leibler (KL) divergence between the new policy $\pi_{\theta'}$ and the previous policy $\pi_\theta$, where $\theta'$ represents the new policy parameters. By doing so, TRPO ensures stable policy updates and prevents catastrophic changes that may result from excessively large updates.

The *Proximal Policy Optimization (PPO)* algorithm (Schulman, Wolski, et al., 2017) is a more recent algorithm that simplifies the TRPO algorithm by replacing the KL divergence constraint with a clipped surrogate objective function. Due to its simple implementation and high performance, it has become one of the most commonly used policy gradient methods. This algorithm is used in Chapter 5 and Chapter 6.

## 2.2 Combinatorial Optimization

### 2.2.1 Combinatorial optimization problems

Combinatorial Optimization (CO) is a fundamental branch of mathematics and computer science concerned with finding the best solution from a finite set of possible options. Combinatorial optimization problems arise in many fields, including engineering, economics, operations research, and their applications are numerous: scheduling, routing, resource allocation, portfolio optimization, logistics, *etc.*

The most iconic combinatorial optimization problem is probably the Travelling Salesman Problem (TSP), which consists in finding the shortest tour that visits all the cities in a given set, and returns to the initial point. It has found applications in several domains, like genome sequencing, data clustering, or aiming telescopes (Applegate et al., 2006). The TSP is a well-studied problem and has been used as a benchmark for many algorithms. There are many popular combinatorial optimization problems, such as:

- Knapsack Problem (KP): given a set of items each characterized by its weight and its value, the goal is to select a subset of items to maximize the total value subject to a constraint on the total weight.
- Vehicle Routing Problem (VRP): this problem is a variation of the TSP. Given a set of customers with demands and a fleet of vehicles, the goal is to find the optimal set of routes for the vehicles to serve all the customers while minimizing the total distance traveled.
- Integer Linear Programming (ILP): given a set of linear constraints and a linear objective function, the goal is to find integer values for the variables that maximize the objective function while satisfying the constraints.
- Boolean Satisfiability problem (SAT): the goal is to find an assignment of truth values that satisfies a given Boolean formula.

A great variety of such problems is defined on graphs, which are a natural way to represent many combinatorial optimization problems. As examples, let us mention:

- Graph Coloring Problem: the goal is to assign colors to the vertices such that no two adjacent vertices have the same color, while minimizing the total number of colors used.
- Maximum Cut: Given an undirected graph, the maximum cut problem is to partition the vertices into two sets such that the number of edges crossing the partition is maximized.

- Minimum Vertex Cover: Given an undirected graph, the minimum vertex cover problem is to find the smallest set of vertices such that every edge has at least one endpoint in the set.

Throughout the thesis, we will refer to "problem" as a general description of an optimization problem, which includes the objective function, constraints, and input/output format, and as a "problem instance", on the other hand, is a specific input to the problem that includes all the necessary data to define it.

For instance, in the case of the TSP, the problem can be described as finding the shortest possible route that visits each city exactly once and returns to the starting city, given a set of cities and the distances between them. On the other hand, a problem instance would be a specific set of cities and their positions, or their distances from each other.

Natural questions arise when considering this variety of problems: "How hard are they?". Is it possible to solve them efficiently? Is there a way to formalize the notion of hardness? This is the subject of the next section.

### 2.2.2 NP-hardness

For an optimization problem, there is an associated decision problem, which is the problem of deciding whether a solution of some quality exists. For example, the decision problem of the TSP is to decide whether a tour of length smaller than some constant $L$ exists. The concept of *NP-hardness*, which we are going to define, is a way to formalize the notion of intractability of a decision problem. Throughout this thesis however, we will often use "NP-hardness" to refer to optimization problems when the associated decision problem is NP-hard.

Historically, Edmonds (1965) was one of the first papers to hint at establishing a mathematical theory of efficient combinatorial algorithms. It contained the conjecture that no polynomial algorithm can solve the TSP, which remains open to this day. We now define more formally these notions.

**Definition 2.8** (Class P (informal definition))**.** *Class P is defined as the set of problems that can be solved in polynomial time (with respect to the size of the input).*

**Definition 2.9** (Class NP (informal definition))**.** *A problem is in NP if proposed solutions can be verified in polynomial time (with respect to the size of the input).*

Intuitively, problems in P should be considered easy, while problems in NP should be considered difficult.

As an example of a problem in NP, consider the previous decision problem of the TSP. If a given instance admits a tour which length is smaller than $L$, a proof can simply consist in exhibiting a valid tour of length smaller than $L$. Checking that the tour has a length smaller than $L$ can be done in polynomial time, and therefore the decision problem of the TSP is in NP.

To "order" the hardness of problems, we can define the notion of reduction between problems:

**Definition 2.10** (Reduction (informal))**.** *A problem $X$ is reducible to problem $Y$ if given a polynomial time algorithm that solves $Y$, it is possible to solve $X$ in polynomial time. Intuitively, if $X$ is reducible to $Y$, then $Y$ is at least as hard as $X$.*

This allows to coin the terms "NP-complete", and "NP-hard" (Knuth, 1974).

**Definition 2.11** (NP-hard problems (informal definition))**.** *A problem is NP-hard if every problem in NP can be reduced to it in polynomial time ("it is at least as hard as every problem in NP"). A problem is NP-complete if it is NP-hard and in NP. This is the hardest class of problems in NP.*

This classification has proved to be very useful in the study of combinatorial optimization problems. A lot of important problems have turned out to be NP-complete, as first shown by Karp (1972), who exhibited 21 NP-complete problems, spanning across a wide range of domains. In fact, all the problems mentioned previously are NP-complete.

From the definition of NP-completeness, it is to be expected that the size of a solution should be polynomial, which implies that the size of the search space of potential solutions is exponential. Although it does not guarantee that NP-complete problems cannot be solved in less than exponential time, there is currently no known polynomial time algorithm for solving these problems. This is the famous *P vs. NP* problem, which is still open to this day. In what follows, we will implicitly focus on NP-hard problems, as P problems are usually easily solvable.

In the next section, we will dive into the various approaches that have been proposed to solve NP-complete problems.

### 2.2.3 Solving hard combinatorial optimization problems

Due to the practical importance of NP-hard problems, and although it is believed that no polynomial time algorithm exists for solving NP-complete problems, many algorithms have been developed to tackle them efficiently.

### Exact methods

Exact methods are a class of optimization algorithms that aim to find the exact optimal solution by exploring the search space and eliminating infeasible or inferior solutions. One of the most widely used algorithms in this class is Branch and Bound (B&B), which decomposes Combinatorial Optimization Problems (COPs) into smaller sub-problems and prunes the search space by using a bounding function to eliminate sub-problems that cannot contain the optimal solution. B&B is particularly effective for solving ILPs by exploiting the fact that dropping the integer constraint results in a well-solvable Linear Programming (LP) problem. Sub-problems are represented as a search tree, where each node represents a relaxed LP version of the initial problem. If the LP is impossible, that node is not processed further. Otherwise, the LP is solved. If a variable has a fractional value, one creates two child nodes by restricting the value of the chosen variable in each node. Interestingly, solving the LP gives an upper bound on the quality of solutions which can be found by descending the tree from this node, which allows pruning the search space.

This pruning mechanism makes this approach very efficient in practice. Furthermore, many complex problems can be formulated as ILP, including well-known problems such as the traveling salesman problem (TSP) (Miller, Tucker, and Zemlin, 1960; Dantzig, Fulkerson, and S. Johnson, 1954). As a result, branch and bound is a widely used and practical method for solving a variety of COPs. In fact, many commercial optimization solvers rely on branch and bound to solve challenging problems.

In practice, when faced with a new COP, one of the first approaches is to attempt to formulate it as an ILP problem, as this approach can often yield efficient and effective solutions.

For some problems, Dynamic Programming (DP) (Bellman, 1957) can be used. It is a method used to solve problems by breaking them down into smaller sub-problems and solving them in a recursive manner. It works by identifying overlapping subproblems, solving each subproblem only once, and then storing the solution for future reference. This can lead to significant improvements in performance, as the time required to solve the same subproblem is greatly reduced. It is often used to solve KP for example, by solving for the optimal value of packing a subset of the items.

### Heuristics

When the previous approaches are not applicable, a vast literature is devoted to the study of *heuristics*. Heuristics are algorithms that do not guarantee an optimal solution, but instead quickly generate near-optimal, or "good enough" solutions.

Not as mathematically sound as exact methods, the field has historically suffered from some suspicion, as illustrated by this citation from Glover (1977): "[Exact] algorithms have long constituted the more respectable side of the family, assuring an optimal solution in a finite number of steps. Methods that merely claim to be clever [...] are accorded lower status. [Exact] algorithms are conceived in analytic purity in the high citadels of academic research, heuristics are midwifed by expediency in the dark corners of the practitioner's lair."

Although, due to their practical efficiency, heuristics have grown in popularity. They are often used to solve large-scale problems, as they are often much faster than exact methods.

Broady speaking, heuristics can be categorized into two categories: *problem-dependant heuristics*, and *meta-heuristics*. Problem-dependant heuristics are tailored to a specific problem, and are often based on the structure of the problem. Meta-heuristics are a class of more general algorithms which can be applied to a wide range of problems.

Problem-dependent heuristics are tailored to take advantage of the unique features of the problem and can often provide high-quality solutions in a relatively short amount of time. These algorithms are generally based on problem-specific knowledge and expertise, such as heuristics based on the geometry of the problem domain or local search methods that exploit the problem structure. Examples of problem-dependent heuristics include the greedy algorithm for the knapsack problem, which selects items in order of decreasing value-to-weight ratios until the knapsack is full. There are more elaborated and efficient such heuristics, like the Lin-Kernighan-Helsgaun (LKH) algorithm for the TSP (Lin and Kernighan, 1973), which uses the structure of TSP tours to iterate from a given proposed solution by swapping edges.

A wide range of meta-heuristics has been proposed, giving rise to a rich taxonomy (Gendreau, Potvin, et al., 2010). Without getting overly specific, we can classify them into two categories: *population-based* and *individual-based* meta-heuristics. Population-based meta-heuristics are based on the idea of maintaining a population of candidate solutions, and iteratively evolving the population to improve the quality of the solutions. Examples of population-based meta-heuristics include genetic algorithms (M. Mitchell, 1998), particle swarm optimization (Kennedy and Eberhart, 1995), and ant colony optimization (Dorigo, Birattari, and Stutzle, 2006). Individual-based meta-heuristics, on the opposite, maintains a single candidate solution, which it iteratively improves, for example by applying local search operators. Examples of individual-based meta-heuristics include simulated annealing (Pincus, 1970), tabu search (Glover, 1986a), and iterated local search.

## 2.3 Reinforcement learning for combinatorial optimization

### 2.3.1 Why and when to use reinforcement learning for combinatorial optimization

Given the vast literature on exact methods, heuristics, and meta-heuristics for combinatorial optimization, it is natural to ask whether we really need to resort to yet another approach. We have also evoked in the Chapter 1 the weaknesses of reinforcement learning in this setting. In this section, we adopt a contrasting viewpoint and explore the contexts where reinforcement learning can be regarded as a promising approach.

In their tour d'horizon, Bengio, Lodi, and Prouvost (2018) gives some arguments on using machine learning for combinatorial optimization.

- It can take into account the specific (unknown) distribution of encountered problem instances. The resulting heuristic can leverage a possible structure of instances in this subdistribution, which would be difficult to achieve without a learning mechanism.
- A Machine Learning (ML) system can learn fast approximations of decisions that would require expensive computations.

These naturally apply to reinforcement learning. Additionally, we can add the following arguments, this time specific to RL.

1. It is a general framework that can be applied to any problem that can be formulated as an MDP with minimal changes.
2. It does not require pre-solved instances to learn from, which would be the case for supervised learning.
3. It can natively deal with uncertainty on the problem data.
4. The problem itself can be stochastic, not fully observable, or not wholly known from the start (*e.g.* a VRP where a new customer can appear anywhere on the map according to some unknown distribution, which is a more realistic setting than the classical VRP where all customers are known in advance).
5. It can be used as a subroutine to improve the performance of an existing algorithm.
6. Given a new, complex problem, it is likely that no good ILP formulation or reduction is known or easily findable. In this case, RL is an easy way to develop a heuristic.
7. RL can be compute-intensive, but most of the cost is paid during training. Once the agent is trained, it can be used to solve new instances of the problem quickly. RL thus involves a large fixed cost, but it can be amortized over the number of instances solved at inference.

8. RL can be used as a learning device: because the resulting policy is not biased toward handcrafted solutions, it can be used to gain new insights on the problem, which in turn can be used to develop better heuristics.

Item 5 can be seamlessly integrated with either an exact or heuristic solver. For instance, researchers such as Etheve et al. (2020) and Parsonson, Laterre, and Barrett (2022) have leveraged reinforcement learning to enhance branching decisions in the context of ILP, while Zheng et al. (2021) have combined a RL agent with LKH to boost its efficiency.

### 2.3.2 Framing a combinatorial optimization problem as a MDP

One natural way to apply reinforcement learning to a combinatorial optimization problem is by framing it as an MDP. This can be achieved using two different approaches. First, *improvement methods* start from a feasible solution and iteratively improve it by making small modifications (actions). Second, *construction methods* build a solution incrementally by selecting one element at a time. For example, when dealing with the TSP, improvement methods would begin with a valid tour and apply local operators, such as swapping the order of two cities in the tour. On the other hand, construction methods would start from an empty tour and add cities one by one until the tour is complete.

Both improvement and construction methods have their own advantages and drawbacks. Improvement methods are generally problem-specific, and require a good understanding of the problem to design sensible improvement operators. But they excel at quickly exploring the solution space. On the other hand, construction methods are more general, and require only the framing of the problem as a MDP. However, these methods struggle to explore the solution space effectively since all solutions are derived from the same underlying policy.

Several improvement methods have been proposed to tackle combinatorial problems. As a quick overview, O. da Costa et al. (2020) focuses on TSP and uses a policy gradient algorithm to learn a policy selecting local operations (2-opt, which corresponds to swapping two edges) given a current solution while Lu, X. Zhang, and S. Yang (2020) and Y. Wu et al. (2021) extends this idea to Capacitated Vehicle Routing Problem (CVRP). X. Chen and Tian (2019), (Hottung and Tierney, 2020) and (Lu, X. Zhang, and S. Yang, 2020) solve several combinatorial problems, including CVRP. X. Chen and Tian (2019) learns one policy to pick a specific part of a solution and one policy to select a modification operator to be applied to this region. (Hottung and Tierney, 2020) propose a large neighborhood search by learning a policy to repair a damaged solution. Hottung, Bhandari, and Tierney (2021) uses a variational auto-encoder to learn a continuous latent space to represent solutions, which can then be searched at test-time using a continuous optimization method.

In the same way, numerous construction methods have been developed.

Bello et al. (2016) uses a pointer network (Vinyals, Fortunato, and Jaitly, 2015) trained using an actor-critic framework. Graph neural networks (Dai et al., 2017), as well as attention networks (Deudon et al., 2018), have also been used to solve TSP. A different line approach has consisted in learning heatmaps of edges likely to belong to the optimal solution, and using this prior together with a search method, like beam search (Joshi, Laurent, and Bresson, 2019), Monte Carlo tree search Fu, Qiu, and Zha, 2021, or dynamic programming Kool, Hoof, Gromicho, et al., 2021.

Solving CVRP has been a more recent line of work with Nazari et al. (2018), which combines an attention mechanism with a recurrent network trained using an actor-critic algorithm. Kool, Hoof, and Welling (2019) proposes an attention-based encoder-decoder architecture. The costly encoder is run once per problem instance, and the resulting embeddings are fed to a small decoder to iteratively roll out the whole trajectory. This work was extended in Kwon et al. (2020), which augments instances (using a different starting position, rotations...) both during training and inference. To better the solution quality, Hottung, Kwon, and Tierney, 2022 runs an active search for each problem instance while only updating a small portion of the networks, speeding up the computations.

Construction methods have also been used beyond routing problems. Kwon et al. (2020) experiments on KP, and Hottung, Kwon, and Tierney (2022) on job scheduling problems. A lot of hard combinatorial problems arise on graphs, like minimum vertex cover (Khalil et al., 2017), job scheduling (C. Zhang et al., 2020; Grinsztajn, Beaumont, et al., 2021a), max cut (Barrett, Clements, et al., 2020; Barrett, Parsonson, and Laterre, 2022), maximal independent set (Z. Li, Q. Chen, and Koltun, 2018)...

We provide a more detailed overview of RL approaches for specific combinatorial optimization problems in the related chapters: Chapter 3 and Chapter 4 for scheduling problems, and Chapter 7 for routing problems. For a more exhaustive overview of RL approaches for various COP, we refer the reader to Powell (2011) and Mazyavkina et al. (2021).

# Part I

# Case Study:
# Dynamic DAG Scheduling

# Chapter 3

# Reinforcement learning for dynamic DAG scheduling

In this chapter, we propose a reinforcement learning approach to solve a realistic scheduling problem and apply it to an algorithm commonly executed in the high-performance computing community, the Cholesky factorization. On the contrary to static scheduling, where tasks are assigned to processors in a predetermined ordering before the beginning of the parallel execution, our method is dynamic: task allocations and their execution ordering are decided at runtime, based on the system state and unexpected events, which allows much more flexibility. To do so, our algorithm uses graph neural networks in combination with the Advantage Actor-Critic (A2C) to build an adaptive representation of the problem on the fly. We show that this approach is competitive with state-of-the-art heuristics used in high-performance computing runtime systems. Moreover, our algorithm does not require an explicit model of the environment, but we demonstrate that extra knowledge can easily be incorporated and improves the performance. We also exhibit key properties provided by this RL approach, and study its transfer abilities to other instances. [1]

## Contents



[1] This chapter is based on the paper (Grinsztajn, Beaumont, et al., 2020) published in the proceedings of the *2020 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning* (*ADPRL*).

## 3.1 Introduction

As outlined in the previous chapter, COPs constitute an important family of fundamental problems: routing problems (*i.e.* TSP, VRP), SAT, graph coloring, task scheduling, and many others. There are various algorithmic approaches, ranging from (provably) exact methods (*e.g.* based on tree search, linear programming, *etc.*) to non (provably) exact/approximate methods (heuristics and meta-heuristics), as discussed in Chapter 2. Those methods are able to solve large-scale COPs, but they require a careful investigation of the problem. On the other hand, real-world applications bring another set of challenges: inherent uncertainty in the definition of the problem and randomness in the process dynamics. For instance, considering a task scheduling problem, task durations and communications delays between tasks are uncertain, and even the very set of tasks to be scheduled may not be known in advance like in an operating system. To effectively address real-world applications, it is crucial to consider COPs that account for uncertainties in the system dynamics.

RL is designed to deal with sequential decision making under uncertainty (Sutton and Barto, 2018). RL algorithms are able to adapt to their environment, adjusting their behavior in response to changes. This property opens the door to tackle COPs which contain uncertainty, or COPs which are not completely defined when their resolution is initiated. Another potential benefit is the ability to generalize to unseen settings, a necessary step toward real-world applications.

In this chapter, we will investigate the potential of RL for a real application, the dynamic scheduling of a set of tasks on a distributed computing system. Modern computer systems contain a variety of resources, interconnected to support parallel and distributed computationally intensive applications. Efficiently executing parallel applications on such systems is critical in many scientific domains. In a High-Performance Computing (HPC) environment, it is very common for an application to be split into several sub-tasks which may be executed in parallel. There usually are some dependencies between those tasks as the results provided by some may be necessary to start others. This structure may naturally be modeled using a Directed Acyclic Graph (DAG): nodes of the DAG are sub-tasks, and directed edges represent dependencies. Task-based runtime systems (Cédric Augonnet et al., 2011; George Bosilca et al., 2013) internally represent the application as a DAG to execute it on a parallel machine. In this case, one of the main duties of such runtime systems is to schedule the different tasks of the DAG onto the available computing resources. The DAG scheduling problem consists in finding the best way of assigning tasks to processing units, so that the task dependencies are respected and the total duration of execution (the makespan) is minimal.

Scheduling a DAG on a set of resources is a combinatorial problem, known to be NP-hard (Ullman, 1975). It is a sequential, stochastic and dynamic problem. Stochasticity intervenes at two levels: on the one hand, the exact computation times of tasks and transfer times of data are unknown, although we can have good prior estimates. On the other hand, the whole DAG is not necessarily known from the start. This requires that the scheduling algorithm implemented in the runtime system is *dynamic*. Moreover, to be generic, the proposed solution must be able to schedule unknown graphs onto any number of computing resources.

In this chapter, we suppose that different types of tasks can have different durations, but we assume that communications can be neglected (either because the target platform is a shared-memory system or because communications and computations can be overlapped). This allows to simplify the problem formulation while keeping its NP-hardness. Our contributions are as follows:

1. We formalize the dynamic DAG scheduling problem as a MDP.
2. We introduce a practical deep RL algorithm able to build a graph representation sequentially and on the fly.
3. In a set of experiments, we demonstrate that our RL approach obtains schedules competitive with state-of-the-art heuristics, even when no explicit knowledge of the environment is available at the cost of more computation time.
4. We further show that our RL approach can generalize, by scheduling yet unseen instances.

## 3.2 Additional related works

Among COPs, task scheduling has attracted a lot of research and presents a rich taxonomy (Leung, 2004). Roughly, it consists in assigning a set of tasks (whose interdependency is represented by a Directed Acyclic Graph – DAG–) to a set of resources while managing different constraints (a resource cannot execute several tasks at the same time, a task cannot start before its predecessor(s) dependencies have been completed, *etc.*). In the literature, several classes of scheduling problems have been studied (Graham et al., 1979). In static problems, the DAG is assumed to be completely known in advance while in dynamic problems, part of the DAG is unveiled as the scheduling algorithm progresses. In the homogeneous setting, all resources are identical while in the heterogeneous case, resources are different and the same task can have different durations depending on the resource it is allocated on. Depending on the input (graph topology, number or type of resources) the problem can be either polynomial or NP-Hard. But even simple cases (*e.g.* tasks with no dependencies and two resources) turn out to be NP-Hard (Garey and D. S. Johnson, 1979).

Few works focus on reinforcement learning for task scheduling in computational graphs. Most of them (Mao, Alizadeh, et al., 2016a; Kumar, Bhambri, and Shambharkar, 2019) consider tasks arriving sequentially and randomly, without a DAG structure. The two papers closest to ours are Mirhoseini, Pham, et al. (2017) and Q. Wu et al. (2018). The first one uses a very realistic environment (communication time, storage capacity of the nodes...), but relies on a basic tabular Q-learning technique, which cannot scale to real-life applications, and does not allow generalization. The second one uses Deep Reinforcement Learning (DRL), but does not allow online scheduling. Moreover, these two papers preprocess the DAG in ways which do not allow the agent to use its whole structure.

Recently, Paliwal et al. (2020) used reinforcement learning to guide a genetic algorithm. However, the scheduling is static, and the environment necessarily deterministic, which are two important limitations with regard to practical constraints.

In Mao, Alizadeh, et al. (2016b), the authors present a reinforcement learning approach to map jobs onto a parallel machine. In contrast to our problem, there are no dependencies between jobs and the goal is to minimize job slowdown.

In Mirhoseini, Pham, et al. (2017), the authors study the task mapping of Deep Graph Neural Network. Their approach is based on the policy gradient algorithm. Yuanxiang Gao, L. Chen, and B. Li (2018) tackles the same problem as Mirhoseini, Pham, et al. (2017) by modeling it as a Markov decision process and using a reinforcement learning approach called proximal policy optimization (Schulman, Levine, et al., 2015b). However, both approaches are not suited for transfer learning as the proposed solutions can only improve the mapping of the input problem.

In Addanki et al. (2019), the authors provide a general approach for mapping task graph using a graph embedding approach called Placeto. In Zhou et al. (2019), the authors introduce GDP, which uses the same approach as Placeto but outperforms it in their experiments. These two approaches allow transfer learning for new graphs but not for new machines (the target platform must be the same as the training one). They also provide the mapping of all tasks at the same time, which is not suitable in case the duration of the tasks are imperfectly known.

Overall, there is yet no generic RL approach that deals with dynamic task graphs and enable generic transfer for both new graphs and new machines.

## 3.3 Task modeling

In this chapter, we focus on the Cholesky factorization problem. The Cholesky factorization is a very common linear algebra routine along with QR and Lower Upper decomposition (LU) (Choi et al., 1996; Buttari et al., 2009). The tiled version is used in several task-based

runtime systems such as StarPU (Cédric Augonnet et al., 2011)or PARSEC (George Bosilca et al., 2013). These runtime systems are in charge of scheduling the tasks onto homogeneous or heterogeneous platforms based on the description of the application by means of a DAG (as depicted in Fig. 3.1). Hence, being able to efficiently schedule the CHOLESKY DAG is of utmost importance. It is indeed characteristic of many applications in linear algebra and scientific computing, in the sense that CHOLESKY factorization involves (i) a large number of tasks, (ii) complex but regular dependencies and (iii) a small number of different kernels [2]. It is therefore a very good benchmark for scheduling algorithms (Agullo, Beaumont, et al., 2016; Agullo, Cédric Augonnet, Dongarra, Ltaief, et al., 2010; Jeannot, 2013) and designing a scheduling algorithm for the dense tiled CHOLESKY factorization is paramount, both practically and theoretically.

### 3.3.1 Tiled CHOLESKY factorization

In this chapter, we focus on the task graph induced by the tiled dense CHOLESKY factorization depicted in Algorithm 3.1

**Algorithm 3.1:** Tiled version of the CHOLESKY factorization.

```
1 for k = 0...T − 1 do
2     A[k][k] ← POTRF(A[k][k])
3     for m = k + 1...T − 1 do
4         A[m][k] ← TRSM(A[k][k], A[m][k])
5     for n = k + 1...T − 1 do
6         A[n][n] ← SYRK(A[n][k], A[n][n])
7         for m = n + 1...T − 1 do
8             A[m][n] ← GEMM(A[m][k], A[n][k], A[m][n])
```

For a given symmetric positive definite matrix $A$, the CHOLESKY algorithm computes a lower triangular matrix $L$ such that $A = LL^T$. In the tiled version, the matrix is decomposed into $T \times T$ square tiles. Each tile is hence a sub-matrix of the original matrix. We denote $A[i][j]$ the tile corresponding to row $i$ and column $j$: the reader should be careful that this $A[i][j]$ is a $N \times N$ sub-matrix of the original matrix, made of the elements of rows and columns (*i.e.* spanning rows from $N \times i$ to $N \times (i+1) - 1$). $N$ is usually several hundreds (typically $380 \times 380$ for standard CPUs and $960 \times 960$ for GPUs). At each step $k$, the algorithm performs a CHOLESKY factorization of the tile $A[k][k]$ located on the diagonal (with the `POTRF` kernel). Then it updates all the tiles below it ($A[k][k+1:T-1]$) using a triangular solve (`TRSM` kernel). The trailing sub-matrix is updated using the `SYRK` kernel for tiles on the diagonal and matrix multiply (`GEMM` kernel) for the remaining tiles (of the lower triangular part).

[2] in this chapter and the following, a "kernel" is a basic operation performed on a sub-matrix. The CHOLESKY factorization algorithm is expressed as a combination of 4 different kernels.

Each kernel POTRF, TRSM, SYRK and GEMM is therefore executed several times during the Cholesky factorization by different *task instances*. Each task instance requires input data produced by other tasks. This leads to a DAG in which nodes are task instances and directed edges are (data) dependencies between two task instances. For example, for $k = 0$ the POTRF kernel updates tile $A[0][0]$. This tile is then used by all the TRSM kernels to update tiles $A[1 : T-1][k]$. Hence, in the graph, there are $T-1$ edges between the task that instantiate the POTRF kernel and $T-1$ tasks that each instantiate a TRSM kernel. Fig. 3.1 illustrates graphically the Cholesky DAG for $T = 5$ (*i.e.* 5 by 5 tiles).

The advantage of the tiled version of the Cholesky factorization is three-fold. First, by working on tiles, the computation of the kernels is very fast and optimized using BLAS (Basic Linear Algebra Subroutine) kernels. Second, tiles enable to deal with large size problems with relatively small DAGs. Third, the tiled version expresses a lot of parallelism which facilitates its execution on modern parallel systems.

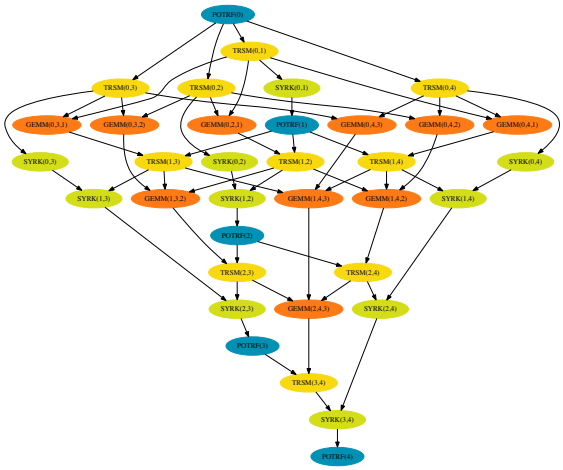



**Figure 3.1** – DAG of the Cholesky Factorization for $T = 5$. The indices of POTRF$(k)$, TRSM$(k, m)$, SYRK$(k, n)$ and GEMM$(k, n, m)$ correspond to the loop indices of Algorithm 3.1.

### 3.3.2 Problem definition

Our scheduling problem can be formalized as follows. For a given value of $T$, the Cholesky factorization is represented by a DAG, $G = (V, E)$, where the set of vertices $V$ corresponds to the set of $n$ tasks in the application, and $E$ is the set of directed edges between tasks, expressing dependencies between the result of the execution of a task, and its use by a subsequent task. One vertex corresponds to the beginning of the factorization process, while another vertex corresponds to its completion. Each task can be either one among the four types (of kernels) that determine its duration. The duration of a task can be deterministic or stochastic. For simplicity, we will consider them deterministic, although our optimization method works

the same way for stochastic durations. We denote by $d(v)$ the duration of the task $v$, and by $W = \sum_{v \in V} d(v)$ the total work needed to complete the instance. In what follows, we assume that the communication cost due to dependencies is zero. This is justified by the two following use cases. First, if the target machine is a multi-core system with shared memory, each core can execute one kernel at a time and the whole matrix is stored in a memory shared by all the cores. Hence, communication between tasks is done through memory access and such accesses are negligible compared to kernel computation time. Second, we can notice that for a tile of size $N$, the amount of data to transfer is of the order of $N^2$ while the complexity of the different kernels is of the order of $N^3$, so that one generally chooses a tile size $N$ large enough to overlap computations and communications, while allowing an efficient use of computation resources (cache,...).

The *makespan* is defined as the completion time of the last task to be executed. Two notions are of importance in what follows. The Critical Path (CP) of a DAG is the longest distance between the start node and the end node, including all the tasks and their duration. The total work ratio is equal to $\frac{W}{p}$ (where $p$ is the number of computing resources of the parallel machine). CP and $\frac{W}{p}$ are both lower bounds of the optimal makespan, which we want to minimize.

### 3.3.3 Reinforcement learning formulation

We model the problem as a MDP $(\mathcal{S}, \mathcal{A}, P, R)$. We now define $\mathcal{S}$, $\mathcal{A}$, $R$, and the objective function used to model the DAG scheduling problem.

#### State Space

The goal is to give the agent as much information about the task DAG as necessary. Computing the optimal solution requires the whole graph, and therefore the "state" should embed the whole graph. However, the whole DAG being potentially arbitrarily large and too cumbersome to be handled in practice, we consider an approximate representation. Hence, we restrict the information represented in a state to information about running tasks and available tasks, along with their descendants: "running" tasks are those currently executed, "available" those that may be executed but can not because of a lack of computing resources, "descendants" are all the tasks that have to wait for running and available tasks to be completed to be run, because they depend on their results. The depth of descendants being considered is left as a parameter in our algorithm; it is denoted by the window $w$. This choice of $w$ corresponds to a trade-off between computational time and accuracy. It is illustrated in Fig. 3.2.

Each node is represented by a set of raw features: these features are expected to encode and summarize the DAG information at the node level. The representation $X_i$ of node $i$ can be

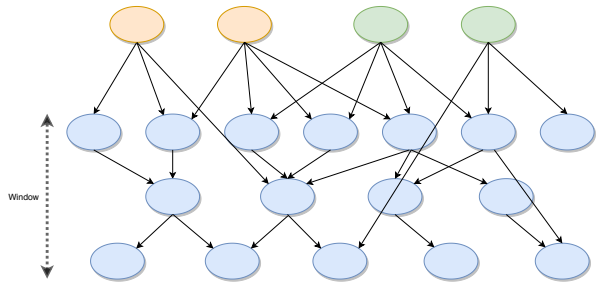



**Figure 3.2** – A state contains information about running tasks, available tasks and their descendants. This plot illustrates these notions: nodes are tasks, edges are dependencies between tasks: and a task/node can not start its execution before all its ancestors have run to completion. In orange: running tasks; in green: available tasks; in blue: descendants of running or available tasks. In this example, the window $w$ is set to 3.

written:

$$X_i = [succ_i, pred_i, type_i, avail_i, run_i, cp_i]$$

where $succ_i$ is the number of successor nodes of $i$, $pred_i$ is the number of predecessor nodes of $i$, $type_i$ is the type of the task encoded as a one-hot vector, $avail_i$ is a binary variable indicating if the task is available, $run_i$ is a binary variable indicating if the task is currently running, and $cp_i$ contains the portion of critical path ahead of the task.

To produce a good schedule for a DAG of tasks, one has to consider not only the current task to schedule, but also the dependencies of the tasks to schedule in the (near) future. Therefore, the state has to combine information about a set of nodes, taking into account the dependencies between them.

For this, we use graph convolution networks (Graph Convolutional Network (GCN)) (Kipf and Welling, 2016) which have been shown to perform well on several graph-related task benchmarks.

Given a graph $G$, a GCN updates the embedding $H^{(l)}$ of each node using local information, according to the formula:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right).$$

Here, $\sigma$ is an activation function, $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected graph with added self-connections, $I_N$ is the identity matrix, $\tilde{D}_{i,i} = \sum_j \tilde{A}_{i,j}$, and $W^{(l)}$ is a layer-specific trainable weight matrix.

The raw features are used as the starting vector for each node, such that ${H^{(0)}}^\top = [\hat{X}_1, \hat{X}_2, \ldots, \hat{X}_n]$. Stacking such GCN layers gives rise to a richer representation of the DAG by combining the properties of neighboring vertices (Kipf and Welling, 2016).

#### Action Space

An action consists in selecting an available task or in doing nothing (pass). If a task $t$ is selected, it is immediately scheduled on a ready-for-use processor. If all processors are ready, it is not possible to pass.

#### Reward and objective function

The reward is given when the final state is reached, or equivalently when the whole input DAG has been computed. Indeed, there is no relevant information available before the whole DAG has been scheduled that could be used as an immediate reward. The reward uses the final makespan given by the whole scheduling trajectory, normalized by a baseline duration. We use a heuristic (the As Soon As Possible (ASAP) algorithm, detailed in Section 3.5.1) to get a baseline duration in this chapter.

Thus, the reward can be written as:

$$R(makespan) = \frac{makespan_{ASAP} - makespan}{makespan_{ASAP}}$$

The lower the makespan the better; therefore the reward becomes positive as soon as the learned policy becomes more efficient than ASAP.

## 3.4 Algorithm

### 3.4.1 Actor-critic

We train an agent to schedule tasks using A2C (Volodymyr Mnih, Badia, et al., 2016). A2C is an actor-critic algorithm that aims at maximizing the objective function directly by gradient ascent:

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left( \sum_t \gamma^t r(s_t, a_t) \right)$$

where $\tau$ is a trajectory sampled according to the policy $\pi$ parametrized by $\theta$, and $\gamma$ a discount factor.

To compute a single update, we first run the current policy up to $t_{max}$ steps or until a terminal state is reached.

As discussed in the previous chapter, A2C uses a policy network (the actor) $\pi_\theta(a_t \mid s_t)$, parameterized by $\theta$, which computes a distribution of probability over the actions, and a value

network (the critic) which estimates the value of a state $V_{\theta_v}(s_t \mid \theta_v)$, and is used to lower the variance in the computation of the advantage function.

The policy update takes the form $\nabla_\theta \log \pi_\theta(a_t \mid s_t) A(s_t, a_t, \theta, \theta_v)$, where the advantage function $A(s_t, a_t, \theta, \theta_v)$ can be written $\sum_{i=0}^{k-1} \gamma^i r(s_{t+i}, a_{t+i}) + V_{\theta_v}(s_{t+k}) - V_{\theta_v}(s_t)$. $k$ is either the time when a terminal state is reached, or $t_{max}$. In the first case, $V_{\theta_v}(s_{t+k}) = 0$. The critic is simply updated in order to minimize the mean square error between the predicted value $V_{\theta_v}(s_t)$ and the real return $\sum_{i=0}^{k-1} \gamma^i r(s_{t+i}, a_{t+i}) + V_{\theta_v}(s_{t+k})$.

Adding the entropy of the policy in the objective function has been shown to improve exploration (Williams and Peng, 1991). We therefore add the term $\beta \nabla_\theta \mathcal{H}(\pi_\theta(s_t))$ to the actor gradient, where $\mathcal{H}$ is the entropy and $\beta$ a hyper-parameter which controls the influence of the entropy regularization.

In practice, many parameters of the actor and the critic are shared, as we will detail in the next section.

### 3.4.2 Architecture

The network architecture is kept as simple as possible in order to minimize the scheduling computation overhead (see Fig. 3.3).

A sub-DAG is fed to the neural network (bottom of Fig. 3.3) and goes through a series of graph convolution layers. The number of layers is a parameter of the algorithm; it should be related to the size of the window $w$: at least $w$ layers are required to let the necessary information flow from the nodes of the sub-DAG to nodes representing ready tasks. Empirically, we found that using exactly $w$ iterations is enough. Between these layers, we use Rectified Linear Unit (ReLU) functions as non-linear activations.

As already mentioned above, these stacked convolution layers produce an internal representation of the sub-DAG input. This representation is used to produce an estimation of the value of the current state, and determine which action to take. A set of 3 1-layer Fully-Connected layer (FC) produces these two results.

### 3.4.3 Scheduling

The scheduling process is done iteratively by placing the available task chosen by the agent on an available device. Once every device has been assigned a task or the agent has decided to pass, there is no environment-agent interaction until the next event, the moment when one or more tasks are completed, and the corresponding computing units become available. Then the agent can choose a new action, and the process goes on until the whole DAG has been computed.

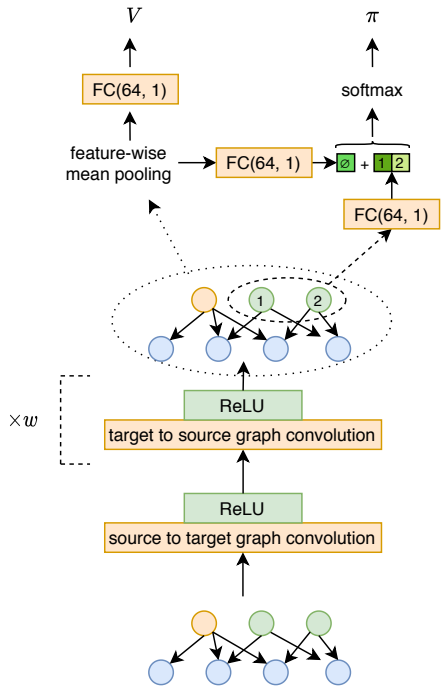



**Figure 3.3** – Overview of the agent architecture. At the bottom, a sub-DAG is input into a stack of $1 + w$ graph convolution layers. An internal representation of the sub-DAG is outputted in which node information has been mixed. For this current state, an estimate of the value $V$ and an action to perform (either do nothing ($\emptyset$) or begin the execution of one of the available tasks - in green, either 1 or 2) are generated. FC$(64, 1)$ is a fully connected layer with an input size of 64 and an output size of 1.

## 3.5 Experiments

### 3.5.1 Reference algorithm

In this chapter, our goal is to analyze the performance of a reinforcement learning-based algorithm for the dynamic scheduling of Cholesky factorization tasks. The dynamic nature of factorization is in practice imposed by the difficulty of accurately predicting computational costs and communication durations in a HPC environment in which the various operations unpredictably influence execution times. In practice, dynamic runtimes rely solely on the description of the machine state and on the tasks already performed, using a task priority system to define which tasks to perform in the event that the number of available resources is less than the number of available tasks. In these dynamic systems, task placement decisions are made a little in advance, taking into account the placement of input data, and this delay is used to transfer task input data if necessary to overlap communication and computation. We have already discussed in Section 3.3.2 how to overlap communications and computations in both the CPU multicore and the GPU cases.

To perform dynamic scheduling, ASAP is a strategy of choice which is the *de facto* standard in most dynamic schedulers. ASAP never leaves a resource inactive if there is an executable task and ASAP chooses among several candidate tasks the one that is the farthest from the end of the computation (the one with the longest critical path). It has been demonstrated in Beaumont, Langou, et al. (2020) that despite its simplicity, this strategy gives excellent results for Cholesky factorization, especially in the case where execution times are similar to what is observed in practice on GPUs. This is therefore the strategy that we use as a baseline to evaluate the performance of the reinforcement learning-based algorithm that we propose: the reader should keep in mind that it is difficult to beat ASAP, or even to perform as well as ASAP. However, ASAP requires the whole DAG: ASAP cannot cope with a dynamic environment in which the DAG is unknown. Our approach does not suffer from this limitation.

We add two other baselines, Random and Greedy. We call Greedy the baseline which prioritizes the tasks which have the largest number of successors. Random consists simply in choosing the task to schedule uniformly among the available tasks. Both baselines are very simple, hence computed very quickly.

For reproducibility purposes, the code used to perform the experiments is available at https://github.com/nathangrinsztajn/DAG-scheduling.

### 3.5.2 Simulated model

In order to be able to iterate rapidly over runs, we do not evaluate the agent performance on a real device but on a simulated environment. We use a different mean duration for each type of task, as shown in table 3.1, according to the data gathered in Beaumont, Langou, et al. (2020) for GPU computations.

| POTRF | SYRK | TRSM | GEMM |
|---|---|---|---|
| 11 | 2 | 8 | 3 |

**Table 3.1** – Task durations used in the DAG model.

### 3.5.3 Results

We perform different types of simulations:

1. performance comparison of our RL approach with several baselines, using Cholesky factorization with different numbers of tiles.

2. a closer look at the performances if we remove the critical path of the node embedding and vary the window parameter $w$.

**Table 3.2** – DAG characteristics for several number of tiles $T$ in the CHOLESKY factorization.

| $T$ | $\|V\|$ | $W$ | Critical Path |
|---|---|---|---|
| 4 | 21 | 116 | 74 |
| 8 | 121 | 536 | 158 |
| 16 | 817 | 3056 | 326 |

3. transfer learning: we perform different kinds of transfers:
   (a) having learned to schedule the DAG of tasks of a CHOLESKY factorization for a given number of tiles $T$, how does this transfer to other numbers of tiles?
   (b) having learned to schedule the DAG of tasks of a CHOLESKY factorization for a given number of computing devices $p$, how does this transfer to other numbers of computing devices?

As our approach is not deterministic, we train 10 agents for each configuration with 10 different seeds. The graph neural network policy was developed using PyTorch Geometric package (Fey and Lenssen, 2019) and trained on the 8 cores of a CPU (no GPU was used), a run taking approximately 1 hour to complete. In the tables, we provide the results of the best of the 10 agents.

Using the notations introduced in section 3.4.1, we set $\beta = 0.02$, $t_{max} = 40$ in all our experiments. Each agent is trained for 10,000 steps and only the best version encountered during the process is kept. The training was done using Adam optimizer, with a learning rate of $0.01$ and $\varepsilon = 0.1$. As some parameters of the actor and the critic are shared, we give the actor update more importance by down-weighting the critic learning rate by $1/2$.

Unless specified otherwise, we take $w = 1$ for training and testing.

We use 3 different DAG obtained for three different numbers of tiles $T$: 4, 8 and 16. These graphs have very distinct characteristics, as shown in Table 3.2. $|V|$ is the number of nodes in the DAG. Please refer to section 3.3.2 for the definition and meaning of $W$ and critical path.

#### RL *vs.* ASAP: performance comparison

Table 3.3 reports the performance of the baselines and our RL approach for different numbers of tiles $T$ and processors $p$. Performance is measured as the makespan of the scheduling computed by a given algorithm. We can see that our RL approach is very competitive with ASAP, and outperforms consistently the other baselines. Again, the reader should keep in mind that ASAP is a very good and hard to beat heuristic.

| $T$ | $p$ | Agent | ASAP | Greedy | Random |
|---|---|---|---|---|---|
| 4 | 4 | **74** | **74** | **74** | 74.8 (0.87) |
| 8 | 4 | 163 | **160** | 173 | 196.5 (5.57) |
| 16 | 4 | 792 | **787** | 814 | 832.9 (6.09) |
| 8 | 2 | **280** | 282 | 286 | 300.2 (5.39) |
| 8 | 6 | **158** | **158** | 174 | 174.2 (3.24) |

**Table 3.3** – Makespan comparison (lower is better). For stochastic baselines, the result is a mean over 10 trajectories, and the standard deviation is given in parentheses.

### A closer look at $w$ and our initial embedding

The length of the critical path ahead added in the initial node embeddings gives information about all the tasks remaining at each step, and supposes an accurate model of the DAG and of the sub-task durations. As this information is not always available, we investigate the performance of our algorithm without such help: the results are given in Table 3.4. We choose to compare those configurations with $T = 8, p = 4$ as it is one of the most difficult RL settings, according to Table 3.3.

We observe that when the critical path is included in the embedding, there is no benefit in enlarging $w$. In fact, $w = 0$ is already almost optimal. On the contrary, without this information, we note that enlarging $w$ greatly improves the performances and seems to have a stabilizing effect on the training. When $w = 4$, the makespan is almost as good as those of agents trained with the critical path. On the other hand, rising $w$ increases the computation time, as can be seen in Fig. 3.4. $w$ should therefore be tuned adequately to ensure a suitable time-performance trade-off for the current case of use. ASAP cannot be used when the critical path is not available.

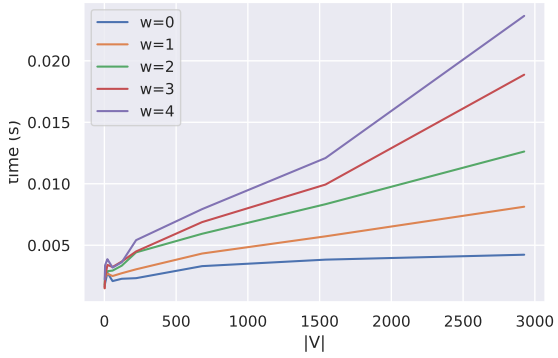



**Figure 3.4** – Mean computation time per action at inference, for several $w$ and several DAG sizes. The computation time per action increases with the number of nodes in the graph and with $w$, as in both cases there are more message-passings to compute during the updates of the node embeddings.

| CP | $w$ | makespan |
|---|---|---|
| $+$ | 0 | 163 (3.28) |
| $+$ | 1 | 163 (4.54) |
| $-$ | 0 | 173 (40.13) |
| $-$ | 1 | 170 (16.53) |
| $-$ | 2 | 171 (0.89) |
| $-$ | 3 | 166 (0.83) |
| $-$ | 4 | 164 (10.58) |

**Table 3.4** – Performance comparison of several settings, for $T = 8$ and $p = 4$. The CP column contains $+$ if we included the critical path in the node embeddings, and $-$ otherwise. We run each configuration 10 times and keep for each one the results of the best agent. The standard deviation of the makespans of the 5 best agents is written in parentheses.

#### Experiments on transfer learning

Table 3.5 reports makespans achieved with transfer learning: we train our agent on a given number of tiles $T$, and we measure the makespan achieved for two other values. When $T_{\text{train}}$ and $T_{\text{test}}$ are equal, the results are those reported in table 3.3. We can see that the transfer is very effective: the makespan obtained with this 0-shot transfer is only slightly worse than the one of a dedicated agent, and still not much worse than ASAP. We conduct the same experiments with the number of computation units $p$. We notice that our agent exhibits good transfer abilities across $p$, beating the Greedy baseline almost in every case. While training a RL agent takes time, this experiment shows that once trained, the RL agent has the ability to schedule different DAGs, something ASAP cannot do.

## Chapter conclusion

In this chapter, we have investigated the use of reinforcement learning as a principled approach to solve scheduling problems involving the need of being able to adapt to a dynamic (runtime) environment. Solving scheduling problems is known to be NP-hard, and remains a challenge for RL. In this chapter, we focused on dynamic scheduling on homogeneous resources without communication costs. We conducted our experiments on a well-known, heavily used numerical procedure, the Cholesky factorization. Experiments show that we can achieve schedules that are as efficient as those obtained by dedicated heuristics; they also show the benefit of using a RL approach to transfer the policy learned on a certain hardware configuration to another or to transfer the scheduling policy learned on a certain graph of tasks to another. To the best of our knowledge, the paper on which this chapter is based was the first to present such an adaptive

**Table 3.5** – Transfer learning experiments through $T$ (number of tiles) and $p$ (number of processing units). The RL agent learns on $T_{\text{train}}$ and $p_{\text{train}}$ and is tested at inference on $T_{\text{test}}$ and $p_{\text{test}}$.

| $\mathbf{T}_{\text{test}}$ | $\mathbf{T}_{\text{train}}$ | $\mathbf{p}_{\text{test}}$ | $\mathbf{p}_{\text{train}}$ | **makespan** |
|---|---|---|---|---|
| 4 | 4 | 4 | 4 | 74 |
| | 8 | | | 74 |
| | 16 | | | 74 |
| 8 | 4 | | | 215 |
| | 8 | | | 163 |
| | 16 | | | 175 |
| 16 | 4 | | | 911 |
| | 8 | | | 805 |
| | 16 | | | 792 |
| 8 | 8 | 2 | 2 | 280 |
| | | | 4 | 285 |
| | | | 6 | 296 |
| | | 4 | 2 | 172 |
| | | | 4 | 163 |
| | | | 6 | 178 |
| | | 6 | 2 | 158 |
| | | | 4 | 159 |
| | | | 6 | 158 |

RL approach featuring dynamic scheduling and transfer learning, and to study the effect of the node-level information available.

The presented approaches have several limitations. As we only considered homogeneous computing resources, examining heterogeneous devices such as CPUs and GPUs would be interesting. Considering other types of tasks, such as LU factorization, which is more complex than Cholesky because of the repeated selection of a pivot, is also a path to investigate. Yet another direction would be to study the impact of noise on task execution times. These three extensions are considered in the following chapter.

# Chapter 4

# RL for dynamic DAG scheduling: stochasticity and heterogeneity

In this chapter, we propose READYS, a reinforcement learning algorithm for the dynamic scheduling of computations modeled as a Directed Acyclic Graph (DAGs). As in the previous chapter, our goal is to develop a scheduling algorithm in which allocation and scheduling decisions are made at runtime, based on the state of the system. We go further than the previous chapter by considering a more realistic scheduling environment, in which the exact computation durations of tasks are not exactly known. We also consider heterogeneous platforms made of a few CPUs and GPUs, and focus on more graphs originating from linear algebra factorization kernels (Cholesky, LU, QR). We first propose to analyze the performance of READYS when learning is performed on a given (platform, kernel, problem size) combination. Using simulations, we show that the scheduling agent obtains performances very similar or even superior to algorithms from the literature, and that it is especially powerful when the scheduling environment contains a lot of uncertainty. We additionally demonstrate that our agent exhibits very promising generalization capabilities. To the best of our knowledge, this was the first work showing that reinforcement learning can be used for dynamic DAG scheduling on heterogeneous resources. [1]

## Contents



[1] This chapter is based on the paper (Grinsztajn, Beaumont, et al., 2021b) published in the proceedings of the *2021 IEEE International Conference on Cluster Computing (CLUSTER)*.

## 4.1 Introduction

In the previous chapter, we proposed an RL approach to deal with tiled Cholesky factorization in simple homogeneous environments; we showed that in this context an RL approach can be competitive with the ASAP heuristic, and that it is possible to transfer the knowledge acquired while solving a problem of size $T$ to solving another problem of size $T'$, hence saving the learning time. In this chapter, we go further in our investigation: we apply this RL approach to a set of factorization algorithms, namely Cholesky, LU, and QR, which are very common in numerical linear algebra routines, and we extend our results to heterogeneous platforms consisting of both CPUs and GPUs. In this context, we consider a scheduling problem in which resource performances are unrelated (Lenstra, Shmoys, and Tardos, 1990), *i.e.* the ratio between the processing time of a task on a GPU and on a CPU depends on the type of the task itself.

In this chapter, we also analyze the effect of noise on task durations. Indeed, while we may have good estimates of the computation durations of tasks and communication times, the exact durations are typically unknown and can vary due to various factors, which can make the scheduling problem challenging. When using static task allocation and scheduling, even with relatively well-known task and communication durations, a drift is observed when the problem size becomes large, which is a source of load imbalance and idle time. This observation is at the heart of the success of dynamic schedulers such as StarPU and ParSEC (Cédric Augonnet et al., 2011; George Bosilca et al., 2013), which rely on less sophisticated scheduling algorithms because decisions are made at runtime and must be very fast, but benefit from a more accurate knowledge of the state of computation and communication resources at the time of making a decision. In practice, dynamic schedulers make task allocation decisions a little in advance of the actual processing of the tasks, which makes it possible to perform the necessary communications as soon as possible, thus ensuring a good overlap of communications with computations.

As in the previous chapter, we assume that communications can be overlapped with computations and can therefore be neglected, which is a reasonable assumption at the scale of the computing node consisting of a few CPUs and GPUs (see Section 4.3). Linear algebra kernels offer a rich context in which the combined use of CPUs and GPUs is relevant. Indeed, the different linear algebra kernels involved in a factorization exhibit very different acceleration rates on the GPUs.

As before, we represent only a part of the DAG of tasks to be scheduled: this part is a window $w$ sliding over the DAG, so that at any time, the algorithm considers only a relevant part of the DAG, consisting in the tasks that are ready for processing (whose all predecessors have been processed) and their descendants at distance $w$, in order to efficiently schedule the next tasks. Moreover, it is reasonable to consider that in practice, the whole DAG is not known in advance. Indeed, numerical tests might typically modify dynamically at runtime the shape of the DAG. Furthermore, scheduling and allocation decisions must be fast compared to the

typical duration of tasks, so it is necessary to restrict the size of the state on which decisions are based.

Our contributions are as follows. We model dynamic scheduling on heterogeneous platforms as a reinforcement learning problem in Section 4.3 and design READYS, a suitable RL agent in Section 4.4. Then, we perform an experimental study considering standard linear algebra kernels in Section 4.5. We compare the performance of READYS with those of classical heuristics in the heterogeneous case, namely the static Heterogeneous Earliest-Finish Time (HEFT) (Topcuoglu, Hariri, and M.-y. Wu, 2002) and the dynamic Minimum Completion Time (MCT) (Sakellariou and Zhao, 2004) heuristics. We show that READYS is competitive with respect to HEFT even when the prediction of task lengths is accurate. When task durations are not exactly known in advance, we show that READYS performs better.

It is worth noting that HEFT relies on complete and accurate knowledge of the DAG, unlike READYS which grounds its decisions on much less information. In Section 4.5.6, we investigate the possibility to transfer the knowledge acquired while solving the problem on a given (matrix) size to another size.

Overall, the paper on which this chapter is based was the first to demonstrate that reinforcement learning is an interesting approach to dynamically schedule tasks when the characteristics of the tasks and computing resources are not perfectly known or are changing over time. We emphasize that this lack of exact knowledge, though often neglected, is actually the situation faced on real HPC systems.

## 4.2 Additional related work

In Mao, Schwarzkopf, et al. (2019), a dynamic scheduling strategy is learned on top of Spark considering online DAG job arrivals. Contrary to our approach, scheduling decisions are made at *stage level* (a stage being a set of independent tasks operating on different input data), leaving fine-grained task-level decisions to Spark. This eliminates the burden of dealing with large DAG, but restricts the approach to jobs with high inherent parallelism.

Orhean, Pop, and Raicu (2018) tackles the problem of heterogeneous distributed systems. But contrarily to us, it relies on a simple q-learning reinforcement algorithm with a hard-coded pre-processing of the DAG into a look-up table, which prevents scaling to complex environments or generalizing to unseen instances.

In the scheduling literature, a lot of work has been done to efficiently perform linear algebra factorizations in parallel on heterogeneous platforms, because of the practical importance of these kernels. We focus here on works using dynamic runtime scheduling strategies. For instance, the Cholesky factorization has been implemented in DAGuE (G. Bosilca et al., 2012),

StarPU (C. Augonnet et al., 2011; Cojean et al., 2019), OmpSs (Duran et al., 2011), and SuperMatrix (Quintana-Ortí et al., 2009). The basic task scheduling strategy consists in (i) analyzing the list of ready tasks (*i.e.* all tasks whose predecessors have already been processed), (ii) taking the most important task (using a priority system typically based on heuristics such as HEFT (Topcuoglu, Hariri, and M.-y. Wu, 2002)), and (iii) placing it on the resource that is likely to complete it as early as possible using estimations as in MCT heuristic (Sakellariou and Zhao, 2004), given the task cost models on the different resources (and the input data transfer times). HEFT and its variants are the de facto heuristics for static scheduling, while MCT is popular for dynamic scheduling: we will use them as reference algorithms to evaluate the performance of READYS in Section 4.5.

## 4.3 Models

### 4.3.1 Problem definition

The scheduling problem can be formalized as follows. As before, we are given a DAG that models the set of tasks to be scheduled. Each vertex corresponds to one task, and each directed edge expresses a dependency between the result of one task and its use by a subsequent task. The target machine is composed of heterogeneous computing units (*i.e.* CPUs and GPUs). We also focus on applications in linear algebra; in such applications, a matrix is processed and the set of tasks to be performed is made of a small number (typically 4 in this chapter) of kernels. One kernel corresponds to a certain processing performed on a sub-matrix/tile of a given size. The execution time of a kernel depends on the computing unit executing it, so that the acceleration factor on a GPU can be larger for a kernel than for another. This is typically the case in practice, due to the suitability of a given kernel on one particular computing unit, or another. Nevertheless, in practice, due to heating conditions or NUMA effects, the duration of the execution of a given task on a given resource is not constant and the variability in the processing time also depends on the resource on which they are performed(Beaumont, Eyraud-Dubois, and Yihong Gao, 2019).

In what follows, and as in the previous chapter, we assume that it is possible to overlap communications with computations, so that we can neglect communication costs. For a matrix of size $M \times M$, there are $T^2$ tiles where $T = M/N$. In the numerical linear algebra applications considered here, each tile is processed several times so that the number of tasks of the DAG is $n = \mathcal{O}(T^3)$.

As before, the *makespan* is the quantity that we aim at minimizing in this chapter. It is noteworthy that in the heterogeneous case there is neither a notion of a critical path (since we do not know in advance where the different tasks will be allocated) nor of total work (since

we do not know in advance the fraction of each task type allocated on each type of resource). Extensions of critical path relying on probabilistic estimates have been proposed in Topcuoglu, Hariri, and M.-y. Wu (2002) and a generalization of the overall work relying on rational number linear programming has been proposed in Agullo, Beaumont, et al. (2016).

Furthermore, since we are interested in learning a dynamic scheduling strategy, we do not assume that the reinforcement learning-based scheduler knows in advance the entire DAG to be scheduled, nor that it is capable of computing global statistics such as task critical path or overall work. The scheduler is myopic, considering only the tasks ready to be processed and their descendants up to a certain depth.

### 4.3.2 Reinforcement learning formulation

A **state** contains the necessary information about the system to be able to decide which action is best to perform to optimize the objective function. In our context, the state should contain information about the status of the ready tasks and their descendants up to a certain depth in the DAG, and the status of the computational resources so that the agent can decide which task to schedule next on which resource. This simple formalism can be difficult to handle because a lot of task-processor pairs can be considered. Instead, each time a decision has to be made, we choose at random one available processor, named the "current processor".

Regarding the DAG, computing the optimal solution requires exponential time and the knowledge of the whole DAG. For the reasons mentioned in Chapter 3, we consider an approximate representation, where we restrict the information represented in a state to the information about running tasks, ready tasks, and some of their descendants. The state of the resources is represented by a vector containing the type of each computing resource (CPU or GPU) and the estimated time at which it will be available, given the task already running on it.

Each vertex of the DAG, i.e. each task, is represented by a set of raw features: these features are expected to encode and summarize the DAG information at this vertex level. We use normalized quantities in order to facilitate policy transfer between graphs of different sizes. The representation $\hat{X}_i$ of Task $i$ is essentially the same as in the previous chapter, with the difference that there is no critical path anymore. It can be written as

$$\hat{X}_i = [|S(i)|, |P(i)|, type(i), ready(i), F(i)],$$

where $S(i)$ is the set of immediate successors of vertex $i$ (and $|S(i)|$ is hence the number of successors of $i$), $P(i)$ is the set of immediate predecessors of $i$ (hence $|P(i)|$ is the number of predecessors of $i$), $type(i)$ is the type of the task encoded, $ready(i)$ is a binary variable indicating if the task $i$ is ready. $F(i)$ replaces the critical path used in the previous chapter,

and summarizes the information about the descendants of task $i$: it is a vector containing the number of descendants of each type normalized by the total number of tasks of each type. More formally, if we denote by $0$ the root of the DAG, we can define the unnormalized form $\bar{F}$ of $F$ recursively by

$$\bar{F}(i) = \left( type(i) + \sum_{c \in S(i)} \frac{\bar{F}(c)}{|P(c)|} \right) \text{ and } F(i) = \frac{\bar{F}(i)}{\bar{F}(0)},$$

where $type(i)$ is the one-hot vector representing the task type. To produce a good schedule for a DAG of tasks, one has to consider not only the current task to schedule, but also the dependencies of the tasks to schedule in the (near) future. Therefore, as before, this representation $\hat{X}_i$ is passed through several GCNs layers.

Each time a computational resource becomes available, an **action** consists in selecting an available task to be run on this computational resource, or in staying idle (action $\emptyset$). This $\emptyset$ action makes it possible to implement more complex schedules than list schedules, where for example slow processors are kept idle. The set of possible actions makes $\mathcal{A}$.

The **transition** function $P$ corresponds to the computer system transiting from one state to another. Each time a task is completed, a computational resource becomes available, we arbitrarily select an available computing unit to be the "current processor". Whether the action is to schedule a task on this processor or to pass, we select a new idle processor as the current processor, and keep iterating until either all processors are executing a task or have been skipped. We then continue the simulation until a new task finishes, and repeat this process until every task has been scheduled.

As in the previous chapter, the reward function $R$ is given at the end of an episode, and is defined as the difference between the makespan of the schedule performed by the RL algorithm and the makespan computed by a baseline algorithm. Here, we use the HEFT heuristic as a baseline. HEFT is a scheduling strategy that assigns the highest priority task (priorities being computed as explained in Section 4.5.3) to the next available resource. If we denote by $makespan$ the makespan achieved by the RL agent and by $makespan$(HEFT) the makespan achieved by HEFT heuristic, the return is defined as:

$$R(\text{makespan}) = \frac{\text{makespan(HEFT)} - \text{makespan}}{\text{makespan(HEFT)}}.$$

$R$ is thus positive whenever the reinforcement algorithm is performing better than HEFT.

## 4.4 Algorithm

### 4.4.1 Actor-critic

As before, we choose A2C (Volodymyr Mnih, Badia, et al., 2016) as our RL algorithm, and use a neural network to represent the policy.

### 4.4.2 Architecture of the RL agent

In this paragraph, we very briefly outline the architecture of READYS. The neural network architecture is kept as simple as possible in order to minimize the scheduling computation overhead (see Fig. 4.1). The input of the neural network(s) is made of the DAG information and the state of the computing resources. As explained earlier, a stack of graph convolution layers mixes the information of the nodes of the DAG, up to a certain depth from the current node. The output of the last convolution layer is used as an internal representation of the DAG. The number of GCN layers is a parameter of the algorithm, as before related to the size of the window $w$ as at least $w$ layers are needed to allow information to flow between the nodes of the sub-DAG to the available tasks. This representation of the DAG is stacked together with the embedding of the computing resources described in 4.3.2 and used to compute final action probabilities. We then sample an action according to this output probability, and schedule the corresponding task on the current processor (or do not schedule anything if it is the $\emptyset$ action). We provide our implementation in Grinsztajn (2021).

## 4.5 Experiments

In this section, we report experiments in which we compare READYS with other well-known and efficient heuristics.

### 4.5.1 Task graphs

We consider three types of DAGs corresponding to Cholesky (Agullo, Beaumont, et al., 2016), LU (Agullo, Cédric Augonnet, Dongarra, Faverge, Langou, et al., 2011), and QR factorizations (Agullo, Cédric Augonnet, Dongarra, Faverge, Ltaief, et al., 2011). These applications are used in many real-life applications and are considered as a good testbed for the evaluation of runtime systems (Choi et al., 1996; Buttari et al., 2009). For instance in Baboulin, Giraud, and Gratton (2005) the authors use a Cholesky factorization to solve an electromagnetic problem ; in Bientinesi et al. (2009) authors use an dense QR factorization for solving an eigenvalue problem that can be applied in quantum chemistry, finite element modeling or multi-variate

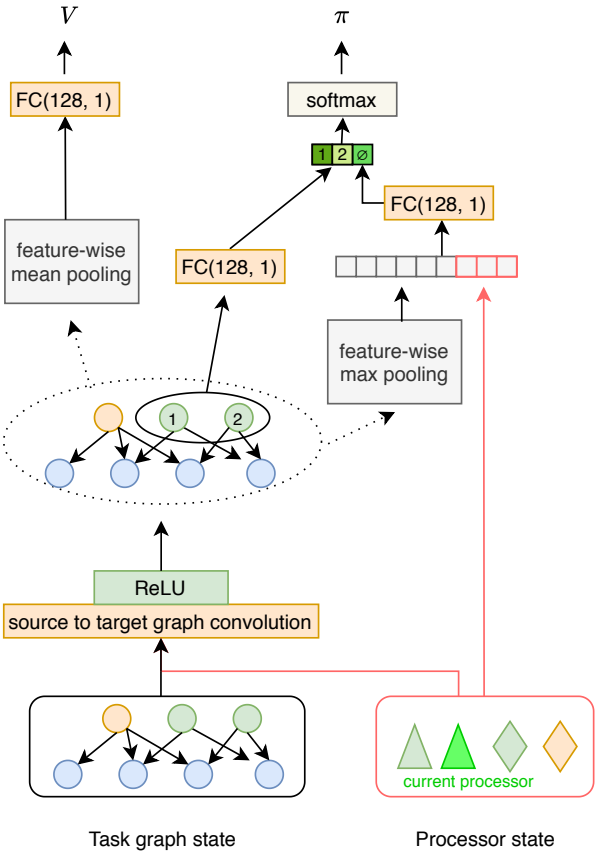



**Figure 4.1** – Overview of the architecture of READYS. At the bottom, a sub-DAG enriched with the computing resource state information is fed into a stack of several graph convolution layers and outputs an internal representation. It is used to estimate the state value $V$ via mean-pooling and one-dimensional projection. The embeddings of available tasks (here 1 and 2 are aggregated into a batch and projected onto a one-dimensional vector, which can be seen as the score of each task. This vector is concatenated with a single real number, the score of the $\emptyset$ action, computed from a projection of the processor state and the max-pooling of the internal DAG representation, and normalized with a softmax to output probabilities $\pi$. FC$(128, 1)$ denotes a fully connected layer with an input size of 128 and an output size of 1. We represent each type of processor (*eg* CPU and GPU) by a different shape. Idle processors are in green, running processors are in orange, and the current processor is in light green.

statistics. From a computer-science point of view, these factorizations involve (i) a large number of tasks, (ii) complex dependencies and (iii) a small number (4) of different kernels. Therefore they constitute a very good benchmark for scheduling algorithms (Agullo, Cédric Augonnet, Dongarra, Ltaief, et al., 2010; Jeannot, 2013) and designing good scheduling policies in this context is both very meaningful theoretically and of extreme practical importance.

### 4.5.2 Simulation model

For a task-processor pair $(i, p)$, we denote by $E(i, p)$ the expected duration of task $i$ executed on $p$, and $d(i, p)$ its actual duration. $d$ is a stochastic variable. In the experiments, $d$ is obtained by adding a Gaussian noise to the expected duration $E$. More formally, we model $d$ as follows:

$$d(i,p) = \max\left[0, \mathcal{N}\Big(E(i,p), \sigma E(i,p)\Big)\right],$$

where $\sigma$ is a parameter of the simulated environment controlling the noise level: the greater $\sigma$, the larger the uncertainty on the duration of each task. We are aware of the limits and drawbacks of this duration model. There does not exist any good model in the literature that would fit the setting under study, but the model proposed here enables us to model some uncertainty on task duration, which is an essential feature of the systems we are considering. We leave to future work either to come up with a good model, or to study the sensitivity of our analysis to various noise models. In our experiments, the expected durations of each kernel of each type of graph for each type of resources (CPU and GPU) are taken from real measurements of the literature (Agullo, Beaumont, et al., 2016; Agullo, Cédric Augonnet, Dongarra, Faverge, Langou, et al., 2011; Agullo, Cédric Augonnet, Dongarra, Faverge, Ltaief, et al., 2011).

### 4.5.3 Baselines

Our goal is to analyze the performance of a reinforcement learning-based algorithm for the dynamic scheduling of Cholesky, LU and QR factorizations. In practice, it is very difficult to accurately predict the computational costs and the communication durations in an HPC environment in which the various running processes unpredictably influence the execution times of each other. This explains the success of dynamic schedulers (Cédric Augonnet et al., 2011; Duran et al., 2011; George Bosilca et al., 2013). Indeed, since it is not possible to schedule and allocate tasks long in advance, in practice, dynamic runtimes rely solely on the description of the machine state and on the tasks already performed, using a task priority mechanism to define which tasks to perform in the event that the number of available resources is less than the number of available tasks. We have already discussed in Section 3.3.2 how to overlap communications and computations in both the CPU multicore and the GPU cases.

We have chosen HEFT (Topcuoglu, Hariri, and M.-y. Wu, 2002) as a reference static algorithm. It is a *static* list-scheduling heuristic that contrary to READYS uses the whole DAG to compute a schedule. This consists in never leaving a resource inactive if there exists a ready-to-be-processed task and breaking ties among candidate tasks by choosing the one that is farthest from the end of the computation. In the homogeneous case, it corresponds to the one with the longest critical path. It has been demonstrated in Beaumont, Langou, et al. (2020) that despite its simplicity, this strategy gives excellent results for Cholesky factorization, especially when execution times are similar to what is observed in practice on GPUs. We also compare READYS to MCT (minimum completion time). MCT is a *dynamic* heuristic that, similarly to our approach, considers tasks one after the other without considering the whole DAG. Each time a task becomes ready it is assigned to the resource where it is expected to complete the soonest (Sakellariou and Zhao, 2004).

### 4.5.4 Training

We perform a grid search on several hyper-parameters of our model, notably the window $w \in [0, 2]$, and the number $g \in [1, 3]$ of GCN layers. The networks are trained using the Adam optimizer with a learning rate of 0.01, while leaving the over hyper-parameters default in PyTorch (Paszke et al., 2019).

Regarding the actor-critic algorithm, we chose a baseline loss scaling of 0.5, and grid-searched the unroll length in $[20, 40, 60, 80]$ and the entropy loss ratio in $[10^{-3}, 5 \times 10^{-3}, 10^{-2}]$. Informally, we noted that training an agent would take approximately 20 minutes on a standard laptop with no GPU.

### 4.5.5 Results

The performance of several models trained on the three types of DAGs, for different number of tiles, CPUs and GPUs are summarized in Figure 4.2. We recall that $T$ is the number of tiles in each dimension of the matrix, hence $T^2$ tiles in total, and that there are $\mathcal{O}(T^3)$ tasks in the considered DAGs (Agullo, Beaumont, et al., 2016; Agullo, Cédric Augonnet, Dongarra, Faverge, Langou, et al., 2011; Agullo, Cédric Augonnet, Dongarra, Faverge, Ltaief, et al., 2011). We compute the improvement over HEFT (static heuristic) and MCT (dynamic heuristic). As soon as $\sigma > 0$, durations are stochastic and reported figures are obtained by averaging the performance over 5 runs/seeds. We see that when $\sigma$ is small, READYS performs similarly to HEFT (red boxes). One must keep in mind that HEFT makes use of the complete DAG to compute a schedule whereas READYS does not as it is fully dynamic and discovers the graph online. As soon as $\sigma$ increases, READYS outperforms HEFT taking advantage of the fact that it discovers task durations by itself. Compared to MCT (blue boxes) which is a fully dynamic

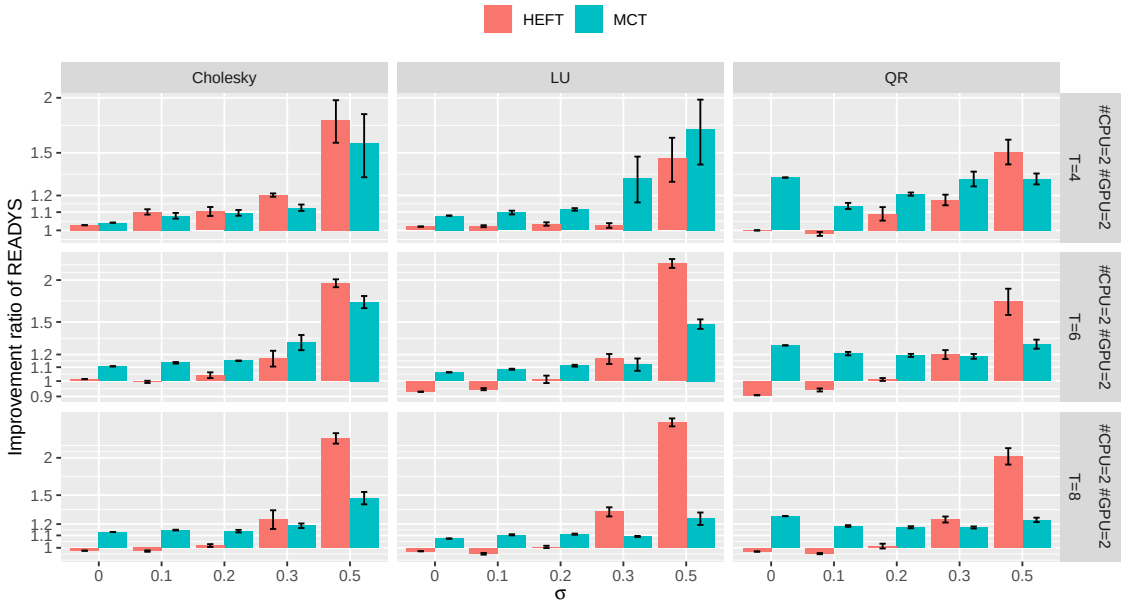



**Figure 4.2** – Makespan improvement over HEFT and MCT according to $T \in \{2, 4, 8\}$ (rows), the noise level $\sigma$, and for each of the 3 tasks we consider (3 columns), when the computing platform is made of 2 CPUs and 2 GPUs. The larger the bars above 1, the better READYS performs w.r.t. to competitors.

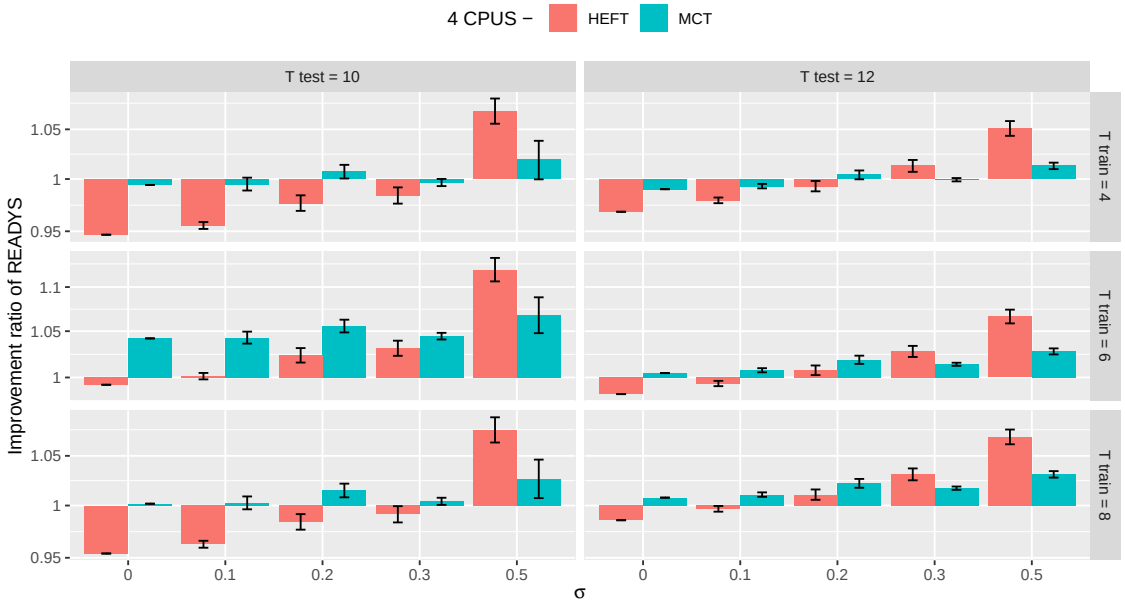



**Figure 4.3** – Transfer learning experiments: Makespan improvement over HEFT for the Cholesky task graph for several noise levels $\sigma$. On the left, the testing DAG is the tiled Cholesky with $T = 10$ tiles, and on the right the tiled Cholesky with $T = 12$ tiles. The computing platform is made of 4 CPUs.

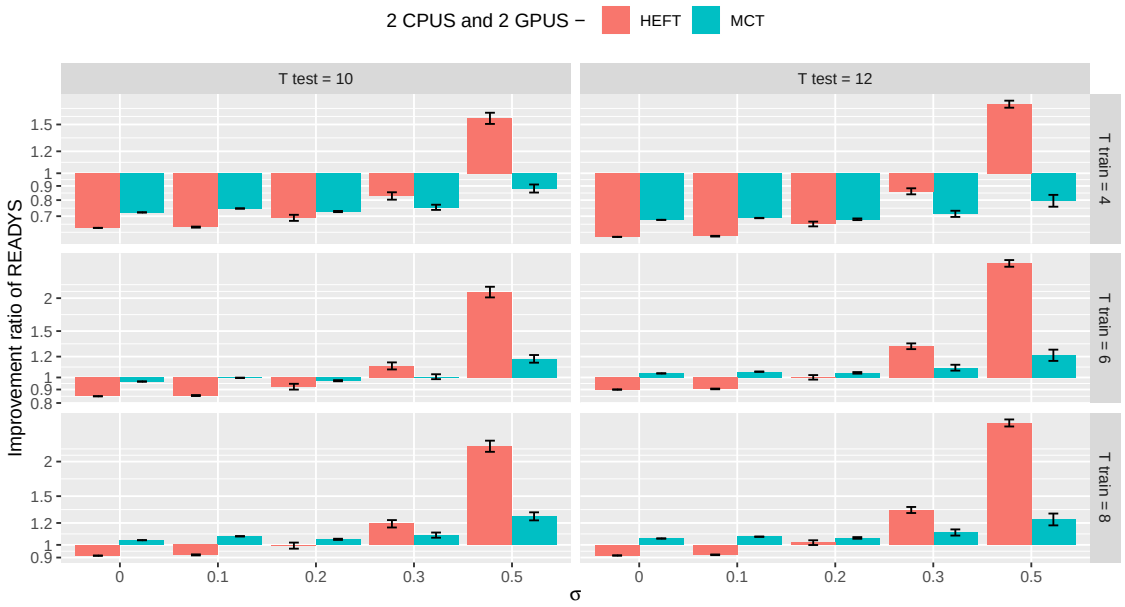



**Figure 4.4** – Transfer learning experiments: Makespan improvement over HEFT for the Cholesky task graph for several noise levels $\sigma$. On the left, the testing DAG is the tiled Cholesky with $T = 10$ tiles, and on the right the tiled Cholesky with $T = 12$ tiles. The computing platform is made of 2 CPUs and 2 GPUs.

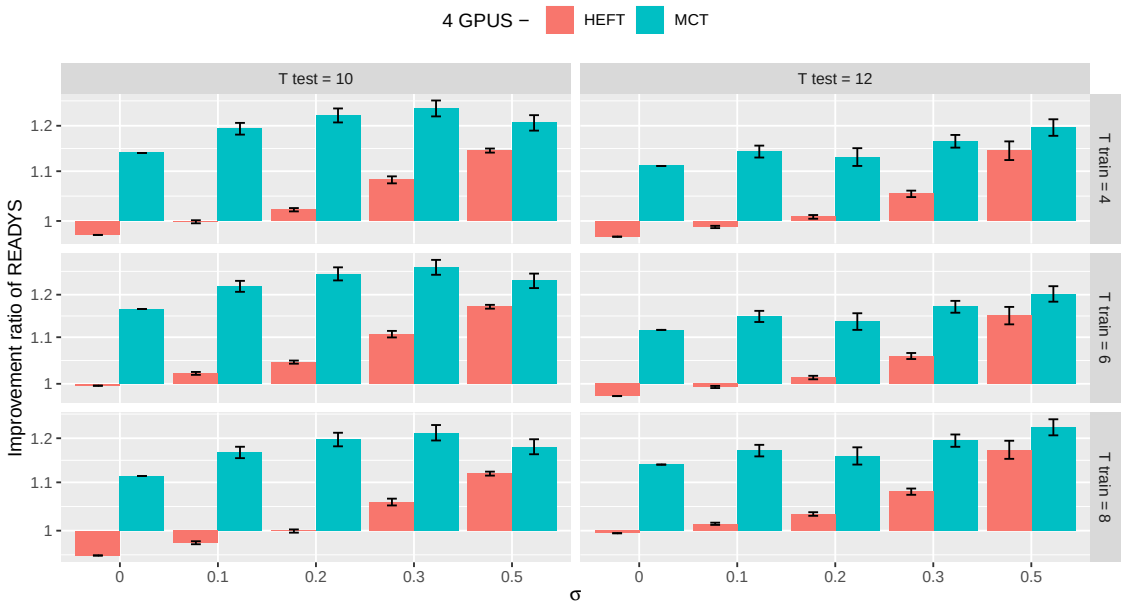



**Figure 4.5** – Transfer learning experiments: Makespan improvement over HEFT for the Cholesky task graph for several noise levels $\sigma$. On the left, the testing DAG is the tiled Cholesky with $T = 10$ tiles, and on the right the tiled Cholesky with $T = 12$ tiles. The computing platform is made of 4 GPUs.

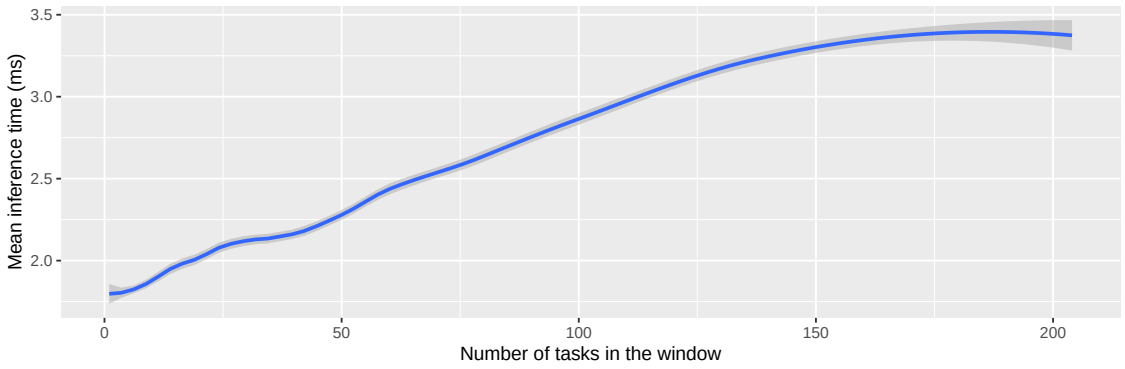



**Figure 4.6** – Mean inference time for the Cholesky DAG with 99% confidence interval.

heuristic like READYS, we see that our approach is much more efficient even for low noise, exhibiting more than 35% improvement in some cases. Moreover, the relative performance of MCT compared to READYS is roughly constant when $\sigma$ varies when the task graph is sufficiently large ($T = 6$ or more). Indeed, they are both insensitive to uncertainty about the duration of tasks (unlike HEFT), because they schedule tasks taking into account the actual state of the system and the unexpected duration of the tasks. On the opposite, HEFT computes the schedule before the execution and is less able to cope with duration variability. It also means that in order to deal with uncertainty, as soon as the input graph is large enough, it is better to make dynamic decisions than to have a complete view of the topology of the graph.

### 4.5.6 Transfer learning

Training a RL agent is well-known to be time-consuming. To overcome this difficulty, we investigate whether an agent trained to schedule a specific DAG is able to schedule other DAGs of different sizes, which is an example of transfer learning.

Reducing learning time is crucial to make reinforcement learning usable in practice. In particular, it is crucial that a scheduling policy learned on a small graph may be transferred to a larger-size graph, on which it would be too expensive to train from scratch.

We consider the Cholesky task graph and apply directly READYS trained on either $T = 4, 6$ or $8$ tiles (that is respectively $20$, $56$ and $120$ tasks) to DAGs of size $10$ and $12$ tiles ($220$ and $364$ tasks). Results are summarized in Figure 4.3, Figure 4.4 and Figure 4.5. As previously, figures are averaged over 5 runs/seeds when stochastic. These experiments exhibit very promising transfer capacities for all three considered computing platform architectures. Models trained for $T = 6$ and $T = 8$ obtain roughly similar performances when used to schedule problems of size 10 or 12, losing by only a few percents against HEFT when $\sigma = 0$, and becoming again competitive as soon as $\sigma > 0.2$. The results are even better when compared to MCT where the improvement is always positive.

We can notice that the performance of READYS varies according to computing platforms, which is not surprising as the optimal scheduling strategy may change a lot. In the case of 4 GPUs for example, scheduling tasks on the critical path is crucial, which can be difficult for baselines like MCT, hence the large improvements of READYS.

As expected, models trained for $T = 4$ obtain weaker performances: the environment used for their training is too different from the testing environment. For instance, the ratio between the different types of kernels is too different in this case. When dealing with large task graphs, this may suggest an alternative strategy to a costly training from scratch: finding an intermediate instance close enough to the initial DAG, which size is small enough so that training time remains short; train on it, and apply the learned policy to the larger instance.

### 4.5.7 Inference time

We further report wall-clock inference time on standard hardware (one CPU, no GPU) in Fig. 4.6 in order to evaluate the scheduling overhead, since scheduling decisions are made at runtime. It increases with the number of tasks in the window (in our experiments, the average number of tasks in the window is 45), but remains in the order of milliseconds, so that the scheduling overhead is reasonable as in the case of tiled algorithm kernels execution time is much larger.

## Chapter Conclusion

In this chapter we address the following question: *can Reinforcement Learning effectively be used to dynamically schedule Directed Acyclic Graphs (DAGs) on heterogeneous systems?* This is both a very difficult question as scheduling is a NP-Hard combinatorial optimization problem and a very important question as dynamic scheduling is used in many task-based runtime systems. To the best of our knowledge, we are the first to positively answer this open question.

To do so, we consider several DAGs arising from linear algebra. We demonstrate the ability of READYS to be competitive with state-of-the-art static scheduling algorithms such as HEFT even when there is no noise in the task duration estimation. This is remarkable as HEFT is a static heuristic that contrary to our approach, has full knowledge of the graph (topology and tasks durations). As READYS makes very little use of prior knowledge about the environment, it is particularly powerful when the uncertainty about the task duration is large or when the environment is stochastic, improving the results obtained by HEFT by a large margin. Compared to a dynamic approach such as MCT, the results are even better as we are able to outperform MCT in all cases, regardless of the uncertainty of task durations.

Learning scheduling algorithms for parallel heterogeneous computing platforms capable of handling stochastic duration is a key feature of our solution, since real execution environments do not generally behave in a deterministic way (*e.g.* regarding resource availability, the execution time of a given task, the communication time of a given transfer). In this case, reinforcement learning is capable of adapting to current execution conditions, dealing with unplanned situations. Moreover, and very importantly, we show that our proposed solution enables *transfer learning*. A model trained on a specific DAG of a (small) given size is able to efficiently apply the learned strategy to larger graphs.

This chapter opens several directions for future works. We useA2C as our reinforcement learning algorithm. Other algorithms that have been recently introduced (*e.g.* (Flet-Berliac et al., 2021c)) may improve our results still further. Future work could include generalizations of transfer performances, using for example techniques from few-shot learning or meta-learning. This ability to generalize and transfer knowledge is crucial: paying the full price of model training is probably the main practical obstacle to using these techniques. More broadly, this work opens new avenues for the use of reinforcement learning for scalable and practical dynamic DAG scheduling.

To this end, we have established a collaboration with the Tadaam and RealOpt teams of high-performance computing at Inria Bordeaux, which have a strong interest in dynamic scheduling problems for HPC applications. The collaboration has been ongoing for nearly a year to create a perfectly realistic simulation environment that is well-suited to reinforcement learning. Through this collaboration, we aim to scale up our tests and ideas to larger environments, as well as provide to the RL community an interesting and challenging real-world combinatorial environment.

# Part II

# Leveraging Structure and Priors

# Chapter 5

# Reversibility-Aware Reinforcement Learning

In the scheduling problem discussed in Part I, incorporating task dependencies was crucial to achieve efficient scheduling. This highlights the importance of incorporating structure, in a broad sense, to solve complex combinatorial optimization problems. In this chapter, we focus on a specific type of structure that arises at the trajectory level: action reversibility. This property was initially introduced as an inductive bias to solve Sokoban, a NP-hard combinatorial game where the agent can easily get trapped in unsolvable positions. However, it is also shown to be effective in other environments. From theoretical considerations, we show that approximate reversibility can be learned through a simple surrogate task: ranking randomly sampled trajectory events in chronological order. Intuitively, pairs of events that are always observed in the same order are likely to be separated by an irreversible sequence of actions. Conveniently, learning the temporal order of events can be done in a fully self-supervised way, which we use to estimate the reversibility of actions from experience, without any priors. We propose two different strategies that incorporate reversibility in RL agents: one for exploration (RAE), and one for control (RAC). [1]

## Contents



[1] This chapter is based on the paper Grinsztajn, Ferret, et al. (2021a) published in the proceedings of the *34th conference on advances in Neural Information Processing Systems* (*NeurIPS 2021*)

## 5.1 Introduction

We address the problem of estimating if and how easily actions can be reversed in the RL context. Irreversible outcomes are often not to be taken lightly when making decisions. As humans, we spend more time evaluating the outcomes of our actions when we know they are irreversible (McAllister, T. R. Mitchell, and Beach, 1979). As such, irreversibility can be positive (*i.e.* takes risk away for good) or negative (*i.e.* leads to later regret). Also, decision-makers are more likely to anticipate regret for hard-to-reverse decisions (Zeelenberg, 1999). All in all, irreversibility seems to be a good prior to exploit for more principled decision-making. In this chapter, we explore the option of using irreversibility to guide decision-making and confirm the following assertion: by estimating and factoring reversibility in the action selection process, safer behaviors emerge in environments with intrinsic risk factors. In addition to this, we show that exploiting reversibility leads to more efficient exploration in environments with undesirable irreversible behaviors, including the famously difficult Sokoban puzzle game.

However, estimating the reversibility of actions is no easy feat. It seemingly requires a combination of planning and causal reasoning in large dimensional spaces. We instead opt for another, simpler approach (see Fig. 5.1): we propose to learn in which direction time flows between two observations, directly from the agents' experience, and then consider *irreversible* the transitions that are assigned a temporal direction with high confidence. *In fine*, we reduce reversibility to a simple classification task that consists in predicting the temporal order of events.

Our contributions are the following: 1) we formalize the link between reversibility and precedence estimation, and show that reversibility can be approximated via temporal order, 2) we propose a practical algorithm to learn temporal order in a self-supervised way, through simple binary classification using sampled pairs of observations from trajectories, 3) we propose two novel exploration and control strategies that incorporate reversibility, and study their practical use for directed exploration and safe RL, illustrating their relative merits in synthetic as well as more involved tasks such as Sokoban puzzles.

## 5.2 Additional related work

To the best of our knowledge, this work is the first to explicitly model the reversibility of transitions and actions in the context of RL, using temporal ordering to learn from trajectories in a self-supervised way, in order to guide exploration and control. Yet, several aspects of the problem we tackle were studied in different contexts, with other motivations; we review these here.

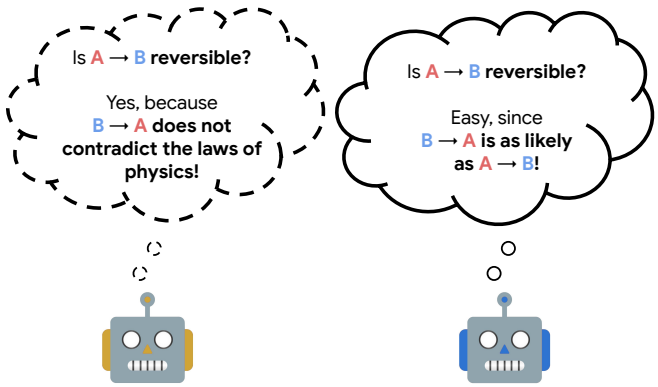



**Figure 5.1** – High-level illustration of how reversibility can be estimated. **Left:** from an understanding of physics. **Right:** ours, from experience.

**Leveraging reversibility in RL.** Kruusmaa, Gavshin, and Eppendahl (2007) estimate the reversibility of state-action couples so that robots avoid performing irreversible actions, since they are more likely to damage the robot itself or its environment. A shortcoming of their approach is that they need to collect explicit state-action pairs and their reversal actions, which makes it hard to scale to large environments. Several works (Savinov et al., 2019; Badia, Sprechmann, Vitvitskyi, D. Guo, et al., 2020; Badia, Piot, et al., 2020) use reachability as a curiosity bonus for exploration: if the current state has a large estimated distance to previous states, it means that it is novel and the agent should be rewarded. Reachability and reversibility are related, in the sense that irreversible actions lead to states from which previous states are unreachable. Nevertheless, their motivations and ours diverge, and we learn reversibility through a less involved task than that of learning reachability. Nair et al. (2020) learn to reverse trajectories that start from a goal state so as to generate realistic trajectories that reach similar goals. In contrast, we use reversibility to direct exploration and/or control, not for generating learning data. Closest to our work, Rahaman et al. (2020) propose to learn a potential function of the states that increases with time, which can detect irreversibility to some extent. A drawback of the approach is that the potential function is learned using trajectories sampled from a random policy, which is a problem for many tasks where a random agent might fail to cover interesting parts of the state space. In comparison, our method does not use a potential function and learns jointly with the RL agent, which makes it a viable candidate for more complex tasks.

**Safe exploration.** Safe exploration aims at making sure that the actions of RL agents do not lead to negative or unrecoverable effects that would outweigh the long-term value of exploration (Amodei et al., 2016). Notably, previous works developed distinct approaches to avoid irreversible behavior: by incremental updates to safe policies (Hans et al., 2008; García and Fernández, 2012), which requires knowing such a policy in advance; by restricting policy search to ergodic policies (Moldovan and Abbeel, 2012) (*i.e.* that can always come back to any

state visited), which is costly; by active exploration (Maillard et al., 2019), where the learner can ask for rollouts instead of exploring potentially unsafe areas of the state space itself; and by computing regions of attraction (Berkenkamp et al., 2016) (the part of the state space where a controller can bring the system back to an equilibrium point), which requires prior knowledge of the environment dynamics.

**Self-supervision from the arrow of time.** Self-supervision has become a central component of modern machine learning algorithms, be it for computer vision, natural language or signal processing. In particular, using temporal consistency as a source of self-supervision is now ubiquitous, be it to learn representations for downstream tasks (Goroshin et al., 2015; Ramanathan et al., 2015; Dadashi et al., 2020), or to learn to detect temporal inconsistencies (Wei et al., 2018). The closest analogies to this chapter are methods that specifically estimate some aspects of the arrow of time as self-supervision. Most are to be found in the video processing literature, and self-supervised tasks include predicting which way the time flows (Pickup et al., 2014; Wei et al., 2018), verifying the temporal order of a subset of frames (Misra, Zitnick, and Hebert, 2016), predicting which video clip has the wrong temporal order among a subset (Fernando, Bilen, et al., 2017) as well as reordering shuffled frames or clips from the video (Fernando, Gavves, et al., 2015; El-Nouby et al., 2019; D. Xu et al., 2019). Bai et al. (2020) notably propose to combine several of these pretext tasks along with data augmentation for video classification. Using time as a means of supervision was also explored for image sequencing (Basha, Moses, and Avidan, 2012), audio (Carr et al., 2021) or EEG processing (Saeed et al., 2020). In RL, self-supervision also gained momentum in recent years (Z. D. Guo, Pires, et al., 2020; Srinivas, Laskin, and Abbeel, 2020; Yarats et al., 2021), with temporal information being featured (Amiranashvili et al., 2018). Notably, several works (Aytar et al., 2018; Dwibedi et al., 2018; Z. D. Guo, Azar, Piot, et al., 2018; Sermanet et al., 2018) leverage temporal consistency to learn useful representations, effectively learning to discriminate between observations that are temporally close and observations that are temporally distant. In comparison to all these works, we estimate the arrow of time through temporal order prediction with the explicit goal of finding irreversible transitions or actions.

## 5.3 Reversibility

**Degree of reversibility.** We start by introducing formally the notion of reversibility. Intuitively, an action is reversible if it can be undone, meaning that there is a sequence of actions that can bring us back to the original state.

**Definition 5.1.** *Given a state $s$, we call* degree of reversibility within $K$ steps *of an action $a$*

$$\phi_K(s,a) \coloneqq \sup_\pi p_\pi(s \in \tau_{t+1:t+K+1} \mid s_t = s, a_t = a), \tag{5.1}$$

*and the* degree of reversibility *of an action is defined as*

$$\phi(s,a) \coloneqq \sup_\pi p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s, a_t = a), \tag{5.2}$$

*with $\tau = \{s_i\}_{i=1\dots T} \sim \pi$ corresponding to a trajectory, and $\tau_{t:t'}$ the subset of the trajectory between the timesteps $t$ and $t'$ (excluded). We omit their dependency on $\pi$ for the sake of conciseness. Given $s \in \mathcal{S}$, the action $a$ is* reversible *if and only if $\phi(s,a) = 1$, and said* irreversible *if and only if $\phi(s,a) = 0$.*

In deterministic environments, an action is either reversible or irreversible: given a state-action couple $(s,a)$ and the unique resulting state $s'$, $\phi_K(s,a)$ is equal to 1 if there is a sequence of less than $K$ actions which brings the agent from $s'$ to $s$, and is otherwise equal to zero. In stochastic environments, a given sequence of actions can only reverse a transition up to some probability, hence the need for the notion of degree of reversibility.

**Policy-dependent reversibility.** In practice, it is useful to quantify the degree of reversibility of an action as the agent acts according to a fixed policy $\pi$, for which we extend the notions introduced above. We simply write :

$$\phi_{\pi,K}(s,a) \coloneqq p_\pi(s \in \tau_{t+1:t+K+1} \mid s_t = s, a_t = a) \text{ and } \phi_\pi(s,a) \coloneqq p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s, a_t = a). \tag{5.3}$$

It immediately follows that $\phi_K(s,a) = \sup_\pi \phi_{\pi,K}(s,a)$ and $\phi(s,a) = \sup_\pi \phi_\pi(s,a)$.

## 5.4 Reversibility estimation via classification

Quantifying the exact degree of reversibility of actions is generally hard. In this section, we show that reversibility can be approximated efficiently using simple binary classification.

### 5.4.1 Precedence estimation

Supposing that a trajectory contains the states $s$ and $s'$, we want to be able to establish *precedence*, that is predicting whether $s$ or $s'$ comes first *on average*. It is a binary classification problem,

which consists in estimating the quantity $\mathbb{E}_{s_t=s,s_{t'}=s'}\left[\mathbb{1}_{t'>t}\right]$. Accordingly, we introduce the precedence estimator which, using a set of trajectories, learns to predict which state of an arbitrary pair is most likely to come first.

**Definition 5.2.** *Given a fixed policy $\pi$, we define the* finite-horizon precedence estimator *between two states as follows:*

$$\psi_{\pi,T}(s,s') = \mathop{\mathbb{E}}_{\tau\sim\pi} \mathop{\mathbb{E}}_{\substack{s_t=s,s_{t'}=s' \\ t,t'<T}} \left[\mathbb{1}_{t'>t}\right]. \tag{5.4}$$

Conceptually, given two states $s$ and $s'$, the precedence estimator gives an approximate probability of $s'$ being visited after $s$, given that both $s$ and $s'$ are observed in a trajectory. The indices are sampled uniformly within the specified horizon $T \in \mathbb{N}$, so that this quantity is well-defined even for infinite trajectories. Additional properties of $\psi$, regarding transitivity for instance, can be found in Appx. A.1.2.

**Remark 5.3.** *The quantity $\psi_{\pi,T}(s,s')$ is only defined for pairs of states which can be found in the same trajectory, and is otherwise irrelevant. In what follows, we implicitly impose this condition when considering state pairs.*

**Theorem 5.4.** *For every policy $\pi$ and $s,s' \in \mathcal{S}$, $\psi_{\pi,T}(s,s')$ converges when $T$ goes to infinity. We refer to the limit as the* precedence estimator, *written $\psi_\pi(s,s')$.*

*Proof.* The proof of this theorem is developed in Appendix A.1.3. □

This result is key to ground theoretically the notion of empirical reversibility $\bar{\phi}$, which we introduce in the next definition. It simply consists in extending the notion of precedence to a state-action pair.

**Definition 5.5.** *We finally define the* empirical reversibility *using the precedence estimator:*

$$\bar{\phi}_\pi(s,a) = \mathop{\mathbb{E}}_{s'\sim P(s,a)} \left[\psi_\pi(s',s)\right]. \tag{5.5}$$

In a nutshell, given that we start in $s$ and take the action $a$, the empirical reversibility $\bar{\phi}_\pi(s,a)$ measures the probability that we go back to $s$, starting from a state $s'$ that follows $(s,a)$. We now show that our empirical reversibility is linked with the notion of reversibility defined in the previous section, and can behave as a useful proxy.

### 5.4.2 Estimating reversibility from precedence

We present here our main theoretical result which relates reversibility and empirical reversibility:

**Theorem 5.6.** *Given a policy $\pi$, a state $s$ and an action $a$, we have: $\bar{\phi}_\pi(s,a) \geq \frac{\phi_\pi(s,a)}{2}$.*

*Proof.* The full proof of the theorem is given in Appendix A.1.3. □

This result theoretically justifies the name of empirical reversibility. From a practical perspective, it provides a way of using $\bar{\phi}$ to detect actions which are irreversible or hardly reversible: $\bar{\phi}_\pi(s,a) \ll 1$ implies $\phi_\pi(s,a) \ll 1$ and thus provides a sufficient condition to detect actions with low degrees of reversibility. This result gives a way to detect actions that are irreversible given a specific policy followed by the agent. Nevertheless, we are generally interested in knowing if these actions are irreversible for any policy, meaning $\phi(s,a) \ll 1$ with the definition of Section 5.3. The next proposition makes an explicit connection between $\bar{\phi}_\pi$ and $\phi$, under the assumption that the policy $\pi$ is stochastic.

**Proposition 5.7.** *We suppose that we are given a state $s$, an action $a$ such that $a$ is reversible in $K$ steps, and a policy $\pi$. Under the assumption that $\pi$ is stochastic enough, meaning that there exists $\rho > 0$ such that for every state and action $s, a$, $\pi(a \mid s) > \rho$, we have: $\bar{\phi}_\pi(s,a) \geq \frac{\rho^K}{2}$. Moreover, we have for all $K \in \mathbb{N}$: $\bar{\phi}_\pi(s,a) \geq \frac{\rho^K}{2}\phi_K(s,a)$.*

*Proof.* The proof is given in Appendix A.1.4. □

As before, this proposition gives a practical way of detecting irreversible moves. If for example $\bar{\phi}_\pi(s,a) < \rho^k/2$ for some $k \in \mathbb{N}$, we can be sure that action $a$ is not reversible in $k$ steps. The quantity $\rho$ can be understood as a minimal probability of taking any action in any state. This condition is not very restrictive: $\varepsilon$-greedy strategies for example satisfy this hypothesis with $\rho = \frac{\varepsilon}{|\mathcal{A}|}$.

In practice, it can also be useful to limit the maximum number of time steps between two sampled states. That is why we also define the windowed precedence estimator as follows:

**Definition 5.8.** *Given a fixed policy $\pi$, we define the* windowed precedence estimator *between two states as follows:*

$$\psi_{\pi,T,w}(s,s') = \mathop{\mathbb{E}}_{\tau\sim\pi} \mathop{\mathbb{E}}_{\substack{s_t=s, s_{t'}=s' \\ t,t'<T \\ |t-t'|\leq w}} \left[\mathbb{1}_{t'>t}\right]. \tag{5.6}$$

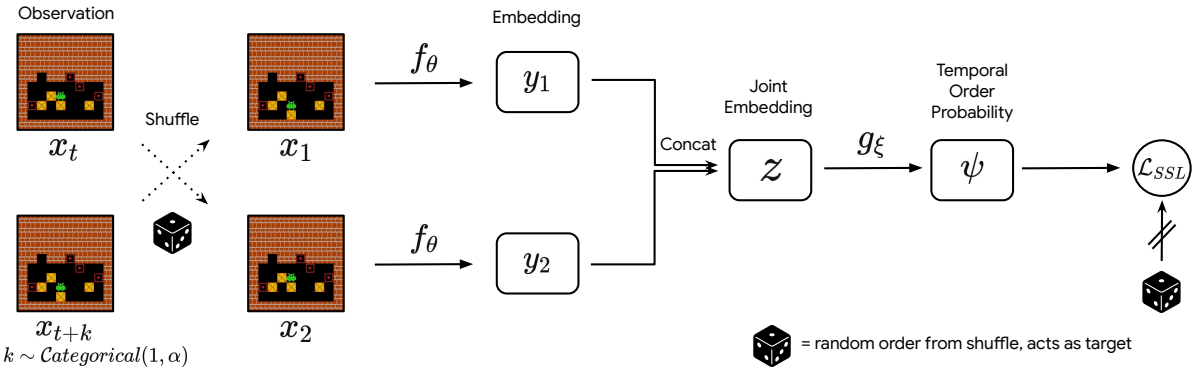



**Figure 5.2** – The proposed self-supervised procedure for precedence estimation.

Intuitively, compared to previous precedence estimators, $\psi_{\pi,T,w}$ is restricted to short-term dynamics, which is a desirable property in tasks where distinguishing the far future from the present is either trivial or impossible.

## 5.5 Reversibility-aware reinforcement learning

Leveraging the theoretically-grounded bridge between precedence and reversibility established in the previous section, we now explain how reversibility can be learned from the agent's experience and used in a practical setting.

**Learning to rank events chronologically.** Learning which observation comes first in a trajectory is achieved by binary supervised classification, from pairs of observations sampled uniformly in a sliding window on observed trajectories. This can be done fully offline, *i.e.* using a previously collected dataset of trajectories for instance, or fully online, *i.e.* jointly with the learning of the RL agent; but also anywhere on the spectrum by leveraging variable amounts of offline and online data.

This procedure is not without caveats. In particular, we want to avoid overfitting to the particularities of the behavior of the agent, so that we can learn meaningful, generalizable statistics about the order of events in the task at hand. Indeed, if an agent always visits the state $s_a$ before $s_b$, the classifier will probably assign a close-to-one probability that $s_a$ precedes $s_b$. This might not be accurate with other agents equipped with different policies, unless transitioning from $s_b$ to $s_a$ is hard due to the dynamics of the environment, which is in fact exactly the cases we want to uncover. We make several assumptions about the agents we apply our method to: 1) agents are learning and thus, have a policy that changes through

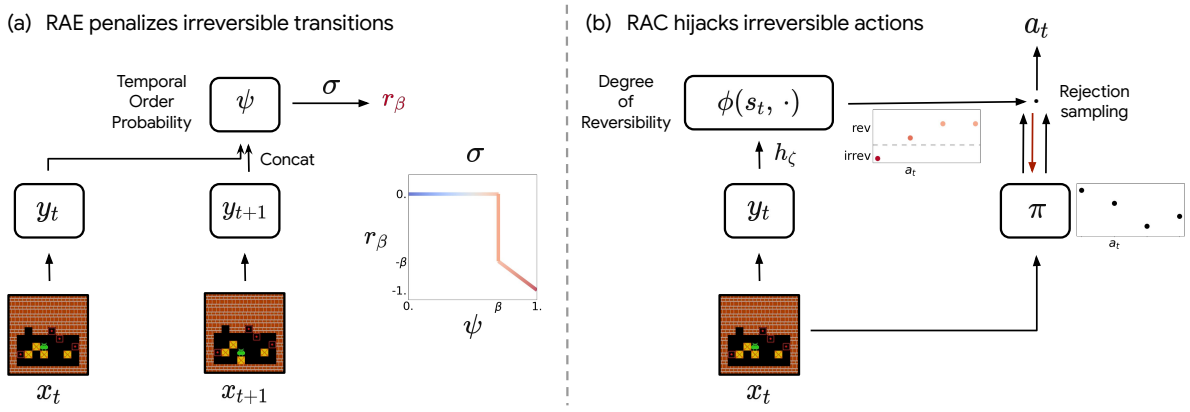



**Figure 5.3** – Our proposed methods for reversibility-aware RL. **(a):** RAE encourages reversible behavior via auxiliary rewards. **(b):** RAC avoids irreversible behavior by rejecting actions whose estimated reversibility is inferior to a threshold.

interactions in the environment, 2) agents have an incentive not to be too deterministic. For this second assumption, we typically use an entropic regularization in the chosen RL loss, which is a common design choice in modern RL methods. These assumptions, when put together, alleviate the risk of overfitting to the idiosyncrasies of a single, non-representative policy.

We illustrate the precedence classification procedure in Fig. 5.2. A temporally-ordered pair of observations, distant of no more than $w$ timesteps, is sampled from a trajectory and uniformly shuffled. The result of the shuffling operation is memorized and used as a target for the binary classification task. A Siamese network creates separate embeddings for the pair of observations, which are concatenated and fed to a separate feed-forward network, whose output is passed through a sigmoid to obtain a probability of precedence. This probability is updated via negative log-likelihood against the result of the shuffle, so that it matches the actual temporal order.

Then, a transition (and its implicit sequence of actions) represented by a starting observation $x$ and a resulting observation $x'$ is deemed irreversible if the estimated precedence probability $\psi(x, x')$ is superior to a chosen threshold $\beta$. Note that we do not have to take into account the temporal proximity of these two observations here, which is a by-product of sampling observations uniformly in a window in trajectories. Also, depending on the threshold $\beta$, we cover a wide range of scenarios, from pure irreversibility ($\beta$ close to 1) to soft irreversibility ($\beta > 0.5$, the bigger $\beta$, the harder the transition is to reverse). This is useful because different tasks call for different levels of tolerance for irreversible behavior: while a robot getting stuck and leading to an early experiment failure is to be avoided when possible, tasks involving human safety might call for absolute zero tolerance for irreversible decision-making. We elaborate on these aspects in Sec. 5.6.

**Reversibility-aware exploration and control.** We propose two different algorithms based on reversibility estimation: Reversibility-Aware Exploration (RAE) and Reversibility-Aware Control (RAC). We give a high-level representation of how the two methods operate in Fig. 5.3.

In a nutshell, RAE consists in using the estimated reversibility of a pair of consecutive observations to create an auxiliary reward function. In our experiments, the reward function is a piecewise linear function of the estimated reversibility and a fixed threshold, as in Fig. 5.3: it grants the agent a negative reward if the transition is deemed too hard to reverse. The agent optimizes the sum of the extrinsic and auxiliary rewards. Note that the specific function we use penalizes irreversible transitions but could encourage such transitions instead, if the task calls for it.

RAC can be seen as the action-conditioned counterpart of RAE. From a single observation, RAC estimates the degree of reversibility of all available actions, and "takes control" if the action sampled from the policy is not reversible enough (*i.e.* has a reversibility inferior to a threshold $\beta$). "Taking control" can have many forms. In practice, we opt for rejection sampling: we sample from the policy until an action that is reversible enough is sampled. This strategy has the advantage of avoiding irreversible actions entirely, while trading-off pure reversibility for performance when possible. RAC is more involved than RAE, since the action-conditioned reversibility is learned from the supervision of a standard, also learned precedence estimator. Nevertheless, our experiments show that it is possible to learn both estimators jointly, at the cost of little overhead.

We now discuss the relative merits of the two methods. In terms of applications, we argue that RAE is more suitable for directed exploration, as it only encourages reversible behavior. As a result, irreversible behavior is permitted if the benefits (*i.e.* rewards) outweigh the costs (*i.e.* irreversibility penalties). In contrast, RAC shines in safety-first, real-world scenarios, where irreversible behavior is to be banned entirely. With an optimal precedence estimator and task-dependent threshold, RAC will indeed hijack all irreversible sampled actions. RAC can be especially effective when pre-trained on offline trajectories: it is then possible to generate fully-reversible, safe behavior from the very first online interaction in the environment. We explore these possibilities experimentally in Sec. 5.6.2.

Both algorithms can be used online or offline with small modifications to their overall logic. The pseudo-code for the online version of RAE and RAC can be found in Appendix A.2.2.

The self-supervised precedence classification task could have applications beyond estimating the reversibility of actions: it could be used as a means of getting additional learning signal or representational priors for the RL algorithm. Nevertheless, we opt for a clear separation between the reversibility and the RL components so that we can precisely attribute improvements to the former, and leave aforementioned studies for future work.

## 5.6 Experiments

The following experiments aim at demonstrating that the estimated precedence $\psi$ is a good proxy for reversibility, and at illustrating how beneficial reversibility can be in various practical cases. We benchmark RAE and RAC on a diverse set of environments, with various types of observations (tabular, pixel-based), using neural networks for function approximation. See Appendix A.3 for details.

### 5.6.1 Reward-free reinforcement learning

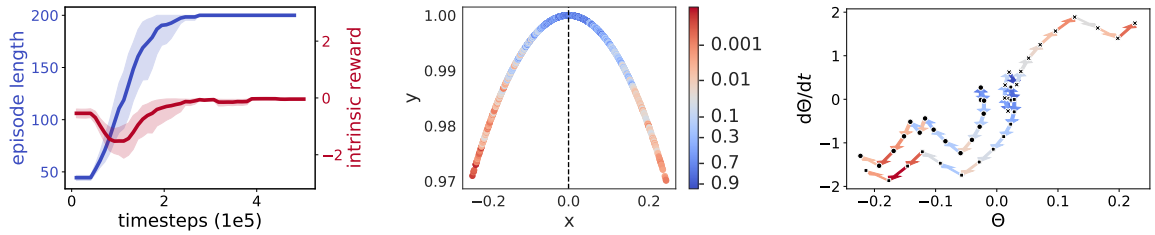
We illustrate the ability of RAE to learn sensible policies without access to rewards. We use the classic pole balancing task Cartpole (Barto, Sutton, and C. W. Anderson, 1983a), using the OpenAI Gym (Brockman et al., 2016) implementation. In the usual setting, the agent gets a reward of 1 at every time step, such that the total undiscounted episode reward is equal to the episode length, and incentivizes the agent to learn a policy that stabilizes the pole. Here, instead, we remove this reward signal and give a PPO agent (Schulman, Wolski, et al., 2017) an intrinsic reward based on the estimated reversibility, which is learned online from agent trajectories. The reward function penalizes irreversibility, as shown in Fig. 5.3. Note that creating insightful rewards is quite difficult: too frequent negative rewards could lead the agent to try and terminate the episode as soon as possible.

We display our results in Fig. 5.4. Fig. 5.4a confirms the claim that RAE can be used to learn meaningful rewards. Looking at the intrinsic reward, we discern three phases. Initially, both the policy and the reversibility classifier are untrained (and intrinsic rewards are 0). In the second phase, the classifier is fully trained but the agent still explores randomly (intrinsic rewards become negative). Finally, the agent adapts its behavior to avoid penalties (intrinsic rewards go to 0, and the length of trajectories increases). Our reward-free agent reaches the score of 200, which is the highest possible score.

To further assess the quality of the learned reversibility, we freeze the classifier after 300k timesteps and display its predicted probabilities according to the relative coordinates of the end of the pole (Fig. 5.4b) and the dynamics of the angle of the pole $\theta$ (Fig. 5.4c). In both cases, the empirical reversibility matches our intuition: the reversibility should decrease as the angle or angular momentum increase, since these coincide with an increasing difficulty to go back to the equilibrium.

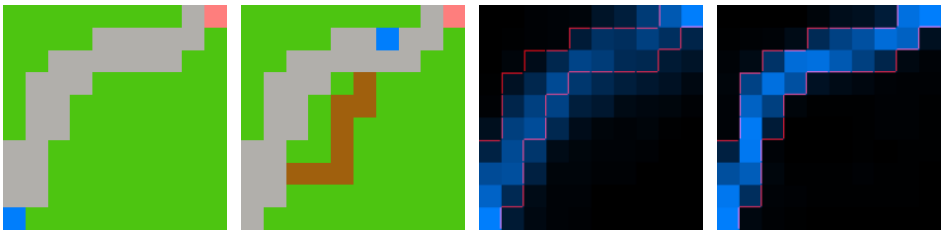
### 5.6.2 Learning reversible policies

In this section, we investigate how RAE can be used to learn reversible policies. When we train an agent to achieve a goal, we usually want it to achieve that goal following implicit safety




(**a**) Training curves (**b**) Relative pole coordinates (**c**) Random trajectories

**Figure 5.4** – **(a):** Training curves of a PPO+RAE agent in reward-free Cartpole. Blue: episode length. Red: intrinsic reward. A 95% confidence interval over 10 random seeds is shown. **(b):** The $x$ and $y$ axes are the coordinates of the end of the pole relatively to the cart position. The color denotes the online reversibility estimation between two consecutive states (logit scale). **(c):** The representation of three random trajectories according to $\theta$ (angle of the pole) and $\frac{d\theta}{dt}$. Arrows are colored according to the learned reversibility of the transitions they correspond to.



(**a**) Initial state (**b**) A trajectory (**c**) PPO (500k) (**d**) PPO+RAE (500k)

**Figure 5.5** – **(a):** The Turf environment. The agent can walk on grass, but the grass then turns brown. **(b):** An illustrative trajectory where the agent stepped on grass pixels. **(c):** State visitation heatmap for PPO. **(d):** State visitation heatmap for PPO+RAE. It coincides with the stone path (red).

constraints. Handcrafting such safety constraints would be time-consuming, difficult to scale for complex problems, and might lead to reward hacking; so a reasonable proxy consists in limiting irreversible side-effects in the environment (Leike et al., 2017).

To quantify side-effects, we propose Turf, a new synthetic environment. As depicted in Fig. 5.5a,5.5b, the agent (blue) is rewarded when reaching the goal (pink). Stepping on grass (green) will spoil it, causing it to turn brown. Stepping on the stone path (grey) does not induce any side-effect.

In Fig. 5.5c,5.5d, we compare the behaviors of a trained PPO agent with and without RAE. The baseline agent is indifferent to the path to the goal, while the agent benefitting from RAE learns to follow the road, avoiding irreversible consequences.

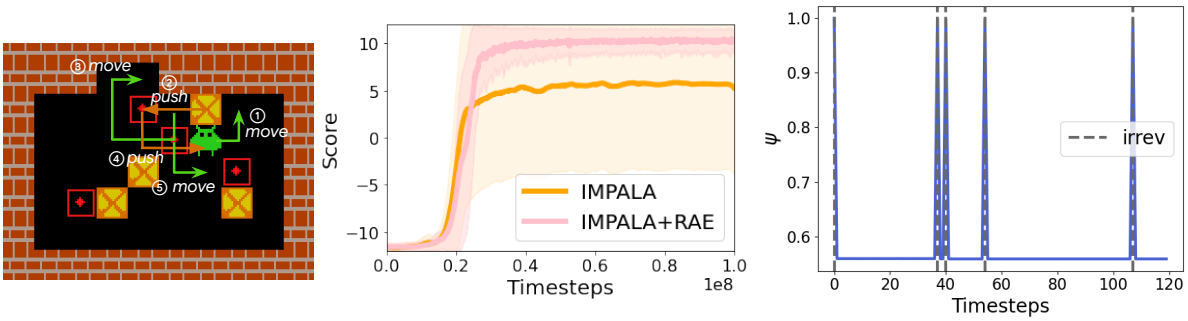
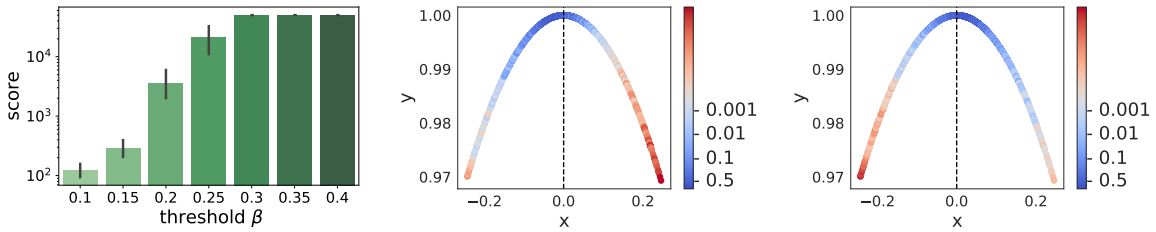



**Figure 5.6** – **(a):** Non-trivial reversibility: pushing the box against the wall can be reversed by pushing it to the left, going around, pushing it down and going back to start. A minimum of 17 moves is required to go back to the starting state. **(b):** Performances of IMPALA and IMPALA+RAE on 1k levels of Sokoban (5 seeds average). **(c):** Evolution of the estimated reversibility along one episode.

### 5.6.3 Sokoban

Sokoban is a popular puzzle game where a warehouse keeper (controlled by the player) must move boxes around and place them in dedicated places. Each level is unique and involves planning, since there are many ways to get stuck. For instance, pushing a box against a wall is often un-undoable, and prevents the completion of the level unless actually required to place the box on a specific target. Sokoban is a challenge to current model-free RL algorithms, as advanced agents require millions of interactions to reliably solve a fraction of levels (Weber et al., 2017; Guez, Mirza, Gregor, et al., 2019). One of the reasons for this is tied to exploration: since agents learn from scratch, there is a long preliminary phase where they act randomly in order to explore the different levels. During this phase, the agent will lock itself in unrecoverable states many times, and further exploration is wasted. It is worth recalling that contrary to human players, the agent does not have the option to reset the game when stuck. In these regards, Sokoban is a great testbed for reversibility-aware approaches, as we expect them to make the exploration phase more efficient, by incorporating the prior that irreversible transitions are to be avoided if possible, and by providing tools to identify such transitions.

We benchmark performance on a set of 1k levels. Results are displayed in Fig. 5.6. Equipping an IMPALA agent (Espeholt et al., 2018) with RAE leads to a visible performance increase, and the resulting agent consistently solves all levels from the set. We take a closer look at the reversibility estimates and show that they match the ground truth with high accuracy, despite the high imbalance of the distribution (*i.e.* few irreversible transitions, see Fig. 5.6c) and complex reversibility estim ation (see Fig. 5.6a).




(**a**) Trajectory lengths (**b**) Coordinates: action 0 (**c**) Coordinates: action 1

**Figure 5.7 – (a):** Mean score of a random policy augmented with RAC on Cartpole+ for several threshold values, with 95% confidence intervals over 10 random seeds (log scale). **(b) and (c):** The $x$ and $y$ axes are the coordinates of the end of the pole relatively to the cart position. The color indicates the estimated reversibility values.

### 5.6.4 Safe control

In this section, we put an emphasis on RAC, which is particularly suited for safety related tasks.

**Cartpole+.** We use the standard Cartpole environment, except that we change the maximum number of steps from 200 to 50k to study long-term policy stability. We name this new environment Cartpole+. It is substantially more difficult than the initial setting. We learn reversibility offline, using trajectories collected from a random policy. Fig. 5.7a shows that a random policy augmented with RAC achieves seemingly infinite scores. For the sake of comparison, we indicate that a DQN (Volodymyr Mnih, Kavukcuoglu, et al., 2015) and the state-of-the-art M-DQN (Vieillard, Pietquin, and Geist, 2020) achieve a maximum score of respectively 1152 and 2801 under a standard training procedure, described in Appendix A.3.5. This can be surprising, since RAC was only trained on random thus short trajectories (mean length of 20). We illustrate the predictions of our learned estimator in Fig. 5.7b,5.7c. When the pole leans to the left ($x < 0$), we can see that moving the cart to the left is perceived as more reversible than moving it to the right. This is key to the good performance of RAC and perfectly on par with our understanding of physics: when the pole is leaning in a direction, agents must move the cart in the same direction to stabilize it.

**Turf.** We now illustrate how RAC can be used for safe online learning: the implicitly safe constraints provided by RAC prevent policies from deviating from safe trajectories. This ensures for example that agents stay in recoverable zones during exploration.

We learn the reversibility estimator offline, using the trajectories of a random policy. We reject actions whose reversibility is deemed inferior to $\beta = 0.2$, and train a PPO agent with RAC. As displayed in Fig. 5.8, PPO with RAC learns to reach the goal without causing any irreversible side-effect (*i.e.* stepping on grass) during the whole training process.

The threshold $\beta$ is a very important parameter of the algorithm. Too low a threshold could lead to overlooking some irreversible actions, while a high threshold could prevent the agent from learning the new task at hand. We discuss this performance/safety trade-off in more details in Appendix. A.3.7.

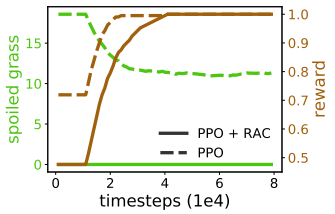



**Figure 5.8** – PPO and RAC (solid lines) vs PPO (dashed lines). At the cost of slower learning (brown), our approach prevents the agent from producing a single irreversible side-effect (green) during the learning phase. Curves are averaged over 10 runs.

## Chapter Conclusion

In this chapter, we formalized the link between the reversibility of transitions and their temporal order, which inspired a self-supervised procedure to learn the reversibility of actions from experience. In combination with two novel reversibility-aware exploration strategies, RAE for directed exploration and RAC for directed control, we showed the empirical benefits of our approach in various scenarios, ranging from safe RL to risk-averse exploration. Notably, and in line with our initial goal, we demonstrated increased performance in procedurally-generated Sokoban puzzles, which we tied to more efficient exploration. This is a first hint in the direction that exploiting specific structures of combinatorial environments can improve the performance of RL agents.

# Chapter 6

# Better state exploration using action sequence equivalence

After the previous chapter targeted at a particular type of trajectory structure, we consider now a more general problem: exploiting prior information about action sequence equivalence: that is, when different sequences of actions produce the same effect. This is something that occurs in many combinatorial environments. In KP for example, where an action corresponds to picking up an item, every action commutes. We propose a new local exploration strategy calibrated to minimize collisions and maximize new state visitations. We show that this strategy can be computed at little cost by solving a convex optimization problem. By replacing the usual $\varepsilon$-greedy strategy in a DQN, we demonstrate its potential in several environments with various dynamic structures. [1]

**Contents**



## 6.1 Introduction

Despite the rapidly improving performance of Reinforcement Learning (RL) agents on a variety of tasks (Volodymyr Mnih, Kavukcuoglu, et al., 2015; Silver, Huang, Christopher J.

[1] This chapter is based on Grinsztajn, Johnstone, et al. (2022) presented at the *Deep Reinforcement Learning Worskshop* at the Neurips 2022 conference

Maddison, et al., 2016), they remain largely sample-inefficient learners compared to humans (Toromanoff, Wirbel, and Moutarde, 2019). Contributing to this is the vast amount of prior knowledge humans bring to the table before their first interaction with a new task, including an understanding of physics, semantics, and affordances (Dubey et al., 2018).

The considerable quantity of data necessary to train agents is becoming more problematic as RL is applied to ever more challenging and complex tasks. Much research aims at tackling this issue, for example through transfer learning (Rusu et al., 2016), meta learning, and hierarchical learning, where agents are encouraged to use what they learn in one environment to solve a new task more quickly. Other approaches attempt to use the structure of Markov Decision Processes (MDP) to accelerate learning without resorting to pretraining. Mahajan and Tulabandhula (2017) and Biza and Jr. (2019) learn simpler representations of MDPs that exhibit symmetrical structure, while Pol et al. (2020) show that environment invariances can be hard-coded into equivariant neural networks.

A fundamental challenge standing in the way of improved sample efficiency is exploration. We consider a situation where the exact transition function of a Markov Decision Process is unknown, but some knowledge of its local dynamics is available under the form of a prior expectation that given sequences of actions have identical results. This way of encoding prior knowledge is sufficiently flexible to describe many useful environment structures, particularly when actions correspond to agent movement. For example, in a gridworld (called RotationGrid hereafter) where the agent can move forward ($\uparrow$) and rotate $90°$ to the left ($\curvearrowleft$) or to the right ($\curvearrowright$), the latter two actions are the *inverse* of each other, in that performing one undoes the effect of the other. During exploration, to encourage the visitation of not yet seen states, it is natural to simply ban sequences of actions that revert to previously visited states, following the reasoning of Tabu search (Glover, 1986b). We observe further that $\curvearrowright\curvearrowright$ and $\curvearrowleft\curvearrowleft$ both lead to the same state (represented as state 4 in Figure 6.1). If actions were uniformly sampled, the chances of visiting this state would be much higher than any of the others. Based on these observations, we introduce a new method taking advantage of Equivalent Action SEquences for Exploration (EASEE), an overview of which can be found in Figure 6.1. EASEE looks ahead several steps and calculates action sampling probabilities to explore as uniformly as possible new states conditionally on the action sequence equivalences given to it. It constructs a partial MDP which corresponds to a local representation of the true MDP around the current state. We then formulate the problem of determining the best distribution over action sequences as a linearly constrained convex optimization problem. Solving this optimization problem is computationally inexpensive and can be done once and for all before learning begins, providing a principled and tractable exploration policy that takes into account environment structure. This policy can easily be injected into existing reinforcement learning algorithms as a substitute for $\varepsilon$-greedy exploration.

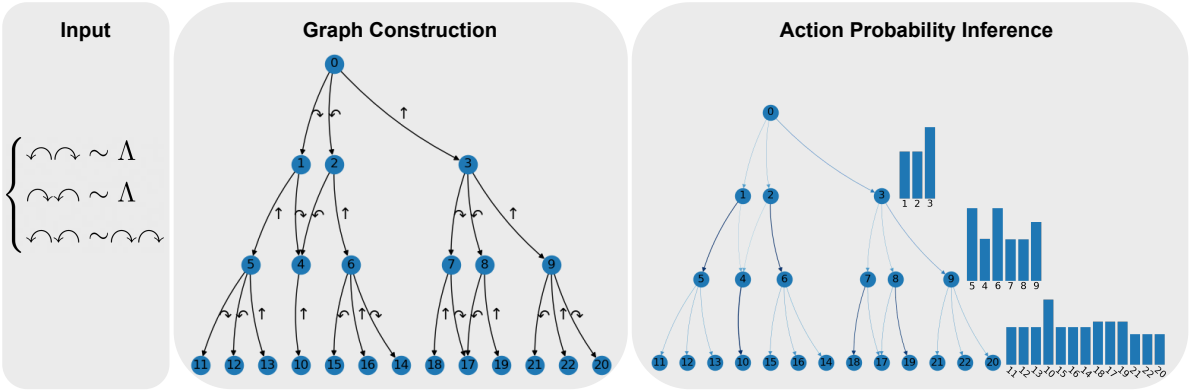



**Figure 6.1** – Illustration of EASEE on RotationGrid environment. The input is information about the dynamics of the environment known in advance under the form of action sequence equivalences ($\Lambda$ denotes the empty action sequence). This is used to construct a representation of all the unique states that can be visited in 3 steps. The probabilities of sampling each action are then determined to explore as uniformly as possible. The probabilities of visiting each unique state are displayed on the right.

Our contribution is threefold. First, we formally introduce the notion of equivalent action sequences, a novel type of structure in Markov Decision Processes. Then, we show that priors on this type of structure can easily be exploited during offline exploration by solving a convex optimization problem. Finally, we provide experimental insights and show that incorporating EASEE into a DQN (Volodymyr Mnih, Kavukcuoglu, et al., 2015) improves agent performance in several environments with various structures.

## 6.2 Additional related work

**Curiosity-driven exploration** The problem of ensuring that agents see sufficiently diverse states has received a lot of attention from the RL community. Many methods rely on intrinsic rewards (Schmidhuber, 1991; Chentanez, Barto, and Singh, 2005; Şimşek and Barto, 2006; Lopes et al., 2012; Marc G. Bellemare et al., 2016; Ostrovski et al., 2017; Pathak et al., 2017) to entice agents to unseen or misunderstood areas. In the tabular setting, these take the form of count-based exploration bonuses which guide the agent toward poorly visited states (*e.g.* Strehl and Littman (2008)). Scaling this method requires the use of function approximators (Burda et al., 2019; Badia, Sprechmann, Vitvitskyi, Z. D. Guo, et al., 2020; Flet-Berliac et al., 2021a). Unlike EASEE, these methods necessitate the computation of non-stationary and vanishing novelty estimates, which require careful tuning to balance learning stability and exploration incentives. Moreover, because these bonuses are learned, and do not allow for the use of prior structure knowledge, they constitute an orthogonal approach to ours.

**Redundancies in trajectories** The idea that different trajectories can overlap and induce redundancies in state visitation is used in Leurent and Maillard (2020) and Czech, Korus, and Kersting (2020) in the case of Monte-Carlo tree search. However, they require a generative model, and propose a new Bellman operator to update node values according to newly uncovered transitions rather than modifying exploration. Closer to our work, Caselles-Dupré, Garcia-Ortiz, and Filliat (2020) study structure in action sequences, but restrict themselves to commutative properties. Tabu search (Glover, 1986b) is a meta-heuristic which uses knowledge of the past to escape local optima. It is popular for combinatorial optimization (Hertz and Werra, 2005). Like our approach, it relies on a local structure: actions which are known to cancel out recent moves are deemed *tabu*, and are forbidden for a short period of time. This prevents cycling around already found solutions, and thus encourages exploration. In Abramson and Wechsler (2003), tabu search is combined with reinforcement learning, using action priors. However, their method cannot make use of more complex action-sequence structure.

**Maximum state-visitation entropy** Our goal to explore as uniformly as possible every nearby state can be seen as a local version of the Maximum State-Visitation Entropy problem (MSVE) (Farias and Van Roy, 2003; Hazan et al., 2019; Lee et al., 2019; Z. D. Guo, Azar, Saade, et al., 2021). MSVE formulates exploration as a policy optimization problem whose solution maximizes the entropy of the distribution of visited states. Although some of these works (Hazan et al., 2019; Lee et al., 2019; Z. D. Guo, Azar, Saade, et al., 2021) can make use of priors about state similarities, they learn a global policy and cannot exploit structure in action sequences.

**Action space structure** The idea of exploiting structure in action spaces is not new. Large discrete action spaces may be embedded in continuous action spaces either by leveraging prior information (Dulac-Arnold et al., 2016) or learning representations (Chandak et al., 2019). Tavakoli, Pardo, and Kormushev (2018) manage high-dimensional action spaces by assuming a degree of independence between each dimension. These methods aim to improve the generalization of policies to unseen actions rather than enhancing exploration. Tennenholtz and Mannor (2019) provide an understanding of actions through their context in demonstrations. Farquhar et al. (2020) introduce a curriculum of progressively growing action spaces to accelerate learning. Certain characteristics of actions, such as reversibility, can be learned through training, as in Grinsztajn, Ferret, et al. (2021b).

## 6.3 Formalism

### 6.3.1 Equivalence over action sequences

We consider a *Markov Decision Process* (MDP) defined as a 5-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \gamma)$, with $\mathcal{S}$ the set of states, $A$ the action set, $T$ the transition function, $R$ the reward function and the discount factor $\gamma$. The set of actions is assumed to be finite $|\mathcal{A}| < \infty$. We restrict ourselves to deterministic MDPs. A possible extension to MDPs with stochastic dynamics is discussed in Appendix B.1.6.

In the following, the notations are borrowed from formal language theory. Sequences of actions are analogous to strings over the set of symbols $\mathcal{A}$ (possible actions). The set of all possible sequences of actions is denoted $\mathcal{A}^\star = \bigcup_{k=0}^{\infty} \mathcal{A}^k$ where $\mathcal{A}^k$ is the set of all sequences of length $k$ and $\mathcal{A}^0$ contains as single element the empty sequence $\Lambda$. We use . for the concatenation operator, such that for $v_1 \in \mathcal{A}^{h_1}, v_2 \in \mathcal{A}^{h_2}, v_1.v_2 \in \mathcal{A}^{h_1+h_2}$. The transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ gives the next state $s'$ when action $a$ is taken in state $s$: $T(s, a) = s'$. We recursively extend this operator to action sequences $T : \mathcal{S} \times \mathcal{A}^\star \to \mathcal{S}$ such that, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \forall w \in \mathcal{A}^\star$:

$$T(s, \Lambda) = s$$
$$T(s, w.a) = T(T(s, w), a)$$

Intuitively, this operator gives the new state of the MDP after a sequence of actions is performed from state $s$.

**Definition 6.1** (Equivalent sequences)**.** *We say that two action sequences $a_1 \dots a_n$ and $a'_1 \dots a'_m \in \mathcal{A}^\star$ are equivalent at state $s \in \mathcal{S}$ if*

$$T(s, a_1 \dots a_n) = T(s, a'_1 \dots a'_m) \tag{6.1}$$

*Two sequences of actions are equivalent over $\mathcal{M}$ if they are equivalent at state $s$ for all $s$ in $\mathcal{S}$. This is written:*

$$a_1 \dots a_n \sim_{\mathcal{M}} a'_1 \dots a'_m \tag{6.2}$$

This means that we consider two sequences of actions to be equivalent when following one or the other will always lead to the same state. When the considered MDP $\mathcal{M}$ is unambiguous, we simplify the notation by writing $\sim$ instead of $\sim_{\mathcal{M}}$.

We argue that some priors about the environments can be easily encoded as a small set of action sequence equivalences. For example, we may know that going left then right is the same

thing as going right then left, that rotating two times to the left is the same thing as rotating two times to the right, or that opening a door twice is the same thing as opening the door once. All these priors can be encoded as a set of equivalences:

**Definition 6.2** (Equivalence set)**.** *Given a MDP $\mathcal{M}$ and several equivalent sequence pairs $v_1 \sim w_1, v_2 \sim w_2, \dots, v_n \sim w_n$, we say that $\Omega = \{\{v_1, w_1\}, \{v_2, w_2\}, \dots \{v_n, w_n\}\}$ is an equivalence set over $\mathcal{M}$.*

Formally, $\Omega$ is a set of pairs of elements of $\mathcal{A}^\star$, such that $\Omega \subset (\mathcal{A}^\star)^2$. By abuse of notation, we write $v \sim w \in \Omega$ if $\{v, w\} \in \Omega$.

Intuitively, it is clear that action sequence equivalences can be combined to form new, longer equivalences. For example, knowing that going left then right is the same thing as going right then left, we can deduce that going two times left then two times right is the same thing as going two times right then two times left. In the same fashion, if opening a door twice produces the same effect as opening it once, opening three times the door does the same. We formalize these notions in what follows. First, we note that equivalent sequences can be concatenated.

**Proposition 6.3.** *If we have two pairs of equivalent sequences over $\mathcal{M}$,* i.e. *$w_1, w_2, w_3, w_4 \in \mathcal{A}^\star$ such that*

$$w_1 \sim w_2$$

$$w_3 \sim w_4$$

*then the concatenation of the sequences are also equivalent sequences:*

$$w_1 \cdot w_3 \sim w_2 \cdot w_4$$

The proof is given in Appendix B.1.1. We are now going to define formally the fact that the equivalence of two sequences can be deduced from an equivalence set $\Omega$. We first consider the previous example where an action $a$ has the effect of opening a door, such that $a.a \sim a$. We can then write $a.a.a \sim (a.a).a \sim (a).a \sim a.a \sim a$ by applying two times the equivalence $a.a \sim a$ and rearranging the parentheses. More generally and intuitively, the equivalence of two action sequences $v$ and $w$ can be deduced from $\Omega$, which we denote $v \sim_\Omega w$, if $v$ can be changed into $w$ iteratively, chaining equivalences of $\Omega$.

More formally, we write $v \sim_\Omega^1 w$ if $v$ can be changed to $w$ in one step, meaning:

$$\exists u_1, u_2, v_1, w_1 \in \mathcal{A}^\star \text{such that} \begin{cases} v = u_1.v_1.u_2 \\ w = u_1.w_1.u_2 \\ v_1 \sim w_1 \in \Omega \end{cases} \tag{6.3}$$

For $n \geq 2$, we say that $v$ can be changed into $w$ in $n$ steps if there is a sequence $v_1, \dots, v_n \in \mathcal{A}^\star$ such that $v \sim^1_\Omega v_1 \sim^1_\Omega \cdots \sim^1_\Omega v_n = w$. Finally, we say that $v \sim_\Omega w$ if there is $n \in \mathbb{N}$ such that $v$ can be changed into $w$ in $n$ steps. The relation $\sim_\Omega$ is thus a formal way of extending equivalences from a fixed equivalence set $\Omega$, and at first glance not connected with $\sim$, which deals with the equivalences of the MDP dynamics. We now show a connection between the two notions.

**Theorem 6.4.** *Given an equivalence set $\Omega$, $\sim_\Omega$ is an equivalence relationship. Furthermore, for $v, w \in \mathcal{A}^\star$, $v \sim_\Omega w \Rightarrow v \sim w$.*

The proof is given in Appendix B.1.2. Given this relation between $\sim$ and $\sim_\Omega$, we will simplify the notation in what follows by writing $\sim$ instead of $\sim_\Omega$ when the equivalence set considered is unambiguous. As $\sim_\Omega$ is an equivalence relationship, it provides a partition over action sequences: two action sequences in the same set lead to the same final state from any given state.

### 6.3.2 Local-dynamics graph

We leverage the equivalences defined above to determine a model of the MDP up to a few timesteps. As traditionally done in Monte-Carlo Tree Search (Coulom, 2007a), an MDP $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ with deterministic dynamics can be locally unrolled to produce a tree, where a node of depth $h$ represents a sequence of actions $v \in \mathcal{A}^h$, and the edges represent transitions between such sequences. The root of the tree corresponds to the empty action sequence $\Lambda$. Here we adopt the same formalism, except that equivalent sequences will point to the same node.

Given a tree $\mathcal{T}$ of depth $d \in \mathbb{N}$ corresponding to a partial unrolling of sequences in $\mathcal{A}^\star$, and an equivalence set $\Omega$, we call *local-dynamics graph* of depth $d$ under equivalence $\Omega$ the graph $\mathcal{G} = (V, E)$ corresponding to the tree $\mathcal{T}$ where nodes are quotiented with the equivalence relation $\sim_\Omega$. Intuitively, it means that nodes corresponding to equivalent action sequences are merged. In this case, the resulting graph is not necessarily a tree. In the following, unless the distinction is necessary, we identify action sequences with their equivalence classes.

The graph $\mathcal{G}$ gives rise to a new, smaller MDP resulting from $\mathcal{M}$: the state space $V$ is the set of action sequences smaller than $d$ quotiented by the equivalence relation $\sim_\Omega$, the action

space $\mathcal{A}$ is untouched. Given a node $n$ corresponding to a sequence $w \in \mathcal{A}^\star$, and an action $a \in \mathcal{A}$, $T(n, a)$ is the node representing the sequence $w.a \in \mathcal{A}^\star$. Nodes representing sequences of length exactly $d$ are *final states*. The initial state $v_0$ is the empty sequence $\Lambda$. This MDP represents the local dynamics induced by $\sim_\Omega$ from a given root state. We detail in the next section how to construct such graphs in practice, and how to use these sub-MDPs for a better exploration.

## 6.4 Method

### 6.4.1 From equivalent actions to local-dynamics graph

Producing the local-dynamics graph involves considering all possible action sequences and merging those that are equivalent. Figure 6.2 illustrates the construction of a local-dynamics graph, given $\mathcal{A} = \{a_1, a_2\}$ and $\Omega = \{a_1a_1 \sim \Lambda, a_2a_1 \sim a_1a_2\}$. Starting from the root node $0$ (first step), we iteratively expand the graph by unrolling the nodes at the edges of the graph. Steps 2 and 3 create nodes 1 and 2 corresponding to action sequences $a_1$ and $a_2$ respectively. In a tree, the expansion of a node corresponding to a sequence $w \in \mathcal{A}^h$ with the action $a \in \mathcal{A}$ always leads to the creation of a new leaf that results from the sequence of actions $w.a \in \mathcal{A}^{h+1}$. However, in a local-dynamics graph the node representing $w.a$ might already be present, in which case we add an edge from $w$ without creating a new node. In Figure 6.2, this case occurs at the $4^{\text{th}}$ and the $6^{\text{th}}$ construction steps. The first case corresponds to adding the action $a_1$ to the node $1$, which represents the action sequence $a_1.a_1$. Since $a_1.a_1 \sim \Lambda \in \Omega$, we simply add an edge from $1$ to $0$. The second case occurs when expanding node $2$ with the action $a_1$, leading to the action sequence $a_2.a_1 \sim_\Omega a_1.a_2$. Since node $3$ already represents $a_1.a_2$, we simply add an edge from node $2$ to node $3$. As a final construction step, we prune edges which go backward in the local-dynamics graph, like $(1, 0)$ in Fig. 6.2, such that the resulting graph is a DAG. This is motivated by the fact that we are interested in finding a good exploration policy: an action which takes us back to a previously visited state should be ignored.

From a practical point of view, the graph construction algorithm takes as input the action set $\mathcal{A}$, the sequence equivalence set $\Omega$, and the desired depth $d$, and outputs a DAG. Informally, it starts from a graph $\mathcal{G} = (V, E)$ reduced to a root state $\{0\}$ and iteratively expands $\mathcal{G}$ until a distance $d$ to the root is reached. We store in each node every action sequence which allows to reach it from any parent nodes. For example, in Fig.6.2, the node $3$ can be reached from the node $0$ with sequences $\{a_1.a_2, a_2.a_1\}$, from node $1$ with sequence $\{a_2\}$ and from node $2$ with sequence $\{a_1\}$, thus the set of sequences stored in node $3$ would be $\{a_1, a_2, a_1.a_2, a_2.a_1\}$. When expanding a node $n$ with an action $a \in \mathcal{A}$, we check every sequence $w$ stored in $n$ if $w.a$ appears in $\Omega$, and if a node corresponding to an equivalent sequence of $w.a$ is already in $V$. If it is the case, we simply add an edge from $n$ to this node, otherwise we create a new node representing

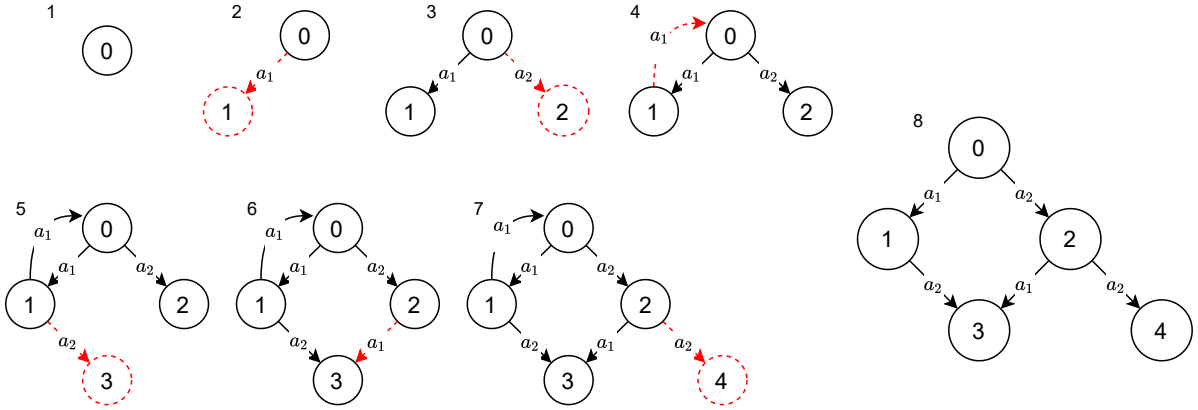



**Figure 6.2** – Example of iterative graph construction with $\Omega = \{a_1a_1 \sim \Lambda, a_2a_1 \sim a_1a_2\}$ and a maximum depth of 2. The $8^{\text{th}}$ construction step corresponds to the pruning of the edge $(1, 0)$.

$w.a$. In practice, this algorithm can be refined: in each node, we only store sequences which are subsequences of sequences in $\Omega$. We provide a more detailed implementation of this algorithm in Appendix B.1.3.

**Proposition 6.5.** *The complexity of this graph construction algorithm is upper bounded by* $\mathcal{O}\left(|\mathcal{A}|^{2d}|\Omega|d\right)$.

The proof is given in Appendix B.1.4. It is to be noted that this upper bound is in general far larger than the actual number of operations. Indeed, it supposes that the number of nodes in the graph is $|\mathcal{A}|^d$, although it can be much smaller thanks to the redundancies induced by $\Omega$. A more precise formula is $\mathcal{O}\left(|V||\mathcal{A}|^d|\Omega|\right)$, where $|V|$ is the number of nodes in the final graph and depends on the structure of $\Omega$. Despite this exponential theoretical complexity, the goal is to use this algorithm locally, thus for small depths. In practice we found that local-dynamics graphs could be computed within a few seconds on a standard laptop.

### 6.4.2 From local-dynamics graph to local exploration policy

Once the local-dynamics graph $(V, E)$ has been constructed, our goal is to find a good local exploration policy in the resulting MDP as defined in Section 6.3.2. We recall that its set of states is $V$, and its actions dynamics are given by the edges $E$. Ideally, we would want to find a policy $\pi$ such that all nodes in the local-dynamics graph are visited equally often.

Given a policy $\pi$, a state $v \in V$ and an action $a \in \mathcal{A}$, we denote $p_{\pi,t}(v)$ and $p_{\pi,t}(v, a)$ the $t$-steps state distribution and state-action distribution respectively. Formally, $p_{\pi,t}(v) = \mathbb{P}_\pi(v_t = v)$ and $p_{\pi,t}(v, a) = \mathbb{P}_\pi(v_t = v, a_t = a)$.

Ideally we would like each $t$-step state distribution to be uniform. However, depending on the exact local-dynamics graph this may or may not be possible. Instead, following the principle of maximum entropy (Jaynes, 1957), we frame the objective of balancing the state distribution at step $t$ as maximizing $\mathcal{H}([p_{\pi,t}(v_0), p_{\pi,t}(v_1), \dots, p_{\pi,t}(v_{|V|-1})] = \mathcal{H}(p_{\pi,t})$, where $\mathcal{H}$ is the Shannon entropy. For a local-dynamics graph of depth $d \in \mathbb{N}$, we define our global objective as maximizing $J(\pi) = \tilde{J}(p_{\pi,1}, \dots, p_{\pi,d}) = \frac{1}{d}\sum_{t=1}^{d} \mathcal{H}(p_{\pi,t})$. Other global objectives are possible, for example optimizing entropy over only the final states, or some other weighted mixture. In practice, over simple experiments, we observed that changes in the entropy mixture hardly induced any variation in the computed policies and agent behavior.

Informally, our objective can be understood as maximizing state diversity locally, for every timestep smaller than $d$. For environments where additional priors about state interests are available, one could adapt the quantity $J$ to compute the entropy on a subset of the most interesting states, therefore biasing exploration toward promising areas.

We consider $\mathcal{K}$, the set of joint distributions $(p_0, p_1, \dots p_d)$ which verifies the following properties:

- $\forall t \leq d, p_t(v, a) \geq 0$
- $\forall v \in V, \sum_{a\in\mathcal{A}} p_0(v, a) = p_0(v) = \mathbb{1}_{v_0}(v)$
- $\forall t < d, \sum_{a\in\mathcal{A}} p_{t+1}(v, a) = \sum_{v'\in V, a\in\mathcal{A}} p_t(v', a)\mathbb{P}(v \mid v', a)$

We denote $D(\mathcal{A})$ the set of distributions over $\mathcal{A}$. From any $(p_0, p_1, \dots, p_d) \in \mathcal{K}$, it is possible to find a time-dependent policy $\pi : V \times \{0, \dots, d\} \to D(\mathcal{A})$ such that $p_0 = p_{\pi,0}, p_1 = p_{\pi,1}, \dots, p_d = p_{\pi,d}$, and for any policy $\pi$ we have $(p_{\pi,0}, p_{\pi,1}, \dots, p_{\pi,t}) \in \mathcal{K}$ (Puterman, 2014).

As the entropy $\mathcal{H}$ is concave, the function $\tilde{J}$ is a concave function over $\mathcal{K}$. Moreover, the constraints defining $\mathcal{K}$ are linear. Therefore,

$$\max_{(p_1,\dots,p_d)\in\mathcal{K}} \tilde{J}(p_1, \dots, p_n) \tag{6.4}$$

can be solved efficiently using any convex solver. In our implementation, we use CVXPY (Diamond and Boyd, 2016; Agrawal et al., 2018). Once $(p_1^\star, \dots, p_d^\star) = \arg\max_{\mathcal{K}} \tilde{J}$ is computed, we can immediately calculate a time-dependent policy $\pi^\star$ from such a distribution (Puterman, 2014) with:

$$\pi_t^\star(v, a) = \frac{p_t^\star(v, a)}{p_t^\star(v)} \tag{6.5}$$

As the local-dynamics graph $(V, E)$ is a DAG, the set of nodes $V_0, V_1, \dots, V_d$ which can be reached respectively at timesteps $t = 0, t = 1, \dots, t = d$ are disjoint. Therefore any time-dependent policy defined on $V$ can be framed as a stationary policy. Considering for example $\pi^\star$,

we can write $\pi^\star(v,\cdot) = \pi_0^\star(v,\cdot)$ if $v \in V_0$, $\pi^\star(v,\cdot) = \pi_1^\star(v,\cdot)$ if $v \in V_1, \dots,$ and $\pi^\star(v,\cdot) = \pi_d^\star(v,\cdot)$ if $v \in V_d$.

### 6.4.3 From local exploration to global policy

The optimal $\pi^\star$ determined in the previous section can then be used to guide exploration. With an $\varepsilon$-greedy policy, each step has a probability $\varepsilon$ of being an exploration step, where an action is sampled uniformly. Instead, we keep in memory the local-dynamics graph, and initialize the current state at $v = \Lambda$. Everytime an action $a$ is performed, $v$ is updated such that $v \leftarrow v.a$, and reinitialized to $\Lambda$ after a sequence of length $d$. At each exploration step, instead of sampling $a$ uniformly, EASEE samples $a$ according to the distribution $\pi^\star(v,\cdot)$. Pseudocode for this process can be found in Appendix B.1.5.

## 6.5 Results

For every experiments, additional details about environments and hyperparameters are given in Appendix B.2.

### 6.5.1 Pure exploration

To get a better understanding of EASEE, we consider two simple gridworld environments with different structures: CardinalGrid and RotationGrid. These environments are both $100 \times 100$ gridworlds, but with different action structures. In CardinalGrid, the agent can move one square in the four cardinal directions $(\rightarrow, \leftarrow, \uparrow, \downarrow)$, whereas in RotationGrid, the agent can move either forward one square $(\uparrow)$, or rotate $90°$ on the spot to the left $(\curvearrowleft)$ or to the right $(\curvearrowright)$. The agent starts in the middle of the grid and can explore for 100 timesteps, after which the environment is reset.

In CardinalGrid, we consider the 4 equivalence sets:

- $\{\rightarrow\leftarrow\sim\leftarrow\rightarrow\}$ (" $\rightarrow$ and $\leftarrow$ commute ")
- $\{\rightarrow\leftarrow\sim\leftarrow\rightarrow, \uparrow\downarrow\sim\downarrow\uparrow\}$ ("all actions commute")
- $\{\rightarrow\leftarrow\sim\leftarrow\rightarrow, \uparrow\downarrow\sim\downarrow\uparrow, \rightarrow\leftarrow\sim \Lambda\}$ ("all actions commute and $\rightarrow\leftarrow\sim \Lambda$")
- $\{\rightarrow\leftarrow\sim\leftarrow\rightarrow, \uparrow\downarrow\sim\downarrow\uparrow, \rightarrow\leftarrow\sim \Lambda, \uparrow\downarrow\sim \Lambda\}$ (" all actions commute and $\rightarrow\leftarrow\sim\uparrow\downarrow\sim \Lambda$ "),

while in RotationGrid, we consider the three equivalence sets:

- $\{\curvearrowright\curvearrowleft\sim \Lambda\}$

- $\{\curvearrowright\curvearrowleft\sim \Lambda, \curvearrowleft\curvearrowright\sim \Lambda\}$
- $\{\curvearrowright\curvearrowleft\sim \Lambda, \curvearrowleft\curvearrowright\sim \Lambda, \curvearrowright\curvearrowright\sim\curvearrowleft\curvearrowleft\}$.

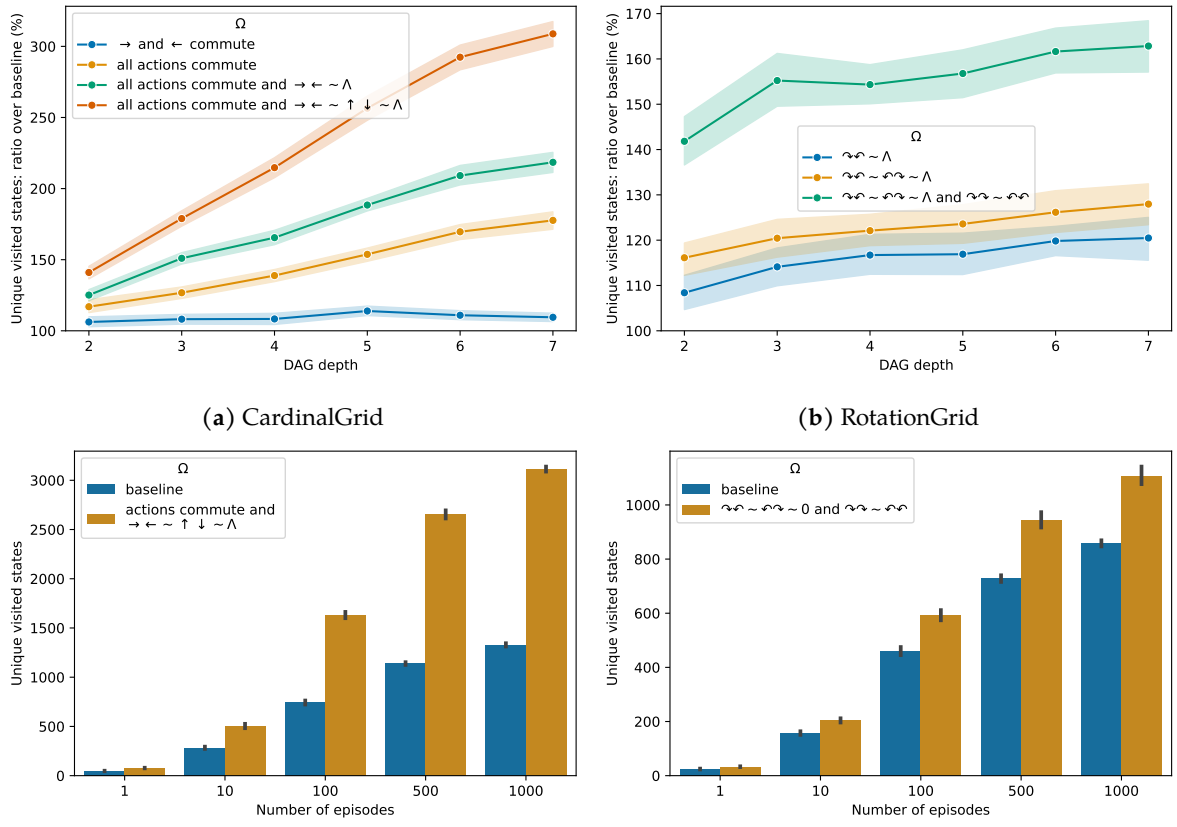
Fig. 6.3 shows the benefits of exploiting the structure of the action space for exploration. Figures 6.3a, 6.3b show the ratio of the number of unique states visited using EASEE over a standard uniform exploration policy. For both environments, a greater equivalence set leads to a more efficient exploration. In the environmentCardinalGrid for example, for a fixed depth of 6, adding the information that $\rightarrow$ and $\leftarrow$ commute and that every actions commute allow to reach respectively 10% and 60% more states in 100 episodes. Furthermore, extra equivalences encoding that $\rightarrow$ is the inverse of $\leftarrow$, and $\uparrow$ the inverse of $\downarrow$ increase the number of new states encountered threefold. It can also be seen that deeper graphs provide better exploration, which is expected: using deeper graphs results in exploiting equivalence priors over longer action sequences.

Figures 6.3c, 6.3d show the number of unique states visited with respect to the total number of episodes of exploration. We see that EASEE benefits exploration in all configurations considered: it allows the agent to visit more states within a single trajectory, and as well as across a thousand. It gives insight about the sample-efficiency gain which can be achieved using EASEE over a standard random policy. In the CardinalGrid setting, EASEE visits more unique states over 100 episodes than uniform exploration over 1000.

### 6.5.2 Minigrid

The Minimalistic Gridworld Environment (MiniGrid) is a suite of environments that test diverse capabilities in RL agents (Chevalier-Boisvert, Willems, and Pal, 2018). We evaluated the influence of adding EASEE to Q-learning on the DoorKey task. The environment is a gridworld split into two rooms separated by a locked door. The agent must collect a key to get to the objective in the other room.The dynamics of the environment are those of RotationGrid with two extra actions: the agent may `PICKUP` the key when facing it and `OPEN` the door when carrying the key. The EASEE version of the Q-learning assumes the following action sequence equivalences:

$$\curvearrowright\curvearrowleft \sim \Lambda$$
$$\curvearrowleft\curvearrowright \sim \Lambda$$
$$\curvearrowleft\curvearrowleft \sim \curvearrowright\curvearrowright$$
$$\texttt{OPEN} \sim \texttt{OPEN} \cdot \texttt{OPEN}$$
$$\texttt{PICKUP} \sim \texttt{PICKUP} \cdot \texttt{PICKUP}$$




(**a**) CardinalGrid (**b**) RotationGrid

(**c**) CardinalGrid (**d**) RotationGrid

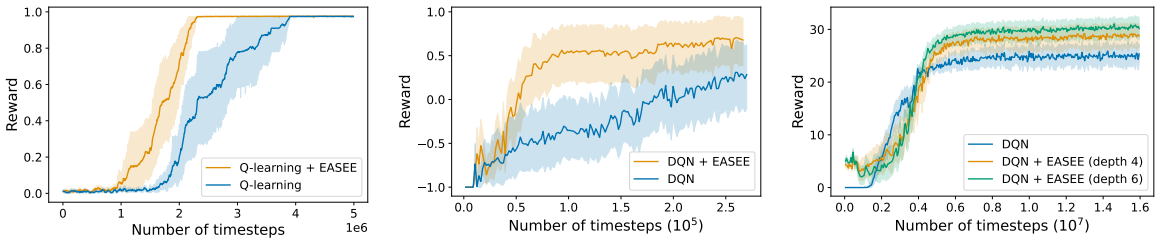
**Figure 6.3 – (a, b):** Ratio of the number of unique visited states during 100 episodes following EASEE over standard $\varepsilon$-greedy policy, for different equivalence sets and depths in the environments CardinalGrid and RotationGrid respectively. **(c, d):** Number of unique visited states according to the number of episodes for EASEE with a fixed depth of 4 compared to standard $\varepsilon$-greedy policy.

The reward over this training is presented in Figure 6.4a. Using a depth of 6, the EASEE augmented version outperforms classic Q-learning.

### 6.5.3 Catcher

We test EASEE on a game of Catcher, where the agent must catch a ball falling vertically with a paddle that can move left and right. It receives a reward of $+1$ when the ball is caught and $-1$ when it is missed. The prior we incorporate into the exploration is that the actions commute *i.e.* $\leftarrow\rightarrow\sim\rightarrow\leftarrow$. For faster learning we restrict each episode to a single ball drop, with the agent starting in the middle of the environment.

We choose a depth of 30 for EASEE. This is also the length of a single episode. The mean reward over training is plotted in Figure 6.4b.




(**a**) DoorKey (15 random seeds) (**b**) Catcher (20 random seeds) (**c**) Freeway (20 random seeds)

**Figure 6.4** – Mean reward over training with 95% confidence intervals.

### 6.5.4 Freeway

We test our method on the Atari 2600 game Freeway (M. G. Bellemare et al., 2013). The agent has to cross a road with multiple lanes without getting hit by the cars, and receives a reward when it reaches safety on the other side. The action space is composed of 3 actions : moving forward of 1 lane ($\uparrow$), moving backward of 1 lane ($\downarrow$), and passing ($-$). As cars arrive randomly, it is not easy to find priors on action equivalences in this environment. Since passing and moving backwards can sometimes be useful to avoid cars we cannot forbid these actions. However, we have prior knowledge that performing these two actions does not lead to visiting new lanes. We restrict the use of these actions with $\Omega = \{\downarrow\sim\downarrow\downarrow, -\downarrow\sim\downarrow-\}$, which has the effect of removing every node which is reached by chaining two $\downarrow$ actions without moving forward, and compute the exploration policy on the remaining nodes. Results can be seen in Fig. 6.4c. Interestingly, incorporating such a prior does not lead to better sample-efficiency, as in the previous environments, but to a better final policy.

## Chapter Conclusion

Implementers of reinforcement learning agents can often provide insights about the environment, despite not knowing its precise dynamics or optimal policy. In this chapter, we argue that some of these insights can be efficiently represented using the notion of action-sequence equivalence, which we formalize. We propose a method to incorporate such priors in classic Q-learning algorithms and demonstrate empirically its ability to improve sample efficiency and performance. More precisely, our approach can be divided into two steps: first, the construction of a graph representing the local dynamics, and then the resolution of a convex optimization problem aiming to balance node visitation. We show that incorporating such prior knowledge can replace standard $\varepsilon$-greedy and improve at little cost RL algorithms, especially in environments with a rich combinatorial structure, like DoorKey and Catcher.

Our results indicate that this approach may have potential beyond environments with perfectly known structures, such as Freeway. In such cases, the exploration policy we determine may not be optimal, but it can still be much closer to optimality than uniformly sampling actions. This suggests that our approach could be useful beyond problems with a strict combinatorial structure.

# Part III

# Dealing with Uncertainty

# Chapter 7

# Population-Based Reinforcement Learning for Combinatorial Optimization

After exploring various problem structures in the previous part and venturing beyond the boundaries of CO, we return to tackling canonical COPs with construction methods. Here, we specifically focus on the uncertainty at the action level, where taking optimal decisions at each step involves extensive computations that cannot be reliably solved by a neural policy. Hence, leading approaches typically incorporate additional search strategies, ranging from stochastic sampling and beam search to explicit fine-tuning, to address this action uncertainty. In this chapter, we propose a novel approach that leverages the benefits of learning a population of complementary policies, which can be simultaneously rolled out during inference. We introduce Poppy, a simple training procedure for populations that induces an unsupervised specialization targeted solely at maximizing the performance of the population, without relying on predefined or hand-crafted notions of diversity. Our results demonstrate that Poppy produces a set of complementary policies and achieves state-of-the-art RL performance on three well-known NP-hard problems: the TSP, the capacitated vehicle routing (CVRP), and 0-1 knapsack (KP) problems. [1]

## Contents



[1] This chapter is based on a preprint (Grinsztajn, Furelos-Blanco, and Barrett, 2022), under review at *ICML 2023*.

## 7.1 Introduction

As the search space of feasible COP solutions typically grows exponentially with the problem size, exact solvers can be challenging to scale; hence, CO problems are often also tackled with handcrafted heuristics using expert knowledge. Whilst a diversity of ML-based heuristics have been proposed, reinforcement learning (RL; Sutton and Barto, 2018) is a promising paradigm, for reasons outlined in Chapter 2. Algorithmic improvements to RL-based CO solvers, coupled with low inference cost, and the fact that they are by design targeted at specific problem distributions, have progressively narrowed the gap with traditional solvers.

We recall that RL methods frame CO as sequential decision-making problems, and can be divided into two families (Mazyavkina et al., 2021). First, *improvement methods* start from a feasible solution and iteratively improve it through small modifications (actions). However, such incremental search cannot quickly access very different solutions, and requires handcrafted procedures to define a sensible action space. Second, *construction methods* incrementally build a solution by selecting one element at a time. In practice, it is often unrealistic for a learned heuristic to solve NP-hard problems in a single shot, therefore these methods are typically combined with search strategies, such as stochastic sampling or beam search. However, just as improvement methods are biased by the initial starting solution, construction methods are biased by the single underlying policy. Thus, a balance must be struck between the exploitation of the learned policy (which may be ill-suited for a given problem instance) and the exploration of different solutions (where the extreme case of a purely random policy will likely be highly inefficient).

In this work, we propose Poppy, a construction method that uses a *population* of agents with suitably diverse policies to improve the exploration of the solution space of hard CO problems. Whereas a single agent aims to perform well across the entire problem distribution, and thus has to make compromises, a population can learn a set of heuristics such that only one of these has to be performant on any given problem instance. However, realizing this intuition presents several challenges: (i) naïvely training a population of agents is expensive and challenging to scale, (ii) the trained population should have complementary policies that propose different solutions, and (iii) the training approach should not impose any handcrafted notion of diversity within the set of policies given the absence of clear behavioral markers aligned with performance for typical CO problems.

Challenge (i) can be addressed by sharing a large fraction of the computations across the population, specializing only lightweight policy heads to realize the diversity of agents. Moreover, this can be done on top of a pre-trained model, which we clone to produce the population. Challenges (ii) and (iii) are jointly achieved by introducing a RL objective aimed at specializing agents on distinct subsets of the problem distribution. Concretely, we derive a policy gradient method for the population-level objective, which corresponds to training only

the agent which performs best on each problem. This is intuitively justified as the performance of the population on a given problem is not improved by training an agent on an instance where another agent already has better performance. Strikingly, we find that judicious application of this conceptually simple objective gives rise to a population where the diversity of policies is obtained without explicit supervision (and hence is applicable across a range of problems without modification) and essential for strong performance.

Our contributions are summarized as follows:

1. We motivate the use of populations for CO problems as an efficient way to explore environments that are not reliably solved by single-shot inference.
2. We derive a new training objective and present a practical training procedure that encourages performance-driven diversity (*i.e.* effective diversity without the use of explicit behavioral markers or other external supervision).
3. We evaluate Poppy on three CO problems: TSP, CVRP, and 0-1 knapsack (KP). In these three problems, Poppy consistently outperforms all other RL-based approaches.

## 7.2 Additional related work

**ML for combinatorial optimization** The first attempt to solve TSP with neural networks is due to Hopfield and Tank (1985), which only scaled up to 30 cities. Recent developments of bespoke neural architectures (Vinyals, Fortunato, and Jaitly, 2015; Vaswani et al., 2017) and performant hardware have made ML approaches increasingly efficient. Indeed, several architectures have been used to address CO problems, such as graph neural networks (Dai et al., 2017), recurrent neural networks (Nazari et al., 2018), and attention mechanisms (Deudon et al., 2018). In this chapter, we use an encoder-decoder architecture that draws from that proposed by Kool, Hoof, and Welling (2019). The costly encoder is run once per problem instance, and the resulting embeddings are fed to a small decoder iteratively rolled out to get the whole trajectory, which enables efficient inference. This approach was furthered by Kwon et al. (2020), who leveraged the underlying symmetries of typical CO problems (*e.g.* of starting positions and rotations) to realize improved training and inference performance using instance augmentations. Kim, J. Park, and j. k. j. (2021) also draw on Kool, Hoof, and Welling and use a hierarchical strategy where a seeder proposes solution candidates, which are refined bit-by-bit by a reviser. Closer to our work, Xin et al. (2021) trains multiple policies using a shared encoder and separate decoders. Whilst this work (MDAM) shares our architecture and goal of training a population, our approach for enforcing diversity differs substantially. MDAM explicitly trades off performance with diversity by jointly optimizing policies and their KL divergence. Moreover, as computing the KL divergence for the whole trajectory is intractable,

MDAM is restricted to only using it to drive diversity at the first timestep. In contrast, Poppy drives diversity by maximizing population-level performance (*i.e.* without any explicit diversity metric), uses the whole trajectory and scales better with the population size (we have used up to 32 agents instead of only 5).

Additionally, ML approaches usually rely on mechanisms to generate multiple candidate solutions (Mazyavkina et al., 2021). As seen in Chapter 2, one such mechanism consists in using improvement methods on an initial solution. However, these approaches have two limitations: they are environment-specific, and the search procedure is inherently biased by the initial solution.

An alternative exploration mechanism is to generate a diverse set of trajectories by stochastically sampling a learned policy, potentially with additional beam search (Joshi, Laurent, and Bresson, 2019), Monte Carlo tree search (Fu, Qiu, and Zha, 2021), dynamic programming (Kool, Hoof, Gromicho, et al., 2021) or active search (Hottung, Kwon, and Tierney, 2022). However, intuitively, the generated solutions tend to remain close to the underlying deterministic policy, implying that the benefits of additional sampled candidates diminish quickly.

**Population-based RL** Populations have already been used in RL to learn diverse behaviors. In a different context, Gupta et al. (2018), Eysenbach et al. (2019), Hartikainen et al. (2020) and Pong et al. (2020) use a single policy conditioned on a set of goals as an implicit population for unsupervised skill discovery. Closer to our approach, another line of work revolves around explicitly storing a set of distinct policy parameters. Hong et al. (2018), Doan et al. (2020), Jung, G. Park, and Sung (2020) and Parker-Holder et al. (2020) use a population to achieve a better coverage of the policy space. However, they enforce explicit attraction-repulsion mechanisms, which is a major difference with respect to our approach where diversity is a pure byproduct of performance optimization.

Our method is also related to approaches combining RL with evolutionary algorithms (EA; Khadka and Tumer, 2018; Khadka, Majumdar, et al., 2019; Pourchot and Sigaud, 2019), which benefit from the sample-efficient RL policy updates while enjoying evolutionary population-level exploration. However, the population is a means to learn a unique strong policy, whereas Poppy learns a set of complementary strategies. More closely related, Quality-Diversity (QD; Pugh, Soros, and Stanley, 2016; Cully and Demiris, 2018) is a popular EA framework that maintains a portfolio of diverse policies. Pierrot et al. (2022) has recently combined RL with a QD algorithm, Map-Elites (Mouret and Clune, 2015); unlike Poppy, QD methods rely on handcrafted behavioral markers, which is not easily amenable to the CO context.

One of the drawbacks of population-based RL is its expensive cost. However, recent approaches have shown that modern hardware as well as targeted frameworks enable efficient

vectorized population training (Flajolet et al., 2022), opening the door to a wider range of applications.

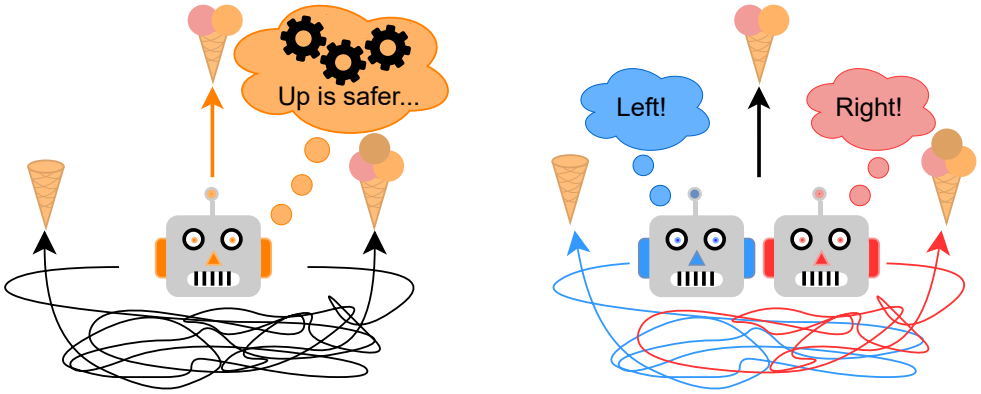



**Figure 7.1** – In this environment, the upward path always leads to a medium reward, while the left and right paths are intricate such that either one may lead to a low reward or high reward with equal probability. **Left**: An agent trained to maximize a sum of rewards converges to taking the safe upward road since it does not have enough information to act optimally. **Right**: A 2-agent population can always take the left and right paths and thus get the largest reward.

## 7.3 Method

### 7.3.1 Background and motivation

**RL Formulation** A CO problem instance $\rho$ sampled from some distribution $\mathcal{D}$ consists of a discrete set of $N$ variables (*e.g.* city locations in TSP). We model a CO problem as a Markov decision process (MDP) defined by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $T$, and a reward function $R$. A state is a trajectory through the problem instance $\tau_t = (x_1, \dots, x_t) \in \mathcal{S}$ where $x_i \in \rho$, and thus consists of an ordered list of variables (not necessarily of length $N$). An action, $a \in \mathcal{A} \subseteq \rho$, consists of choosing the next variable to add; thus, given state $\tau_t = (x_1, \dots, x_t)$ and action $a$, the next state is $\tau_{t+1} = T(\tau_t, a) = (x_1, \dots, x_t, a)$. Let $\mathcal{S}^* \subseteq \mathcal{S}$ be the set of *solutions*; that is, states that comply with the problem's constraints (*e.g.*, a sequence of cities such that each city is visited once and ends with the starting city in TSP). The reward function $R : \mathcal{S}^* \to \mathbb{R}$ maps solutions into scalars. We assume the reward is maximized by the optimal solution (*e.g.* $R$ returns the negative tour length in TSP).

A *policy* $\pi_\theta$ parameterized by $\theta$ can be used to generate solutions for any instance $\rho \sim \mathcal{D}$ by iteratively sampling the next action $a \in \mathcal{A}$ according to the probability distribution $\pi_\theta(\cdot \mid \rho, \tau_t)$. We learn $\pi_\theta$ using REINFORCE (Williams, 1992b). This method aims at maximizing the RL objective $J(\theta) \doteq \mathbb{E}_{\rho\sim\mathcal{D}}\, \mathbb{E}_{\tau\sim\pi_\theta,\rho}\, R(\tau)$ by adjusting $\theta$ such that good trajectories are more likely to be sampled in the future. Formally, the policy parameters $\theta$ are updated by gradient ascent

using $\nabla_\theta J(\theta) = \mathbb{E}_{\rho\sim\mathcal{D}}\,\mathbb{E}_{\tau\sim\pi_\theta,\rho}(R(\tau) - b_\rho)\nabla_\theta \log(p_\theta(\tau))$, where $p_\theta(\tau) = \prod_t \pi_\theta(a_{t+1} \mid \rho, \tau_t)$ and $b_\rho$ is a baseline. The gradient of the objective, $\nabla_\theta J$, can be estimated empirically using Monte Carlo simulations.

**Motivating example** We argue for the benefits of training a population using the example in Figure 7.1. In this environment, there are three actions: **Left**, **Right**, and **Up**. **Up** leads to a medium reward, while **Left**/**Right** lead to low/high or high/low rewards (the configuration is determined with equal probability at the start of each episode). Crucially, the left and right paths are intricate, so the agent cannot easily infer from its observation which one leads to a higher reward. Then, the best strategy for a single agent is to always go **Up**, as the guaranteed medium reward (2 scoops) is higher than the expected reward of guessing left or right (1.5 scoops). In contrast, two agents in a population can go in opposite directions and always find the maximum reward. There are two striking observations: (i) the agents do not need to perform optimally for the population performance to be optimal (one agent gets the maximum reward), and (ii) the average performance is worse than in the single-agent case.

Specifically, the discussed phenomenon can occur when (i) some optimal actions are too difficult to infer from observations and (ii) choices are irreversible (*i.e.* it is not possible to recover from a suboptimal decision). These conditions usually hold when solving hard CO problems. In these situations, as shown above, maximizing the performance of a population will require agents to specialize and likely yield better results than in the single-agent case.

### 7.3.2 Poppy

We present the components of Poppy: a RL objective encouraging agent specialization, and an efficient training procedure taking advantage of a pre-trained policy.

**Population-based objective** At inference, reinforcement learning methods usually sample several candidates to find better solutions. This process, though, is not anticipated during training, which optimizes the 1-shot performance with the usual RL objective $J(\theta) = \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau\sim\pi_\theta,\rho}R(\tau)$ previously presented in Section 7.3.1. Intuitively, given $K$ trials, we would like to find the best set of policies $\{\pi_1, \dots, \pi_K\}$ to rollout once on a given problem. This gives the following population objective:

$$J_{\text{pop}}(\theta_1, \dots, \theta_K) \doteq \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1\sim\pi_{\theta_1},\dots,\tau_K\sim\pi_{\theta_K}} \max\left[R(\tau_1), \dots, R(\tau_K)\right],$$

where each trajectory $\tau_i$ is sampled according to the policy $\pi_{\theta_i}$. Maximizing $J_{\text{pop}}$ leads to finding the best set of $K$ agents which can be rolled out in parallel for any problem.

**Theorem** (Policy gradient for populations)**.** *The gradient of the population objective is:*

$$\nabla J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) = \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1\sim\pi_{\theta_1},\dots,\tau_K\sim\pi_{\theta_K}}\big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big)\nabla \log p_{\theta_{i^*}}(\tau_{i^*}), \tag{7.1}$$

*where:* $i^* = \arg\max_{i\in\{1,\dots,K\}}\big[R(\tau_i)\big]$ (*index of the agent that got the highest reward*) *and* $i^{**} = \arg\max_{i\neq i^*}\big[R(\tau_i)\big]$ (*index of the agent that got the second highest reward*).

The proof is provided in Appendix C.2.1. Remarkably, it corresponds to rolling out every agent and only training the one that got the highest reward on any problem. This formulation applies across various problems and directly optimizes for population-level performance without explicit supervision or handcrafted behavioral markers.

Optimizing the presented objective does not provide any strict diversity guarantee. However, note that diversity maximizes our objective in the highly probable case that, within the bounds of finite capacity and training, a single agent does not perform optimally on all subsets of the training distribution. Therefore, intuitively (and, as we will show, practically) diversity emerges over training in the pursuit of maximizing the objective.

**Algorithm 7.1:** Poppy training

**Inputs:** problem distribution $\mathcal{D}$, number of agents $K$, batch size $B$, number of training steps $H$, pre-trained parameters $\theta$.
$\theta_1, \theta_2, \dots, \theta_K \leftarrow \texttt{CLONE}(\theta)$;
`/* Clone the pre-trained agent` $K$ `times. */`
**for** *step 1 to* $H$ **do**
  $\rho_i \leftarrow \texttt{Sample}(\mathcal{D})\ \forall i \in 1, \dots, B$ ;
  $\tau_i^k \leftarrow \texttt{Rollout}(\rho_i, \theta_k)\ \forall i \in 1, \dots, B, \forall k \in 1, \dots, K$ ;
  $k_i^* \leftarrow \arg\max_{k\leq K} R(\tau_i^k)\ \forall i \in 1, \dots, B$ ;
  `/* Select the best agent for each problem` $\rho_i$`. */`
  $\nabla L(\theta_1, \theta_2, \dots, \theta_K) \leftarrow \frac{1}{B}\sum_{i\leq B}\texttt{REINFORCE}(\tau_i^{k_i^*})$ ;
  `/* Propagate the gradients through these only. */`
  $(\theta_1, \theta_2, \dots, \theta_K) \leftarrow (\theta_1, \theta_2, \dots, \theta_K) - \alpha\nabla L(\theta_1, \theta_2, \dots, \theta_K)$ ;

**Training procedure** The training procedure consists of two phases:

1. We train (or reuse) a single agent using an architecture suitable for solving the CO problem at hand. Here, we adopt the training process described in Kwon et al. (2020), which has been shown to be efficient for TSP, CVRP and KP.

2. The agent trained in Phase 1 is cloned $K$ times to form a $K$-agent population. The population is trained as described in Algorithm 7.1: only the best agent is trained on any problem. Agents implicitly specialize in different types of problem instances during this phase.

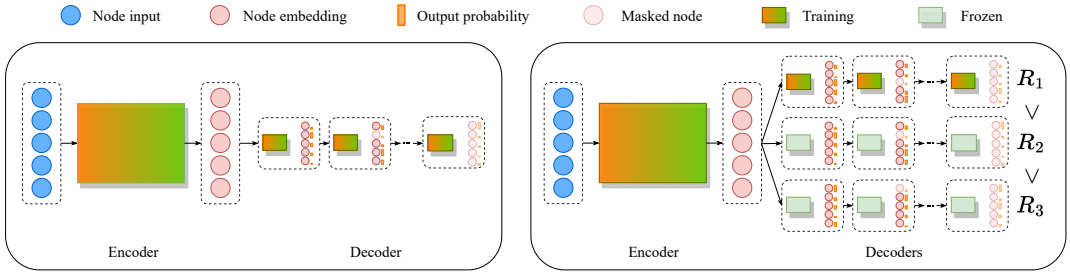



**Figure 7.2** – Phases of the training process with a model using static instance embeddings. **Left** (**Phase 1**): the encoder and the decoder are trained from scratch. **Right** (**Phase 2**): the decoder is cloned $K$ times, and the whole model is trained using the Poppy training objective (*i.e.* the gradient is only propagated through the decoder that yields the highest reward).

Phase 1 enables training the model without the computational overhead of a population. Moreover, we informally note that applying the Poppy objective directly to a population of untrained agents can be unstable. Randomly initialized agents are often ill-distributed, hence a single (or few) agent(s) dominate the performance across all instances. In this case, only the initially dominating agents receive a training signal, further widening the performance gap. Whilst directly training a population of untrained agents for population-level performance may be achievable with suitable modifications, we instead opt for the described pre-training approach as it is efficient and stable.

**Architecture** To reduce the memory footprint of the population, some of the model parameters can be shared. Here, the architecture for TSP, CVRP, and KP uses the attention model by Kool, Hoof, and Welling (2019), which decomposes the policy model into two parts: (i) a large encoder $h_\psi$ that takes an instance $\rho$ as input and outputs embeddings $\omega$ for each of the variables in $\rho$, and (ii) a smaller decoder $q_\phi$ that takes the embeddings $\omega$ and a partial trajectory $\tau_t$ as input and outputs the probabilities of each possible action. Figure 7.2 illustrates the training phases of such a model.

We exploit this framework to build a population of $K$ agents. The encoder $h_\psi$ is shared as a common backbone for the whole population, whereas the decoders $q_{\phi_1}, q_{\phi_2}, \ldots, q_{\phi_K}$ are unique to each agent. This is motivated by (i) the encoder learning general representations useful for all agents, and (ii) reducing the overhead of training a population and keeping the total number of parameters low. A discussion on the model sizes is provided in Appendix C.1.1.

## 7.4 Experiments

We evaluate Poppy on three CO problems: TSP, CVRP, and KP. To emphasize its generality, we use the same hyperparameters for each problem, taken from Kwon et al. (2020). We run Poppy for populations of 4, 8, 16, or 32 agents, which exhibit various time-performance tradeoffs.

**Training** One training step corresponds to computing policy gradients over the same batch of 64 randomly generated instances for each agent in the population. Training time varies with problem complexity and training phase. For instance, in TSP with 100 cities, Phase 1 takes 4.5M steps (5 days), whereas Phase 2 takes 400k training steps and lasts 1-4 days depending on the population size. Our JAX-based implementation uses environments from the Jumanji suite (Bonnet et al., 2022). All experiments were run on a v3-8 TPU.

**Inference** We greedily roll out each agent in the population, and use the augmentations proposed by Kwon et al. (2020) for TSP and CVRP. To give a sense of the performance of Poppy with a larger time budget, we additionally introduce a simple sampling procedure in Appendix C.3, which we show to be competitive with the active search method described by Hottung, Kwon, and Tierney (2022).

**Starting points** Following Kwon et al. (2020), we generate multiple solutions for each instance $\rho$ by considering a set of $P \in [1, N]$ *starting points*, where $N$ is the number of instance variables. For example, a starting point in a TSP instance could be any of its cities. Therefore, across the different training phases, agents generate trajectories for (instance, starting point) pairs. The average reward across starting points is used as the REINFORCE baseline. We consider each (instance, starting point) pair as a separate problem and, thus, train the best agent for each of them. We refer the reader to Appendix C.1.2 for details.

**Baselines** We compare Poppy against exact solvers, heuristics, and state-of-the-art ML methods. Some baseline performances taken from Fu, Qiu, and Zha (2021), Xin et al. (2021) and Hottung, Kwon, and Tierney (2022) were obtained with different hardware (Nvidia GTX 1080 Ti, RTX 2080 Ti, and Tesla V100 GPUs, respectively) and framework (PyTorch); thus, for fairness, we mark these times with $*$ in our tables. As a comparison guideline, we informally note that these GPU inference times should be approximately divided by 2 to get the converted TPU time.

### 7.4.1 Traveling Salesman Problem (TSP)

Given a set of $n$ cities, the goal in TSP is to visit every city and come back to the starting city while minimizing the total traveled distance.

**Setup** We use the architecture used by Kool, Hoof, and Welling (2019) and Kwon et al. (2020) with slight modifications (see Appendix C.4). The training is done on instances of size $n = 100$. The testing instances are taken from Kool, Hoof, and Welling (2019) for $n = 100$, and from Hottung, Kwon, and Tierney (2022) for $n \in \{125, 150\}$. We compare Poppy to (i) the specialized supervised learning (SL) methods GCN-BS (Joshi, Laurent, and Bresson, 2019), CVAE-Opt (Hottung, Bhandari, and Tierney, 2021), and DPDP (Kool, Hoof, Gromicho, et al., 2021); (ii) the RL methods MDAM (Xin et al., 2021) with and without beam search, Att-GCRN+Monte-Carlo Tree Search (MCTS) (Fu, Qiu, and Zha, 2021), the improvement methods 2-Opt-DL (O. da Costa et al., 2020), LIH (Y. Wu et al., 2021) and, finally, POMO with greedy rollouts, POMO with 16 stochastic rollouts (to match Poppy 16 runtime), and POMO with an ensemble of 16 decoder heads trained in parallel (*i.e.* using the architecture of Poppy's Phase 2 but trained without $J_{\text{poppy}}$).

**Results** Table 7.1 displays the average tour length, the optimality gap, and the total runtime for each test set. The best algorithm remains Concorde as it is a highly specialized TSP solver. Remarkably, Poppy 16 with greedy rollouts reaches the best performance across every category in just a few minutes, except for one case where DPDP performs better; however, DPDP tackles routing problems specifically, requires pre-solved instances, and makes use of expert knowledge. Compared to DPDP, Poppy improves 0-shot performance, suggesting that it is more robust to distribution shifts. Att-GCRN+MCTS is known for being scalable to larger TSP instances than TSP100; however, it is outperformed by Poppy, showing that it trades off performance for scale. Finally, we emphasize that specialization is crucial to achieving state-of-the-art performance: Poppy 16 outperforms POMO 16 (ensemble), which also trains 16 agents in parallel but without the $J_{\text{poppy}}$ objective (*i.e.* without specializing to serve as an ablation of our proposed objective).

**Analysis** Figure 7.3 helps understand the resulting behavior from using Poppy. The left plot shows that whilst the population-level performance improves with population size, the average performance of a random agent from the population on a random instance gets worse, which can be interpreted as specialization. Interestingly, it shows a stronger specialization for larger populations, which $J_{\text{poppy}}$ appears to balance. To further analyze this phenomenon, we display in the rightmost figure the number of agents reaching the best performance among the population. We observe that the best performance is reached by a single agent in more

**Table 7.1** – TSP results.

| | | **Inference** (10k instances) | | | **0-shot** (1k instances) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $n = 100$ | | | $n = 125$ | | | $n = 150$ | | |
| | Method | Obj. | Gap | Time | Obj. | Gap | Time | Obj. | Gap | Time |
| | Concorde | 7.765 | 0.000% | 82M | 8.583 | 0.000% | 12M | 9.346 | 0.000% | 17M |
| | LKH3 | 7.765 | 0.000% | 8H | 8.583 | 0.000% | 73M | 9.346 | 0.000% | 99M |
| SL | GCN-BS | 7.87 | 1.39% | 40M* | - | - | - | - | - | - |
| SL | CVAE-Opt | - | 0.343% | 6D* | 8.646 | 0.736% | 21H* | 9.482 | 1.45% | 30H* |
| SL | DPDP | 7.765 | 0.004% | 2H* | 8.589 | 0.070% | 31M* | 9.434 | 0.94% | 44M* |
| RL | MDAM (greedy) | 7.93 | 2.19% | 36S* | - | - | - | - | - | - |
| RL | MDAM (beam search) | 7.79 | 0.38% | 44M* | - | - | - | - | - | - |
| RL | Att-GCRN+MCTS | - | 0.037% | 15M* | - | - | - | - | - | - |
| RL | 2-Opt-DL | 7.83 | 0.87% | 41M* | - | - | - | - | - | - |
| RL | LIH | 7.87 | 1.42% | 2H* | - | - | - | - | - | - |
| RL | POMO | 7.774 | 0.13% | 37S | 8.605 | 0.26% | 6S | 9.393 | 0.50% | 10S |
| RL | POMO (16 samples) | 7.770 | 0.073% | 9M | 8.597 | 0.16% | 1M | 9.385 | 0.41% | 2M |
| RL | POMO 16 (ensemble) | 7.773 | 0.10% | 9M | 8.603 | 0.23% | 1M | 9.393 | 0.50% | 2M |
| RL | **Poppy 4** | 7.767 | 0.029% | 2M | 8.590 | 0.079% | 23S | 9.364 | 0.19% | 38S |
| RL | **Poppy 8** | 7.766 | 0.015% | 5M | 8.587 | 0.046% | 45S | 9.360 | 0.14% | 1M |
| RL | **Poppy 16** | **7.765** | **0.008%** | 9M | **8.585** | **0.029%** | 1M | **9.356** | **0.10%** | 2M |

than 25% of the test instances, and in almost 50% by three agents or less, which shows that they have diverse behaviors. Additional analyses are made in Appendix C.4.1.

### 7.4.2 Capacitated Vehicle Routing Problem (CVRP)

Given a vehicle with limited capacity departing from a depot node and a set of $n$ nodes with different demands, the goal is to find an optimal set of routes such that each node (except for the depot) is visited exactly once and has its demand covered. The vehicle's capacity diminishes by the demand of the visited node (which must be fully covered) and it is restored when the depot is visited.

**Setup** We use the test instances used by Kwon et al. (2020) for $n = 100$, and the sets from Hottung, Kwon, and Tierney (2022) to evaluate generalization to larger problems. We evaluate Poppy with populations of 4, 8, and 32 agents. We compare Poppy to the heuristic solver LKH3 (Helsgaun, 2017), taken as a reference to compute the gaps although its performances are not optimal. We also report results for the supervised ML methods CVAE-Opt (Hottung, Bhandari, and Tierney, 2021) and DPDP (Kool, Hoof, Gromicho, et al., 2021), and RL methods MDAM (Xin et al., 2021), LIH (Y. Wu et al., 2021), NeuRewriter (X. Chen and Tian, 2019), NLNS

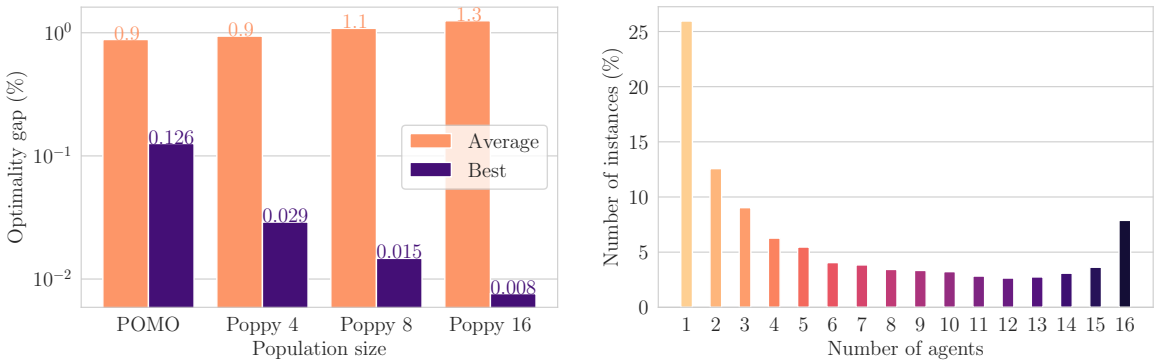



**Figure 7.3** – Analysis of Poppy on TSP100. **Left:** With $J_{\text{poppy}}$, the average performance gets worse as the population size increases, but the population-level performance improves. **Right:** Proportion of test instances where any number of Poppy 16 agents reaches the exact same best solution. The best performance is reached by only a single agent in 26% of the cases.

(Hottung and Tierney, 2020), and POMO (Kwon et al., 2020). As for TSP, we evaluate POMO in three settings: with greedy rollouts, 32 stochastic samples to match the runtime of our largest population, and an ensemble of 32 decoders trained in parallel.

**Results** Table 7.2 shows Poppy has the best performances among RL approaches, *e.g.* Poppy 32 has the same runtime as POMO with 32 stochastic rollouts while dividing by 2 the optimal gap for CVRP100.[2] Interestingly, this ratio increases on the generalization instance sets with $n = 125$ and $n = 150$, suggesting that Poppy is more robust to distributional shift. DPDP performs best among all ML-based approaches. However, it relies on a costly problem-specific beam search as well as pre-solved instances.

### 7.4.3 0-1 Knapsack (KP)

We evaluate Poppy on KP to demonstrate its applicability beyond routing problems. Given a set of $n$ items with specific weights and values and a bag of limited capacity, the goal is to determine which items should be added to the bag such that the total weight does not exceed the bag's capacity and the value is maximal.

**Setup** We use the setting employed by Kwon et al. (2020): an action corresponds to putting an item in the bag, which is filled iteratively until no more items can be added. We evaluate Poppy on 3 population sizes (4, 8, and 16) against the optimal solution based on dynamic programming, a greedy heuristic, and POMO with greedy rollouts, 16 stochastic samples, or an ensemble of 16 decoders.

[2] A fine-grained comparison between POMO with stochastic sampling and Poppy is in Appendix C.5.

**Table 7.2** – CVRP results.

| | | **Inference** (10k instances) $n = 100$ | | | **0-shot** (1k instances) $n = 125$ | | | **0-shot** (1k instances) $n = 150$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Method | | Obj. | Gap | Time | Obj. | Gap | Time | Obj. | Gap | Time |
| LKH3 | | 15.65 | 0.000% | 6D | 17.50 | 0.000% | 19H | 19.22 | 0.000% | 20H |
| SL | CVAE-Opt | - | 1.36% | 11D* | 17.87 | 2.08% | 36H* | 19.84 | 3.24% | 46H* |
| SL | DPDP | 15.63 | −0.13% | 23H* | 17.51 | 0.07% | 3H* | 19.31 | 0.48% | 5H* |
| RL | MDAM (greedy) | 16.40 | 4.86% | 45S* | - | - | - | - | - | - |
| RL | MDAM (beam search) | 15.99 | 2.23% | 53M* | - | - | - | - | - | - |
| RL | LIH | 16.03 | 2.47% | 5H* | - | - | - | - | - | - |
| RL | NeuRewriter | 16.10 | - | 66M* | - | - | - | - | - | - |
| RL | NLNS | 15.99 | 2.23% | 62M* | 18.07 | 3.23% | 9M* | 19.96 | 3.86% | 12M* |
| RL | POMO | 15.76 | 0.76% | 2M | 17.68 | 1.02% | <1M | 19.58 | 1.85% | 1M |
| RL | POMO (32 samples) | 15.70 | 0.32% | 43M | 17.59 | 0.50% | 8M | 19.48 | 1.35% | 12M |
| RL | POMO 32 (ensemble) | 15.72 | 0.49% | 43M | 17.63 | 0.72% | 8M | 19.49 | 1.37% | 12M |
| RL | **Poppy 4** | 15.72 | 0.46% | 5M | 17.62 | 0.64% | 2M | 19.48 | 1.33% | 2M |
| RL | **Poppy 8** | 15.70 | 0.32% | 11M | 17.59 | 0.49% | 3M | 19.45 | 1.19% | 5M |
| RL | **Poppy 32** | **15.67** | **0.13%** | 43M | **17.55** | **0.24%** | 8M | **19.39** | **0.86%** | 12M |

**Results** Table 7.3 shows that Poppy leads to improved performance with a population of 16 agents, dividing the optimality gap with respect to POMO by 45 and 12 on KP100 and KP200 respectively, and by 12 and 2 with respect to POMO with 16 stochastic samples for the exact same runtime. We emphasize that although these gaps are small, these differences are still significant: Poppy 16 is strictly better than POMO in 34.30% of the KP100 instances, and better in 99.95%.

## Chapter Conclusion

Poppy is a population-based RL method for CO problems. It uses a RL objective that incurs agent specialization with the purpose of maximizing population-level performance. Crucially, Poppy does not rely on handcrafted notions of diversity to enforce specialization. We show that Poppy achieves state-of-the-art performance on three popular NP-hard problems: TSP, CVRP, and KP.

This work opens the door to several directions for further investigation. Firstly, we have experimented on populations of at most 32 agents; therefore, it is unclear what the consequences of training larger populations are. Whilst even larger populations could reasonably be expected to provide stronger performance, achieving this may not be straightforward. Aside from the increased computational burden, we also hypothesize that the population performance

**Table 7.3** – KP results.

| | **Testing** (10k instances) $n = 100$ | | | **Testing** (1k instances) $n = 200$ | | |
|---|---|---|---|---|---|---|
| Method | Obj. | Gap | Time | Obj. | Gap | Time |
| Optimal | 40.437 | - | | 57.729 | - | |
| Greedy | 40.387 | 0.1250% | | 57.672 | 0.0986% | |
| POMO | 40.428 | 0.0224% | 8S | 57.718 | 0.0191% | 4S |
| POMO (16 samples) | 40.435 | 0.0060% | 2M | 57.727 | 0.0032% | 1M |
| POMO 16 (ensemble) | 40.429 | 0.021% | 2M | 57.719 | 0.0170% | 1M |
| **Poppy 4** | 40.434 | 0.0081% | 33S | 57.723 | 0.0099% | 16S |
| **Poppy 8** | 40.436 | 0.0032% | 1M | 57.726 | 0.0058% | 33S |
| **Poppy 16** | **40.437** | **0.0005%** | 2M | **57.728** | **0.0015%** | 1M |

could eventually collapse once no additional specialization niches can be found, leaving agents with null contributions behind. Exploring strategies to scale and prevent such collapses is an interesting direction for future work.

Secondly, our work has built on the current state-of-the-art RL for CO approaches in a single- or few-shot inference setting to demonstrate the remarkable efficacy of a population-based approach. However, there are other paradigms that we could consider. For example, active-search methods allow an increased number of solving attempts per problem and, in principle, such methods for inference-time adaption of the policy could be combined with an initially diverse population to further boost performance. Indeed, we investigate the performance of Poppy with a larger time budget in Appendix C.3 and find that Poppy combined with a simple sampling procedure and no fine-tuning already matches, or even surpasses, the state-of-the-art active search approach of Hottung, Kwon, and Tierney (2022). An alternative direction to active search would be to consider an even more lightweight decoder that, whilst less expressive, could allow for a larger population to be trained. We leave a detailed investigation of the tradeoff between population size and the expressivity of each single agent to future work.

Finally, we recall that the motivation behind Poppy was dealing with problems where predicting optimal actions from observations is too difficult to be solved reliably by a single agent. We believe that such settings may not be strictly limited to canonical CO problems, and that population-based approaches offer a promising direction for many challenging RL applications. In this direction, we hope that approaches (such as Poppy) that alleviate the need for handcrafted behavioral markers, whilst still realizing performant diversity, could broaden the range of applications of population-based RL.

# Chapter 8

# Efficient Search of Human Adversarial Policies in Chess

After discussing uncertainty at the action level in the previous chapter, we now shift our focus to planning under uncertainty. To illustrate this concept, we choose a specific combinatorial game as a use case: chess, where uncertainty arises from the unknown opponent's behavior. Specifically, we approach the problem of winning against an opponent of a particular level as a planning problem using a fixed dataset of human games. Drawing on a custom MCTS designed for offline data, and applying it to millions of online human games, we derive HUman-adversarial Search TaiLored at Exploiting Recurrent mistake (Hustler), a system whose plays especially target the weaknesses of players of specific strengths. Our results reveal that this opponent-aware mechanism significantly improves the speed at which winning positions are attained, outperforming Stockfish, one of the strongest chess engines to date. This finding suggests that even super-human AI game programs can benefit from the use of data collected on human-played games, resulting in substantial benefits when assisting or playing against human beings. [1]

**Contents**



## 8.1 Introduction

The general goal of artificial intelligence systems designed for two-player games is to find the optimal strategy. This optimal strategy takes the form of minimax optimum for perfect

[1] This chapter is based on the preprint Grinsztajn and Preux (2023), under review at *IJCAI 2023*.

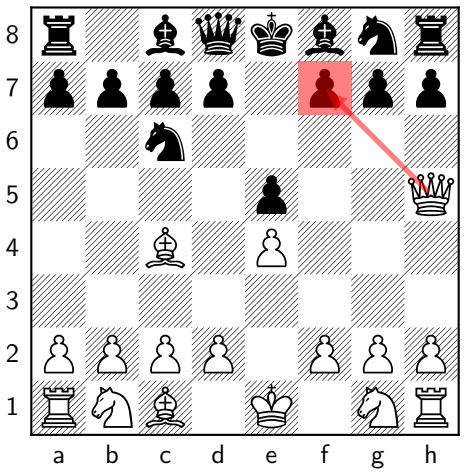



**Figure 8.1** – Setup of the famous Scholar's mate. The white queen is threatening to go to f7 and checkmate the black king. If Black is a beginner, this player thus risks being defeated after only 4 moves. On the other hand, if Black plays the best response pawn to g6 to block the queen, they will enjoy a slightly better position. That is why this opening is never played against high-level players, or by chess engines.

information games (*e.g.* chess, go...), or Nash equilibrium in imperfect information games (*e.g.* poker, diplomacy (Brown and Sandholm, 2019)). These two approaches have in common the assumption that the opponent always plays the best possible response. They thus aim at being robust against any possible adversarial strategy. This mechanism ensures that they are not exploitable, and they have proven to be very efficient in practice, reaching a super-human level in several games (Silver, Huang, Chris J. Maddison, et al., 2016; Silver, Hubert, et al., 2018; Brown and Sandholm, 2019). Although, they may not necessarily be the most efficient means of winning against human opponents who might be subject to bias in specific situations, and who do not always play the best response (A. Anderson, Kleinberg, and Mullainathan, 2017).

On the other hand, in strategic games, human beings tend to adapt their play-style to match the level of their opponents. An instance of this behavior is the classic chess opening known as Scholar's mate, displayed in Fig. 8.1. It is often employed by strong players against weaker opponents, as it can lead to a quick victory in just four moves if they don't see the threat. Similarly, in Poker, players often attempt to identify weaknesses, patterns, and biases in their opponents' play as this can provide a significant advantage.

The Scholar's mate can be seen as an adversarial strategy targeted at weak players: it is not the optimal policy, but it is very likely to lead to a fast checkmate given the average behavior of low-level opponents. The objective of the chapter is to automate the discovery of such adversarial policies for a wide range of levels, from amateurs to semi-professional players.

To that goal, we introduce `data Monte Carlo Tree Search (d-MCTS)`, a specific MCTS designed to search a large and fixed dataset of game trajectories, and present Hustler (HUman-

adversarial Search TaiLored at Exploiting Recurrent mistake), a system capable of playing human-adversarial policies for a wide range of player level. The contributions of the chapter are as follows:

1. We propose a custom MCTS for large offline datasets.
2. We use `d-MCTS` on a large database of online chess games to infer adversarial policies targeted at specific Elo ranges.
3. We propose a quantitative evaluation scheme for these strategies and show that it reaches winning positions in fewer moves than super-human chess engines in practical games. We further analyze some concrete examples of these strategies.

### 8.1.1 Additional related work

**Human policy in chess** Chess has been widely used in academic research as a testbed of artificial intelligence, with the current engines being far above human level (Silver, Hubert, et al., 2018; Stockfish developers, 2022; Pascutto, Gian-Carlo, 2019).

Interestingly, the fields of economics and psychology have also employed chess as a model task environment for studying various cognitive processes like perception, memory, and problem-solving (Charness, 1992). Additionally, research has been conducted on understanding the risk preference of human players (Holdaway and Vul, 2021), and on quantifying the non-transitivity in chess using human game data (Sanjaya, J. Wang, and Y. Yang, 2022).

In recent years, there has been a growing interest in developing AI that can analyze and mimic the human style in chess. In McIlroy-Young, Sen, et al. (2020), researchers introduced Maia, an AI trained on human chess games that predicts human moves more accurately than existing engines. Other related works include McIlroy-Young, R. Wang, et al. (2021) which manages to identify players from their moves and style alone, McIlroy-Young, R. Wang, et al. (2022) which develops predictive models for the behavior of individual players in chess, and Krishnan and Martens (2022) which proposes an interpretable symbolic model of chess policy.

**Adversarial attacks in reinforcement learning** Adversarial attacks have been a topic of interest, especially in computer vision (Goodfellow, Shlens, and Szegedy, 2015; Szegedy et al., 2014). The goal is to find small perturbations to apply to an input image such that it is consistently misclassified.

This line of research has extended toward RL, with the objective of finding an adversary policy in the context of two-player games (Gleave et al., 2020; X. Wu et al., 2021) to attack a RL agent. Similar approaches have been applied to board game AI, like AlphaZero (Silver, Hubert, et al., 2018) in Go. In Timbers et al. (2022), the authors introduce a deep reinforcement

learning algorithm for learning the best response to an agent. In Lan et al. (2022), the authors show that adversarial states can be attained by adding meaningless moves to a position.

**Understanding human errors** Human-adversarial examples have been first studied in Elsayed et al. (2018), where they show that specific image perturbations fooling multiple machine learning models can also fool time-limited humans. In the context of chess specifically, several research papers study chess databases to understand when and why human players commit blunders. A. Anderson, Kleinberg, and Mullainathan (2017) takes into account the complexity of the position at hand, fatigue, and time pressure to help to model human decision, while T. Biswas and Kenneth W Regan (2015) and T. T. Biswas and K. W. Regan (2015) derives for example the search-depth of human player analysis.

## 8.2 Method

### 8.2.1 Background on MCTS

MCTS is a heuristic search algorithm that has been widely used in the field of artificial intelligence, particularly for game playing (Silver, Huang, Chris J. Maddison, et al., 2016; Silver, Hubert, et al., 2018). It was first proposed in Coulom (2007b) to improve the performance of traditional tree search algorithms for Go.

The key idea behind MCTS is to guide the search process by using simulation and evaluation to estimate the value of each potential move. In particular, MCTS relies on the principle of "exploration and exploitation", where the algorithm balances the need to explore different moves in order to gather more information with the need to exploit the best moves known so far in order to maximize the chances of winning.

To implement a MCTS, a tree data structure is used to represent states of the game (nodes), the possible moves (edges), and their outcomes (child node). At each step of the search, the algorithm selects the next move to explore based on a combination of the estimated value of the move and the amount of exploration that has been conducted on that move so far. As the search progresses, the algorithm continually updates the estimates of the value of each move based on the results of the simulations and the evaluations of the resulting positions.

The MCTS algorithm is typically based on four steps, applied at each iterations (Chaslot et al., 2008):

- Selection: starting at the root node, a child selection policy is used to descend through the tree until a leaf is reached.

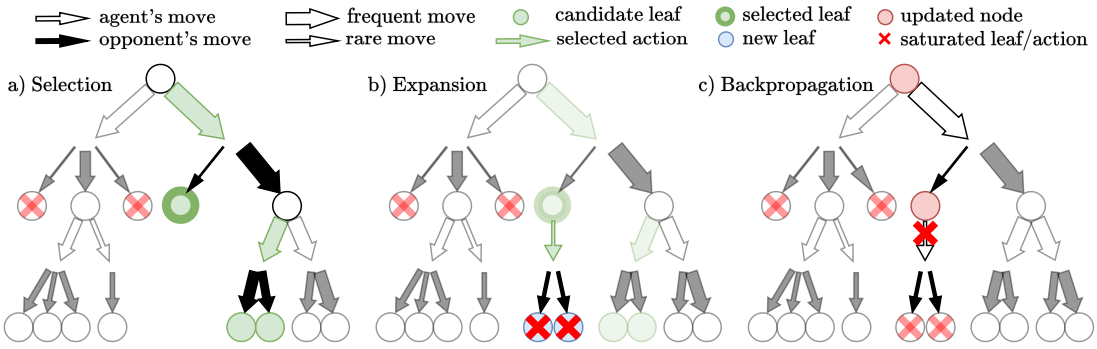



**Figure 8.2** – The three phases of d-MCTS.

- Expansion: one or more child nodes are added to the selected node according to the available moves.
- Simulation: a simulation is run from one of these new nodes according to some default policy (*e.g.* choosing uniform random moves, or heuristics specific to the game…) until the game finishes.
- Backpropagation: the result of the simulation is "backed up" from leaves back to the root through the selected nodes to update their statistics (*e.g.* number of games played from this node, number of wins…)

The node statistics updated at each step are used to compute the child selection policy during the selection phase.

### 8.2.2 MCTS using historical data

The classical MCTS method described above requires specifying the opponent policy during the simulation phase. Usually, it is not a problem, as the same default policy can be used for both the player and their opponent. However, in our setting, we want to find adversarial policies against human players, for whom we only have access to a finite set of historical games. A possible solution would be to train a model to imitate their policy. Though, it looks difficult in practice, because any discrepancy with the real human policy could be exploited by the MCTS and would invalidate the results. To overcome this issue, we modify the usual MCTS algorithm and propose d-MCTS, a version which does not require a simulation phase and can use a fixed dataset of games only.

In our situation, the two players do not have a symmetric role. In what follows, we will write "the attacker" to refer to the side searching for an adversarial policy, and "the victim" for its opponent, whose policy is fixed and represents the average human behavior at a given

ranking. We adopt RL notations, considering the attacker to be the agent, and the victim's behavior to be part of the environment. Therefore, a state $s$ can be any game position where it's the attacker's turn to play, and an action $a \in \mathcal{A}(s)$ can be any legal move in the position.

Formally, we suppose that we have access to a (large) set of previously recorded games. For each game state $s$, we have at our disposal the number of trajectories that went through it that we denote $n_{\text{data}}(s)$, the number of times that the action $a \in \mathcal{A}(s)$ was taken $n_{\text{data}}(s, a)$, as well as the win ratio of those trajectories $V(s)$. We write $\hat{\pi}(a \mid s) = \frac{n_{\text{data}}(a,s)}{n_{\text{data}}(s)}$ the empirical action distribution in the dataset (representing the average policy of the attacker), and $\hat{p}(s' \mid s, a)$ the stochasticity of the environment (representing the empirical policy of the victim).

**Saturated nodes and actions** Because we only have at our disposal a finite dataset, we may have a lot of games played in certain positions, and very few from others. To avoid overfitting to positions with a high win ratio but a very small number of games, we specify a minimum number of trajectories $N_{\text{min}}$ under which a node is not expandable. Intuitively, it sets an exploration frontier: positions encountered in fewer than $N_{\text{min}}$ games will not be explored further. Formally, we write $\partial\mathcal{T} = \{s \in \mathcal{T} \mid n_{\text{data}}(s) < N_{\text{min}}\}$ the set of such nodes in the partially expanded tree $\mathcal{T}$. The depth of this frontier depends on the popularity of each specific game line in our dataset: the allowed depth will be greater when playing frequent actions.

To prevent bumping against these non-expandable nodes when descending through the tree, we call "saturated actions" the actions whose descendant leaves are all in $\partial\mathcal{T}$. For $s \in \mathcal{T}$, we write $\mathcal{A}_{\text{sat}}(s)) \subset \mathcal{A}(s)$ the subset of saturated actions. Intuitively, taking such an action is pointless as it necessarily leads to a non-expandable leaf, and should thus be forbidden.

For each node encountered, we keep several statistics in memory:

- $N(s)$ the number of times the node $s \in \mathcal{T}$ has been visited during the search.
- $N(s, a)$ the number of times the action $a \in \mathcal{A}(s)$ has been chosen during the search.
- $V_{\text{policy}}(s)$ the win rate from the position $s$.
- $V_{\text{search}}$ the win rate from the position $s$ if saturated actions are forbidden.
- The set of saturated actions in each node.

Each time a new node $s$ is added to the tree, both $V_{\text{policy}}$ and $V_{\text{search}}$ and are initialized as the empirical win ratio, that is $V_{\text{policy}}(s) = V_{\text{search}}(s) = V(s)$. Intuitively, $V_{\text{policy}}$ corresponds to the win rate when playing the best adversarial policy discovered and is used at test time once the search is over. $V_{\text{search}}$ is the win rate when playing the best adversarial policy discovered, but removing the part of the trees which cannot be explored further. It is used to guide the selection phase during the search.

**Selection** We start at the root node of the tree and traverse down to the leaves. Specifically, at each visited node, we choose the action that maximizes the following expression:

$$a* = \underset{a \notin \mathcal{A}_{\text{sat}}(s))}{\arg\max} \underset{s' \sim \hat{p}(s,a)}{\mathbb{E}} \left[ V_{\text{search}}(s') \right] + c \times U(s,a) \tag{8.1}$$

This action selection strategy aims at balancing the exploitation of the current knowledge with the necessity of exploring new moves to be able to improve the current policy. Specifically, $V(s')$ favors exploitation while $U(s,a)$ is a bonus which favors exploration, and $c$ is a parameter that balances the two behaviors. We define the exploration term as a variation of `Upper Confidence bounds applied to Trees (UCT)` (Kocsis and Szepesvári, 2006), which we combine with the prior probability $\hat{\pi}$, as done in Silver, Huang, Chris J. Maddison, et al. (2016):

$$U(s,a) = \sqrt{\frac{\hat{\pi}(a \mid s) \log(N(s))}{N(s,a)}} \tag{8.2}$$

Intuitively, if the value of all states is the same, this term ensures that every action is played with the frequency $\hat{\pi}$. This is particularly useful in our case because it makes our search policy more likely to follow frequent moves, and thus to keep expanding nodes with a high number of games. We continue this process until a leaf is reached.

**Expansion** The victim's policy being stochastic, the situation is as if the transitions were non-deterministic. Following the selection policy can thus lead to several possible leaves. Denoting $\mathcal{L}(\mathcal{T})$ the set of leaves of the tree $\mathcal{T}$, the leaf $s_l \in \mathcal{L}(\mathcal{T})$ to expand is selected according to the following quantity:

$$s_l = \underset{\substack{s \in \mathcal{L}(\mathcal{T}) \\ s \notin \partial\mathcal{T}}}{\arg\max} (1 - V(s)) \cdot p_s \tag{8.3}$$

where $V(s)$ is the value of the leaf $s$, and $p_s$ the probability of reaching $s$ starting from the root node and following the previous selection phase. Intuitively, it encourages the selection of future positions which are both probable (with $p_s$), and have room for improvement $(1 - V(s))$. The leaf $s_l$ is then expanded: all its children are added to $\mathcal{T}$.

**Backpropagation** Starting with the expanded leaf and iterating through the sub-tree up to the root, node values are updated according to the equation:

**Algorithm 8.1:** d-MCTS

**Inputs:** Initial state $s_0$, budget $n$
Initialize tree: $\mathcal{T} = \{s_0\}$ ;
**while** $|\mathcal{T}| < n$ **do**
    Build subtree $\tilde{\mathcal{T}}$ according to (8.1) ;
    Select leaf $s_l \in \tilde{\mathcal{T}}$ with largest contribution with (8.3) ;
    Expand $s_l$ ;
    Backpropagate node statistics according to (8.4) and (8.5) ;
    Backpropagate saturation information ;
Return $\arg\max_{\substack{a \in \mathcal{A}(s) \\ n_{\text{data}}(s,a) \geq N_{\min}}} \mathbb{E}_{s' \sim \hat{p}(s,a)} \left[ V_{\text{policy}}(s') \right]$ ;

$$V_{\text{search}}(s) = \max_{a \notin \mathcal{A}_{\text{sat}}(s))} \mathop{\mathbb{E}}_{s' \sim \hat{p}(s,a)} \left[ V_{\text{search}}(s') \right] \tag{8.4}$$

$$V_{\text{policy}}(s) = \max_{\substack{a \in \mathcal{A}(s) \\ n_{\text{data}}(s,a) \geq N_{\min}}} \mathop{\mathbb{E}}_{s' \sim \hat{p}(s,a)} \left[ V_{\text{policy}}(s') \right] \tag{8.5}$$

The quantity $V_{\text{search}}$ is only updated from non-saturated actions. This is motivated by the fact that during the selection phase, saturated actions are unplayable, and thus should not influence the choice of the next leaf to expand. On the other hand, $V_{\text{policy}}$ is updated with the full tree, provided that actions have been played sufficiently in the dataset to avoid overfitting.

Additionally to backpropagating these statistics, we also update saturated actions for each node traversed.

## 8.3 Results

### 8.3.1 Setting

**Dataset** Our train dataset is composed of 165,600,218 blitz online chess games played on the Lichess website during 6 months, from 2021-08 to 2022-01. Blitz are fast-paced games of chess in which each player has typically between 3 and 8 minutes of time to think for the entire game. Because of the limited time available, players must take decisions quickly and cannot afford to spend a lot of time analyzing each potential move. The validation and test datasets are composed of the same type of games, played respectively from 2021-05 to 2021-07 and from 2022-02 to 2022-04 and composed of 80,508,054 and 76,799,666 games. Using a time period for the test dataset that is posterior to those of the train and validation sets ensures that there is no

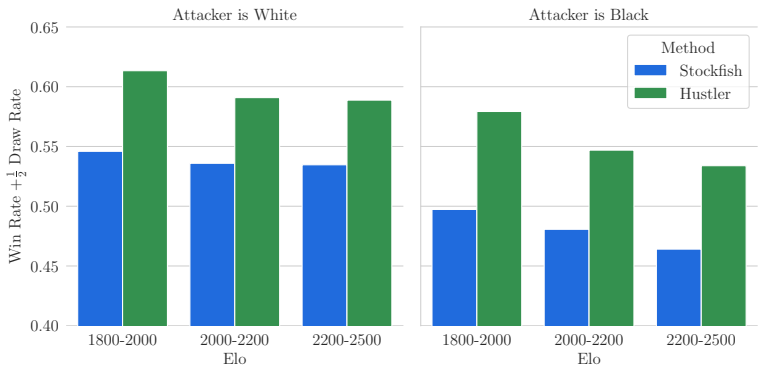



**Figure 8.3** – For each Elo level and each attacker side, the mean reward obtained by Hustler and Stockfish, after 15 moves on the test dataset. Hustler consistently outperforms Stockfish in every scenario.

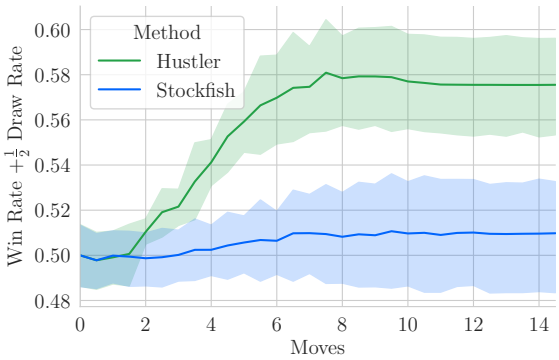



**Figure 8.4** – The reward obtained by Hustler and Stockfish on the test dataset depends on the number of moves considered from the starting position. The curve displayed corresponds to the average over each Elo category for both Black and White as the attacker.

overfitting to a specific distribution, and is aligned with how this tool could be used in practice (using the most recent games to train and test the output policy on the next games).

Games were split into 3 categories depending on the mean Elo of the two players: amateurs (Elo in 1800-2000), strong amateurs (Elo in 2000-2200), and very strong amateurs or semi-professionals (2200-2500).

**Baseline** As a baseline, we use Stockfish version 14.1 (Stockfish developers, 2022), one of the strongest chess engines to date. To limit the computation time, we fix a maximum depth of 20. Ten years ago, Ferreira (2013) evaluated the far weaker Houdini 1.5 at depth 20 to be around 2894 Elo, which is already more than the human FIDE Elo record. Given the progress of chess engines, it is fair to estimate Stockfish to be super-human in these conditions.

**Reward** In chess tournaments, a win is traditionally worth 1 point, a draw $\frac{1}{2}$, and a loss 0. We keep this scale to evaluate the games in our dataset, instead of the win rate. Concretely, for a position $s$ from which $n_{\text{win}}$ games end with the player winning, $n_{\text{draw}}$ with a draw and $n_{\text{loss}}$ with a loss, we have:

$$V(s) = \frac{n_{\text{win}} + \frac{1}{2} n_{\text{draw}}}{n_{\text{win}} + n_{\text{draw}} + n_{\text{loss}}} \tag{8.6}$$

This also ensures that values are always between 0 and 1.

**Parameters** We compute the tree $\mathcal{T}$ from the initial board position, with a budget of $n = 100,000$ nodes. For each state $s$ in the tree which is not a leaf, the corresponding policy consists simply in taking the action defined as:

$$a = \underset{\substack{a \in \mathcal{A}(s) \\ n_{\text{data}}(s,a) \geq N_{\text{min}}}}{\arg\max} \underset{s' \sim \hat{p}(s,a)}{\mathbb{E}} \left[ V_{\text{policy}}(s') \right] \tag{8.7}$$

We grid-search the two hyperparameters of d-MCTS $c \in \{0.1, 0.3, 0.5\}$ and $N_{\text{min}} \in \{50, 100\}$ on the validation dataset.

### 8.3.2 Quantitative evaluation

**Evaluation protocol** A perfect evaluation protocol for the resulting adversarial policies would be to let players of the same level play against each other, while giving access to one of them to the adversarial policy for the first $m$ moves (a move being an action from the attacker, followed by an action from the victim). If this attacking player wins reliably despite being of the same strength as the victim, it would indeed mean that using the adversarial policy results in taking a quick advantage.

We approximate this protocol with our fixed test dataset: one difficulty is that if the policy to analyze plays rare moves, the game could end up in a position that is not in our dataset. To evaluate the adversarial policies, we roll out trajectories from the initial board state. To keep this practical, we use a beam search of width 100 for $m = 15$ moves. Concretely, after every move, we keep the 100 most likely positions, according to the test distribution of the victim's action. We do not expand positions which correspond to a mate, and do not take any action if i) the position is not in $\mathcal{T}$ or is a leaf of $\mathcal{T}$ (as the policy is not defined for this position) or if ii) the action would lead to positions met in less than 50 games in the test dataset. This second condition ensures that the positions in the beam search contain enough games to evaluate win rates reliably.

Once this process is done, we are thus left with 100 positions which have high probabilities of being reached. We then evaluate the win, draw, and loss rates of those positions using the test dataset. Intuitively, when evaluating a specific policy $\pi$, this protocol corresponds to evaluating a player drawing its move from $\pi$ for at most 15 moves, and then playing by themselves, against an opponent playing by themselves from the beginning.

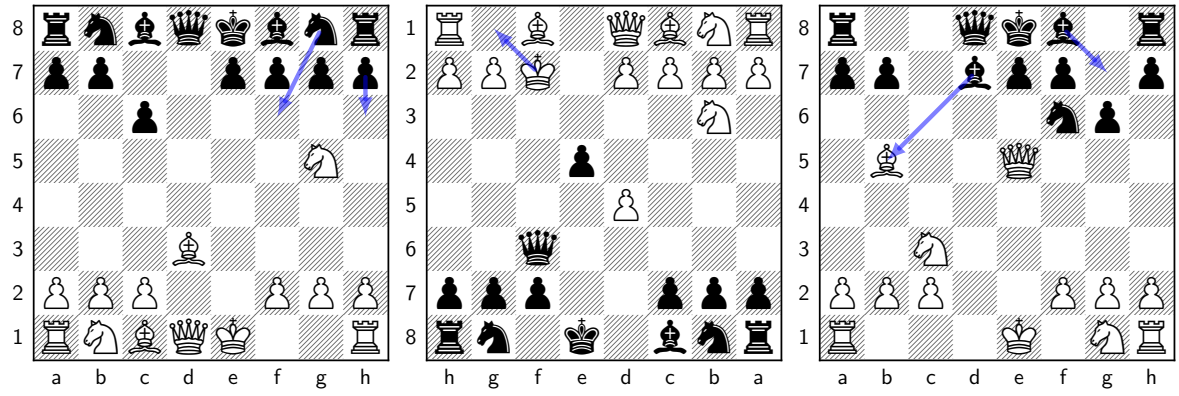
Interestingly, this evaluation scheme is aligned with the way chess players usually train and play. During the opening, most moves come from theoretical lines and have been memorized. Depending on the opening and the level of the players, after 5 to 30 moves, the game reaches a new or unknown position for one of the players, and at this point this player has to find the correct moves without external help.

**Results** We use this evaluation protocol to evaluate the adversarial policies obtained for each Elo category, both when the attacker is White and the attacker is Black. We can see in Fig. 8.3 that Hustler consistently outperforms Stockfish for every situation considered. Unsurprisingly, scores are better for White, as they have the advantage of making the first move. The score also decreases with the Elo range considered, which intuitively indicates that it is easier to fool a weak opponent than a strong opponent. But interestingly, the score is still high in the maximum Elo range tested (2200-2500), which indicates that even very high-level players can be successfully attacked.

In Figure 8.4, we display the scores when varying the number of moves $m$ played using Hustler or Stockfish. As Stockfish plays (almost) optimally, the positions reached become better and better as the number of moves played using its policy rises. Although, the score increases very slowly, which means that these positions are still equal in practice: Stockfish's moves are super-human, though not very threatening for a good amateur player at this stage of the game. On the contrary, Hustler's score shows a sharp rise between the 2nd and 8 moves. It shows that it reaches quickly unbalanced positions, where the attacker has grabbed an advantage.

### 8.3.3 Qualitative examples

Generally speaking, we observe that Hustler plays very aggressively, often gambiting pawns or even full pieces to reach positions where the victim has a high chance of playing a blunder (game-loosing move). These traps can be more or less subtle, depending on the victim's level. To give a sense of Hustler's strategy, we display and analyze one game that could happen against victims of each Elo category. We greedily play the best moves when it is Hustler's turn, and the most likely moves when it is the victim's turn according to the test dataset.



(**a**) 1800-2000 (**b**) 2000-2200 (**c**) 2200-2500

**Figure 8.5** – Human-adversarial chess positions reached by Hustler, ordered by victim's Elo ((**a**): 1800-2000, (**b**): 2000-2200, (**c**): 2200-2500). In blue are displayed the victims' most common moves, which are all blunders, and are played more than 50% of the time.

**Chess notations** We use the algebraic chess notation system. Each piece is represented by a unique pictogram, ♘ for knight, ♗ for bishop, ♖ for rook, ♕ for queen, and ♔ for king, except for the pawn which is the default moving piece when no symbol is used. The squares on which the pieces move are indicated by a combination of a letter and a number, with "a" through "h" representing the columns and "1" through "8" representing the rows. Captures are indicated by the symbol "x" between the piece symbol and the square on which it moves. For example, dxe4 would indicate that a pawn in column "d" capture a piece on the square "e4", and ♘xg5 that a knight captures a piece on the square g5. Check is indicated by the symbol "+" after the move. Castling is indicated by the notation "O-O" for kingside castling and "O-O-O" for queenside castling.

In what follows, teal-colored moves are made by Hustler, while blue moves are frequent moves at the level considered. After each victim's move, we display its probability according to the test dataset. We additionally write "??" after a move if it is a blunder.

**Elo: 1800-2000** We display the beginning of a possible game starting with the Caro Kann defense (**1. e4 c6**), where Hustler has the white pieces.

**1. e4 c6 (8%) 2. ♘f3 d5 (91%) 3. d3 d×e4 (71%) 4. ♘g5 e×d3 (67%) 5. ♗×d3 ♘f6?? (56%)** ( 5... h6?? (23%))

After the initial first move, every move from the victim has a very high probability (respectively 91%, 71%, and 67% for the second, third and fourth move). This gives a 43% probability of performing the three actions in a row and obtaining the position shown in Fig. 8.5a. Once the position is attained, the victim has a 79% chance to play a game-losing move, either by 5...♘f6

or 5…h6. Both these plays make sense intuitively are they are ways of defending the pawn in h7, but fall for the tactic **6.** ♘×f7 ♔×f7 **7.** ♗g6+ h×g6 **8.** ♕×d8 winning the black queen. The best and only moves to keep a slight edge with Black according to Stockfish is 5…♘d7 or 5…♕c7. Although, they are only played respectively 3% and 0.3% of the time by players between 1800 and 2000 Elo. In practice, this position is very likely a win for white, after only 5 moves.

Additionally, if the victim does not play 4…exd3, another popular line transposes toward the same trap: **4…** ♘f6 (19%) **5.** ♘c3 e×d3 (55%) **6.** ♗×d3 h6?? (37%)

**Elo: 2000-2200** We give an example where Hustler has the black pieces, and the victim starts with the common 1. e4:

**1.** e4 (56%) e5 **2.** ♘f3 (66%) d5 **3.** e×d5 (48%) e4 **4.** ♘d4 (30%) ♗c5 **5.** ♘b3 (89%) ♗×f2+ **6.** ♔×f2 (97%) ♕f6+ **7.** ♔g1?? (52%)

White is winning according to Stockfish after the black move 6…♕f6+, giving the position displayed in Fig.8.5b. However, more than half of the time, a victim between 2000 and 2200 Elo would play the blunder 7. ♔g1, leading to a forced mate after 7…♕b6. Although very strong, this move is thus difficult to anticipate in practice even for strong amateurs. It is likely the case because i) the queen is the only developed black piece, so the situation does not feel dangerous, and ii) it is unintuitive that at this point of the game, no white piece can block the check (even 8. d4 does not work because the black pawn e4 can take en passant).

If white responds with the correct 7. ♔e1 instead of 7. ♔g1, Stockfish gives white a huge advantage. However, in practice, the position is still tricky to handle: the white player only wins 51% of the 43 games of the test set.

**Elo: 2200-2500** In the following game, Hustler has the white pieces, and is against very high-level players between 2200 and 2500 Elo playing the classical Sicilian defense (1. e4 c5).

**1.** e4 c5 (34%) **2.** ♘c3 ♘c6 (43%) **3.** d3 g6 (57%) **4.** ♗e3 d6 (56%) **5.** d4 c×d4 (95%) **6.** ♗×d4 ♘×d4 (89%) **7.** ♕×d4 ♘f6 (95%) **8.** e5 d×e5 (93%) **9.** ♗b5 ♗d7 (97%) **10.** ♕×e5 ♗g7?? (51%) ( 10… ♗×b5?? (32%))

The critical position after 10. ♕xe5 is displayed in Fig. 8.5c. The tactic after the blunder 10…Bg7?? is not easy to anticipate, even for skilled players:

**11.** O-O-O O-O **12.** ♗×d7 ♘×d7 **13.** ♕b5 and the knight in d7 cannot be defended.

The other frequent black move 10…Bxb5?? does not hold for long after: ( 10… ♗×b5?? (32%) 11. ♘×b5 ♖c8(74%) 12. ♖d8) and the black king is surrounded.

The best move and the only one to keep a small advantage with black according to the computer is ( 10... a6 (16%)), very rarely played. The resulting evaluation of Stockfish shows only a small edge for Black: even in the rare case where the victim would find the best defense, the game remains very much balanced.

These examples give a sense of Hustler's policy. The stronger the victim, the longer and subtler tactics tend to be.

## Chapter Conclusion

We have presented a novel approach for exploiting human data in game-playing AI. Specifically, we have designed a new MCTS capable of handling offline data and applied it to a dataset of millions of online human chess games. The resulting policy, Hustler, successfully targets weaknesses of players of specific strengths, and our results reveal that it reaches winning positions against humans faster than one of the strongest super-human engines Stockfish.

We believe that this work opens several promising future directions. Professional chess players prepare for their next games and tournaments by memorizing a lot of opening lines. This search for new opening lines is driven by "theoretical novelty", meaning a new move or idea that had hardly been played and was previously unknown to chess theory. The benefit is to surprise the opponent, who will have to deal with a completely new situation on the board. In this sense, a major part of the preparation at the professional level is adversarial: finding unusual moves that are likely to drive the opponent toward making mistakes. Future versions of this work, using professional games with longer time control, possibly combined with chess engine evaluations, could help chess players with their preparations. It could also be possible to target specific players or groups of players, provided that enough game data is available. Furthermore, our search method is based purely on historical data, and does not generalize to unseen positions. It would be interesting to train a model to robustly predict human plays, and then use an adversarial search mechanism on top.

More generally, the approach of using historic data to find the best "practical" moves, incorporating the possible mistakes of the opponent, is a different paradigm from the usual approach aiming at finding the absolute best play in a given position. It could be used in a lot of games, like go, poker, and real-time strategy games. Furthermore, it may be especially appealing in imperfect information games, where a Nash equilibrium is not necessarily the "optimal" strategy given the specific players around the table. Finally, the MCTS for offline data introduced in this chapter is not limited to two-player games: it is sufficiently general to be applied more broadly to stochastic environments where a lot of data is available.

# Chapter 9

# General Conclusion and Perspectives

In this thesis, we have explored the potential of reinforcement learning for combinatorial optimization problems. We have discussed the main limitations and advantages of using this approach, and presented several case studies to illustrate its effectiveness in various contexts.

After a general background introduction to reinforcement learning and combinatorial optimization, we motivated the use of RL for COPs in different contexts: when the problem at hand has not been studied before and cannot be easily reduced to a known problem, when there is uncertainty or stochasticity on the problem (as it is often the case in real-world problems), when the goal is to learn a heuristic on a specific sub-distribution of problems frequently encountered in practice *etc.*

The first part of the thesis was devoted to a case study of the problem of dynamically scheduling interdependent jobs on machines, which provided some insights into the common challenges of using reinforcement learning for realistic combinatorial optimization. The second part dealt with the study of structured problems in reinforcement learning, using two examples: reversibility and sequence equivalence. Although initially motivated by COPs, these two chapters are more general and can be applied to other problems. In Chapter 5, action (ir-)reversibility was initially considered a helpful inductive bias for the combinatorial game Sokoban, where an agent can easily get stuck in positions where the instance becomes unsolvable. But we showed that it can also have additional applications in other domains, such as safety. The third part dealt with uncertainty in combinatorial problems, along two different prisms: how to handle uncertainty at the action level, and how to handle uncertainty at the data level. Chapter 7 outlines a population-based RL approach to explore efficiently several heuristics, shown to be performant on a wide range of combinatorial problems. Chapter 8 uses chess as a use case to develop a method to plan using a finite dataset.

We now provide some perspectives on possible future works, and what we believe are important questions about the use of reinforcement learning for combinatorial optimization.

**Combining reinforcement learning with evolutionary algorithms** Combinatorial problems are characterized by numerous local optima and a vast search space, making it challenging to find the global optimal solution. Reinforcement learning approaches are particularly susceptible to getting trapped quickly during training due to these issues, as observed in Chapter 3. One promising approach to overcome this limitation is to maintain a population of solutions by combining RL with evolutionary algorithms. This hybrid approach has been studied in other contexts (Khadka and Tumer, 2018; Khadka, Majumdar, et al., 2019; Pourchot and Sigaud, 2019; Pierrot et al., 2022) like continuous control, and could be a fruitful direction for CO. Poppy, introduced in Chapter 7, can be seen as a first step in this direction, as it maintains a population of specialized agents. Scaling such approaches to larger populations and larger instance sizes could be interesting future work.

**From canonical combinatorial problems to real-world problems** One common criticism of RL is that it tends to focus too much on games and synthetic benchmarks, rather than real-world applications. Interestingly, COP can act as a first middle ground: they share many qualities with games that make them amenable to RL, such as fast and perfect simulators without sim-to-real issues, while having a wide range of applications, as outlined in Chapter 1. A venue for RL can thus be to tackle complex and applied combinatorial problems, like chip placement and design (Liao et al., 2019; Mirhoseini, Goldie, et al., 2020). Problems involving a large combinatorial search, like theorem proving (Kaliszyk et al., 2018; Lample et al., 2022), or discovering faster matrix multiplication algorithms (Fawzi et al., 2022) are also interesting venues.

Moreover, RL can also be applied to real-world combinatorial problems that involve uncertainty and stochasticity. In such cases, RL allows for the learning of heuristics in an end-to-end fashion, whereas traditional methods would first need to model the stochasticity and then solve the problem, which can be challenging. Some examples of such problems include HPC (Chapter 3, Chapter 4), routing in a city with stochastic demand and traffic, and inventory management *etc.* However, the success of this approach depends on the available environments and the research ecosystem. To this end, recent initiatives such as the competition on vehicle routing problems (Kool, Bliek, et al., 2022) and the Jumanji library (Bonnet et al., 2022), which focuses on providing a library of combinatorial problems with a particular emphasis on industrial problems, are promising developments.

**Reinforcement learning from human feedback: a combinatorial problem?** Reinforcement learning from human feedbacks (Knox and Stone, 2008; Christiano et al., 2017, RLHF) consists in learning policies using human preferences as a reward function. This approach has recently been used in the context of large language models (Ziegler et al., 2019). Indeed, these large models are trained to imitate large corpus of text data, which although powerful, is not perfectly aligned with their intended use. For example, they are prone to generate untruthful, unhelpful

or even toxic text. RLHF is an efficient way to improve the performance of these models (Ouyang et al., 2022). RLHF features an agent which generates a sequence of discrete actions, with a reward given at the very end of the sequence. Interestingly, this is very similar to combinatorial problems, as studied, for example, in Chapter 3, Chapter 4, and Chapter 7. This similarity suggests that some techniques developed for COP could be useful for RLHF. For example, the use of a population, as discussed in Chapter 7, could be adapted for RLHF to maintain a diversity of solutions.

**Tool-augmented reinforcement learning** Some computations involved to find solutions to combinatorial problems are very easy to perform, but hard to learn. In Chapter 7 for instance, when solving the TSP, an agent must probably learn implicitly to compute distances between cities and compare lengths of possible sub-paths. These kinds of tasks are hard for a neural network, as it needs to be robust to every city position, and require very high precision, but they are strikingly easy to compute with any calculator. Some difficulty of learning to solve TSP with RL thus revolves around teaching an agent to perform high-precision arithmetics, which seems a bit wasteful.

A promising direction for alleviating this issue could be to connect the agent to external tools, that it can learn to use during the training process. This line of work has been explored recently for large language models. Cobbe et al. (2021), for example, trains a language model to make use of an external calculator which is called after the use of a special token. These approaches, although promising, remain largely unexplored for reinforcement learning in general and combinatorial optimization in particular.

# Appendix A

# Complements on Chapter 5

We organize the supplementary material as follows: in Appendix A.1, we include the proofs of results from the main text, as well as additional formalism; in Appendix A.2, we provide additional details about the proposed algorithms, including pseudo-code and figures that did not fit in the main text; and in Appendix A.3, we detail our experimental procedure, including hyperparameters for all methods/

## A.1 Mathematical elements and proofs

### A.1.1 Possible definitions of reversibility

In this section, we present several intuitive definitions of reversibility in MDPs. We chose the third definition as our reference, which we argue presents several advantages over the others, although they can be interesting in specific contexts. Indeed, Eq. (A.5) is simpler than Eq. (A.1), as it does not depend on the discount factor, and more general than Eq. (A.3), as it does not enforce a fixed number of timesteps for going back to the starting state.

**Discounted reward.**

$$\phi_{\pi,K}(s,a) \coloneqq \sum_{k>t}^{K} \gamma^{k-t} p_\pi(s_{t+k} = s \mid s_t = s, a_t = a)\,, \tag{A.1}$$

$$\phi_{\pi}(s,a) \coloneqq \sum_{k>t}^{\infty} \gamma^{k-t} p_\pi(s_{t+k} = s \mid s_t = s, a_t = a). \tag{A.2}$$

**Fixed time step.**

$$\phi_{\pi,K}(s,a) \coloneqq \sup_{k \leq K} p_\pi(s_{t+k} = s \mid s_t = s, a_t = a)\,, \tag{A.3}$$

$$\phi_\pi(s,a) \coloneqq \sup_{k \in \mathbb{N}} p_\pi(s_{t+k} = s \mid s_t = s, a_t = a). \tag{A.4}$$

**Undiscounted reward.**

$$\begin{aligned}\phi_{\pi,K}(s,a) &\coloneqq \sum_{k=1}^{K} p_\pi(s_{t+k} = s, s_{t+k-1} \neq s, \dots, s_{t+1} \neq s \mid s_t = s, a_t = a)\,, \\ &= p_\pi(s \in \tau_{t+1:t+K+1} \mid s_t = s, a_t = a)\,. \end{aligned} \tag{A.5}$$

$$\begin{aligned}\phi_\pi(s,a) &\coloneqq \sum_{k=1}^{\infty} p_\pi(s_{t+k} = s, s_{t+k-1} \neq s, \dots, s_{t+1} \neq s \mid s_t = s, a_t = a)\,, \\ &= p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s, a_t = a). \end{aligned} \tag{A.6}$$

### A.1.2 Additional properties

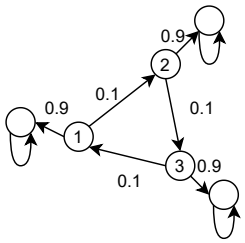



**Figure A.1** – Counter-example for additional property 4. The initial state is sampled uniformly amongst $\{0, 1, 2\}$.

We write $s \to s'$ if $\psi_\pi(s, s') \geq 0.5$ ("it is more likely to go from $s$ to $s'$ than to go from $s'$ to $s$") and $s \Rightarrow s'$ if $\psi_\pi(s, s') = 1$ ("it is possible to go from $s$ to $s'$, but it is not possible to come back to $s$ from $s'$").

1. $\psi_\pi(s, s') + \psi_\pi(s', s) = 1$
2. if $s_0 \Rightarrow s_1 \Rightarrow s_2$ then $s_0 \Rightarrow s_2$ (transitivity for $\Rightarrow$)
3. if $s_0 \to s_1 \to \cdots \to s_i \Rightarrow s_{i+1} \to \cdots \to s_t$ then $s_0 \Rightarrow s_t$
4. in general $s_1 \to s_2$ and $s_2 \to s_3$ doesn't imply $s_1 \to s_3$

Proofs:

(1) $\psi_\pi(s, s') + \psi_\pi(s', s) = \mathbb{E}_{\tau \sim \pi}\, \mathbb{E}_{t \neq t' | s_t = s, s_{t'} = s'} \left[ \mathbb{1}_{t' > t} + \mathbb{1}_{t' < t} \right] = \mathbb{E}_{\tau \sim \pi}\, \mathbb{E}_{t \neq t' | s_t = s, s_{t'} = s'} \left[ 1 \right] = 1.$

(2) and (3): As (3) is stronger than (2), we only prove (3). If it is possible to have $s_0$ after $s_t$ in a trajectory, then it is possible to have $s_i$ after $s_t$. As we have a positive probability of seeing $s_t$ after $s_{i+1}$, we have a positive probability of seeing $s_i$ after $s_{i+1}$, which contradicts $s_i \Rightarrow s_{i+1}$.

(4) A counter example can be found in Fig. A.1. In this case we clearly have $s_1 \to s_2$, $s_2 \to s_3$ and $s_3 \to s_1$.

### A.1.3 Proofs of theorem 5.4 and theorem 5.6

In the following, we prove simultaneously Theorem 5.4 and Theorem 5.6. We begin by two lemmas.

**Lemma A.1.** *Given a trajectory $\tau$, we denote by $\#_T(s \to s')$ the number of pairs $(s, s')$ in $\tau_{1:T}$ such that $s$ appears before $s'$. We present a simple formula for $\psi(s', s)$ according to the structure of the state trajectory:*

$$\psi_{\pi,T}(s, s') = \frac{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s')\right]}{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s)\right]}\,. \tag{A.7}$$

*Proof.* In order to simplify the notations, we leave implicit the fact that indices are always sampled within $[0, T]$.

$$\psi_{\pi,T}(s, s') = \mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t\neq t' | s_t=s, s_{t'}=s'} \left[\mathbb{1}_{t'>t}\right], \tag{A.8}$$

$$= \frac{\mathbb{E}_\pi\, \mathbb{E}_{t\neq t'}\left[\mathbb{1}_{t'>t}\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right]}{\mathbb{E}_\pi\, \mathbb{E}_{t\neq t'}\left[\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right]}\,. \tag{A.9}$$

$$\tag{A.10}$$

Similarly, we have:

$$\mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t'>t} \left[\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right] = \frac{\mathbb{E}_\pi\, \mathbb{E}_{t\neq t'}\left[\mathbb{1}_{t'>t}\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right]}{\mathbb{E}_{t\neq t'}\left[\mathbb{1}_{t'>t}\right]}\,. \tag{A.11}$$

Combining it with our previous equation:

$$\psi_{\pi,T}(s, s') = \frac{\mathbb{E}_\pi\, \mathbb{E}_{t'>t}\left[\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right]\, \mathbb{E}_{t\neq t'}\left[\mathbb{1}_{t'>t}\right]}{\mathbb{E}_\pi\, \mathbb{E}_{t\neq t'},\left[\mathbb{1}_{s_t=s}\mathbb{1}_{s_{t'}=s'}\right]}\,, \tag{A.12}$$

$$= \frac{1}{2} \frac{\mathbb{E}_\pi \mathbb{E}_{t'>t} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right]}{\mathbb{E}_\pi \mathbb{E}_{t \neq t'} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right]} . \tag{A.13}$$

$$\tag{A.14}$$

Looking at the denominator, we can notice:

$$\mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t \neq t'} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right] = \frac{1}{2} \mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t<t'} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right] + \frac{1}{2} \mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t'<t} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right] , \tag{A.15}$$

$$= \frac{1}{2} \mathop{\mathbb{E}}_{\pi} \mathop{\mathbb{E}}_{t<t'} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} + \mathbb{1}_{s_t=s'} \mathbb{1}_{s_{t'}=s} \right] , \tag{A.16}$$

which comes from the fact that $t$ and $t'$ play a symmetrical role. Thus,

$$\psi_{\pi,T}(s, s') = \frac{\mathbb{E}_{\tau \sim \pi} \mathbb{E}_t \mathbb{E}_{t'>t} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right]}{\mathbb{E}_{\tau \sim \pi} \mathbb{E}_t \mathbb{E}_{t'>t} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} + \mathbb{1}_{s_t=s'} \mathbb{1}_{s_{t'}=s} \right]} . \tag{A.17}$$

Since

$$\mathop{\mathbb{E}}_{\tau \sim \pi} \left[ \#_T(s \to s') \right] = \sum_{i<j \leq T} \mathbb{1}_{s_i=s} \mathbb{1}_{s_j=s'} , \tag{A.18}$$

$$= \binom{T}{2} \sum_{i<j \leq T} \frac{1}{\binom{T}{2}} \mathbb{1}_{s_i=s} \mathbb{1}_{s_j=s'} , \tag{A.19}$$

$$= \binom{T}{2} \mathop{\mathbb{E}}_{\tau \sim \pi} \mathop{\mathbb{E}}_{t} \mathop{\mathbb{E}}_{t'>t} \left[ \mathbb{1}_{s_t=s} \mathbb{1}_{s_{t'}=s'} \right] , \tag{A.20}$$

we get:

$$\psi_{\pi,T}(s, s') = \frac{\binom{T}{2} \mathbb{E}_{\tau \sim \pi} \left[ \#_T(s \to s') \right]}{\binom{T}{2} \mathbb{E}_{\tau \sim \pi} \left[ \#_T(s \to s') + \#_T(s' \to s) \right]} , \tag{A.21}$$

$$\psi_{\pi,T}(s, s') = \frac{\mathbb{E}_{\tau \sim \pi} \left[ \#_T(s \to s') \right]}{\mathbb{E}_{\tau \sim \pi} \left[ \#_T(s \to s') + \#_T(s' \to s) \right]} . \tag{A.22}$$

$$\tag{A.23}$$

□

**Lemma A.2.** *Assume that we are given a fixed trajectory where $s$ appears $k \in \mathbb{N}$ times, in the form of :*

$$s_0 \underbrace{\longrightarrow}_{n_0(s')} s \underbrace{\longrightarrow}_{n_1(s')} s \underbrace{\longrightarrow}_{n_2(s')} s \underbrace{\longrightarrow}_{n_3(s')} s \cdots \underbrace{\longrightarrow}_{n_{k-1}(s')} s \underbrace{\longrightarrow}_{n_k(s')}, \tag{A.24}$$

*where $n_i(s')$ denotes the number of times $s'$ appears between the $i^{th}$ and the $(i+1)^{th}$ occurrence of $s$.*

*In this case,*

$$\#_T(s \to s') = \sum_{i=0}^{k} i \times n_i(s'). \tag{A.25}$$

*If we suppose that $n_1(s') = n_2(s') = \cdots = n_{k-1}(s')$, we also have*

$$\#_T(s \to s') - \#_T(s' \to s) = k\left(n_k(s') - n_0(s')\right). \tag{A.26}$$

*Proof.* Eq. (A.25) comes directly from $\#_T(s \to s') = \sum_{i=1}^{k}\sum_{j=i}^{k} n_j(s') = \sum_{i=0}^{k} i \times n_i(s')$. To prove Equ. (A.26), we first notice that $\#_T(s \to s') + \#_T(s' \to s) = k \times \sum_{i=0}^{k} n_i(s')$. Thus

$$\#_T(s \to s') - \#_T(s' \to s) = 2 \times \#_T(s \to s') - \left(\#_T(s \to s') + \#_T(s' \to s)\right), \tag{A.27}$$

$$= 2\left(k\, n_k(s') + n_1(s') \sum_{i=0}^{k-1} i\right) - \left(k\, n_k(s') + k\, n_0(s') + k\,(k-1)\, n_1(s')\right), \tag{A.28}$$

$$= k\, n_k(s') - k\, n_0(s'). \tag{A.29}$$

□

**Theorem.** *For every policy $\pi$ and $s, s' \in \mathcal{S}$, $\psi_{\pi,T}(s, s')$ converges when $T$ goes to infinity.*

**Theorem.** *Given a policy $\pi$, a state $s$, and an action $a$, we can link reversibility and empirical reversibility with the inequality: $\bar{\phi}_\pi(s,a) \geq \frac{\phi_\pi(s,a)}{2}$.*

*Proof.* For a policy $\pi$ and $s, s' \in \mathcal{S}$, we define $\hat{\phi}_\pi(s, s')$ the quantity $p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s')$ such that $\phi_\pi(s,a) = \mathbb{E}_{s' \sim P(s,a)}\left[\hat{\phi}_\pi(s, s')\right]$ In order to prove the theorem, we first prove that $\psi_T(s', s)$ converges to a quantity denoted by $\psi(s', s)$, and that:

$$\forall s, s' \in \mathcal{S}, \frac{\hat{\phi}^\pi(s, s')}{2} \leq \psi(s', s). \tag{A.30}$$

We subdivide our problem into four cases, depending on whether $s$ and $s'$ are recurrent or transient.

**Case 1:** $p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s) < 1$ and $p_\pi(s' \in \tau_{t+1:\infty} \mid s_t = s') = 1$ ($s$ is transient and $s'$ is recurrent for the Markov chain induced by $\pi$). Informally, this means that if a trajectory contains the state $s'$ we tend to see $s'$ an infinite number of times, and we only see $s$ a finite number of times in a given trajectory.

This implies $\hat{\phi}_\pi(s, s') = p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s') = 0$, as recurrent states can only be linked to other recurrent states (Norris, 1998). It is not possible to find trajectories where $s$ appears after $s'$, thus $\psi_T(s', s) = 0 = \psi(s', s)$. Equ. (A.30) becomes "$0 \leq 0$".

**Case 2:** $p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s) = 1$ and $p_\pi(s' \in \tau_{t+1:\infty} \mid s_t = s') < 1$ ($s$ is recurrent and $s'$ is transient for the Markov chain induced by $\pi$).

As before, this implies $\hat{\phi}_\pi(s', s) = p_\pi(s' \in \tau_{t+1:\infty} \mid s_t = s) = 0$, and thus it is not possible to see in a trajectory $s$ after $s'$. It implies $\psi_T(s', s) = 1 = \psi(s', s)$, so Equ. (A.30) is verified.

**Case 3:** $p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s) = 1$ and $p_\pi(s' \in \tau_{t+1:\infty} \mid s_t = s') = 1$ ($s$ is recurrent and $s'$ is recurrent for the Markov chain induced by $\pi$). We denote by $T_k$ the random variable corresponding to the time of the $k^{\text{th}}$ visit to $s$. A trajectory can be represented as follows:

$$s_0 \underbrace{\longrightarrow}_{n_1(s')} s \underbrace{\longrightarrow}_{n_2(s')} s \underbrace{\longrightarrow}_{n_3(s')} s \underbrace{\longrightarrow}_{n_4(s')} s \cdots \underbrace{\longrightarrow}_{n_k(s')} s = s_{T_k} \underbrace{\longrightarrow}_{n_{k+1}(s')} , \tag{A.31}$$

where, writing $\sim$ the equality in distribution, $n_2(s') \sim n_3(s') \sim \cdots \sim n_k(s')$ and $\mathbb{E}_\tau\, n_2(s') = \mathbb{E}_\tau\, n_3(s') = \cdots = \mathbb{E}_\tau\, n_k(s')$ using the strong Markov property. From Lemma A.1 we get:

$$\psi_{\pi,T}(s, s') = \frac{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s')\right]}{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s)\right]}, \tag{A.32}$$

$$= \frac{1}{2}\frac{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s) + \#_T(s \to s') - \#_T(s' \to s)\right]}{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s)\right]}, \tag{A.33}$$

$$= \frac{1}{2} + \frac{\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') - \#_T(s' \to s)\right]}{2\ \mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s)\right]}. \tag{A.34}$$

We can see from Lemma A.2 :

$$\mathop{\mathbb{E}}_{\tau}\left[\#_{T_k}(s \to s') - \#_{T_k}(s' \to s)\right] = -k \mathop{\mathbb{E}}_{\tau} n_1(s'). \tag{A.35}$$

Thus,

$$\frac{\mathbb{E}_\tau\left[\#_{T_k}(s \to s') - \#_{T_k}(s' \to s)\right]}{\mathbb{E}_{\tau\sim\pi}\left[\#_{T_k}(s \to s') + \#_{T_k}(s' \to s)\right]} = \frac{-k\ \mathbb{E}_\tau\, n_1(s')}{k\ \mathbb{E}_\tau\, n_1(s') + k^2\ \mathbb{E}_\tau\, n_2(s')} \tag{A.36}$$

$$\xrightarrow[k\to\infty]{} 0. \tag{A.37}$$

Given $t \in \mathbb{N}$ and a trajectory $\tau$, we denote $\#_T(s)$ the random variable corresponding to the number of times when $s$ appear before $t$, such that a trajectory has the following structure :

$$s_0 \underbrace{\longrightarrow}_{n_1(s')} s \underbrace{\longrightarrow}_{n_2(s')} s \underbrace{\longrightarrow}_{n_3(s')} s \underbrace{\longrightarrow}_{n_4(s')} \dots \underbrace{\longrightarrow}_{n_k(s')} s = s_{T_{\#_T(s)}} \underbrace{\longrightarrow s_t \longrightarrow}_{n_{k+1}(s')} s = s_{T_{\#_T(s)+1}}\,. \tag{A.38}$$

$$\frac{\mathbb{E}_\tau\left[\#_T(s \to s') - \#_T(s' \to s)\right]}{\mathbb{E}_\tau\left[\#_T(s \to s') + \#_T(s' \to s)\right]} \leq \frac{\mathbb{E}_\tau\left[\#_{\#_T(s)}(s \to s') - \#_{\#_T(s)}(s' \to s)\right] + \mathbb{E}_\tau\, \#_T(s) n_{k+1}(s')}{\mathbb{E}_\tau\, \#_{\#_T(s)}(s \to s') + \mathbb{E}_\tau\, \#_{\#_T(s)}(s' \to s)}\,, \tag{A.39}$$

$$\xrightarrow[T\to\infty]{} 0 \text{ as in Equ. (A.36).} \tag{A.40}$$

And,

$$\frac{\mathbb{E}_\tau\left[\#_T(s \to s') - \#_T(s' \to s)\right]}{\mathbb{E}_\tau\left[\#_T(s \to s') + \#_T(s' \to s)\right]} \geq \frac{\mathbb{E}_\tau\left[\#_{\#_T(s)}(s \to s') - \#_{\#_T(s)}(s' \to s)\right] - \mathbb{E}_\tau \sum_{i=1}^{\#_T(s)+1} n_i(s')}{\mathbb{E}_\tau\left[\#_{\#_T(s)}(s \to s') + \#_{\#_T(s)}(s' \to s)\right]}\,, \tag{A.41}$$

$$\xrightarrow[T\to\infty]{} 0 \tag{A.42}$$

Therefore,

$$\frac{\mathbb{E}_\tau\left[\#_T(s \to s') - \#_T(s' \to s)\right]}{\mathbb{E}_\tau\left[\#_T(s \to s') + \#_T(s' \to s)\right]} \xrightarrow[T\to\infty]{} 0 \text{ , and finally,} \tag{A.43}$$

$$\psi_{\pi,T}(s, s') = \frac{1}{2} + \frac{\mathbb{E}_\tau\left[\#_T(s \to s') - \#_T(s' \to s)\right]}{2\,\mathbb{E}_{\tau\sim\pi}\left[\#_T(s \to s') + \#_T(s' \to s)\right]} \xrightarrow[T\to\infty]{} \frac{1}{2}\,. \tag{A.44}$$

As $\hat{\phi}_\pi(s,s') = 1$ here, we immediately have $\frac{\hat{\phi}_\pi(s,s')}{2} = \psi(s',s)$. We can notice that the inequality is tight in this case.

**Case 4:** $p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s) < 1$ and $p_\pi(s' \in \tau_{t+1:\infty} \mid s_t = s') < 1$ ($s$ is transient and $s'$ is transient for the Markov chain induced by $\pi$). To simplify the following formulas, we will write $\alpha = p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s')$. Here, we denote by $\#(s)$ the random variable corresponding to the total number of visits of the state $s$, and $\#(s \to s')$ the number of pairs such that $s$ appears before $s'$. $\#(s)$ follows the geometric distribution $G\left(1 - p_\pi(s \in \tau_{t+1:\infty} \mid s_t = s)\right)$.

$\#_T(s \to s')$ converges almost surely to $\#(s \to s')$, and we have $\#_T(s \to s') \leq \#(s \to s')$. Therefore, using the dominated convergence theorem, $\mathbb{E}_\tau\left[\#_T(s \to s')\right] \xrightarrow[T\to\infty]{} \mathbb{E}_\tau\left[\#(s \to s')\right]$, and thus:

$$\psi_{\pi,T}(s',s) = \frac{\mathbb{E}_\tau\left[\#_T(s' \to s)\right]}{\mathbb{E}_\tau\left[\#_T(s \to s') + \#_T(s' \to s)\right]} \xrightarrow[T\to\infty]{} \frac{\mathbb{E}_\tau \#(s' \to s)}{\mathbb{E}_\tau\left[\#(s \to s') + \#(s' \to s)\right]} = \psi^\pi(s',s)\,. \tag{A.45}$$

This time, we consider a trajectory $\tau$ where $s$ appears $k$ times after $s'$, such that it is of the form:

$$\underbrace{s' \dots s'}_{n_0(s')\geq 0} \longrightarrow \underbrace{s \dots s}_{n_1(s)>0} \longrightarrow \underbrace{s' \dots s'}_{n_1(s')>0} \longrightarrow \underbrace{s \dots s}_{n_2(s)>0} \longrightarrow \cdots \longrightarrow \underbrace{s' \dots s'}_{n_{k-1}(s')>0} \longrightarrow \underbrace{s \dots s}_{n_k(s)>0} \longrightarrow \underbrace{s' \dots s'}_{n_k(s')\geq 0} \longrightarrow \tag{A.46}$$

Here, $n_0(s')$ is the number of times when $s'$ appears in the trajectory before the first appearance of $s'$, $n_i(s)$ is the number of times when $s$ appears between two occurrences of $s'$, and $n_k(s')$ the number of times when $s'$ appears after the last appearance of $s$. From the strong Markov property, $n_1(s') \sim n_2(s') \sim \cdots \sim n_{k-1}(s')$ and $n_1(s) \sim n_2(s) \sim \cdots \sim n_k(s)$. Note also that these variables are all independent. Here $k$ is a random variable following the geometric distribution $G(\alpha)$ where $\alpha = p(s \in \tau_{t:\infty} \mid s_t = s')$. Notice that when $n_k(s') > 0$, we have $n_k(s) \sim n_1(s)$ and $n_k(s') \sim n_1(s')$.

Using these two simplifications, one can write:

$$\mathop{\mathbb{E}}_{\tau}\Big[\#(s' \to s) - \#(s \to s')\Big|k\Big] \geq \mathop{\mathbb{E}}_{\tau}\Big[\#(s' \to s) - \#(s \to s')\Big|k, n_k(s') > 0\Big]\,, \tag{A.47}$$

$$\geq \mathop{\mathbb{E}}_{\tau}\Big[n_0(s')\big[n_1(s) + (k-1)n_1(s) + n_k(s)\big] - n_1(s)\big[kn_1(s') + n_k(s')\big] + \tag{A.48}$$

$$n_k(s)\big[kn_1(s') - n_k(s')\big] - n_k(s')(k-1)n_1(s)\Big|k, n_k(s') > 0\Big]\,, \tag{A.49}$$

$$\geq -k \mathop{\mathbb{E}}_{\tau}\Big[n_1(s)\Big|k\Big] \mathop{\mathbb{E}}_{\tau}\Big[n_k(s')\Big|k, n_k(s') > 0\Big] \text{ as in Lemma A.2}\,, \tag{A.50}$$

$$\geq -k \mathop{\mathbb{E}}_{\tau}(n_1(s)) \mathop{\mathbb{E}}_{\tau}(n_1(s'))\,. \tag{A.51}$$

Likewise,

$$\mathop{\mathbb{E}}_{\tau}\left[\#(s' \to s) + \#(s \to s') \,|\, k\right] = \mathop{\mathbb{E}}_{\tau}\Big[k\, n_1(s)n_k(s') + k\, n_0(s')n_1(s) + k\,(k-1)n_1(s)n_1(s')\Big|k\Big]\,, \tag{A.52}$$

$$= k\Big[\mathop{\mathbb{E}}_{\tau}\left[n_1(s)\right] \mathop{\mathbb{E}}_{\tau}\left[n_1(s')\right] + \mathop{\mathbb{E}}_{\tau}\left[n_1(s)\right] \mathop{\mathbb{E}}_{\tau}\left[n_0(s')\right]\Big] + k(k-1) \mathop{\mathbb{E}}_{\tau}\left[n_1(s)\right] \mathop{\mathbb{E}}_{\tau}\left[n_1(s')\right]. \tag{A.53}$$

Thus,

$$\frac{\mathbb{E}_\tau\left[\#(s' \to s) - \#(s \to s')\right]}{\mathbb{E}_\tau\left[\#(s \to s') + \#(s' \to s)\right]} = \frac{\sum_{i=1}^{\infty} p(k=i)\, \mathbb{E}_\tau\left[\#(s' \to s) - \#(s \to s') \,|\, k=i\right]}{\sum_{i=1}^{\infty} p(k=i)\, \mathbb{E}_\tau\left[\#(s \to s') + \#(s' \to s) \,|\, k=i\right]}\,, \tag{A.54}$$

$$\geq -\frac{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\, i\, \mathbb{E}_\tau\left[n_1(s)\right]\, \mathbb{E}_\tau\left[n_1(s')\right]}{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\Big[i\left(\mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_1(s')\right] + \mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_0(s')\right]\right) + i\,(i-1)\, \mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_1(s')\right]\Big]}\,, \tag{A.55}$$

$$\geq -\frac{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha) i\, \mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_1(s')\right]}{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\Big[i\, \mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_1(s')\right] + i\,(i-1)\, \mathbb{E}_\tau\left[n_1(s)\right] \mathbb{E}_\tau\left[n_1(s')\right]\Big]}\,, \tag{A.56}$$

$$\geq -\frac{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\, i}{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\left[i + i\,(i-1)\right]}\,, \tag{A.57}$$

$$\geq -\frac{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\, i}{\sum_{i=1}^{\infty} \alpha^{i-1}(1-\alpha)\, i^2}\,, \tag{A.58}$$

$$\geq -\frac{\frac{1}{1-\alpha}}{\frac{1+\alpha}{(1-\alpha)^2}}\,, \tag{A.59}$$

$$\geq -\frac{1-\alpha}{1+\alpha}\,. \tag{A.60}$$

From Lemma A.1,

$$\psi^{\pi}(s', s) = \frac{1}{2}\left(1 + \frac{\mathbb{E}_{\tau}\left[\#(s' \to s) - \#(s \to s')\right]}{\mathbb{E}_{\tau\sim\pi}\left[\#(s \to s') + \#(s' \to s)\right]}\right), \tag{A.61}$$

$$\geq \frac{1}{2}\left(1 - \frac{1-\alpha}{1+\alpha}\right), \tag{A.62}$$

$$\geq \frac{\alpha}{1+\alpha}, \tag{A.63}$$

$$\geq \frac{\alpha}{2} = \frac{\hat{\phi}_{\pi}(s, s')}{2}. \tag{A.64}$$

As a quick summary, we divided our problem in 4 cases, and proved that in each case, for every pair of states $s, s'$, we have $\psi^{\pi}(s', s) \geq \frac{\hat{\phi}_{\pi}(s,s')}{2}$.

To end the proof, we simply take the expectation over the distribution of the next states:

$$\mathop{\mathbb{E}}_{s'\sim P(s,a)} \psi_{\pi}(s', s) \geq \frac{1}{2} \mathop{\mathbb{E}}_{s'\sim P(s,a)} \hat{\phi}_{\pi}(s, s'), \tag{A.65}$$

$$\bar{\phi}_{\pi}(s, a) \geq \frac{\phi_{\pi}(s, a)}{2}. \tag{A.66}$$

□

### A.1.4 Proof of proposition 5.7

**Proposition** (Proposition 5.7). *We suppose that we are given a state $s$, an action $a$ such that $a$ is reversible in $K$ steps, a policy $\pi$ and $\rho > 0$. Then, $\bar{\phi}_{\pi}(s, a) \geq \frac{\rho^K}{2}$, where $\mathcal{A}$ denotes the number of actions. Moreover, we have for all $K \in \mathbb{N}$: $\bar{\phi}_{\pi}(s, a) \geq \frac{\rho^K}{2}\phi_K(s, a)$.*

*Proof.* We first prove the second part of the proposition, which is more general. From Definition 5.1, and as the set of policies is closed, there is a policy $\pi^*$ such that $\phi_K(s, a) = p_{\pi^*}(s \in \tau_{t+1:t+K+1} \mid s_t = s, a_t = a)$. We begin by noticing that $\pi$ has a probability at least equal to $\rho$ to copy the policy $\pi^*$ in every state.

It can be stated more formally:

$$\forall s \in \mathcal{S}, \mathop{\mathbb{E}}_{a\sim\pi(s), a^*\sim\pi^*(s)}(\mathbb{1}_{a=a'}) = \sum_{a\in\mathcal{A}} p_{\pi}(a \mid s) p_{\pi^*}(a \mid s) \geq \rho\Big(\sum_{a\in\mathcal{A}} p_{\pi^*}(a \mid s)\Big) = \rho. \tag{A.67}$$

Then, we have:

$$\phi_{\pi,K}(s,a) = p_\pi(s \in \tau_{t+1:t+K+1} \mid s_t = s, a_t = a)\,, \tag{A.68}$$

$$= \mathop{\mathbb{E}}_{\pi}\left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \tag{A.69}$$

$$= \mathop{\mathbb{E}}_{s_{t+2},\dots,s_{t+K+1}\sim\pi} \mathop{\mathbb{E}}_{s_{t+1}\sim p(s_t,a_t)} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \tag{A.70}$$

$$= \mathop{\mathbb{E}}_{s_{t+3},\dots,s_{t+K+1}\sim\pi} \mathop{\mathbb{E}}_{\substack{a_{t+1}\sim\pi(s_{t+1})\\ s_{t+2}\sim p(s_{t+1},a_{t+1})}} \mathop{\mathbb{E}}_{s_{t+1}\sim p(s_t,a_t)} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \tag{A.71}$$

$$= \mathop{\mathbb{E}}_{s_{t+3},\dots,s_{t+K+1}\sim\pi} \mathop{\mathbb{E}}_{\substack{a_{t+1}\sim\pi(s_{t+1}),a'_{t+1}\sim\pi^*(s_{t+1})\\ s_{t+2}\sim p(s_{t+1},a_{t+1})}} \mathop{\mathbb{E}}_{s_{t+1}\sim p(s_t,a_t)} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \tag{A.72}$$

$$\geq \mathop{\mathbb{E}}_{s_{t+3},\dots,s_{t+K+1}\sim\pi} \mathop{\mathbb{E}}_{\substack{a_{t+1}\sim\pi(s_{t+1}),a^*_{t+1}\sim\pi^*(s_{t+1})\\ s_{t+2}\sim p(s_{t+1},a_{t+1})\\ s_{t+1}\sim p(s_t,a_t)}} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a, a_{t+1} = a^*_{t+1}\right]\mathbb{1}_{a_{t+1}=a^*_{t+1}}\,, \tag{A.73}$$

$$\geq \rho \mathop{\mathbb{E}}_{s_{t+3},\dots,s_{t+K+1}\sim\pi} \mathop{\mathbb{E}}_{s_{t+1},s_{t+2}\sim\pi^*} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \text{and iterating the same process,}, \tag{A.74}$$

$$\geq \rho^K \mathop{\mathbb{E}}_{s_{t+1},s_{t+2},\dots,s_{t+K+1}\sim\pi^*} \left[\mathbb{1}_{s\in\tau_{t+1:t+K+1}} \mid s_t = s, a_t = a\right], \tag{A.75}$$

$$\geq \rho^K \phi_K(s,a)\,. \tag{A.76}$$

We can conclude using Theorem 5.6: $\bar{\phi}_\pi(s,a) \geq \frac{\phi_\pi(s,a)}{2} \geq \frac{\phi_{\pi,K}(s,a)}{2} \geq \frac{\rho^K}{2}\phi_K(s,a)$.

□

## A.2 Additional details about reversibility-aware RL

### A.2.1 Learning a reversibility estimator

We illustrate how the reversibility estimator is trained in Fig. A.2. We remind the reader that it is a component that is specific to RAC. See Algorithm 23 for the detailed procedure of how to train it jointly with the standard precedence estimator and the RL agent.

### A.2.2 Pseudo-code for RAE and RAC

We give the pseudo-code for the online versions of RAE (Algorithm 17) and RAC (Algorithm 23). The rejection sampling policy $\bar{\pi}$ under approximate reversibility $\phi$ and threshold $\beta$

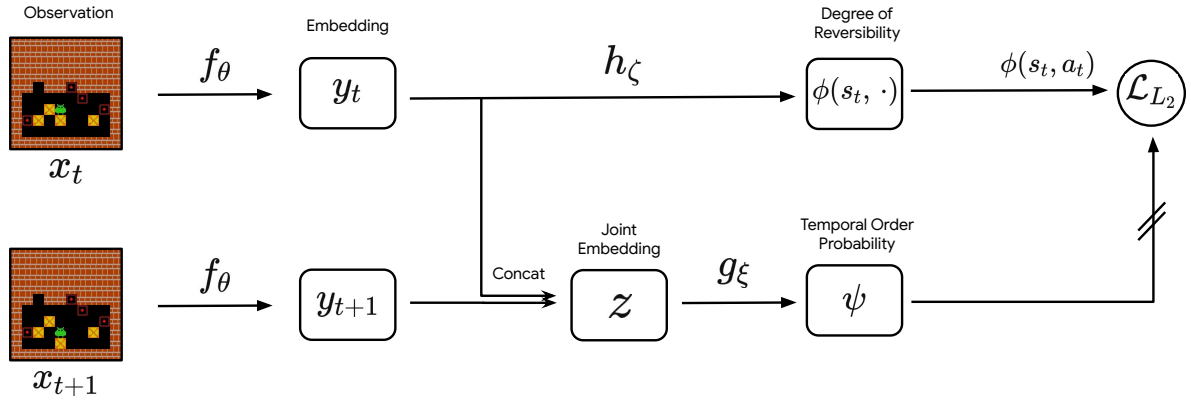



**Figure A.2** – The training procedure for the reversibility estimator used in RAC.

**Algorithm A.1:** RAE: Reversibility-Aware Exploration (online)

Initialize the agent weights $\Theta$ and number of RL updates per trajectory $k$;
Initialize the precedence classifier weights $\theta, \xi$, window size $w$, threshold $\beta$ and learning rate $\eta$;
Initialize the replay buffer $\mathcal{B}$;
**for** *each iteration* **do**
    `/* Collect interaction data and train the agent. */`
    Sample a trajectory $\tau = \{x_i, a_i, r_i\}_{i=1...T}$ with the current policy;
    Incorporate irreversibility penalties $\tau' = \{x_i, a_i, r_i + r_\beta(\psi_{\theta,\xi}(x_i, x_{i+1}))\}_{i=1...T}$;
    Store the trajectory in the replay buffer $\mathcal{B} \leftarrow \mathcal{B} \cup \tau$;
    Do $k$ RL steps and update $\Theta$;
    `/* Update the precedence classifier. */`
    **for** *each training step* **do**
        Sample a minibatch $\mathcal{D}_{batch}$ from $\mathcal{B}$;
        `/* Self-supervised precedence classification, loss in Eq.(A.78). */`
        $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{SSL}(\mathcal{D}_{batch})$;
        $\xi \leftarrow \xi - \eta \nabla_\xi \mathcal{L}_{SSL}(\mathcal{D}_{batch})$;
    **end**
**end**

**Algorithm A.2:** RAC: Reversibility-Aware Control (online)

Initialize the agent weights $\Theta$ and number of RL updates per trajectory $k$;
Initialize the precedence classifier weights $\theta, \xi$, window size $w$, threshold $\beta$ and learning rate $\eta$;
Initialize the reversibility estimator weights $\zeta$;
Initialize the replay buffer $\mathcal{B}$;
**for** *each iteration* **do**
    `/* Collect interaction data with the modified control policy and train the agent. */`
    Sample a trajectory $\tau$ under the rejection sampling policy $\bar{\pi}$ from eq.(A.77) ;
    Store the trajectory in the replay buffer $\mathcal{B} \leftarrow \mathcal{B} \cup \tau$;
    Do $k$ RL steps and update $\Theta$;
    `/* Update the precedence classifier. */`
    **for** *each training step* **do**
        Sample a minibatch $\mathcal{D}_{batch}$ from $\mathcal{B}$;
        `/* Self-supervised precedence classification, loss in Eq.(A.78). */`
        $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{SSL}(\mathcal{D}_{batch})$;
        $\xi \leftarrow \xi - \eta \nabla_\xi \mathcal{L}_{SSL}(\mathcal{D}_{batch})$;
    **end**
    `/* Update the reversibility estimator, loss in Eq.(A.79). */`
    **for** *each training step* **do**
        Sample a minibatch $\mathcal{D}_{batch}$ from $\mathcal{B}$;
        `/* Regression of the precedence classifier probabilities. */`
        $\zeta \leftarrow \zeta - \eta \nabla_\zeta \mathcal{L}_{L2}(\mathcal{D}_{batch}, \psi_{\theta,\xi})$;
    **end**
**end**

is expressed as follows:

$$\bar{\pi}(a|x) = \begin{cases} 0 & \text{if } \phi(x,a) < \beta \\ \pi(a|x)/Z & \text{otherwise, with } Z = \sum_{a' \in \mathcal{A}} \mathbb{1}\{\phi(x,a') \geq \beta\}\pi(a'|x) \end{cases}. \tag{A.77}$$

This is equivalent, on average, to sampling from the policy $\pi$ until an action that is reversible enough is found.

The loss we use to train the precedence estimator has the expression:

$$\mathcal{L}_{SSL}(\mathcal{D}_{batch}) = \frac{1}{|\mathcal{D}_{batch}|} \sum_{(x,x',y) \in \mathcal{D}_{batch}} -y \log\left(\psi_{\theta,\xi}(x,x')\right) + (1-y)\log\left(1 - \psi_{\theta,\xi}(x,x')\right), \tag{A.78}$$

where $y$ is the binary result of the shuffle, with value $1$ if observations were not shuffled (thus in the correct temporal order), and $0$ otherwise. Pairs of observations $(x, x')$ can be separated by up to $w$ timesteps.

The loss we use to train the reversibility estimator (in RAC only) has the expression:

$$\mathcal{L}_{L2}(\mathcal{D}_{batch}, \psi_{\theta,\xi}) = \frac{1}{2\,|\mathcal{D}_{batch}|} \sum_{(x,a,x') \in \mathcal{D}_{batch}} \left(\psi_{\theta,\xi}(x,x') - \phi_\zeta(x,a)\right)^2, \tag{A.79}$$

where $(x, a, x')$ are triples of state, action and next state sampled from the collected trajectories.

The offline versions of both RAE and RAC can be derived by separating each online algorithm into two parts: 1) training the precedence classifier (and the reversibility estimator for RAC), which is achieved by removing the data collection and RL steps and by using a fixed replay buffer; and 2) training the RL agent, which is the standard RL procedure augmented with modified rewards for RAE, and modified control for RAC, using the classifiers learned in the first part without fine-tuning.

## A.3 Experimental details

### A.3.1 Reward-free reinforcement learning

**Cartpole.** The observation space is a tabular 4-dimensional vector: (cart position $x$, cart velocity $\dot{x}$, pole angle $\theta$, pole velocity $\dot{\theta}$). The discrete action space consists of applying a force left or right. The episode terminates if the pole angle is more than $\pm 12°$ ($|\theta| \leq 0.209$ radians), if the cart position is more than $\pm 2.4$, or after 200 time-steps. It is considered solved when the average return is greater than or equal to 195.0 over 100 consecutive trials.

**Architecture and hyperparameters.** The reversibility network inputs a pair of observations and produces an embedding by passing each one into 2 fully connected layers of size 64 followed by ReLU. The two embeddings are concatenated, and projected into a scalar followed by a sigmoid activation. We trained this network doing 1 gradient step every 500 time steps, using the Adam optimizer (Diederik P Kingma and Ba, 2015a) and a learning rate of 0.01. We used batches of 128 samples, that we collected from a replay buffer of size 1 million. The penalization threshold $\beta$ was fine-tuned over the set $[0.5, 0.6, 0.7, 0.8, 0.9]$ and eventually set to 0.7. We notice informally that it was an important parameter. A low threshold could lead to over penalizing the agent leading the agent to terminate the episode as soon as possible, whereas a high threshold could slow down the learning.

Regarding PPO, both the policy network and the value network are composed of two hidden layers of size 64. Training was done using Adam and a learning rate of 0.01. Other PPO hyperparameters were defaults in Raffin et al. (2019), except that we add an entropy cost of 0.05.

### A.3.2 Learning reversible policies

**Environment.** The environment consists of a 10 × 10 pixel grid. It contains an agent, represented by a single blue pixel, which can move in four directions: up, down, left, right. The pink pixel represents the goal, green pixels grass and grey pixels a stone path. Stepping on grass spoils it and the corresponding pixel turns brown, as shown in Fig. 5.5b. A level terminates after getting to the goal, or after 120 timesteps. Upon reaching the goal, the agent receives a reward of +1, every other action being associated with 0 reward.

**Architecture and hyperparameters.** The reversibility network takes a pair of observations as input and produces an embedding by passing each observation through 3 identical convolutional layers of kernel size 3, with respectively 32, 64 and 64 channels. The convolutional outputs are flattened, linearly projected onto 64 dimensional vectors and concatenated. The resulting vector is projected into a scalar, which goes through a final sigmoid activation.

As done for Cartpole, we trained this network doing 1 gradient step every 500 time steps, using the Adam optimizer with a learning rate of 0.01. We used minibatches of 128 samples, that we collected from a replay buffer of size 1M. The penalization threshold $\beta$ was set to 0.8, and the intrinsic reward was weighted by 0.1, such that the intrinsic reward was equal to $-0.1\,\mathbb{1}_{\psi(s_t,s_{t+1})>0.8}\,\psi(s_t, s_{t+1})$.

For PPO, both the policy network and the value network are composed of 3 convolutional layers of size 32, 64, 64. The output is flattened and passed through a hidden layer of size 512. Each layers are followed by a ReLU activation. Policy logits (size 4) and baseline function (size

1) were produced by a linear projection. Other PPO hyperparameters were defaults in Raffin et al. (2019), except that we add an entropy cost of 0.05.

### A.3.3 Sokoban

We use the Sokoban implementation from Schrader (2018). The environment is a 10x10 grid with a unique layout for each level. The agent receives a -0.1 reward at each timestep, a +1 reward when placing a box on a target, a -1 reward when removing a box from a target, and a +10 reward when completing a level. Observations are of size $(10, 10, 3)$. Episodes have a maximal length of 120, and terminate upon placing the last box on the remaining target. At the beginning of each episode, a level is sampled uniformly from a set of 1000 levels, which prevents agents from memorizing puzzle solutions. The set is obtained by applying random permutations to the positions of the boxes and the position of the agent, and is pre-computed for efficiency. All levels feature four boxes and targets.

We use the distributed IMPALA implementation from the Acme framework (Hoffman et al., 2020) as our baseline agent in these experiments. The architecture and hyperparameters were obtained by optimizing for sample-efficiency on a single held-out level. Specifically, the agent network consists of three 3x3 convolutional layers with 8, 16 and 16 filters and strides 2, 1, and 1 respectively; each followed by a ReLU nonlinearity except the last one. The outputs are flattened and fed to a 2-layer feed-forward network with 64 hidden units and ReLU nonlinearities. The policy and the value network share all previous layers, and each have a separate final one-layer feed-forward network with 64 hidden units and ReLU nonlinearities as well. Regarding agent hyperparameters, we use 64 actors running in parallel, a batch size of 256, an unroll length of 20, and a maximum gradient norm of 40. The coefficient of the loss on the value is 0.5, and that of the entropic regularization 0.01. We use the Adam optimizer with a learning rate of 0.0005, a momentum decay of 0 and a variance decay of 0.99.

The precedence estimator network is quite similar: it consists of two 3x3 convolutional layers with 8 filters each and strides 2 and 1 respectively; each followed by a ReLU nonlinearity except the last one. The outputs are flattened and fed to a 3-layer feed-forward network with 64 hidden units and ReLU nonlinearities, and a final layer with a single neuron. We use dropout in the feed-forward network, with a probability of 0.1. Precedence probabilities are obtained by applying the sigmoid function to the outputs of the last feed-forward layer. The precedence estimator is trained offline on 100k trajectories collected from a random agent. It is trained on a total of 20M pairs of observations sampled with a window of size 15, although we observed identical performance with larger sizes (up to 120, which is the maximal window size). We use the Adam optimizer with a learning rate of 0.0005, a momentum decay of 0.9, a variance decay of 0.999. We also use weight decay, with a coefficient of 0.0001. We use a threshold $\beta$ of 0.9. We selected hyperparameters based on performance on validation data.

### A.3.4 Reversibility-aware control in cartpole+

**Learning $\psi$.** The model architecture is the same as described in Appendix A.3.1. The training is done offline using a buffer of 100k trajectories collected using a random policy. State pairs are fed to the classifier in batches of size 128, for a total of 3M pairs. We use the Adam optimizer with a learning rate of 0.01. We use a window $w$ equal to 200, which is the maximum number of timesteps in our environment.

**Learning $\phi$.** We use a shallow feed-forward network with a single hidden layer of size 64 followed by a ReLU activation. From the same buffer of trajectories used for $\psi$, we sample 100k transitions and feed them to $\phi$ in batches of size 128. As before, training is done using Adam and a learning rate of 0.01.

### A.3.5 DQN and M-DQN in cartpole+

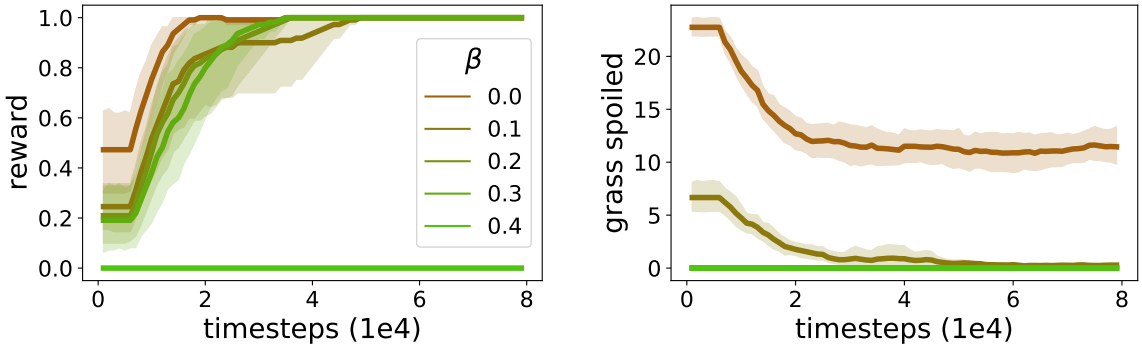
We use the same architecture for DQN and M-DQN. The network is a feed-forward network composed of two hidden layers of size 512 followed by ReLU activation. In both cases, we update the online network every 4 timesteps, and the target network every 400 timesteps. We use a replay buffer of size 50k, and sample batches of size 128. We use the Adam optimizer with a learning rate of 0.001.

We train both algorithms for 2M timesteps. We run an evaluation episode every 1000 timesteps, and report the maximum performance encountered during the training process. We perform a grid search for the discount factor $\gamma \in [0.99, 0.999, 0.9997]$, and for M-DQN parameters $\alpha \in [0.7, 0.9, 0.99]$ and $\tau \in [0.008, 0.03, 0.1]$. The best performances were obtained for $\alpha = 0.9, \tau = 0.03$, and $\gamma = 0.99$.

### A.3.6 Reversibility-aware control in Turf

**Learning $\psi$.** We use the same model architecture as in RAE (Appendix A.3.2), and the same offline training procedure that was used for Cartpole+ (Appendix A.3.4). The window $w$ was set to 120, which is the maximum number of steps in Turf.

**Learning $\phi$.** The architecture is similar to $\psi$, except for the last linear layers: the output of the convolutional layers is flattened and fed to a feed-forward network with one hidden layer of size 64 followed by a ReLU. Again, we used the exact same training procedure as in the case of Cartpole+ (Appendix A.3.4).




(a) (b)

**Figure A.3 – (a):** Reward learning curve for PPO+RAC and several thresholds $\beta$ (average over 10 random seeds). A threshold of 0 means actions are never rejected, and corresponds to the standard PPO. **(b):** Number of irreversible side-effects (grass pixels stepped on). For $\beta$ between 0.2 and 0.4, 0 side-effects are induced during the whole learning.

### A.3.7 Safety and performance trade-off in Turf

We investigate the performance-to-safety trade-off induced by reversibility-awareness in Turf. In Fig. A.3a, we see that the agent is not able to reach the goal when the threshold is greater than 0.4: with too high a threshold, every action leading to the goal could be rejected. We also see that it solves the task under lower threshold values, and that lowering $\beta$ results in faster learning. On the other hand, Fig. A.3b shows that achieving zero irreversible side-effects during the learning is only possible when $\beta$ is greater than 0.2. In this setting, the optimal thresholds are thus between 0.2 and 0.3, allowing the agent to learn the new task while eradicating every side-effect.

This experiment gives some insights on how to tune $\beta$ in new environments. It should be initialized at 0.5 and decreased progressively, until the desired agent behaviour is reached. This would ensure that the chosen threshold is the maximal threshold such that the environment can be solved, while having the greatest safety guarantees.

## A.4 Stochastic MDPs

To study how reversibility-awareness helps in stochastic MDPs, we use a 2D cliff-walking gridworld where stochasticity comes from the wind: additionally to its move, the agent is pushed towards the cliff with a fixed probability. The agent gets a +1 reward for each timestep it stays alive, with a maximum of 250 timesteps. A reversibility-aware agent with a well calibrated threshold should avoid most moves that push it towards the cliff. We use a 6x8 grid, with a maximum of 250 timesteps per episode, and report results averaged over 5000 runs. We provide

two tables: Table A.1 with the average scores of a random policy and Table A.2 with the average scores of a random policy equipped with RAC. Rows correspond to varying stochasticity and columns to varying threshold values.

**Table A.1** – Scores for a random policy in the 2D cliff-walking gridworld, where $p$ is the probability of being pushed by the wind. Higher is better.

| $p$ \ Threshold | 0. | 0.1 | 0.2 | 0.3 | 0.4 |
|---|---|---|---|---|---|
| 0. | 57.5 | 57.7 | 61.2 | 58.2 | 57.7 |
| 0.1 | 29.8 | 28.8 | 29.5 | 30.2 | 29.6 |
| 0.2 | 18.6 | 18.5 | 19.3 | 18.9 | 18.8 |
| 0.3 | 13.4 | 13.3 | 13.9 | 13.6 | 13.4 |
| 0.4 | 10.5 | 10.7 | 10.4 | 10.2 | 10.2 |

**Table A.2** – Scores for a random policy with RAC in the 2D cliff-walking gridworld, where $p$ is the probability of being pushed by the wind. Higher is better.

| $p$ \ Threshold | 0. | 0.1 | 0.2 | 0.3 | 0.4 |
|---|---|---|---|---|---|
| 0. | 59.1 | 250.0 | 250.0 | 250.0 | 250.0 |
| 0.1 | 29.2 | 56.0 | 56.3 | 80.2 | 248.5 |
| 0.2 | 18.7 | 26.7 | 29.2 | 85.8 | 238.6 |
| 0.3 | 13.2 | 16.8 | 19.6 | 77.6 | 250.0 |
| 0.4 | 10.4 | 12.5 | 24.9 | 152.2 | 250.0 |

We can notice that:

- RAC significantly improves performance (reaching maximum or near-maximum score),
- a well-tuned threshold value is crucial to get decent performance with RAC,
- the optimal threshold increases with the stochasticity of the environment (but seems to quickly converge).

# Appendix B

# Complements on Chapter 6

We organize the supplementary material as follows. In Appendix B.1 we include the proofs of results from the main text, as well as additional details about the proposed algorithms, including pseudo-code. In Appendix B.2 we detail our experimental procedure, including hyperparameters for all methods used.

## B.1 Technical elements and proofs

### B.1.1 Proof of Proposition 6.3

**Proposition** (Proposition 6.3)**.** *If we have two pairs of equivalent sequences over $\mathcal{M}$,* i.e. *$w_1$, $w_2$, $w_3$, $w_4 \in \mathcal{A}^\star$ such that*

$$w_1 \sim w_2$$

$$w_3 \sim w_4$$

*then the concatenation of the sequences are also equivalent sequences:*

$$w_1 \cdot w_3 \sim w_2 \cdot w_4$$

*Proof.* For any $s \in \mathcal{S}$, we have $T(s, w_1) = T(s, w_2)$ as $w_1 \sim w_2$. We apply the same property for $w_3$ and $w_4$ on the state $T(s, w_1)$:

$$T(T(s, w_1), w_3) = T(T(s, w_2), w_4)$$
$$T(s, w_1.w_3) = T(s, w_2.w_4)$$

Therefore $w_1.w_3 \sim w_2.w_4$. □

### B.1.2 Proof of theorem 6.4

**Theorem.** *Given an equivalence set $\Omega$, $\sim_\Omega$ is an equivalence relationship. Furthermore, for $v, w \in \mathcal{A}^\star$, $v \sim_\Omega w \Rightarrow v \sim w$.*

**$\sim_\Omega$ is an equivalence relation** Let $u, v, w \in \mathcal{A}^\star$.

*Proof.*

- We immediately have $v \sim_\Omega^1 v$ by choosing $v_1 = \Lambda$ in (6.3), and therefore $v \sim_\Omega v$, thus $\sim_\Omega$ is reflexive.
- It is clear from its definition that $\sim_\Omega^1$ is symmetric, as $\sim$ is symmetric. Then, we suppose that $v \sim_\Omega w$. We have $n \in \mathbb{N}$ and $v_1, \dots, v_n \in \mathcal{A}^\star$ such that $v \sim_\Omega^1 v_1 \sim_\Omega^1 \cdots \sim_\Omega^1 v_n \sim_\Omega^1 w$, therefore $w \sim_\Omega^1 v_n \sim_\Omega^1 \cdots \sim_\Omega^1 v_1 \sim_\Omega^1 v$, thus $w \sim_\Omega v$. Hence $\sim_\Omega$ is symmetric.
- If $u \sim_\Omega v$ and $v \sim_\Omega w$, we have $n_1, n_2 \in \mathbb{N}$, and $u_1, \dots, u_{n_1} \in \mathcal{A}^\star$, $v_1, \dots, v_{n_2} \in \mathcal{A}^\star$, such that $u \sim_\Omega^1 u_1 \sim_\Omega^1 \cdots \sim_\Omega^1 u_{n_1} \sim_\Omega^1 v$ and $v \sim_\Omega^1 v_1 \sim_\Omega^1 \cdots \sim_\Omega^1 v_{n_2} \sim_\Omega^1 w$. It is then clear that $u \sim_\Omega^1 u_1 \sim_\Omega^1 \cdots \sim_\Omega^1 u_{n_1} \sim_\Omega^1 v \sim_\Omega^1 v_1 \sim_\Omega^1 \cdots \sim_\Omega^1 v_{n_2} \sim_\Omega^1 w$, and thus $u \sim_\Omega w$. Therefore $\sim_\Omega$ is transitive.

The relation $\sim_\Omega$ is reflexive, symmetric and transitive. Therefore it is an equivalence relation. □

**$\sim_\Omega$ implies $\sim$**

*Proof.* Let $v, w \in \mathcal{A}^\star$. From Proposition 6.3, we immediately get $v \sim_\Omega^1 w \Rightarrow v \sim_\mathcal{M} w$. Then we can prove by immediate induction that $\forall n \in \mathbb{N}, v_1, \dots, v_n \in \mathcal{A}^\star, v \sim_\Omega^1 v_1 \sim_\Omega^1 \cdots \sim_\Omega^1 v_n \sim_\Omega^1 w \Rightarrow v \sim w$, from which we deduce $v \sim_\Omega w$ implies $v \sim w$. □

### B.1.3 Graph construction algorithm

We present in Algorithm 1 an overview of the graph construction algorithm. It takes as input the action set $A$, the sequence equivalence set $\Omega$, and the desired depth $d$, and outputs a DAG. Informally, it starts from a graph $\mathcal{G} = (V, E)$ reduced to a root state $\{0\}$ and iteratively expands $\mathcal{G}$ until a distance $L$ to the root is reached. For a node $n \in V$ we store in $\mathcal{E}(v)$ sequences which reach $v$, and are prefixes of sequences of $\Omega$. When expanding a state $v \in V$ using an action $a \in A$ (Line. 13), we look at every partial sequence $s \in \mathcal{E}(v)$. If $s.a$ is in $\Omega$, it means that we

**Algorithm B.1:** Graph Construction

**Input** Action set $A$;
**Input** Equivalence set $\Omega$;
**Input** Maximum tree depth $d$;
Initialize the graph $\mathcal{G} = (V, E)$ with $V = \{0\}$ and $V = \emptyset$ ;
Initialize the set of states to expand $\mathcal{S} = \{0\}$ ;
Initialize the current tree depth $l = 0$;
Initialize a dictionary $\mathcal{E}$ which stores partial sequences of $\Omega$ for each state of $V$ ;
**while** $l < d$ **and** $\mathcal{S} \neq \emptyset$ **do**
    newStates = {} ;
    **for** *each state in* $\mathcal{S}$ **do**
        **for** *each action in* $A$ **do**
            `/* create a node corresponding to` $T(state, action)$ `*/`
            newState = `expandNode`(state, action, $\Omega$, $\mathcal{E}$) ;
            **if** *newState not in* $V$ **then**
                `/* Because of sequence redundancies, the state may already appear in the graph. */`
                $V \leftarrow V \cup$ {newState} ;
                newStates $\leftarrow$ newStates $\cup$ {newState} ;
            **end**
            $E \leftarrow E \cup \{$(state, newState)$\}$ ;
            `/* Update the equivalences` $\mathcal{E}$`(newState) to account for the new ways of reaching newState */`
            $\mathcal{E} \leftarrow$ `UpdateDic`(newState, $\mathcal{E}$, $\Omega$) ;
        **end**
    **end**
    $l \leftarrow l + 1$;
    $\mathcal{S} \leftarrow$ newStates ;
**end**
`/* Prune edges such that the resulting graph is a DAG. */`
$T =$ `GraphToDAG`($\mathcal{G}$) ;
**Output** DAG $\mathcal{G}$;

have found a redundant sequence. If the equivalent sequence has already been computed, it means that a node $u$ representing $T(v, s.a)$ has previously been added in $\mathcal{G}$. Otherwise, we add a new node $u$. In both case, we update the equivalences $\mathcal{E}(u)$ to account for the new ways of reaching $u$ (Line. 21).

### B.1.4 Graph construction complexity

As shown in Section 6.3.2, constructing the graph necessitates three intricate loops: The first one goes over every internal node $n \in V$, the second one loops over the set of actions $\mathcal{A}$, and the last one loops over every partial sequence which allows to reach $v$ from a parent node. Inside these three loops, one has to compare the partial sequence with every sequence of $\Omega$. As sequence length in $\Omega$ can be bounded by $d$, the complexity cost inside the tree loops is bounded by $\mathcal{O}(|\Omega|d)$. The total complexity is therefore lower than $\mathcal{O}(|V||\mathcal{A}||\mathcal{A}|^{d-1}|\Omega|d) = \mathcal{O}(|V||\mathcal{A}|^{d}|\Omega|d)$. As $|V| \leq |\mathcal{A}|^d$, the complexity can also be bounded by $\mathcal{O}(|\mathcal{A}|^{2d}|\Omega|d)$.

### B.1.5 Modified DQN

Our modified version of the DQN algorithm can be found at Algorithm 20.

**Algorithm B.2:** Modified DQN

```
Initialize replay memory 𝒟 and Q-networks Q_θ and Q_θ';
Determine local-dynamics graph 𝒢 and the associated optimal exploration policy π*;
for episode = 1 to M do
    Initialize new episode;
    for t = 1 to T do
        ε ← set new ε value with ε-decay (ε usually anneals linearly or is constant);
        Initialize at empty sequence v ← Λ;
        if 𝒰([0,1]) < ε then
            Sample exploring action a_t ∼ π*(v, ·);
        else
            Select greedy action a_t;
        end
        v ← v.a_t (append a_t to the end of sequence v);
        Execute a_t and observe next state s_{t+1} and reward r_t ;
        Store (s_t, a_t, r_t, s_{t+1}) in replay buffer 𝒟 Update θ and θ' normally with
          minibatches from replay buffer 𝒟;
        if Length(v)=d then
            Reset v ← Λ;
        end
    end
end
```

### B.1.6 Possible extension to the stochastic case

In this section we discuss the possibility of extending EASEE to the case of MDPs with stochastic transitions. EASEE relies on three components: the formalization of action sequence equivalences (Def. 6.1), the construction of a local-dynamics graph (Section 6.3.2), and the construction of a local exploration policy by solving a convex problem (Section 6.4.2). We now detail for each step the necessary changes to adapt EASEE to $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \gamma)$, a MDP with stochastic dynamics.

- **Action Sequence equivalences**: the difference with the deterministic case here is that given an action $a \in \mathcal{A}$ and a state $s \in \mathcal{S}$, $T(s, a)$ is not a state but a distribution over the set of states $\mathcal{S}$. Therefore every equality considered in Section 6.3.1 has now to be understood not as an equality between two states but between two distributions. Other than this the formalism can be kept identical. Intuitively, two sequences of actions are equivalent if they lead to the same state distribution from any given state, *i.e.* if they produce the same effect everywhere.
- **Local-Dynamics Graph**: Here, the formalism can again be kept identical. A node in the local-dynamics graph will not represent a state anymore, but rather a distribution over $\mathcal{S}$.
- **Local Exploration Policy**: Solving directly the objective given in (6.4) would lead to maximize the diversity among state distributions encountered. As is, it would not necessarily lead to a better diversity among states, as two different distributions can have an almost similar support. Therefore, adapting EASEE to a stochastic setting would require encoding additional priors about the distributions represented by the nodes of the local-dynamics graph, which we leave for future work. If we suppose that the distributions encountered have disjoint supports, and that their entropy is the same, EASEE can be applied without modification.

## B.2 Experimental details

### B.2.1 Gridworlds

We tested EASEE on the DoorKey task. An illustration of the initial state is given in Fig. B.1. The agent is represented by the red triangle. The yellow key is necessary to open the yellow door. The two room are respectively $12 \times 17$ and $4 \times 17$ grids. The agent has $3249$ timesteps to reach the goal and receive a reward of $1$ before the environment is reset.

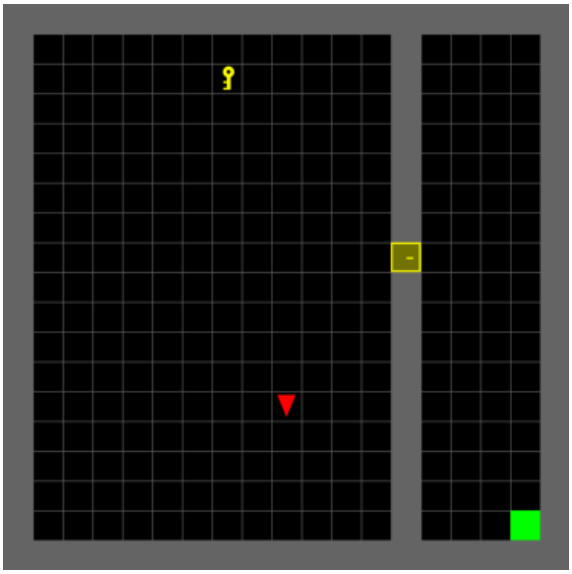


**Figure B.1** – Example of initial state of DoorKey environment.

### B.2.2 Catcher

The paddle is $1$ block wide. The environment is $60$ blocks wide and $30$ blocks high. The ball and the paddle both move at a rate of $1$ block per timestep, so each episode lasts $30$ timesteps.

We use the same architecture for the DQN with and without EASEE. Each observation is a $60 \times 30$ image. The feature extractor network is a CNN composed of 3 convolution layers with kernel size 3 followed by ReLU activation. In both cases, we update the online network every 4 timesteps, and the target network every $10^3$ timesteps. We use a replay buffer of size $10^4$, and sample batches of size 32. We use the Adam optimizer with a learning rate of $10^{-4}$.

We train for $3.10^5$ timesteps. The exploration parameter $\varepsilon$ is linearly annealed from 1 down to 0.05 over 20% of the training period. Other DQN hyperparameters were defaults in Raffin et al. (2019).

### B.2.3 Freeway

**Environment** In Freeway, the agent has to cross a road with multiple lanes without getting hit by the cars. It only receives a reward when it safely reaches the other side of the road. An illustration is given in Fig. B.2. The agent is represented by the yellow chicken.

We use the default preprocessing in Raffin et al. (2019), which follows guidelines of M. G. Bellemare et al. (2013). More precisely, the environment is initialized with a random number of up to 30 no-op actions. The frame is recast as a $84 \times 84 \times 3$ image, and the number of frames

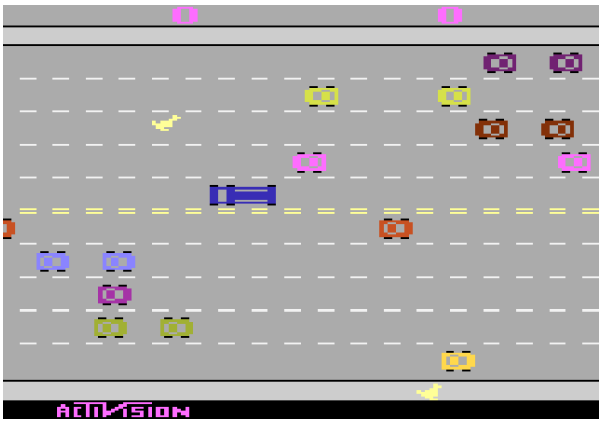



**Figure B.2** – The Freeway environment from Atari 2600.

to skip between each observation is set to 4. The reward is scaled to [-1, 1]. An observation corresponds to 4 stacked game frames.

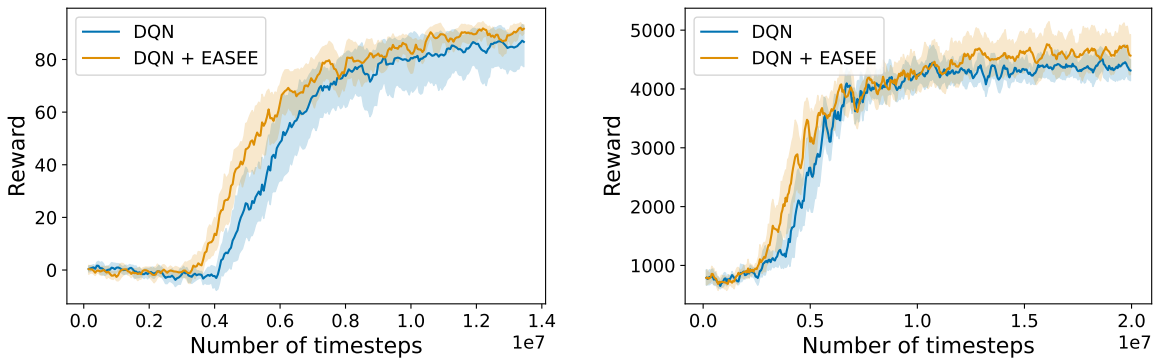
**Architecture and hyperparameters** We use the same architecture for the DQN with and without EASEE. Input images first go through a convolutional neural network, with the same architecture as in Volodymyr Mnih, Kavukcuoglu, et al. (2015). We update the online network every 4 timesteps, and the target network every $10^3$ timesteps. We use a replay buffer of size $10^5$, and sample batches of size 32. We use the Adam optimizer with a learning rate of $10^{-4}$.

We train for $10^7$ timesteps. The exploration parameter $\varepsilon$ is linearly annealed from 1 down to 0.01 over 10% of the training period, which are the default in Raffin (2018) for Atari games. Other DQN hyperparameters were defaults in Raffin et al. (2019).

### B.2.4 Additional experiments

**Environments** We experimented EASEE on two other Atari environments, where the action sequence structures are less straight-forward. The three environments are preprocessed as explained in AppendixB.2.3.

- **Boxing**: This game shows a top-down view of two boxers. The player can move in all four directions, and punch his opponent (pressing the "`FIRE`" button). The action space is composed of 18 actions : `NOOP`, `FIRE`, `UP`, `RIGHT`, `LEFT`, `DOWN`, `UPRIGHT`, `UPLEFT`, `DOWNRIGHT`, `DOWNLEFT`, `UPFIRE`, `RIGHTFIRE`, `LEFTFIRE`, `DOWNFIRE`, `UPRIGHTFIRE`, `UPLEFTFIRE`, `DOWNRIGHTFIRE`, `DOWNLEFTFIRE`. We incorporated priors by decomposing actions, in the form of `UPRIGHT` $\sim$ `UP.RIGHT`, `UPLEFT` $\sim$ `UP.LEFT`, `UPRIGHTFIRE` $\sim$ `UPRIGHT . FIRE`, `UPLEFTFIRE` $\sim$ `UPLEFT . FIRE`, *etc.*




(**a**) Boxing (**b**) Carnival

**Figure B.3** – Performances of DQN and DQN + EASEE on the Atari 2600 games Boxing, Carnival. A 95% confidence interval over 10 random seeds is shown.

- **Carnival**: The goal of the game is to shoot at targets, which include rabbits, ducks, owls, scroll across the screen in alternating directions, and sometimes come at the player. The player can only move in 1 direction, such that the action space is composed of 6 actions: [`NOOP`, `FIRE`, `RIGHT`, `LEFT`, `RIGHTFIRE`, `LEFTFIRE`]. As `NOOP` is not an useful action, EASEE could get an edge simply by adding the equivalence `NOOP` $\sim \Lambda$. For a fair comparison, we restricted the action space to meaningful actions by removing `NOOP` for both EASEE and the baseline. We limited ourselves to the commutative property of `RIGHT` and `LEFT`: `RIGHT` . `LEFT` $\sim$ `LEFT` . `RIGHT`.

**Architecture and hyperparameters** We use the same architecture and the exact same DQN parameters as in AppendixB.2.3. In all three environments, EASEE is used with a depth of 4.

**Results** We can see the results on Fig.B.3. We can see a slight gain for Boxing, and a marginal improvement for Carnival. This can come from various factors:

- When the number of action sequence equivalences considered is small compared to the number of actions, as it is the case for Carnival, the exploration policy computed with EASEE is very much like a uniform policy. It logically makes its performances converge toward those of a standard DQN.
- The action sequence equivalences considered here are only approximately true. In boxing, it is only approximately true that `UPRIGHT` $\sim$ `UP`.`RIGHT` for example. In Carnival, `RIGHT` and `LEFT` commute as long as the player is not at the edges of the screen, in which case `RIGHT` or `LEFT` could have no effect. In both cases, this induces a bias that may harm performance.

# Appendix C

# Complements on Chapter 7

## C.1 Additional details on Poppy

### C.1.1 Number of parameters

Table C.1 shows the total number of parameters of our models as a function of the population size. Since the decoder represents less than $10\%$ of the parameters, scaling the population size can be done efficiently. For instance, a population of 16 agents roughly doubles the model size.

**Table C.1** – Number of model parameters for different population sizes.

| | | | Population size | | | | |
|---|---|---|---|---|---|---|---|
| | Encoder | Decoder | 1 | 4 | 8 | 16 | 32 |
| Parameters | 1,190,016 | 98,816 | 1,288,832 | 1,585,280 | 1,980,544 | 2,771,072 | 4,352,128 |
| Extra parameters | - | - | 0% | 23% | 54% | 115% | 238% |

### C.1.2 Training details

In Section 7.3.2 (see "Training Procedure"), we described that Poppy consists of two phases. In a nutshell, the first phase consists of training our model in a single-agent setting (*i.e.*, an encoder-decoder model with a single decoder head), whereas the second phase consists of keeping the encoder and cloning the previously trained decoder $K$ times (where $K$ is the number of agents) and specialize them using the $J_{\text{poppy}}$ objective. Algorithm C.1 shows the low-level implementation details of the training of the population (*i.e.*, Phase 2) omitted in Algorithm 7.1 for simplicity; namely, given $K$ agents and $P$ starting points, $P \times K$ trajectories are rolled out for each instance, among which only $P$ are effectively used for training.

**Algorithm C.1:** Poppy training with starting points

**Inputs:** problem distribution $\mathcal{D}$, number of starting points per instance $P$, number of agents $K$, batch size $B$, number of training steps $H$, a pretrained encoder $h_\psi$ and a set of $K$ decoders $q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}$

**for** *step 1 to* $H$ **do**

$\rho_i \leftarrow \texttt{Sample}(\mathcal{D})\ \forall i \in 1, \dots, B$ ;

$\alpha_{i,1}, \dots, \alpha_{i,P} \leftarrow \texttt{SelectStartPoints}(\rho_i, P)\ \forall i \in 1, \dots, B$ ;

$\tau^k_{i,p} \leftarrow \texttt{Rollout}(\rho_i, \alpha_{i,p}, h_\psi, q_{\phi_k})\ \forall i \in 1, \dots, B, \forall p \in 1, \dots, P, \forall k \in 1, \dots, K$ ;

$b^k_i \leftarrow \frac{1}{P} \sum_p R(\tau^k_{i,p})$ ;

$k^*_{i,p} \leftarrow \arg\max_{k \leq K} R(\tau^k_{i,p})\ \forall i \in 1, \dots, B, \forall p \in 1, \dots, P$ ;

`/* Select the best agent per (instance, starting point). */`

$\nabla L(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) \leftarrow -\frac{1}{BP} \sum_{i,p} (R(\tau^{k^*_{i,p}}_{i,p}) - b^{k^*_{i,p}}_i) \nabla \log p_{\psi, \phi_{k^*_{i,p}}}(\tau^{k^*_{i,p}}_{i,p})$ ;

`/* Propagate gradients through these only. */`

$(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) \leftarrow (h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K}) - \alpha \nabla L(h_\psi, q_{\phi_1}, q_{\phi_2}, \dots, q_{\phi_K})$ ;

## C.2 Mathematical Elements

### C.2.1 Gradient derivation

We recall that the population objective for $K$ is defined as:

$$J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_n) \doteq \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1 \sim \pi_{\theta_1}, \dots, \tau_K \sim \pi_{\theta_K}} \max\left[R(\tau_1), \dots, R(\tau_K)\right].$$

**Theorem** (Policy gradient for the population objective)**.** *The gradient of the population objective is given by:*

$$\nabla J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_n) = \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1 \sim \pi_{\theta_1}, \dots, \tau_K \sim \pi_{\theta_K}} (R(\tau_{i^*}) - R(\tau_{i^{**}})) \nabla \log p_{\theta_{i^*}}(\tau_{i^*}), \tag{C.1}$$

*where:*

$$i^* = \arg\max\left[R(\tau_1), \dots, R(\tau_K)\right], \qquad \text{(index of the best trajectory)}$$
$$i^{**} = \arg\text{second}\max\left[R(\tau_1), \dots, R(\tau_K)\right], \qquad \text{(index of the second best trajectory)}$$

*Proof.* We first derive the gradient with respect to $\theta_1$ for convenience. As all the agents play a symmetrical role in the objective, the same procedure can be applied to any index.

$$\nabla_{\theta_1} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) = \nabla_{\theta_1} \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1 \sim \pi_{\theta_1}, \dots, \tau_K \sim \pi_{\theta_K}} \max_{i \in \{1,2,\dots,K\}} \left[R(\tau_i)\right] \tag{C.2}$$

$$= \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1, \dots, \tau_K} \max_{i \in \{1,2,\dots,K\}} \left[R(\tau_i)\right] \nabla_{\theta_1} \log p(\tau_1, \dots, \tau_K) \tag{C.3}$$

$$= \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1, \dots, \tau_K} \max_{i \in \{1,2,\dots,K\}} \left[R(\tau_i)\right] \nabla_{\theta_1} \log(\pi_{\theta_1}(\tau_1) \dots \pi_{\theta_K}(\tau_K)) \tag{C.4}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} (\log \pi_{\theta_1}(\tau_1) + \cdots + \log \pi_{\theta_K}(\tau_K)) \tag{C.5}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \tag{C.6}$$

We also have for any problem instance $\rho$ and any trajectories $\tau_2, \dots, \tau_K$:

$$\mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1) = \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1) \tag{C.7}$$

$$= \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \mathbb{E}_{\tau_1\sim\pi_{\theta_1}} \frac{\nabla_{\theta_1} \pi_{\theta_1}(\tau_1)}{\pi_{\theta_1}(\tau_1)} \tag{C.8}$$

$$= \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \sum_{\tau_1} \nabla_{\theta_1} \pi_{\theta_1}(\tau_1) \tag{C.9}$$

$$= \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} \sum_{\tau_1} \pi_{\theta_1}(\tau_1) \tag{C.10}$$

$$= \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} 1 = 0 \tag{C.11}$$

$$\tag{C.12}$$

Intuitively, $\max_{i\in\{2,\dots,K\}}[R(\tau_i)]$ does not depend on the first agent, so this derivation simply shows that $\max_{i\in\{2,\dots,K\}}[R(\tau_i)]$ can be used as a baseline for training $\theta_1$.

Subtracting this to the quantity obtained in Equation C.6, we have:

$$\nabla_{\theta_1} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) = \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1,\dots,\tau_K} \max_{i\in\{1,2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \tag{C.13}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_2,\dots,\tau_K}\mathbb{E}_{\tau_1} \max_{i\in\{1,2,\dots,K\}} [R(\tau_i)] \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \tag{C.14}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_2,\dots,\tau_K}\mathbb{E}_{\tau_1} \left( \max_{i\in\{1,2,\dots,K\}} [R(\tau_i)] - \max_{i\in\{2,\dots,K\}} [R(\tau_i)] \right) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \tag{C.15}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_2,\dots,\tau_K}\mathbb{E}_{\tau_1} \mathbb{1}_{i^*=1} (R(\tau_1) - R(\tau_{i^{**}})) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1), \tag{C.16}$$

$$= \mathbb{E}_{\rho\sim\mathcal{D}}\mathbb{E}_{\tau_1,\dots,\tau_K} \mathbb{1}_{i^*=1} (R(\tau_{i^*}) - R(\tau_{i^{**}})) \nabla_{\theta_1} \log \pi_{\theta_1}(\tau_1). \tag{C.17}$$

$$\tag{C.18}$$

Equation (C.16) comes from the fact that $\left(\max_{i\in\{1,2,\dots,K\}}[R(\tau_i)] - \max_{i\in\{2,\dots,K\}}[R(\tau_i)]\right)$ is 0 if the best trajectory is not $\tau_1$, and $R(\tau_1) - \max_{i\in\{2,\dots,K\}}[R(\tau_i)] = R(\tau_1) - R(\tau_{i^{**}})$ otherwise.

Finally, for any $j \in \{1, \dots, K\}$, the same derivation gives:

$$\nabla_{\theta_j} J_{\text{pop}}(\theta_1, \theta_2, \dots, \theta_K) = \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1, \dots, \tau_K} \mathbb{1}_{i^*=j} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_j} \log \pi_{\theta_j}(\tau_j). \tag{C.19}$$

Therefore, we have:

$$\nabla_\theta = \sum_{j=1}^{n} \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1, \dots, \tau_K} \mathbb{1}_{i^*=j} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_j} \log \pi_{\theta_j}(\tau_j), \tag{C.20}$$

$$\nabla_\theta = \mathbb{E}_{\rho \sim \mathcal{D}} \mathbb{E}_{\tau_1, \dots, \tau_K} \big(R(\tau_{i^*}) - R(\tau_{i^{**}})\big) \nabla_{\theta_{i^*}} \log \pi_{\theta_{i^*}}(\tau_{i^*}), \tag{C.21}$$

$$\tag{C.22}$$

which concludes the proof. □

## C.3 Comparison to active search

To give a sense of the performance of Poppy with a larger time budget, we implement a simple sampling strategy. Given a population of $K$ agents, we first greedily rollout each of them on every starting point, and evenly distribute any remaining sampling budget across the most promising $K$ (agent, starting point) pairs for each instance with stochastic rollouts. This two-step process is motivated by the idea that is not useful to sample several times an agent on an instance where it is outperformed by another one.

**Setup** For both TSP and CVRP, we use the same test instances as in Tables 7.1 and 7.2. We generate a total of $200 \times 8 \times N$ candidate solutions per instance (where 8 corresponds to the augmentation strategy by Kwon et al. (2020) and $N$ is the number of starting points), accounting for both the first and second phases. We evaluate our approach against POMO (Kwon et al., 2020) and EAS (Hottung, Kwon, and Tierney, 2022) with the same budget. As EAS has three different variants, we compare against EAS-Tab since it is the only one that does not backpropagate gradients through the network, similarly to our approach; thus, it should match Poppy's compute time on the same hardware.

**Results** Tables C.2 and C.3 show the results for TSP and CVRP, respectively. With extra sampling, Poppy reaches a performance gap of $0.002\%$ on TSP100, and establishes a state-of-the-art for general ML-based approaches, even when compared to supervised methods. For CVRP, adding sampling to Poppy makes it on par with DPDP and EAS, depending on the problem size, and it is only outperformed by the active search approach EAS, which gives large

improvements on CVRP. As the two-step sampling process used for Poppy is very rudimentary compared to the active search method described in Hottung, Kwon, and Tierney (2022), it is likely that combining the two approaches could further boost performance, which we leave for future work.

**Table C.2** – TSP results (active search).

| | | **Inference** (10k instances) | | | **0-shot** (1k instances) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $n = 100$ | | | $n = 125$ | | | $n = 150$ | | |
| | Method | Obj. | Gap | Time | Obj. | Gap | Time | Obj. | Gap | Time |
| | Concorde | 7.765 | 0.000% | 82M | 8.583 | 0.000% | 12M | 9.346 | 0.000% | 17M |
| | LKH3 | 7.765 | 0.000% | 8H | 8.583 | 0.000% | 73M | 9.346 | 0.000% | 99M |
| SL | GCN-BS | 7.87 | 1.39% | 40M* | - | - | - | - | - | - |
| SL | CVAE-Opt | - | 0.343% | 6D* | 8.646 | 0.736% | 21H* | 9.482 | 1.45% | 30H* |
| SL | DPDP | 7.765 | 0.004% | 2H* | 8.589 | 0.070% | 31M* | 9.434 | 0.94% | 44M* |
| RL | POMO (200 samples) | 7.769 | 0.056% | 2H | 8.594 | 0.13% | 20M | 9.376 | 0.31% | 32M |
| RL | EAS | 7.768 | 0.048% | 5H* | 8.591 | 0.091% | 49M* | 9.365 | 0.20% | 1H* |
| RL | **Poppy 16 (200 samples)** | **7.765** | **0.002%** | 2H | **8.584** | **0.009%** | 20M | **9.351** | **0.05%** | 32M |

**Table C.3** – CVRP results (active search).

| | | **Inference** (10k instances) | | | **0-shot** (1k instances) | | | **0-shot** (1k instances) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $n = 100$ | | | $n = 125$ | | | $n = 150$ | | |
| | Method | Obj. | Gap | Time | Obj. | Gap | Time | Obj. | Gap | Time |
| | LKH3 | 15.65 | 0.000% | 6D | 17.50 | 0.000% | 19H | 19.22 | 0.000% | 20H |
| SL | CVAE-Opt | - | 1.36% | 11D* | 17.87 | 2.08% | 36H* | 19.84 | 3.24% | 46H* |
| SL | DPDP | 15.63 | −0.13% | 23H* | 17.51 | 0.07% | 3H* | 19.31 | 0.48% | 5H* |
| RL | POMO (200 samples) | 15.67 | 0.18% | 4H | 17.56 | 0.33% | 43M | 19.43 | 1.08% | 1H |
| RL | EAS | **15.62** | **-0.14%** | 8H* | 17.49 | 0.00% | 80M* | 19.36 | 0.72% | 2H* |
| RL | **Poppy 32 (200 samples)** | **15.62** | **-0.14%** | 4H | **17.49** | **-0.10%** | 42M | **19.32** | **0.50%** | 1H |

## C.4 Problems

We here describe the details of the CO problems we have used to evaluate Poppy, including instance generation and training details (*e.g.* architecture, hyperparameters). In the case of TSP and CVRP, we show some example solutions obtained by a population of agents. Besides, we thoroughly analyze the performance of the populations in TSP.

### C.4.1 Traveling Salesman Problem (TSP)

#### Instance generation

The $n$ cities that constitute each problem's instance are uniformly sampled from $[0, 1]^2$.

## Complements on Chapter 7

### Training details

**Architecture** We use the same model as Kool, Hoof, and Welling (2019) and Kwon et al. (2020) except for the batch-normalization layers, which are replaced by layer-normalization to ease parallel batch processing. We invert the mask used in the decoder computations (*i.e.*, masking the available cities instead of the unavailable ones) after experimentally observing faster convergence rates. The results reported for POMO were obtained with the same implementation changes to keep the comparison fair. These results are on par with those reported in the original POMO paper (Kwon et al., 2020).

**Hyperparameters** To match the setting used by Kwon et al. (2020), we use the Adam optimizer (Diederik P. Kingma and Ba, 2015b) with a learning rate $\mu = 10^{-4}$, and a $L_2$ penalization of $10^{-6}$. The encoder is composed of 6 multi-head attention layers with 8 heads each. The dimension of the keys, queries and values is 16. Each attention layer is composed of a feed-forward layer of size 512, and the final node embeddings have 128 dimensions. The decoders are composed of 1 multi-head attention layer with 8 heads and 16-dimensional key, query and value.

The number of starting points $P$ is 50 for each instance. We determined this value after performing a grid-search based on the first training steps with $P \in \{20, 50, 100\}$.

### Example solutions

Figure C.2 shows some trajectories obtained from a 16-agent population on TSP100. Even though they look similar, small decisions differ between agents, thus frequently leading to different solutions. Interestingly, some agents (especially 6 and 11) give very poor trajectories. We hypothesize that it is a consequence of specializing since agents have no incentive to provide a good solution if another agent is already better on this instance.

### Population analysis

Figure C.1 shows some additional information about individual agent performances. In the left figure, we observe that each agent gives on average the best solution for $35\%$ of the instances, and that for around $2.5\%$ it gives the unique best solution across the population. These numbers are evenly distributed, which shows that every agent contributes to the whole population performance. Furthermore, we observe the performance is quite evenly distributed across the population of Poppy 16; hence, showing that the population has not collapsed to a few high-performing agents, and that Poppy benefits from the population size, as shown in the bottom figure. On the right is displayed the performance of several sub-populations of agents for Poppy 4, 8 and 16. Unsurprisingly, any fixed size sub-population is better when sampled

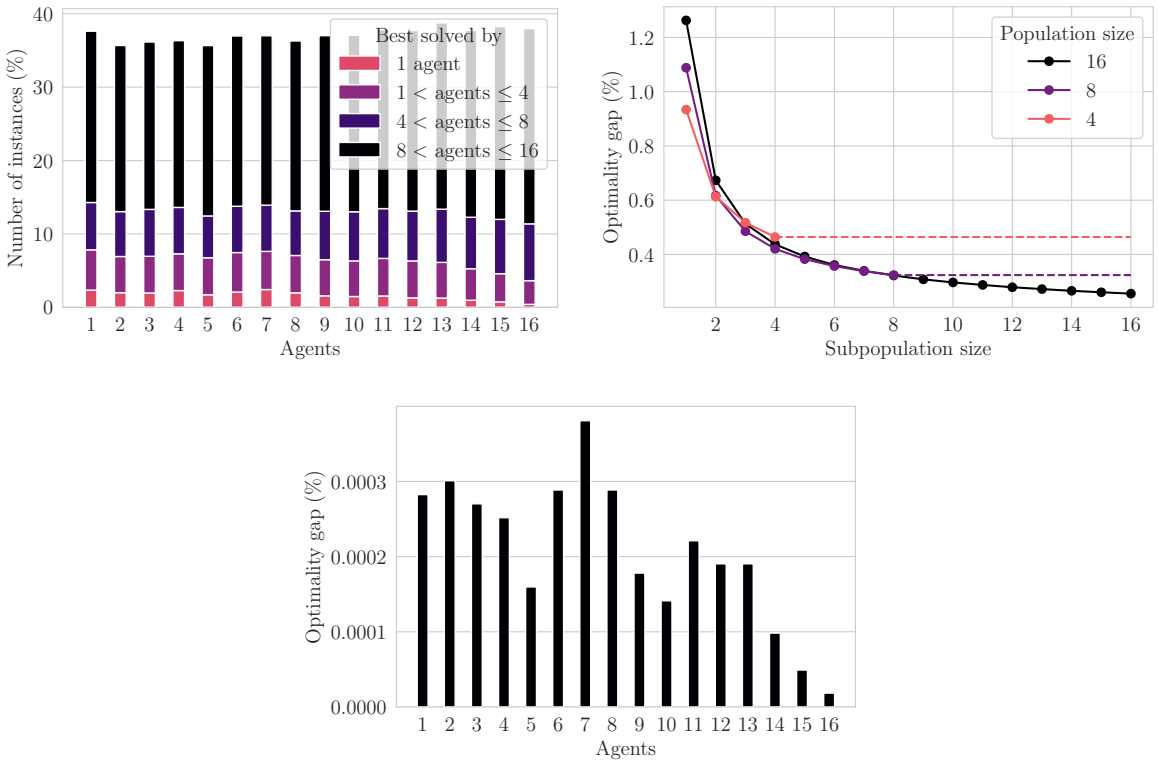



**Figure C.1 – Left**: Proportion of instances that each agent solves best among the population for Poppy 16 on TSP100. Colors indicate the number of agents in the population giving the same solution for these sets of instances. **Right**: The mean performance of 1,000 randomly drawn sub-populations for Poppy 1, 4, 8 and 16. **Bottom**: Optimality gap loss suffered when removing any agent from the population using Poppy 16. Although some agents contribute more (*e.g.* 2, 7) and some less (*e.g.* 15, 16), the distribution is relatively even with all agents contributing, even though no explicit mechanism enforces this behavior

from smaller populations: Poppy 16 needs 4 agents to recover the performance of Poppy 4, and 8 agents to recover the performance of Poppy 8 for example. This highlights the fact that agents have learned complementary behaviors which might be sub-optimal if part of the total population is missing.

### C.4.2 Capacitated Vehicle Routing Problem (CVRP)

#### Instance generation

The locations of the $n$ customer nodes and the depot are uniformly sampled in $[0, 1]^2$. The demands are uniformly sampled from the discrete set $\{\frac{1}{D}, \frac{2}{D}, \dots, \frac{9}{D}\}$ where $D = 50$ for CVRP100, $D = 55$ for CVRP125, and $D = 60$ for CVRP150. The maximum vehicle capacity is 1. The deliveries cannot be split: each customer node is visited once, and its whole demand is taken off the vehicle's remaining capacity.

#### Training details

**Architecture** We use the same model as in TSP. However, unlike TSP, the mask is not inverted; besides, it does not only prevent the agent from revisiting previous customer nodes, but also from visiting the depot if it is the current location, and any customer node whose demand is higher than the current capacity.

**Hyperparameters** We use the same hyperparameters as in TSP except for the number of starting points $P$ per instance used during training, which we set to 100 after performing a grid-search with $P \in \{20, 50, 100\}$.

#### Example solutions

Figure C.3 shows some trajectories obtained by 16 agents from a 32-agent population on CVRP100. Unlike TSP, the agent/vehicle performs several tours starting and finishing in the depot.

### C.4.3 0-1 Knapsack (KP)

#### Instance generation

Item values and weights are uniformly sampled in $[0, 1]$. The bag capacity is fixed to $25$.

### C.4.4 Training details

**Training** For this environment, and contrary to TSP and CVRP, training an agent is lightning-fast as it takes only a few minutes. In this specific case, we noticed it was not necessary to train a single decoder first. Instead, (i) we directly train a population in parallel from scratch, and (ii) specialize the population exactly as done in the other environments.

**Architecture** We use the same model as in TSP. However, the mask used when decoding is not inverted, and the items that do not fit in the bag are masked together with the items taken so far.

**Hyperparameters** We use the same hyperparameters as in TSP except for the number of starting points $P$ used during training, which we set to 100 after performing a grid-search with $P \in \{20, 50, 100\}$.

## C.5 Time-performance tradeoff

We present on Figure C.4 a comparison of the time-performance Pareto front between Poppy and POMO as we vary respectively the population size and the amount of stochastic sampling. Poppy consistently provides better performance for a fixed number of trajectories. Strikingly, in almost every setting, matching Poppy's performance by increasing the number of stochastic samples does not appear tractable.

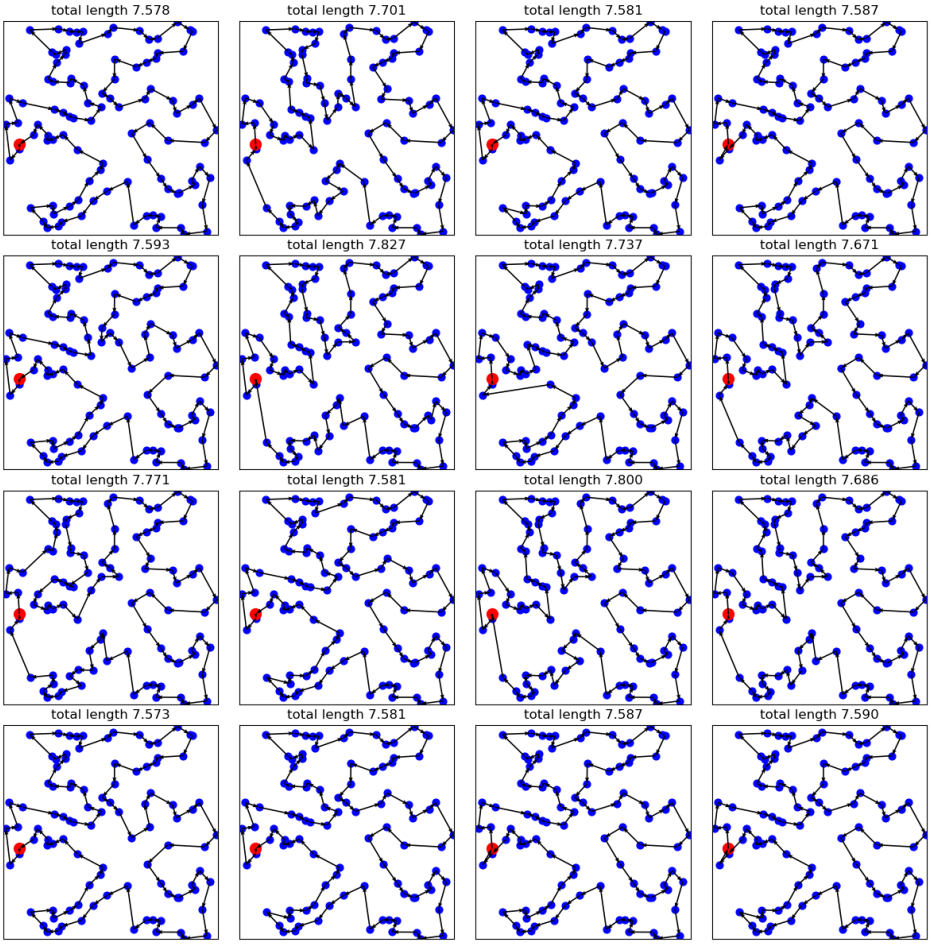



**Figure C.2** – Example TSP trajectories given by Poppy for a 16-agent population from one starting point (red).

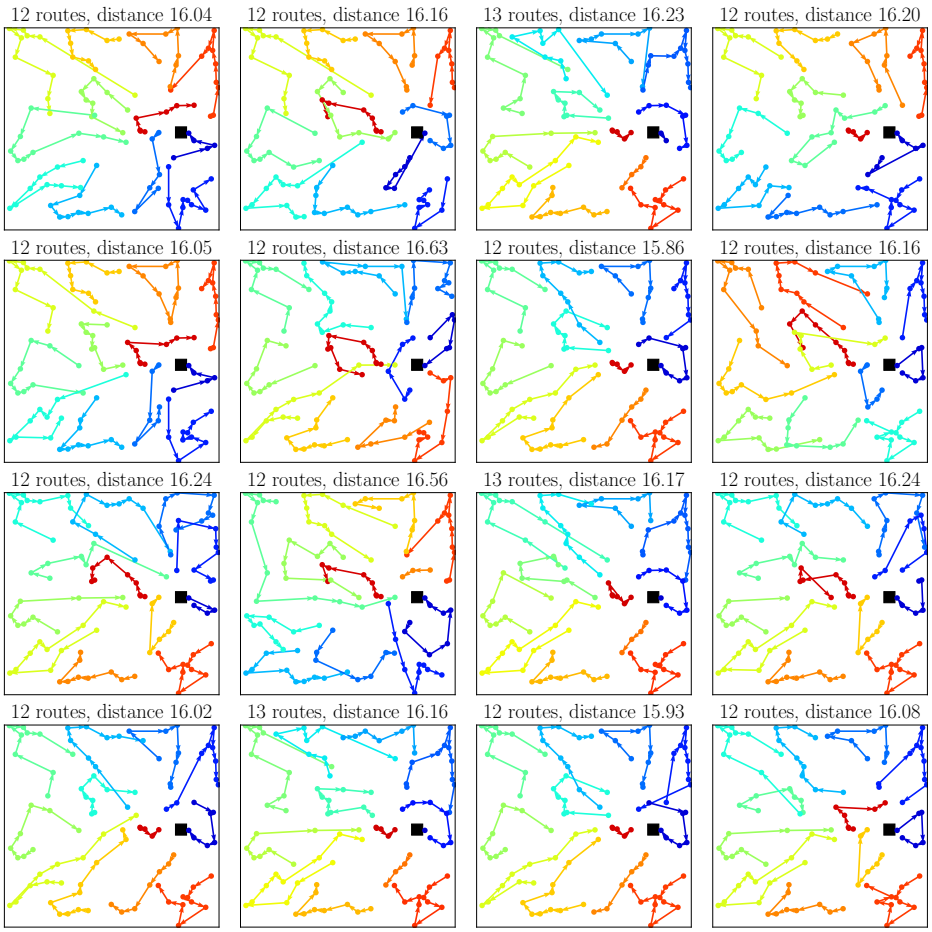
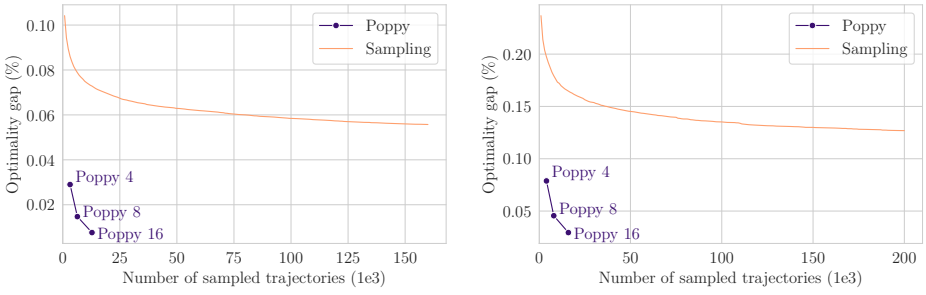


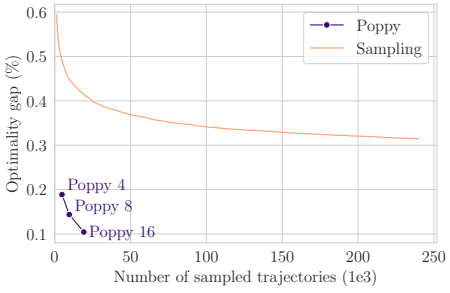

**Figure C.3** – Example CVRP trajectories given by Poppy for 16 agents from a 32-agent population. The depot is displayed as a black square. The edges from/to the depot are omitted for clarity.



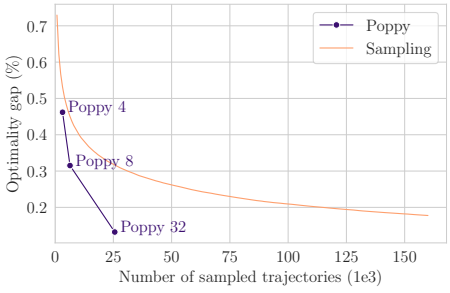

(**a**) TSP100



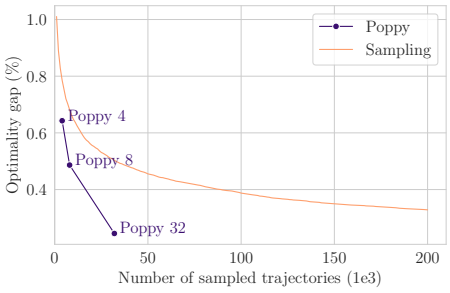

(**b**) TSP125



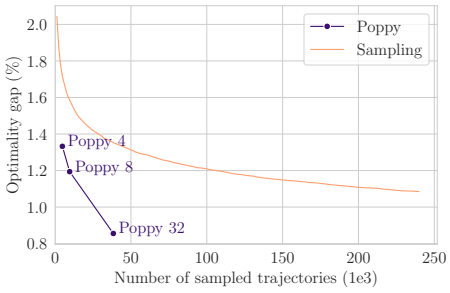

(**c**) TSP150



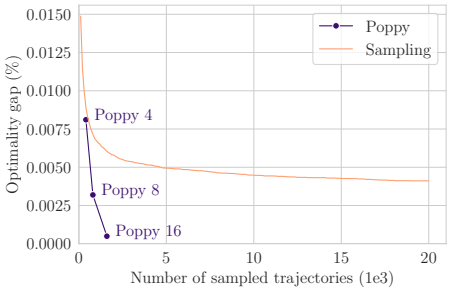

(**d**) CVRP100



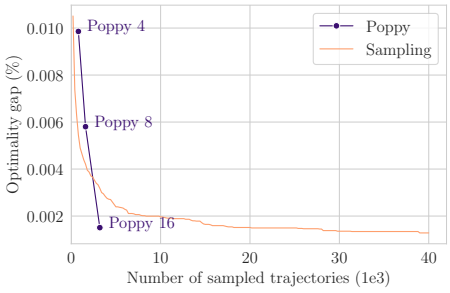

(**e**) CVRP125




(**f**) CVRP150




(**g**) KP100




(**h**) KP200

**Figure C.4** – Comparison of the time-performance Pareto front of Poppy and POMO, for each problem used in the chapter. The x-axis is the number of trajectories sampled per test instance, while the y-axis is the gap with the optimal solution for TSP and KP, and the gap with LKH3 for CVRP.

# Appendix D

# Complements on Chapter 8

## D.1 Additional details

### D.1.1 Hyperparameter selection

We provide in Table D.1 the hyperparameters selected on the validation dataset and used for our experiments.

| Elo | Attacker side | $N_{\text{min}}$ | $c$ |
|---|---|---|---|
| 1800 | White | 100 | 0.1 |
| | Black | 100 | 0.1 |
| 2000 | White | 50 | 0.1 |
| | Black | 100 | 0.1 |
| 2200 | Black | 100 | 0.1 |
| | White | 100 | 0.1 |

**Table D.1** – Hyperparameters selected on the validation dataset.

### D.1.2 Quantitative evaluation: number of moves

In our quantitative evaluation setting, the number of moves actually performed by Hustler of Stockfish depends on the test data available. We provide in Table D.2 the average number of moves for each approach in each setting. Because Hustler's policy is not defined for every position, it plays less than Stockfish in every case, despite showing better performances.

| Elo | Attacker | Method | Average move number |
|---|---|---|---|
| 1800 | White | Hustler | 8.33 |
| | | stockfish | 8.84 |
| | Black | Hustler | 6.87 |
| | | stockfish | 8.05 |
| 2000 | White | Hustler | 7.18 |
| | | stockfish | 9.52 |
| | Black | Hustler | 6.44 |
| | | stockfish | 7.98 |
| 2200 | White | Hustler | 7.68 |
| | | stockfish | 9.72 |
| | Black | Hustler | 6.28 |
| | | stockfish | 7.54 |

**Table D.2** – Average number of moves for each approach in each setting.

## D.2 Additional results

### D.2.1 Effect of $N_{\text{min}}$

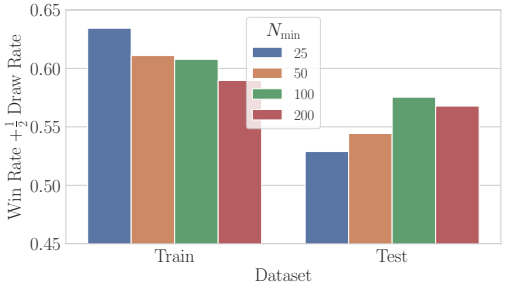



**Figure D.1** – Sensitivity analysis of $N_{\text{min}}$. $N_{\text{min}}$ empirically controls the degree of exploitation of the training set: the smaller $N_{\text{min}}$, the higher the score on the training set, but at the risk of overfitting. The scores are averaged over each Elo category and attacker side.

Intuitively, $N_{\text{min}}$ controls the trade-off between over-fitting (when $N_{\text{min}}$ is too small), and under-fitting (when $N_{\text{min}}$ is too large). Fig. D.1 shows this empirically: although the train score decreases as $N_{\text{min}}$ increases, the best test score is attained for $N_{\text{min}} = 100$.

### D.2.2 Ablation study: $V_{\text{search}}$

We provide an ablation study over the use of $V_{\text{search}}$ in Fig. D.2. We can see that using $V_{\text{search}}$ in addition to $V_{\text{policy}}$ improves performance both at train and test time, outlining its positive effect on the search procedure.

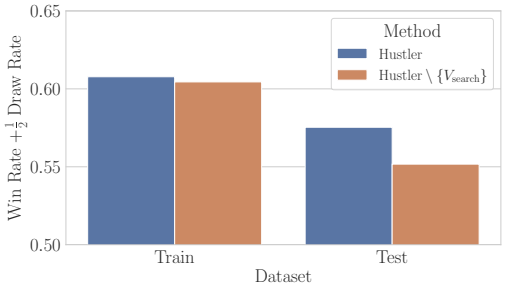
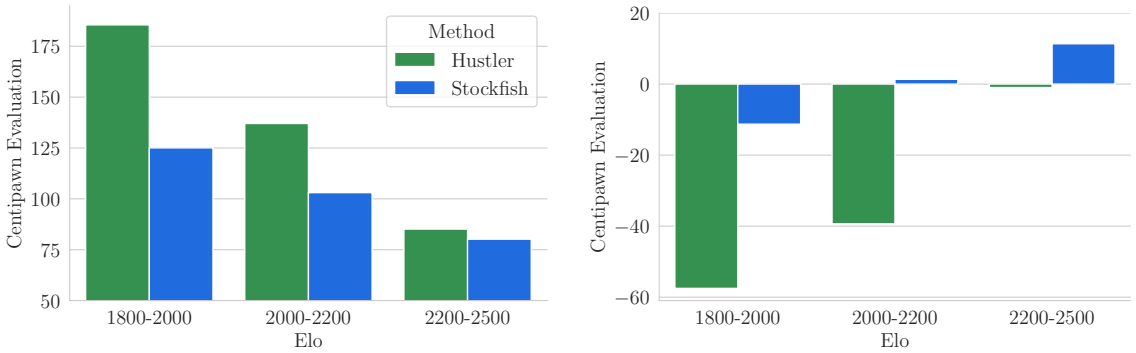



**Figure D.2** – We evaluate d-MCTS without $V_{\text{search}}$, as an ablation study, thus keeping only one value for each node as done traditionally. The scores are averaged over each Elo category and attacker side. Using $V_{\text{search}}$ improves both train and test performance, which outlines that it makes the search more efficient.




(**a**) Attacker is White.

(**b**) Attacker is Black.

**Figure D.3** – Stockfish evaluation (centipawn).

### D.2.3 Centipawn evaluation

We display in Fig. D.3 Stockfish evaluation of the 100 positions reached at the end of the beam search during the quantitative evaluation phase. Even using Stockfish's evaluation, Hustler outperforms Stockfish in every situation.

# List of Figures

# List of Algorithms

# List of Tables

# List of References


Abramson, Myriam and Harry Wechsler (2003). Tabu Search Exploration for On-Policy Reinforcement Learning. In *International Joint Conference on Neural Networks*.

Addanki, Ravichandra, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh (2019). Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879*.

Agrawal, Akshay, Robin Verschueren, Steven Diamond, and Stephen Boyd (2018). A rewriting system for convex optimization problems. *Journal of Control and Decision* 5.1, pp. 42–60.

Agullo, Emmanuel, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, et al. (2011). LU factorization for accelerator-based systems. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*. IEEE, pp. 217–224.

Agullo, Emmanuel, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, et al. (2011). QR factorization on a multicore node enhanced with multiple GPU accelerators. In *IPDPS'11*. IEEE.

Agullo, Emmanuel, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, et al. (2010). Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*.

Agullo, Emmanuel, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar (2016). Are static schedules so bad? A case study on Cholesky factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 1021–1030.

Amiranashvili, Artemij, Alexey Dosovitskiy, Vladlen Koltun, and Thomas Brox (2018). Motion perception in reinforcement learning with dynamic objects. In *Conference on Robot Learning*.

Amodei, Dario, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané (2016). Concrete Problems in AI Safety. *arXiv preprint arXiv:1606.06565*.

Anderson, Ashton, Jon Kleinberg, and Sendhil Mullainathan (2017). Assessing human error against a benchmark of perfection. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11.4, p. 45.

Applegate, D. L., R. E. Bixby, V. Chvatal, and W. J. Cook (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Augonnet, C., S. Thibault, R. Namyst, and P.-A. Wacrenier (Feb. 2011). StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23, pp. 187–198.

Augonnet, Cédric, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23.2, pp. 187–198.

Aytar, Yusuf, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas (2018). Playing hard exploration games by watching youtube. In *Advances in Neural Information Processing Systems*.

Baboulin, Marc, Luc Giraud, and Serge Gratton (2005). A parallel distributed solver for large dense symmetric systems: applications to geodesy and electromagnetism problems. *The International Journal of High Performance Computing Applications* 19.4, pp. 353–363.

Badia, Adrià Puigdomènech, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, et al. (2020). Agent57: Outperforming the atari human benchmark. In *International Conference on Machine Learning*.

Badia, Adrià Puigdomènech, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, et al. (2020). Never give up: Learning directed exploration strategies. In *International Conference on Learning Representations*.

Badia, Adrià Puigdomènech, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, Bilal Piot, Steven Kapturowski, et al. (2020). Never Give Up: Learning Directed Exploration Strategies. In *International Conference on Learning Representations*.

Bai, Yutong, Haoqi Fan, Ishan Misra, Ganesh Venkatesh, Yongyi Lu, Yuyin Zhou, et al. (2020). Can Temporal Information Help with Contrastive Self-Supervised Learning? *arXiv preprint arXiv:2011.13046*.

Barrett, Thomas D., William R. Clements, Jakob N. Foerster, and A. I. Lvovsky (2020). Exploratory Combinatorial Optimization with Reinforcement Learning. In *In Proceedings of the 34th National Conference on Artificial Intelligence, AAAI*.

Barrett, Thomas D., Christopher W. F. Parsonson, and Alexandre Laterre (2022). Learning to Solve Combinatorial Graph Partitioning Problems via Efficient Exploration. *arXiv preprint arXiv:2205.14105*.

Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson (1983a). Neuronlike adaptive elements that can solve difficult learning control problems. In *IEEE Transactions on Systems, Man, and Cybernetics*.

— (1983b). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5, pp. 834–846.

Basha, Tali, Yael Moses, and Shai Avidan (2012). Photo sequencing. In *European Conference on Computer Vision*.

Beaumont, Olivier, Lionel Eyraud-Dubois, and Yihong Gao (2019). Influence of tasks duration variability on task-based runtime schedulers. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 16–25.

Beaumont, Olivier, Julien Langou, Willy Quach, and Alena Shilova (2020). A Makespan Lower Bound for the Scheduling of the Tiled Cholesky Factorization based on ALAP scheduling. In *Proceedings of the 26th International European Conference on Parallel and Distributed Computing (EuroPar 2020)*, pp. 1–12.

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bellemare, Marc G., Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos (2016). Unifying Count-Based Exploration and Intrinsic Motivation. In *Advances in Neural Information Processing Systems*.

Bellman, Richard (1957). *Dynamic Programming*. USA: Princeton University Press.

Bello, Irwan, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio (2016). Neural Combinatorial Optimization with Reinforcement Learning. *arXiv preprint arXiv:1611.09940*.

Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost (2018). Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *CoRR* abs/1811.06128.

Berkenkamp, Felix, Riccardo Moriconi, Angela Schoellig, and Andreas Krause (2016). Safe Learning of Regions of Attraction for Uncertain, Nonlinear Systems with Gaussian Processes. In *Conference on Decision and Control*.

Bientinesi, Paolo, Francisco D Igual, Daniel Kressner, and Enrique S Quintana-Ortí (2009). Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, pp. 387–395.

Biswas, T. T. and K. W. Regan (2015). Quantifying depth and complexity of thinking and knowledge. In *Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART)*.

Biswas, Tamal and Kenneth W Regan (2015). Measuring level-k reasoning, satisficing, and human error in game-play data. In *2015 IEEE 14th International Conference on Machine Learning and Applications*. Miami, FL: IEEE, pp. 941–947.

Biza, Ondrej and Robert Platt Jr. (2019). Online Abstraction with MDP Homomorphisms for Deep Learning. In *International Conference on Autonomous Agents and Multiagent Systems*.

Bonnet, Clément, Donal Byrne, Victor Le, Laurence Midgley, Daniel Luo, Cemlyn Waters, et al. (2022). *Jumanji: Industry-Driven Hardware-Accelerated RL Environments*. Version 0.1.1.

Bosilca, G., A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra (2012). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing* 38.(1-2), pp. 37–51.

Bosilca, George, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra (2013). Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15.6, pp. 36–45.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, et al. (2016). *OpenAI Gym*.

Brown, Noam and Tuomas Sandholm (2019). Superhuman AI for multiplayer poker. *Science* 365.6456, pp. 885–890.

Burda, Yuri, Harrison Edwards, Amos J. Storkey, and Oleg Klimov (2019). Exploration by Random Network Distillation. In *International Conference on Learning Representations*.

Buttari, Alfredo, Julien Langou, Jakub Kurzak, and Jack Dongarra (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35.1, pp. 38–53.

Carr, Andrew Newberry, Quentin Berthet, Mathieu Blondel, Olivier Teboul, and Neil Zeghidour (2021). Self-Supervised Learning of Audio Representations from Permutations with Differentiable Ranking. In *IEEE Signal Processing Letters*.

Caselles-Dupré, Hugo, Michael Garcia-Ortiz, and David Filliat (2020). On the sensory commutativity of action sequences for embodied agents. *arXiv preprint arXiv:2002.05630*.

Chandak, Yash, Georgios Theocharous, James Kostas, Scott M. Jordan, and Philip S. Thomas (2019). Learning Action Representations for Reinforcement Learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 941–950.

Charness, Neil (1992). The impact of chess research on cognitive science. *Psychological research* 54.1, pp. 4–9.

Chaslot, Guillaume, Sander Bakkes, István Szita, and Pieter Spronck (2008). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Chen, Xinyun and Yuandong Tian (2019). Learning to Perform Local Rewriting for Combinatorial Optimization. In *Advances in Neural Information Processing Systems*.

Chentanez, Nuttapong, Andrew G. Barto, and Satinder Singh (2005). Intrinsically Motivated Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Chevalier-Boisvert, Maxime, Lucas Willems, and Suman Pal (2018). *Minimalistic Gridworld Environment for OpenAI Gym*. https://github.com/maximecb/gym-minigrid.

Choi, Jaeyoung, Jack J Dongarra, L Susan Ostrouchov, Antoine P Petitet, David W Walker, and R Clint Whaley (1996). Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* 5.3, pp. 173–184.

Christiano, Paul F, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei (2017). Deep Reinforcement Learning from Human Preferences. In *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, et al. Vol. 30. Curran Associates, Inc.

Cobbe, Karl, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, et al. (2021). *Training Verifiers to Solve Math Word Problems*.

Cojean, Terry, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier (2019). Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Computing* 83, pp. 73–92.

Coulom, Rémi (2007a). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*.

— (2007b). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Springer Berlin Heidelberg, pp. 72–83.

Cully, Antoine and Yiannis Demiris (2018). Quality and Diversity Optimization: A Unifying Modular Framework. *IEEE Transactions on Evolutionary Computation* 22.2, pp. 245–259.

Czech, Johannes, Patrick Korus, and Kristian Kersting (2020). Monte-Carlo Graph Search for AlphaZero. *arXiv preprint arXiv:2012.11045*.

Dadashi, Robert, Léonard Hussenot, Matthieu Geist, and Olivier Pietquin (2020). Primal wasserstein imitation learning. In *International Conference on Learning Representations*.

Dai, Hanjun, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song (2017). Learning Combinatorial Optimization Algorithms over Graphs. In *Advances in Neural Information Processing Systems*.

Dantzig, G., R. Fulkerson, and S. Johnson (1954). Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America* 2.4, pp. 393–410.

Deudon, Michel, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau (2018). Learning Heuristics for the TSP by Policy Gradient. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer International Publishing, pp. 170–181.

Diamond, Steven and Stephen Boyd (2016). CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17.83, pp. 1–5.

Doan, Thang, Bogdan Mazoure, Moloud Abdar, Audrey Durand, Joelle Pineau, and R. Devon Hjelm (2020). Attraction-Repulsion Actor-Critic for Continuous Control Reinforcement Learning. *arXiv preprint arXiv:1909.07543*.

Dorigo, Marco, Mauro Birattari, and Thomas Stutzle (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine* 1.4, pp. 28–39.

Dubey, Rachit, Pulkit Agrawal, Deepak Pathak, Thomas L. Griffiths, and Alexei A. Efros (2018). Investigating Human Priors for Playing Video Games. In *International Conference on Machine Learning*.

Dulac-Arnold, Gabriel, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, et al. (2016). Deep Reinforcement Learning in Large Discrete Action Spaces. *arXiv preprint arXiv::1512.07679*.

Duran, Alejandro, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, et al. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21.02, pp. 173–193.

Dwibedi, Debidatta, Jonathan Tompson, Corey Lynch, and Pierre Sermanet (2018). Learning actionable representations from visual observations. In *International Conference on Intelligent Robots and Systems*.

Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics* 17, pp. 449–467.

Elsayed, Gamaleldin F., Shreya Shankar, Brian Cheung, Nicolas Papernot, Alexey Kurakin, Ian J. Goodfellow, et al. (2018). Adversarial Examples that Fool both Computer Vision and Time-Limited Humans. In *Neural Information Processing Systems*.

Espeholt, Lasse, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*.

Etheve, Marc, Zacharie Alès, Côme Bissuel, Safia Kedad-Sidhoum, and Olivier Juan (2020). Reinforcement Learning for Variable Selection in a Branch and Bound Algorithm. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research. CPAIOR:*

*International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 176–185.

Eysenbach, Benjamin, Abhishek Gupta, Julian Ibarz, and Sergey Levine (2019). Diversity is All You Need: Learning Skills without a Reward Function. In *International Conference on Learning Representations*.

Farias, D. P. de and B. Van Roy (2003). The Linear Programming Approach to Approximate Dynamic Programming. *Operations Research*.

Farquhar, Gregory, Laura Gustafson, Zeming Lin, Shimon Whiteson, Nicolas Usunier, and Gabriel Synnaeve (2020). Growing Action Spaces. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 3040–3051.

Fawzi, Alhussein, Mateusz Balog, Angela Huang, and et al. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610.1, pp. 47–53.

Fernando, Basura, Hakan Bilen, Efstratios Gavves, and Stephen Gould (2017). Self-supervised video representation learning with odd-one-out networks. In *Conference on Computer Vision and Pattern Recognition*.

Fernando, Basura, Efstratios Gavves, Jose M Oramas, Amir Ghodrati, and Tinne Tuytelaars (2015). Modeling video evolution for action recognition. In *Conference on Computer Vision and Pattern Recognition*.

Ferreira, Diogo R. (2013). The Impact of the Search Depth on Chess Playing Strength. *J. Int. Comput. Games Assoc.* 36, pp. 67–80.

Fey, Matthias and Jan E. Lenssen (2019). Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Flajolet, Arthur, Claire Bizon Monroc, Karim Beguir, and Thomas Pierrot (2022). Fast Population-Based Reinforcement Learning on a Single Machine. In *International Conference on Machine Learning*.

Flet-Berliac, Yannis, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021a). Adversarially Guided Actor-Critic. In *International Conference on Learning Representations*.

Flet-Berliac, Yannis, Reda Ouhamma, Odalric-Ambrym Maillard, and Philippe Preux (2021b). Learning Value Functions in Deep Policy Gradients using Residual Variance. In *9th International Conference on Learning Representations (ICLR) 2021*.

— (2021c). Learning Value Functions in Deep Policy Gradients using Residual Variance. In *Proc. ICLR*.

Fu, Zhang-Hua, Kai-Bin Qiu, and Hongyuan Zha (2021). Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, pp. 7474–7482.

Gao, Yuanxiang, Li Chen, and Baochun Li (2018). Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pp. 1676–1684.

García, Javier and Fernando Fernández (2012). Safe Exploration of State and Action Spaces in Reinforcement Learning. *Journal of Artificial Intelligence Research*.

Garey, Michael R and David S Johnson (1979). *Computers and intractability*. Vol. 174. freeman San Francisco.

Gendreau, Michel, Jean-Yves Potvin, et al. (2010). *Handbook of metaheuristics*. Vol. 2. Springer.

Gleave, Adam, Michael Dennis, Neel Kant, Cody Wild, Sergey Levine, and Stuart Russell (2020). Adversarial Policies: Attacking Deep Reinforcement Learning. In *International Conference on Learning Representation*.

Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences* 8, pp. 156–166.

— (1986a). Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research* 13.5, pp. 533–549.

— (1986b). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*.

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy (2015). Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun.

Goroshin, Ross, Joan Bruna, Jonathan Tompson, David Eigen, and Yann LeCun (2015). Unsupervised learning of spatiotemporally coherent metrics. In *International Conference on Computer Vision*.

Graham, Ronald L, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*. Vol. 5. Elsevier, pp. 287–326.

Greensmith, Evan, Peter L. Bartlett, and Jonathan Baxter (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research 5* 5, pp. 1471–1530.

Grinsztajn, Nathan (2021). *RL for Dynamic Scheduling* https://github.com/nathangrinsztajn/RL_for_dynamic_scheduling.

Grinsztajn, Nathan, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux (2020). Geometric deep reinforcement learning for dynamic DAG scheduling. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. Virtual: IEEE Press.

— (2021a). READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling. In *IEEE International Conference on Cluster Computing (CLUSTER)*.

— (2021b). Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. Virtual: IEEE Press.

Grinsztajn, Nathan, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021a). There Is No Turning Back: A Self-Supervised Approach for Reversibility-Aware Reinforcement Learning. In *Advances in Neural Information Processing Systems (Neurips)*. Virtual.

Grinsztajn, Nathan, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021b). There Is No Turning Back: A Self-Supervised Approach for Reversibility-Aware Reinforcement Learning. *arXiv preprint arXiv:2106.04480*.

Grinsztajn, Nathan, Daniel Furelos-Blanco, and Thomas D. Barrett (2022). *Population-Based Reinforcement Learning for Combinatorial Optimization*. Submitted to ICML 2023, under review.

Grinsztajn, Nathan, Toby Johnstone, Johan Ferret, and Philippe Preux (Dec. 2022). Better state exploration using action sequence equivalence. In *NeurIPS workshop on Deep Reinforcement Learning*. Virtual, United States.

Grinsztajn, Nathan, Louis Leconte, Philippe Preux, and Edouard Oyallon (2021). Interferometric Graph Transform for Community Labeling. *CoRR* abs/2106.05875.

Grinsztajn, Nathan and Philippe Preux (2023). *Efficient Search of Human Adversarial Policies in Chess*. Submitted to IJCAI 2023, under review.

Grinsztajn, Nathan, Philippe Preux, and Edouard Oyallon (May 2021). Low-Rank Projections of GCNs Laplacian. In *ICLR 2021 Workshop GTRL*. Virtual.

Guez, Arthur, Mehdi Mirza, Karol Gregor, et al. (2019). An investigation of Model-free planning. In *International Conference on Machine Learning*.

Guo, Zhaohan Daniel, Mohammad Gheshlaghi Azar, Bilal Piot, Bernardo A Pires, and Rémi Munos (2018). Neural predictive belief representations. *arXiv preprint arXiv:1811.06407*.

Guo, Zhaohan Daniel, Mohammad Gheshlaghi Azar, Alaa Saade, Shantanu Thakoor, Bilal Piot, Bernardo Ávila Pires, et al. (2021). Geometric Entropic Exploration. *arXiv preprint arXiv:2101.02055*.

Guo, Zhaohan Daniel, Bernardo Avila Pires, Bilal Piot, Jean-Bastien Grill, Florent Altché, Rémi Munos, et al. (2020). Bootstrap latent-predictive representations for multitask reinforcement learning. In *International Conference on Machine Learning*.

Gupta, Abhishek, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine (2018). Meta-Reinforcement Learning of Structured Exploration Strategies. In *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc.

Hans, Alexander, Daniel Schneegaß, Anton Maximilian Schäfer, and S. Udluft (2008). Safe exploration for reinforcement learning. In *European Symposium on Artificial Neural Networks*.

Hartikainen, Kristian, Xinyang Geng, Tuomas Haarnoja, and Sergey Levine (2020). Dynamical Distance Learning for Semi-Supervised and Unsupervised Skill Discovery. In *International Conference on Learning Representations*.

Hazan, Elad, Sham M. Kakade, Karan Singh, and Abby Van Soest (2019). Provably Efficient Maximum Entropy Exploration. In *International Conference on Machine Learning*.

Helsgaun, K. (2017). An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde University, Tech. Rep.*

Hertz, A. and D. Werra (2005). The tabu search metaheuristic: How we used it. *Annals of Mathematics and Artificial Intelligence*.

Hoffman, Matt, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, et al. (2020). Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979*.

Holdaway, Cameron and Edward Vul (2021). Risk-taking in adversarial games: What can 1 billion online chess games tell us? In *Proceedings of the 43rd Annual Meeting of the Cognitive Science Society*, pp. 986–992.

Hong, Zhang-Wei, Tzu-Yun Shann, Shih-Yang Su, Yi-Hsiang Chang, and Chun-Yi Lee (2018). Diversity-Driven Exploration Strategy for Deep Reinforcement Learning. In *International Conference on Neural Information Processing Systems*.

Hopfield, J J and W David Tank (1985). "Neural" computation of decisions in optimization problems. *Biological cybernetics* 52.3, pp. 141–152.

Hottung, André, Bhanu Bhandari, and Kevin Tierney (2021). Learning a latent search space for routing problems using variational autoencoders. In *International Conference on Learning Representations*.

Hottung, André, Yeong-Dae Kwon, and Kevin Tierney (2022). Efficient Active Search for Combinatorial Optimization Problems. In *International Conference on Learning Representations*.

Hottung, André and Kevin Tierney (2020). Neural Large Neighborhood Search for the Capacitated Vehicle Routing Problem. In *24th European Conference on Artificial Intelligence (ECAI 2020)*.

Jaynes, E. T. (1957). Information Theory and Statistical Mechanics. *Phys. Rev.*

Jeannot, Emmanuel (2013). Symbolic mapping and allocation for the Cholesky factorization on NUMA machines: Results and optimizations. *The International journal of high performance computing applications* 27.3, pp. 283–290.

Joshi, Chaitanya K., Thomas Laurent, and Xavier Bresson (2019). An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem. *arXiv preprint arXiv:1906.01227*.

Jung, Whiyoung, Giseung Park, and Youngchul Sung (2020). Population-Guided Parallel Policy Search for Reinforcement Learning. In *International Conference on Learning Representations*.

Kaliszyk, Cezary, Josef Urban, Henryk Michalewski, and Mirek Olšák (2018). *Reinforcement Learning of Theorem Proving*.

Karp, Richard M. (1972). Reducibility among Combinatorial Problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, pp. 85–103.

Kennedy, J. and R. Eberhart (1995). Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4, pp. 1942–1948.

Khadka, Shauharda, Somdeb Majumdar, Tarek Nassar, Zach Dwiel, Evren Tumer, Santiago Miret, et al. (2019). Collaborative Evolutionary Reinforcement Learning. In *International Conference on Machine Learning*.

Khadka, Shauharda and Kagan Tumer (2018). Evolution-Guided Policy Gradient in Reinforcement Learning. In *Conference on Neural Information Processing Systems*.

Khalil, Elias, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song (2017). Learning Combinatorial Optimization Algorithms over Graphs. In *Advances in Neural Information Processing Systems*.

Kim, Minsu, Jinkyoo Park, and joungho kim joungho (2021). Learning Collaborative Policies to Solve NP-hard Routing Problems. In *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., pp. 10418–10430.

Kingma, Diederik P and Jimmy Ba (2015a). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

— (2015b). Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kipf, Thomas N. and Max Welling (2016). Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907.

Knox, W. Bradley and Peter Stone (2008). TAMER: Training an Agent Manually via Evaluative Reinforcement. In *IEEE 7th International Conference on Development and Learning, (ICDL 2008)*. IEEE, pp. 292–297.

Knuth, D. E. (1974). A Terminological Proposal. *SIGACT News* 6.1, pp. 12–18.

Kocsis, Levente and Csaba Szepesvári (2006). Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 282–293.

Kool, Wouter, Laurens Bliek, Danilo Numeroso, Robbert Reijnen, Reza Refaei Afshar, Yingqian Zhang, et al. (2022). *EURO Meets NeurIPS 2022 Vehicle Routing Competition*. NeurIPS 2022 Competition.

Kool, Wouter, Herke van Hoof, Joaquim Gromicho, and Max Welling (2021). Deep Policy Dynamic Programming for Vehicle Routing Problems. *arXiv preprint arXiv:2102.11756*.

Kool, Wouter, Herke van Hoof, and Max Welling (2019). Attention, Learn to Solve Routing Problems! In *International Conference on Learning Representations*.

Krishnan, Abhijeet and Chris Martens (2022). Towards the Automatic Synthesis of Interpretable Chess Tactics. en. In *Explainable Agency in Artificial Intelligence Workshop 36th AAAI Conference on Artificial Intelligence*.

Kruusmaa, Maarja, Yuri Gavshin, and Adam Eppendahl (2007). Don't do things you can't undo: Reversibility models for generating safe behaviours. In *International Conference on Robotics and Automation*.

Kumar, Vaibhav, Siddhant Bhambri, and Prashant Giridhar Shambharkar (2019). Multiple Resource Management and Burst Time Prediction using Deep Reinforcement Learning. In *Eighth International Conference on Advances in Computing, Communication and Information Technology (CCIT)*, p. 8.

Kwon, Yeong-Dae, Byoungjip Kim Jinho Choo, Youngjune Gwon Iljoo Yoon, and Seungjai Min (2020). POMO: Policy Optimization with Multiple Optima for Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Lample, Guillaume, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, et al. (2022). *HyperTree Proof Search for Neural Theorem Proving*.

Lan, Li-Cheng, Huan Zhang, Ti-Rong Wu, Meng-Yu Tsai, I-Chen Wu, and Cho-Jui Hsieh (2022). Are AlphaZero-like Agents Robust to Adversarial Perturbations? In *Neural Information Processing Systems*.

Lee, Lisa, Benjamin Eysenbach, Emilio Parisotto, Eric P. Xing, Sergey Levine, and Ruslan Salakhutdinov (2019). Efficient Exploration via State Marginal Matching. *arXiv preprint arXiv:1906.05274*.

Leike, Jan, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, et al. (2017). AI Safety Gridworlds. *arXiv preprint arXiv:1711.09883*.

Lenstra, Jan Karel, David B Shmoys, and Éva Tardos (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming* 46.1-3, pp. 259–271.

Leung, Joseph YT (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC press.

Leurent, Edouard and Odalric-Ambrym Maillard (2020). Monte-Carlo Graph Search: the Value of Merging Similar States. In *Asian Conference on Machine Learning*.

Li, Zhuwen, Qifeng Chen, and Vladlen Koltun (2018). Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. In *Advances in Neural Information Processing Systems*.

Liao, Haiguang, Wentai Zhang, Xuliang Dong, Barnabas Poczos, Kenji Shimada, and Levent Burak Kara (2019). *A Deep Reinforcement Learning Approach for Global Routing*.

Lin, S. and B. W. Kernighan (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* 21.2, pp. 498–516.

Lopes, Manuel, Tobias Lang, Marc Toussaint, and Pierre-yves Oudeyer (2012). Exploration in Model-based Reinforcement Learning by Empirically Estimating Learning Progress. In *Advances in Neural Information Processing Systems*.

Lu, Hao, Xingwen Zhang, and Shuang Yang (2020). A Learning-based Iterative Method for Solving Vehicle Routing Problems. In *International Conference on Learning Representations*.

Mahajan, Anuj and Theja Tulabandhula (2017). Symmetry Learning for Function Approximation in Reinforcement Learning. In *International Joint Conference on Artificial Intelligence*.

Maillard, Odalric-Ambrym, Timothy Mann, Ronald Ortner, and Shie Mannor (2019). Active Rollouts in MDP with Irreversible Dynamics. *Hal preprint hal-02177808*.

Mao, Hongzi, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula (2016a). Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16*. the 15th ACM Workshop. Atlanta, GA, USA: ACM Press, pp. 50–56.

— (2016b). Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 50–56.

Mao, Hongzi, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh (2019). Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*.

Mazyavkina, Nina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev (2021). Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research* 134, p. 105400.

McAllister, Daniel W, Terence R Mitchell, and Lee Roy Beach (1979). The contingency model for the selection of decision strategies: An empirical test of the effects of significance, accountability, and reversibility. In *Organizational Behavior and Human Performance*.

McIlroy-Young, Reid, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson (Aug. 2020). Aligning Superhuman AI with Human Behavior: Chess as a Model System. en. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Virtual Event CA USA: ACM, pp. 1677–1687.

McIlroy-Young, Reid, Russell Wang, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson (2021). Detecting Individual Decision-Making Style: Exploring Behavioral Stylometry in Chess. In *Neural Information Processing Systems (NeurIPS 2021)*.

— (2022). Learning Models of Individual Behavior in Chess. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '22. New York, NY, USA: Association for Computing Machinery, pp. 1253–1263.

Miller, C. E., A. W. Tucker, and R. A. Zemlin (1960). Integer Programming Formulation of Traveling Salesman Problems. *J. ACM* 7.4, pp. 326–329.

Mirhoseini, Azalia, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, et al. (2020). Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*.

Mirhoseini, Azalia, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, et al. (2017). Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, pp. 2430–2439.

Misra, Ishan, C Lawrence Zitnick, and Martial Hebert (2016). Shuffle and learn: unsupervised learning using temporal order verification. In *European Conference on Computer Vision*.

Mitchell, Melanie (1998). *An Introduction to Genetic Algorithms*. MIT Press.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. *CoRR* abs/1602.01783.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, et al. (Feb. 2015). Human-level control through deep reinforcement learning. *Nature* 518.7540, pp. 529–533.

Moldovan, Teodor Mihai and Pieter Abbeel (2012). Safe Exploration in Markov Decision Processes. In *International Conference on Machine Learning*.

Mouret, Jean-Baptiste and Jeff Clune (2015). Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*.

Nair, Suraj, Mohammad Babaeizadeh, Chelsea Finn, Sergey Levine, and Vikash Kumar (2020). Time reversal as self-supervision. In *International Conference on Robotics and Automation*.

Nazari, Mohammadreza, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takáč (2018). Reinforcement Learning for Solving the Vehicle Routing Problem. In *Advances in Neural Information Processing Systems*.

Nguyen, Manh Hung, Nathan Grinsztajn, Isabelle Guyon, and Lisheng Sun-Hosoya (2021). MetaREVEAL: RL-based Meta-learning from Learning Curves. In *Workshop on Interactive*

*Adaptive Learning co-located with European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (*ECML PKDD 2021*). Virtual, Bilbao.

Nguyen, Manh Hung, Lisheng Sun-Hosoya, Nathan Grinsztajn, and Isabelle Guyon (July 2022). Meta-learning from Learning Curves: Challenge Design and Baseline Results. In *IJCNN 2022 - International Joint Conference on Neural Networks*. Padua, Italy: IEEE, pp. 1–8.

Norris, James R. (1998). *Markov chains*. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press.

El-Nouby, Alaaeldin, Shuangfei Zhai, Graham W Taylor, and Joshua M Susskind (2019). Skip-Clip: Self-Supervised Spatiotemporal Representation Learning by Future Clip Order Ranking. *arXiv preprint arXiv:1910.12770*.

O. da Costa, Paulo R. de, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay (2020). Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning. In *Asian Conference on Machine Learning*.

Orhean, Alexandru Iulian, Florin Pop, and Ioan Raicu (July 2018). New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing* 117, pp. 292–302.

Ostrovski, Georg, Marc G. Bellemare, Aäron van den Oord, and Rémi Munos (2017). Count-Based Exploration with Neural Density Models. In *International Conference on Machine Learning*.

Ouyang, Long, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, et al. (2022). Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*.

Paliwal, Aditya, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, et al. (2020). *Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs*. en.

Parker-Holder, Jack, Aldo Pacchiano, Krzysztof Choromanski, and Stephen Roberts (2020). Effective Diversity in Population Based Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Parsonson, Christopher W. F., Alexandre Laterre, and Thomas D. Barrett (2022). Reinforcement Learning for Branch-and-Bound Optimisation using Retrospective Trajectories. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence* (*AAAI-23*).

Pascutto, Gian-Carlo (Apr. 4, 2019). *Leela Zero*. Version 0.17.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell (2017). Curiosity-Driven Exploration by Self-Supervised Prediction. In *International Conference on Machine Learning*.

Pickup, Lyndsey C, Zheng Pan, Donglai Wei, Yi Chang Shih, Changshui Zhang, Andrew Zisserman, et al. (2014). Seeing the arrow of time. In *Conference on Computer Vision and Pattern Recognition*.

Pierrot, Thomas, Valentin Macé, Félix Chalumeau, Arthur Flajolet, Geoffrey Cideron, Karim Beguir, et al. (2022). Diversity Policy Gradient for Sample Efficient Quality-Diversity Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*.

Pincus, M. (1970). A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems. *Journal of the Operations Research Society of America* 18.6.

Pol, Elise van der, Daniel Worrall, Herke van Hoof, Frans Oliehoek, and Max Welling (2020). MDP Homomorphic Networks: Group Symmetries in Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Pong, Vitchyr H., Murtaza Dalal, Steven Lin, Ashvin Nair, Shikhar Bahl, and Sergey Levine (2020). Skew-Fit: State-Covering Self-Supervised Reinforcement Learning. In *International Conference on Machine Learning*.

Pourchot, Aloïs and Olivier Sigaud (2019). CEM-RL: Combining evolutionary and gradient-based methods for policy search. In *International Conference on Learning Representations*.

Powell, Warren B. (2011). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. 2nd. Wiley Series in Probability and Statistics. Hoboken, NJ, USA: Wiley.

Pugh, Justin K., Lisa B. Soros, and Kenneth O. Stanley (2016). Quality Diversity: A New Frontier for Evolutionary Computation. *Frontiers in Robotics and AI* 3.

Puterman, Martin L (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Quintana-Ortí, E. S., G. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan (2009). Programming matrix algorithms-by-blocks for thread-level parallelism. 36.3.

Raffin, Antonin (2018). *RL Baselines Zoo*. https://github.com/araffin/rl-baselines-zoo.

Raffin, Antonin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann (2019). *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3.

Rahaman, Nasim, Steffen Wolf, Anirudh Goyal, Roman Remme, and Yoshua Bengio (2020). Learning the arrow of time for problems in reinforcement learning. In *International Conference on Learning Representations*.

Ramanathan, Vignesh, Kevin Tang, Greg Mori, and Li Fei-Fei (2015). Learning temporal embeddings for complex video analysis. In *International Conference on Computer Vision*.

Rusu, Andrei A., Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, J. Kirkpatrick, Koray Kavukcuoglu, et al. (2016). Progressive Neural Networks. *arXiv preprint arXiv:1606.04671*.

Saeed, Aaqib, David Grangier, Olivier Pietquin, and Neil Zeghidour (2020). Learning from Heterogeneous EEG Signals with Differentiable Channel Reordering. In *International Conference on Acoustics, Speech and Signal Processing*.

Sakellariou, Rizos and Henan Zhao (2004). A hybrid heuristic for DAG scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, p. 111.

Sanjaya, Ricky, Jun Wang, and Yaodong Yang (May 2022). Measuring the Non-Transitivity in Chess. en. *Algorithms* 15.5. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, p. 152.

Savinov, Nikolay, Anton Raichuk, Raphaël Marinier, Damien Vincent, Marc Pollefeys, Timothy Lillicrap, et al. (2019). Episodic curiosity through reachability. In *International Conference on Learning Representations*.

Schmidhuber, J. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers. In *International Conference on Simulation of Adaptive Behavior*.

Schrader, Max-Philipp B. (2018). *gym-sokoban*. https://github.com/mpSchrader/gym-sokoban.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015a). Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 1889–1897.

— (2015b). Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sermanet, Pierre, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, et al. (2018). Time-contrastive networks: Self-supervised learning from video. In *International Conference on Robotics and Automation*.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, et al. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529.7587, pp. 484–489.

Silver, David, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature* 529, pp. 484–503.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362.6419, pp. 1140–1144.

Şimşek, Özgür and Andrew G. Barto (2006). An Intrinsic Reward Mechanism for Efficient Exploration. In *International Conference on Machine Learning*.

Srinivas, Aravind, Michael Laskin, and Pieter Abbeel (2020). Curl: Contrastive unsupervised representations for reinforcement learning. In *International Conference on Machine Learning*.

Stockfish developers (2022). *Stockfish*. https://stockfishchess.org.

Strehl, Alexander L. and Michael L. Littman (2008). An analysis of model-based Interval Estimation for Markov Decision Processes. *Journal of Computer and System Sciences*.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press.

Sutton, Richard S., David A. McAllester, Satinder Singh, and Yishay Mansour (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, et al. (2014). Intriguing properties of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Tavakoli, Arash, Fabio Pardo, and Petar Kormushev (2018). Action Branching Architectures for Deep Reinforcement Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence,* (*AAAI-18*), *the 30th innovative Applications of Artificial Intelligence* (*IAAI-18*), *and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence* (*EAAI-18*).

Tennenholtz, Guy and Shie Mannor (2019). The Natural Language of Actions. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 6196–6205.

Tesauro, Gerald (1995). Temporal Difference Learning and TD-Gammon. *J. Int. Comput. Games Assoc.* 18.2, p. 88.

Timbers, Finbarr, Nolan Bard, Edward Lockhart, Marc Lanctot, Martin Schmid, Neil Burch, et al. (July 2022). Approximate Exploitability: Learning a Best Response. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. Ed. by Lud De Raedt. Main Track. International Joint Conferences on Artificial Intelligence Organization, pp. 3487–3493.

Topcuoglu, Haluk, Salim Hariri, and Min-you Wu (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13.3, pp. 260–274.

Toromanoff, Marin, Emilie Wirbel, and Fabien Moutarde (2019). Is Deep Reinforcement Learning Really Superhuman on Atari? Leveling the playing field. *arXiv preprint arXiv:1908.04683*.

Ullman, J.D. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences* 10.3, pp. 384–393.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, et al. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems*.

Vieillard, Nino, Olivier Pietquin, and Matthieu Geist (2020). Munchausen Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). Pointer Networks. In *Advances in Neural Information Processing Systems*.

Watkins, Christopher J. C. H. and Peter Dayan (1992). Q-learning. *Machine Learning* 8.3, pp. 279–292.

Weaver, Lex and Nigel Tao (2001). The Optimal Reward Baseline for Gradient-Based Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Weber, Théophane, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, et al. (2017). Imagination-augmented agents for deep reinforcement learning. In *Advances in Neural Information Processing Systems*.

Wei, Donglai, Joseph J Lim, Andrew Zisserman, and William T Freeman (2018). Learning and using the arrow of time. In *Conference on Computer Vision and Pattern Recognition*.

Williams, Ronald J. (1992a). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, pp. 229–256.

— (1992b). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, pp. 229–256.

Williams, Ronald J. and Jing Peng (1991). Function Optimization using Connectionist Reinforcement Learning Algorithms. *Connection Science* 3.3, pp. 241–268.

Wu, Qing, Zhiwei Wu, Yuehui Zhuang, and Yuxia Cheng (2018). Adaptive DAG Tasks Scheduling with Deep Reinforcement Learning. In *Algorithms and Architectures for Parallel Processing*. Ed. by Jaideep Vaidya and Jin Li. Vol. 11335. Cham: Springer International Publishing, pp. 477–490.

Wu, Xian, Wenbo Guo, Hua Wei, and Xinyu Xing (2021). Adversarial Policy Training against Deep Reinforcement Learning. In *USENIX Security Symposium*.

Wu, Yaoxin, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim (2021). Learning Improvement Heuristics for Solving Routing Problems. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–13.

Xin, Liang, Wen Song, Zhiguang Cao, and Jie Zhang (2021). Multi-Decoder Attention Model with Embedding Glimpse for Solving Vehicle Routing Problems. In *In Proceedings of the 35th National Conference on Artificial Intelligence, AAAI*.

Xu, Dejing, Jun Xiao, Zhou Zhao, Jian Shao, Di Xie, and Yueting Zhuang (2019). Self-supervised spatiotemporal learning via video clip order prediction. In *Conference on Computer Vision and Pattern Recognition*.

Yarats, Denis, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto (2021). Reinforcement Learning with Prototypical Representations. In *International Conference on Machine Learning*.

Zeelenberg, M (1999). Anticipated Regret, Expected Feedback and Behavioral Decision Making. In *Journal of Behavioral Decision Making*.

Zhang, Cong, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Chi Xu (2020). Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

Zheng, Jiongzhi, Kun He, Jianrong Zhou, Yan Jin, and Chu-Min Li (2021). Combining Reinforcement Learning with Lin-Kernighan-Helsgaun Algorithm for the Traveling Salesman Problem. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI-21)*.

Zhou, Yanqi, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, et al. (2019). GDP: Generalized Device Placement for Dataflow Graphs. *arXiv preprint arXiv:1910.01578*.

Ziegler, Daniel M., Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, et al. (2019). Fine-Tuning Language Models from Human Preferences. *arXiv preprint arXiv:1909.08593*.