ULB FACULTÉ DES SCIENCES

UNIVERSITÉ LIBRE DE BRUXELLES

Monte Carlo Tree Search with Advice

Thesis presented by Debraj CHAKRABORTY
in fulfilment of the requirements of the PhD Degree in Computer Science
academic year 2022-2023

Supervisor : Professor Jean-François Raskin
Formal Methods and Verification
Département d'informatique

Thesis jury :

Emmanuel FILIOT, (Université Libre de Bruxelles, chair)
Gilles GEERAERTS, (Université Libre de Bruxelles)
Jan KŘETÍNSKÝ, (Technical University Munich)
Kim G. LARSEN, (Aalborg University)

Acknowledgements

I would like to thank my supervisor Jean-François for his guidance, patience and encouragement throughout the whole journey. In these four years, I have learnt a lot from him: starting from approaching a problem to formalizing that approach and presenting it. I am fortunate to have him as my advisor. A big part of the work in this thesis would not be possible without the help of Damien. I am also grateful to work with Guillermo and Shibashis during the course of the PhD.

I would also like to thank the members of the jury, Emmanuel Filiot, Gilles Geeraerts, Jan Křetínský and Kim G. Larsen, who agreed to review this thesis and provided valuable feedbacks.

This work is partially supported by the ARC project Non-Zero Sum Game Graphs: Applications to Reactive Synthesis and Beyond (Fédération Wallonie-Bruxelles), and by the EOS project Verifying Learning Artificial Intelligence Systems (F.R.S.-FNRS and FWO). Computational resources used for the experiments have been provided by CÉCI, funded by F.R.S.-FNRS and by the Walloon Region.

I thank the people who had been in the formal methods group at ULB. Thanks to Raphaël, Nicolas, Ismaël, Shibashis, Léo, Marie, Damien, Sarah, Edwin, Ayrat, Mrudula, Léonard, Anirban, Sayan and others not only for providing a welcoming work environment but also for all the board game evenings, food, fries and drinks we had together. I also want to thank my friends from CMI: Suman, Anirban, Ritam, Sougata, Prantar, Nisarg, Siddarth and others for having occasional chats, online board games and sharing their OTT streaming service accounts with me.

I would also like to take the opportunity to thank my teachers from school and the professors from CMI for guiding me in the path of learning. In particular, I owe my interest in formal verification to Srivathsan for his instructive teaching and guidance. Finally and most importantly, I am forever grateful to my parents and my uncles for their affection, continuous support and encouragement.

দেবরাজ চক্রবর্ত্তী

Abstract

Markov decision processes (MDPs) are mathematical frameworks to model sequential decision-making. They are discrete-time stochastic models where the controller chooses actions based on the current state and then receives a reward and the state of the MDP is updated based on a probabilistic transition function. A strategy is a mapping from the execution of the system so far to the decisions available for the controller that tells how the controller should behave in the system.

In this thesis, we study how to efficiently combine techniques from formal methods and learning for online computation of a strategy that aims at optimizing the expected long term reward in large systems modelled as MDPs. This strategy is computed with receding horizon and using Monte Carlo tree search (MCTS). We augment the MCTS algorithm with the notion of advice which guides the search in the relevant part of the tree using exact methods. Such an advice can be symbolically written as a logical formula and computed on-the-fly using model checking tools. We show that the classical theoretical guarantees of Monte Carlo tree search are still maintained after this augmentation.

To lower the latency of MCTS algorithms with advice, we propose to replace advice coming from exact algorithms with an artificial neural network (NN). For this purpose, we implemented an expert imitation framework to train the neural network in order to replace the expert advice by a lower-latency neural advice. This neural network can also be used as a full-fledged strategy when minimal latency is required. This imitation framework relies on a data generation algorithm which leverages formal methods in order to obtain noise-free data. We use statistical model checking to detect when additional samples are needed and generate these samples on demand when the performance of the learnt neural network does not match the quality of the strategy computed offline.

To demonstrate the practical interest of our techniques, we implemented the frameworks on different systems modelled as MDPs: in the game of Pac-Man and Frozen Lake and also for safe and optimal scheduling of jobs in a task system.

Contents

Chapter 1

Introduction

A *decision-making procedure* is a system with the following sub-elements: a *controller* who is making the decision and an *environment* which is the part of the system not controlled by the controller. The evolution of the system is partly dependent on the behaviour of the controller and partly on the environment which could be antagonistic or stochastic in nature. A *strategy* is a way how the controller should behave in the system. In other words, it is a mapping from the execution of the system so far to the decisions available for the controller. There is also a *reward* which defines how good a decision taken by the controller is. The objective of the controller is to find a strategy that optimizes the reward accumulated in the long term.

When a system fails or runs inefficiently, it leads to financial consequences or even loss of lives. For example, in late 1970s, the state of Arizona, USA was spending about 14 million USD extra a year for maintenance of their existing highways until they developed a pavement management system based on a Markov decision process model to improve allocation of its limited resources while ensuring quality of the roads [GKW82]. A software flaw in the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation [LT93]. This shows the necessity of constructing safe and efficient controller strategy for systems.

The importance of this has triggered numerous works in different research communities within computer science, most notably in formal methods, and in artificial intelligence and machine learning. The works done in these research communities have respective weaknesses and complementary strengths. On the one hand, algorithms developed in formal methods are generally complete and provide strong guarantees on the optimality of computed solutions, but they tend to be applicable to models of moderate size only. On the other hand, algorithms developed in artificial intelligence and machine learning usually scale to larger models but only provide weaker guarantees. Instead of opposing the two sets of algorithms, this thesis tries to combine the strengths of the two approaches in order to offer new hybrid algorithms that scale better while providing guarantees.

In this thesis, as a running example, we will use a version of the game Pac-Man, a

very popular arcade video game, created by Toru Iwatani in 1980 :

Example 1.1 In this game, Pac-Man has to eat food pills in an enclosed grid as fast as possible while avoiding the ghosts. The agents (Pac-Man and the ghosts) can travel in four directions unless they are blocked by the walls in the grid, and ghosts cannot reverse their direction.

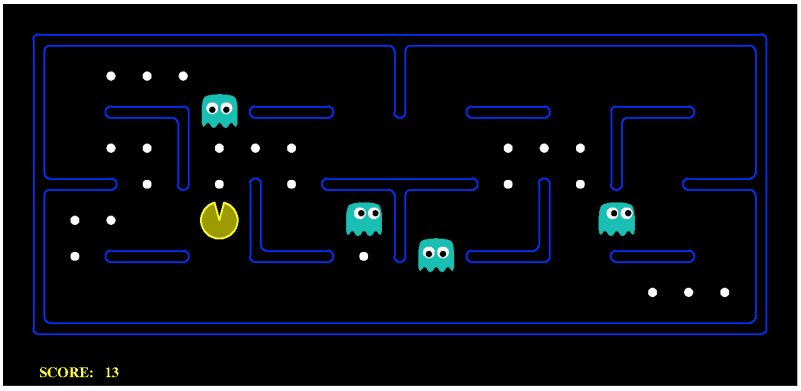

Figure 1.1: PAC-MAN with 4 ghosts in a 9×21 grid. The figure has been generated using the code from [DK].

The score decreases by 1 at each step, and increases by 10 whenever Pac-Man eats a food pill. A win (when the Pac-Man eats all the food pills in the grid) increases the score by 500. Similarly, a loss (when the Pac-Man gets eaten by a ghost), decreases the score by 500. Here the ghosts can be considered the part of the environment, not controlled by the Pac-Man. The objective of the game is to find a strategy that optimizes the accumulated score. □

We have chosen this system as a running example for reasons. The state space of the underlying system is way too large for the state-of-the-art implementations of exact algorithms. Indeed, the reachable state space of the small grid shown here has approximately 10^{16} states. This calls for heuristic approaches used in learning. Even in this case, we will show that, formal methods can play an important role and help the heuristic techniques.

1.1 Background

Formal methods In formal methods, the system is represented as a mathematical model and the desired properties are described as a logical formula. This is useful to assert

the correctness of the controller (*model checking*) or to automatically design a controller strategy satisfying the specification (*synthesis*).

Model checking was introduced in the early 1980s by Clarke and Emerson [CE81] and Queille and Sifakis [QS82]. This uses brute force to examine the entire state-space in the model to check whether the specification is satisfied. Given its sound mathematical foundation, it has been proved to be an effective technique. Typical properties that can be checked using model checking are of a qualitative nature like: Can Pac-Man stay safe in next 8 steps irrespective of whatever legal move the ghosts make? The answer of these properties can be either 'yes' or 'no'. While qualitative specifications are sufficient to model yes-no properties, formal methods also help to answer quantitative properties like: Given a strategy for Pac-Man, what is the score we can expect if the ghosts move uniformly in the grid? For further details on model checking, we refer to [BK08].

In a synthesis problem, the goal is to find a strategy such that if the controller follows that strategy, the system would satisfy the given specification. This problem was originally stated by Alonzo Church [Chu57] in the context of circuits. In [McN65], McNaughton treated this problem in the framework of infinite-duration games. Büchi and Landweber [BL69] solved it for specifications definable by monadic second order logic. Algorithms for synthesis problems has been efficiently implemented in multiple tools, for example Acacia+ [Boh+12] can synthesis strategies for specifications definable in linear temporal logic.

Markov decision process A popular mathematical framework used to model decision-making procedures with discrete state space is *Markov decision process* or MDP which was introduced by Richard Bellman [Bel57a] in 1957. In an MDP, the transition of the states satisfy *Markov property* i.e. given a state and a decision taken by the controller, the immediate reward and the state evolutions are independent of all previous states and decisions of the controller. We call the decisions available at a state *actions*. MDPs have been used to model systems in various disciplines, including robotics, economics, biology and manufacturing.

The *value* of a state is the amount of reward the controller can expect over the time starting from that state. Bellman showed that these values form a functional equation, now named as *Bellman equation*, and also showed a method, now popular as *dynamic programming* [Bel57b], to solve these type of equations. Solving this equation also gives us a strategy for the controller that optimizes the expected reward. The theory of MDP was

further developed and extended by Howard [How60], Blackwell [Bla62; Bla65] and others. We refer to [Put94] as a comprehensive book on Markov decision processes.

Model checking algorithms for Markov decision processes has been efficiently implemented in tools like Storm [Deh+17] and PRISM [KNP11].

Planning with receding horizon control The dynamic programming techniques suffer from what Bellman called 'the curse of dimensionality', meaning that their computational requirements grow with the number of states. So it often becomes impractical to calculate the whole strategy beforehand. In this case, we can apply an online iterative approach where the controller, upon visiting a new state, computes a good action and takes it. Then the state evolves stochastically to a new state according to the dynamics specified by the MDP and the same process is repeated from the new state. This known as *decision time planning* [SB18].

Specifically, we plan using *receding horizon control* [KH06] which has been used in process control since the 1980s. In this approach, the controller fixes a horizon H and finds a strategy that optimizes a combination of total reward that can be accumulated in H steps and an estimation of the value of the state which will be reached in H steps. Then the controller takes an action based on this strategy. For example, in the game of Pac-Man, we can guess that a state is bad if the Pac-Man is surrounded by ghosts and far from the food. Then instead of computing a full plan involving faraway food pills and ghosts, it is more intuitive for a player to aim for eating food pills nearby and trying to avoid ghosts that are close as long as Pac-Man does not end up in such a bad state in near future.

Monte Carlo tree search Receding horizon techniques are often coupled with heuristic search algorithms that avoid the full exploration at the expense of approximation. These algorithms are based on Monte Carlo methods[1], developed by Stanisław Ulam and John von Neumann in the late 1940s, which use random simulations to solve intractable problems. Abramson [Abr87] first implemented this in the context of games. The recent, most popular and most successful heuristic search algorithm is *Monte Carlo Tree Search* or MCTS.

MCTS uses random simulations to identify the most promising action at the current state by iteratively building a search tree which dictates which part of the MDP to simulate

[1] named after a casino in Monaco, as a codename, as the research was part of the development of fission weapons in Los Alamos National Laboratory, USA.

from. This idea of 'adaptive' sampling was introduced in [Cha+05] which talked about balancing between *exploration* (looking in the part of the state-space that has not been sampled enough yet) and *exploitation* (looking in the part of the state-space that appears to be promising). The algorithm UCT (Upper Confidence Bounds applied to Trees) was developed in [KS06] which deals with this exploration vs exploitation dilemma. MCTS showed promising performance in games with very large state space like Go [Cou06]. The most significant implementation of MCTS is in the computer program AlphaGo [Sil+16], developed by Google Deepmind, which was able to defeat Lee Sedol, one of the strongest player in history, in a five-game match with a score of four to one.

Artificial neural network *Artificial neural networks* or ANN were originally created by Warren McCulloch and Walter Pitts [MP43] as models which loosely model the neurons in a brain. These are widely used to approximate non-linear functions from a dataset of known input and outputs. With the advancement of digital electronics, development of practical artificial neural networks became feasible in the 1980s which made it popular in the field of artificial intelligence and machine learning.

A *feed-forward neural network* is a directed graph divided in multiple layers where each layer is a function acting on the output of the previous layer. Cybenko [Cyb89] showed that a network with a single hidden layer containing a large enough number of sigmoid[2] units can approximate any continuous function to any degree of accuracy. Despite this 'universal approximation' property of single-hidden-layer networks, in practice approximating complex functions needs special architecture with multiple layers. Study of these 'deep' neural networks are known as *deep learning*[GBC16].

Instead of explicitly writing the value of states in an MDP into a table during dynamic programming, neural networks can be used to learn the values (as done during *deep Q-learning* [Mni+15]). They can also be used to mimic an 'expert' strategy given training data of the encountered states and actions performed using that strategy (as done during *imitation learning*).

[2] a function of the form $f(x) = \frac{1}{1+e^{-x}}$.

1.2 Contributions

Symbolic advice While MCTS may offer reasonable performances, they usually need considerable adjustments that depend on the application to really perform well. One way to adapt MCTS to a particular application is to bias the search towards promising subspaces by supplying domain-specific knowledge. We show that this can be done using techniques from formal methods. More precisely, a formal specification can be used to guide the simulations to improve the quality of the search. We call these specifications *advice*. In the game of Pac-Man, since getting eaten by a ghost leads to an irrecoverably bad reward, an example of good advice would be to avoid getting eaten by a ghost[3]. This can be used to prune the search tree by removing the part of the tree containing the losing states. For example, we can ask a question as follows:

> Q: If Pac-Man takes the action 'North' from our current state, can he make sure to stay safe in next 8 steps irrespective of whatever legal moves the ghosts make?

If the answer to our question is 'No', we can ignore the action 'North' and never simulate from the subtree under the action 'North'.

On the theoretical side, we study the impact of using symbolic advice on the guarantees offered by MCTS. We identify sufficient conditions for the symbolic advice to preserve the convergence guarantees of the MCTS algorithm. On a more practical side, we show how symbolic advice can be implemented using different formal method techniques. These results have been partially reported in [BCR20].

To demonstrate the practical interest of our techniques, we implemented a framework to run MCTS algorithms guided by symbolic advice on systems modelled as MDPs. We applied our implementation on different systems: in the game of Pac-Man and also for safe and optimal scheduling of jobs in a task system. The last work is reported in [Bus+21].

Imitation learning of expert strategies Our another contribution is to create an *expert imitation framework* to train an artificial neural network in order to replace exact advice by lower-latency neural advice, or even to imitate the *expert* strategy that can be computed offline. This imitation framework relies on a data generation algorithm which leverages

[3] This could also be considered a piece of advice in real life; although statistically, compared to Pac-Man, a sane human encounters much less number of ghosts in their life.

formal methods in order to obtain 'perfect data' for our samples on the one hand and to generate additional samples on the other hand, as long as statistical model checking[4] indicates that it is required to improve the quality of the imitation.

In general, we define a ranking of actions for every state such that the maximally ranked elements are those played by the strategy. Intuitively, the ranking tells us how good every action is from the current state. We propose to train a neural network to learn such a ranking function as an offline step. This neural network can then be used as a full-fledged strategy or as a neural advice to efficiently guide MCTS. This neural advice generated aims for an expected reward comparable with the expert advice, for a fraction of its online latency.

In order to stop the data aggregation loop, we monitor the practical performance of the neural network. Classical metrics to evaluate a neural network may not be representative of the expected reward the neural network will obtain when used as an advice or a full strategy. Indeed, a strategy may make mistakes at crucial moments despite being almost always correct in its decisions, leading to vastly different outcomes. Thus, we propose using statistical model-checking to compute an approximation of the expected reward of our learnt strategies.

We implemented this framework to imitate strategies and advice in two systems: in the game of Pac-Man and Frozen-Lake.

1.3 Related works

Simulation-based techniques has been used in statistical model checking [Dac+16; YS02], which uses Monte Carlo methods to model check larger systems to get approximate results that hold with high probability. The learning based methods used in planning and reinforcement learning has been explored in [Brá+14; KPR18] for quantitative objectives and in [Ash+17; KM17; Aga+22] for long-term rewards. Works has been done for partially observable MDPs [Cha+17b] and other models which are too big to analysis using classical model checking methods. Monte Carlo tree search has been used for priced timed automata when the models are too big for existing complete methods [Jen+22].

In [Ash+18], MCTS has been combined with bounded real-time dynamic program-

[4] In statistical model checking, quantitative properties are checked by simulating enough executions of the system till an estimation of the actual value can be computed with required precision and the confidence level.

ming (BRTDP) technique to create hybrid algorithms to verify reachability in large MDPs. In [BW15], the authors have integrated shallow minimax search in the MCTS framework. In [Als+18; Jan+14], the authors provide a general framework to add safety properties to reinforcement learning algorithms via *shielding*. These techniques analyse statically the full state space of the game in order to compute a set of unsafe actions to avoid. A variation of shielding called safe padding has been studied in [HAK20].

In [GGR18], the scheduling problem was introduced but the authors made the assumption that the underlying distributions of the tasks are known. In the work mentioned in the thesis, we drop this assumption here and provide learning algorithms.

Using deep learning to replace expert (but expensive) policies by learnt strategies is known to be advantageous when the expert policy is unable to meet real-time (latency) constraints (see, e.g. [Iva+19, Section 5.2] and [Her+18]). In order to obtain a satisfactory dataset to train on, we propose a sharp variant of the DAgger algorithm, a dataset aggregation technique introduced in [RGB11; RB10]. A notable difference is that we propose to use model checkers instead of human experts in order to get better-quality data. We also identify so-called *counterexample* configurations in order to guide the aggregation loop to the most interesting states. This is reminiscent of CEGAR approaches for hybrid systems such as [CDS19] that identify states violating a property then focus the deep learning procedures on such states.

1.4 Organization of this thesis

The thesis is organized in multiple chapters. Chapter 2 introduces necessary notions and recalls the results needs for the next chapters. In Section 2.4, we also present an algorithm to find practical strategies for reachability in an MDP. In Chapter 3 and 4, we introduced techniques from formal methods that can be used in decision-time planning to find optimal strategies in MDPs. Specifically, in Chapter 4, we introduced the notion of advice and described their used in Monte Carlo tree search in Chapter 5. Chapter 7 discusses learning techniques to imitate a strategy generated by exact of heuristic methods. Chapter 5 and 8 shows applications of the techniques presented in this thesis on different systems modelled as MDPs.

Chapter 2

Preliminaries

2.1 Transition systems and games

In formal methods, *Transition systems* are used to describe the behaviour of a system. Transition systems are *directed graphs* where the vertices represent *states* of the system and edges model the *transitions* which denotes how the states of the system change. The states are often labelled with *atomic propositions* which describes the characteristics satisfied by that state. Formally we define the transition system like this:

Definition 2.1 (Transition system)

A (finite) transition system or a TS is a tuple $T = (S, E, AP, L)$*, where*

- S *is a finite set of states,*
- E *is a relation in* $S \times S$ *such that* (s, s') *denotes that there is a transition available from* s *to* s'*,*
- AP *is a finite set of atomic proposition,*
- L *is the labelling function from* S *to* 2^{AP} *such that* $L(s)$ *denotes the propositions in* AP *that is satisfied by the state* s*.*

The set of propositions AP often is not explicitly defined. In that case, it is assumed that $AP = S$ with labelling function $L(s) = \{s\}$.

For a transition system $T = (S, E, AP, L)$, we will call the graph $G = (S, E)$ the *underlying graph* of T.

Paths The transition system starts from some state s_0 in S and evolves according to the relation E. That means, if s is the current state, a next state s' is chosen in a *nondeterministic* fashion, such that $(s, s') \in E$. We call a sequence of states $p = q_0 q_1 \dots q_i$ an i-length *path* if for all $t \in [0, i-1]$, $(q_t, q_{t+1}) \in E$. An *infinite path* is an infinite sequence $p = q_0 q_1 \dots$ such that for all $t \in \mathbb{N}$, $(q_t, q_{t+1}) \in E$. We denote the finite prefix of length t of a infinite path $p = q_0 q_1 \dots$ by $p_{|t} = q_0 q_1 \dots q_t$. We respectively denote the last and first state of a path

$p = s_0 a_0 s_1 \dots s_n$ by $\mathsf{last}(p) = s_n$ and $\mathsf{first}(p) = s_0$. Let $p = s_0 s_1 \dots s_i$ and $p' = s'_0 s'_1 \dots s'_j$ be two paths such that $s_i = s'_0$. Then, $p \cdot p'$ denotes the path $s_0 s_1 \dots s_i s'_1 \dots s'_j$.

Example 2.1 Consider the transition system TS described in Figure 2.1 which models the movement of a robot. When the robot moves, it either falls (e.g., due to some defect in its motor) or it keeps moving. From the FALLEN state, the robot can start moving again.

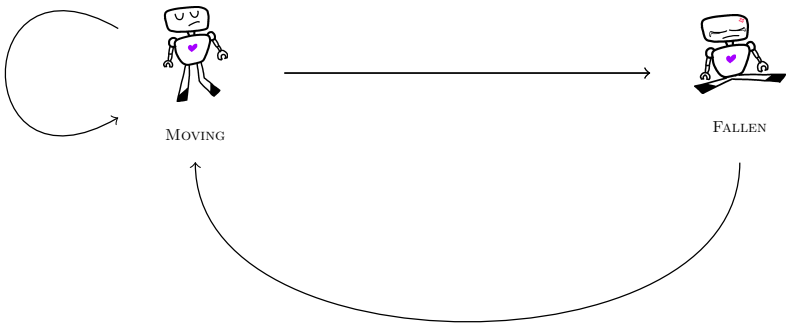

Figure 2.1: A transition system

This transition system contains two states. From state MOVING, there are two transitions available leading either to the state FALLEN or to itself. From the state FALLEN, there is only one transition. □

2.1.1 Two player games

A *game* is played between two players in a transition system, namely player 0 and player 1. The set of states S is partitioned into two sets S_0 and S_1. The states in S_0 is controlled by player 0 so that player 0 can decide the next state the system would evolve to. Similarly, the vertices in S_1 are controlled by player 1.

The winning condition in the game is defined by a set of infinite paths $\mathsf{Win}_0 \subseteq S^\omega$. Player 0 wins the game if the infinite path generated during the game is in Win_0. Player 1 wins the game if the infinite path generated during the game is not in Win_0. Formally we define 2-player game as follows:

Definition 2.2 (2-player game)

A (2-player) game is a tuple $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ *where*

- $T = (S, E, AP, L)$ *is a transition system,*
- (S_0, S_1) *is a partition of the set of states* S *in* T*,*
- Win_0 *is a set of infinite paths.*

For a game $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ where $T = (S_0 \uplus S_1, E, AP, L)$, we call the tuple (S, E, S_0, S_1) the *underlying game graph* of $\mathcal{G}$.

Strategies A (deterministic) strategy *for player* 0 in the 2-player game $\mathcal{G}$ is a function $\sigma : S^*S_0 \to S$ that assigns to each path $s_0 s_1 \dots s_n$, such that $s_n \in S_0$, a state $s_{n+1} \in S$ with $(s_n, s_{n+1}) \in E$.

For a game $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$, a state s and a strategy σ for player 0, let $\mathsf{Paths}_{\mathcal{G}}(s, \sigma)$ be the infinite paths $p = s_0 s_1 \dots$ in T such that $s_0 = s$ and for all $t \in \mathbb{N}$, if $s_t \in S_0$ then $\sigma(p_{|t}) = s_{t+1}$. A strategy σ is winning for player 0 if $\mathsf{Paths}_{\mathcal{G}}(s, \sigma) \subseteq \mathsf{Win}_0$.

Analogously, a *strategy for player* 1 is a function $\tau : S^*S_1 \to S$ that assigns to each path prefix $s_0 s_1 \dots s_n$, with $S_n \in S_1$, a state $s_{n+1} \in S$ with $(s_n, s_{n+1}) \in E$. For a game $\mathcal{G} = (G, S_0, S_1, \mathsf{Win}_0)$, a state s and a strategy τ for player 1, let $\mathsf{Paths}_{\mathcal{G}}(s, \tau)$ be the infinite paths $p = s_0 s_1 \dots$ in T such that $s_0 = s$ and for all $t \in \mathbb{N}$, if $s_t \in S_1$ then $\tau(p_{|t}) = s_{t+1}$. A strategy τ is winning for player 1 if all paths in $\mathsf{Paths}_{\mathcal{G}}(s, \tau) \cap \mathsf{Win}_0 = \emptyset$.

A *non-deterministic strategy* for player 0 is a function $\sigma : S^*S_0 \to 2^S$ that assigns to each path $s_0 s_1 \dots s_n$, with $s_n \in S_0$, a set of states $S_{n+1} \subseteq S$ with $(s_n, s_{n+1}) \in E$ for all $s_{n+1} \in S_{n+1}$. Similar to deterministic strategies, for a game $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$, a state s and a non-deterministic strategy σ for player 0, let $\mathsf{Paths}_{\mathcal{G}}(s, \sigma)$ be the infinite paths $p = s_0 s_1 \dots$ in T such that $s_0 = s$ and for all $t \in \mathbb{N}$, if $s_t \in S_0$ then $s_{t+1} \in \sigma(p_{|t})$. We call this strategy winning for player 0 if $\mathsf{Paths}_{\mathcal{G}}(s, \sigma) \subseteq \mathsf{Win}_0$.

For a non-deterministic strategy σ and a deterministic strategy σ', we say $\sigma' \subseteq \sigma$ if for all paths $p \in S^*$, $\sigma'(p) \in \sigma(p)$. A strategy σ is *memoryless* if $\sigma(p)$ depends only on $\mathsf{last}(p)$.

Reachability and safety games Depending on the condition φ, we get different types of 2-player games. For example:

Definition 2.3 (Reachability game)

Let $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ *where* $T = (S, E, AP, L)$, $F \subseteq S$ *and* $p = s_0 s_1 \ldots$ *is in* Win_0 *if there exists an* $i \in \mathbb{N}$ *such that* $s_i \in F$. $\mathcal{G}$ *is called a* reachability game.
Given a horizon $H \in \mathbb{N}$, *let* $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ *where* $T = (S, E, AP, L)$, $F \subseteq S$ *and* $p = s_0 s_1 \ldots$ *is in* Win_0 *if there exists an* $i \leq H$ *such that* $s_i \in F$. $\mathcal{G}$ *is called a* finite horizon reachability game.

Winning strategies (if exist) for both players in these games can be found by iteratively constructing *attractor sets* using backward breadth-first search in time $\mathcal{O}(|S|+|E|)$ [BCJ18, Theorem 1]. Also, the resulting strategies are memoryless.

Definition 2.4 (Safety game)

Let $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ *where* $T = (S, E, AP, L)$, $F \subseteq S$ *and* $p = s_0 s_1 \ldots$ *is in* Win_0 *if for all* $i \in \mathbb{N}$, $s_i \in F$. $\mathcal{G}$ *is called a* safety game.
Given a horizon $H \in \mathbb{N}$, *let* $\mathcal{G} = (T, S_0, S_1, \mathsf{Win}_0)$ *where* $T = (S, E, AP, L)$, $F \subseteq S$ *and* $p = s_0 s_1 \ldots$ *is in* Win_0 *if for all* $i \leq H$, $s_i \in F$. $\mathcal{G}$ *is called a* finite horizon safety game.

Note that safety games are reachability games for player 1 with the target set $S \setminus F$. So for these games also memoryless winning strategies (if exist) for both players are computed in time $\mathcal{O}(|S| + |E|)$.

Theorem 2.1 ([BJW02, Fact 7])

In a safety game $\mathcal{G}$, *there exists a unique memoryless winning non-deterministic strategy* σ *for player* 0 *such that for all strategies* σ' *winning for player* 0 *in* $\mathcal{G}$, $\sigma' \subseteq \sigma$.

We call this non-deterministic strategy in safety games, *most general strategy* for safety.

2.2 Probability

We briefly introduce the concepts of probability measure and random variables which will be useful for reasoning in the following chapters. The reader may refer to books on

probability theory e.g. [Ros10; AD99] for further details.

Given a set of outcomes Ω, a *σ-algebra* $\mathcal{F} \subseteq 2^{\Omega}$ is a set consisting of subsets of Ω that contains the empty set $\emptyset$ and is closed under complementation and countable unions. The elements of the set $\mathcal{F}$ is called *events*.

A *probability measure* on $(\Omega, \mathcal{F})$ is a function $\mathbb{P} : \mathcal{F} \to [0, 1]$ such that $\mathbb{P}(\Omega) = 1$ and if $\{A_i\}_{i=1}^{\infty} \subseteq \mathcal{F}$ is a countable collection of pairwise disjoint events, then

$$\mathbb{P}(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \mathbb{P}(A_i) \,.$$

A *probability space* is a tuple $(\Omega, \mathcal{F}, \mathbb{P})$ where $\mathcal{F}$ is a σ-algebra on Ω and $\mathbb{P}$ is a probability measure on $(\Omega, \mathcal{F})$. For an event $e \in \mathcal{F}$, the value $\mathbb{P}(e)$ is called the *probability* of the event e. For the case when Ω is countable, we can directly define a *probability distribution* $\mathbb{P} : \Omega \to [0, 1]$ such that $\sum_{\omega \in \Omega} \mathbb{P}(\omega) = 1$. In that case, the probability of an event e is $\sum_{\omega \in e} \mathbb{P}(\omega)$. For a countable set S, we denote the set of all probability distribution by $\mathcal{D}(S)$. The *support* of a distribution $d \in \mathcal{D}(S)$ is the set $\mathsf{Supp}(d) = \{s \in S \mid d(s) > 0\}$.

We say two distributions d and d' are *structurally identical* if $\mathsf{Supp}(d) = \mathsf{Supp}(d')$. Given two structurally identical distributions d and d', for $0 < \epsilon < 1$, we say that d is *ϵ-close* to d', denoted $d \sim^{\epsilon} d'$, if $\mathsf{Supp}(d) = \mathsf{Supp}(d')$, and for all $a \in \mathsf{Supp}(d)$, we have that $|d(a) - d'(a)| \leq \epsilon$.

A (real-valued) *random variable* is a function $X : \Omega \to \mathbb{R}$. The probability that X takes a value in a set $S \subseteq \mathbb{R}$ is written as

$$\mathbb{P}(X \in S) = \mathbb{P}(\{\omega \in \Omega \mid X(\Omega) \in S\}) \,.$$

Example 2.2 The possible outcomes of a fair coin toss have two outcomes $\Omega = \{\text{heads}, \text{tails}\}$. With 2^{Ω} as the σ-algebra, the probability measure $\mathbb{P}$ is given by

$$\mathbb{P}(\phi) = 0,\ \mathbb{P}(\{\text{heads}\}) = \mathbb{P}(\{\text{tails}\}) = \frac{1}{2},\ \mathbb{P}(\{\text{heads,tails}\}) = 1.$$

We can then have a real-valued random variable X that models a payoff of 1 for a successful bet on heads as follows:

$$X(\omega) = \begin{cases} 1 & \text{if } \omega = \text{heads}, \\ 0 & \text{if } \omega = \text{tails}. \end{cases}$$

This gives a probability distribution for X where $\mathbb{P}(X = 1) = \mathbb{P}(X = 0) = \frac{1}{2}$. □

A set of n random variables $X_1, X_2 \ldots X_n$ are *independent* if for any sequence of sets

$S_1, S_2 \ldots S_n$,

$$\mathbb{P}(\bigwedge_{i=1}^{n} X_i \in S_i) = \prod_{i=1}^{n} \mathbb{P}(X_i \in S_i) \ .$$

We state the following inequality which is a useful property for random variables:

Theorem 2.2 (Chernoff-Hoeffding inequality [Hoe63, Theorem 2])

Let $X_1, X_2, \ldots X_n$ be independent random variables in $[0, 1]$. Let $S_n = \sum_n X_i$. Then for all $t > 0$, the following bounds hold:

$$\mathbb{P}\Big[\ |S_n - \mathbb{E}[S_n]| \geq t\Big] \leq 2 \cdot \exp\left(-\frac{2t^2}{n}\right)$$

2.2.1 Learning discrete finite distributions

To learn an unknown discrete distribution d defined on a finite domain $\mathsf{Supp}(d)$, we collect independent and identically distributed (i.i.d.) samples from that distribution and infer a model of it. Formally, given a sequence $\mathbb{S} = (s_j)_{j \in J}$ of samples drawn i.i.d. from the distribution d, we denote by $d(\mathbb{S}) : \mathsf{Supp}(d) \to [0, 1]$, the function that maps every element $a \in \mathsf{Supp}(d)$ to its relative frequency in $\mathbb{S}$, *i.e.*,

$$d(\mathbb{S})(a) = \frac{|\{j \in J \mid s_j = a\}|}{|J|} \ .$$

The following lemma tells us that if the size of S is large enough then the model $d(\mathbb{S})$ is close to the actual d with high probability:

Lemma 2.1

For all finite discrete distributions d with $|Supp(d)| = r$, for all $\epsilon, \gamma \in (0, 1)$ such that $\min_{a \in \mathsf{Supp}(d)}(d(a)) > \epsilon$, if $\mathbb{S}$ is a sequence of $n \geq \frac{r}{2\epsilon^2}(\ln 2r - \ln \gamma)$ i.i.d. samples drawn from d, then $d \sim^{\epsilon} d(\mathbb{S})$ with probability at least $1 - \gamma$.

Proof For a distribution d, and an element e in $\mathsf{Supp}(d)$, suppose we collect m independent and identically distributed (i.i.d.) samples from d. Let $X_1^{d_e}, \ldots, X_m^{d_e}$ be independent and identically distributed random variables where

$$X_m^{d_e} = \begin{cases} 1 & \text{if } s_j = e, \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\mathbb{E}(X_m^{d_e}) = d(e)$. Let $S_m^{d_e} = \sum_m X_n^{d_e}$. Then, $S_m^{d_e} = d(\mathbb{S})(e)$, the relative frequency

of e. As the variables are i.i.d., we have $\mathbb{E}(S_n^{d_e}) = m \cdot d(e)$.

Then from Theorem 2.2, we have:

$$\mathbb{P}\Big[\,|S_m - \mathbb{E}[S_m]| \geq \epsilon\Big] \leq 2 \cdot \exp\left(-\frac{2\epsilon^2}{m}\right).$$

Putting $m \geq \frac{1}{2\epsilon^2}(\ln 2r - \ln \gamma)$, we get,

$$\mathbb{P}\Big[\,|S_m - \mathbb{E}[S_m]| \geq \epsilon\Big] \leq \frac{\gamma}{r}.$$

Since there are r elements in the domain, we need a total of at least $m \cdot r$ samples to approximate d, and hence the result. □

2.3 Probabilistic systems

2.3.1 Markov chain

Markov chains are transition systems where the state of the system evolves according to a probability distribution. This distribution only depends on the current state of the system and not on, for example, the whole history of the evolution of the system till the current state. A Markov chain can be defined as follows:

Definition 2.5 (Markov chain)

A (discrete-time) Markov chain or an MC is a tuple $M = (S, P, AP, L)$, where

- *S is a countable set of states,*
- *P is a mapping from S to $\mathcal{D}(S)$ such that $P(s)(s')$ denotes the probability of moving from state s to state s' in a single transition,*
- *AP is a finite set of atomic proposition,*
- *L is the labelling function from S to 2^{AP} such that $L(s)$ denotes the propositions in AP that are satisfied by the state s.*

For ease of notation, we denote the value $P(s)(s')$ as $P(s, s')$. The set of propositions AP often is not explicitly defined. In that case, it is assumed that $AP = S$ with labelling function $L(s) = \{s\}$.

Paths For a Markov chain M, a *path* of length $i > 0$ is a sequence of $i+1$ consecutive states. We say that $p = s_0 s_1 \dots s_i$ is an i-length *path* in the MC M if for all $t \in [0, i-1]$, $s_{t+1} \in \mathsf{Supp}(P(s_t))$. We also consider states to be paths of length 0.

An *infinite path* is an infinite sequence $p = s_0 s_1 s_2 \dots$ of states such that for all $t \in \mathbb{N}$, $s_{t+1} \in \mathsf{Supp}(P(s_t))$. We denote the finite prefix of length t of a finite or infinite path $p = s_0 s_1 \dots$ by $p_{|t} = s_0 \dots s_t$. For a finite or infinite path $p = s_0 s_1 \dots$, we denote its $(i+1)^{th}$ state by $p[i] = s_i$. We respectively denote the last and first state of a path $p = s_0 s_1 \dots s_n$ by $\mathsf{last}(p) = s_n$ and $\mathsf{first}(p) = s_0$. Let $p = s_0 s_1 \dots s_i$ and $p' = s'_0 s'_1 \dots s'_j$ be two paths such that $s_i = s'_0$. Then, $p \cdot p'$ denotes $s_0 s_1 \dots s_i s'_1 \dots s'_j$.

For an MC M, the set of all finite paths of length i is denoted by Paths^i_M. We denote the set of all finite paths in M by Paths_M and the set of finite paths of length at most H by $\mathsf{Paths}^{\leq H}_M$. The set of all infinite paths is denoted by Paths^ω_M.

If $p \in \mathsf{Paths}^i_M$ and $i \leq j$, then let $\mathsf{Paths}^j_M(p)$ denote the set of paths p' in Paths^j_M such that there exists $p'' \in \mathsf{Paths}^{j-i}_M$ with $p' = p \cdot p''$. In particular, let $\mathsf{Paths}^i_M(s)$ denote the set of paths p in Paths^i_M such that $\mathsf{first}(p) = s$. Similarly, if $p \in \mathsf{Paths}_M$, then let $\mathsf{Paths}_M(p)$ denote the set of paths p' in Paths_M such that there exists $p'' \in \mathsf{Paths}_M$ with $p' = p \cdot p''$. For $p \in \mathsf{Paths}_M$, let $\mathsf{Paths}^\omega_M(p)$ denote the set of paths p' in Paths^ω_M such that there exists $p'' \in \mathsf{Paths}^\omega_M$ with $p' = p \cdot p''$. $\mathsf{Paths}^\omega_M(p)$ is called the *cylinder set* of p.

σ-algebra and probability measures The σ-algebra $\mathcal{F}^M$ associated with the MC M is the smallest σ-algebra that contains the cylinder sets $\mathsf{Paths}^\omega_M(p)$ for all $p \in \mathsf{Paths}_M$. For a state s in S, a measure is defined for the cylinder sets as

$$\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s_0 s_1 \dots s_i)) = \begin{cases} \prod_{t=0}^{i-1} P(s_t, s_{t+1}) & \text{if } s_0 = s \\ 0 & \text{otherwise.} \end{cases}$$

Also $\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s)) = 1$ and $\mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(s')) = 0$ for all states $s' \neq s$. From Carathéodory's extension theorem [AD99, section 1.3.10], this can be extended to a unique probability measure $\mathbb{P}_{M,s}$ on the aforementioned σ-algebra. In particular, if $\mathcal{C} \subseteq \mathsf{Paths}_M$ is a set of finite paths forming pairwise disjoint cylinder sets, then

$$\mathbb{P}_{M,s}(\cup_{p \in \mathcal{C}} \mathsf{Paths}^\omega_M(p)) = \sum_{p \in \mathcal{C}} \mathbb{P}_{M,s}(\mathsf{Paths}^\omega_M(p)) \,.$$

Example 2.3 Consider the Markov chain M described in Figure 2.2 which models the movements of a robot. When the robot moves, with probability 0.15, the robot falls; and

with probability 0.85, it keeps moving. From the fallen state, the robot can start moving again with probability 0.5 in every try. This Markov chain contains two states, namely MOVING and FALLEN. The probability distributions are represented in the dotted lines.

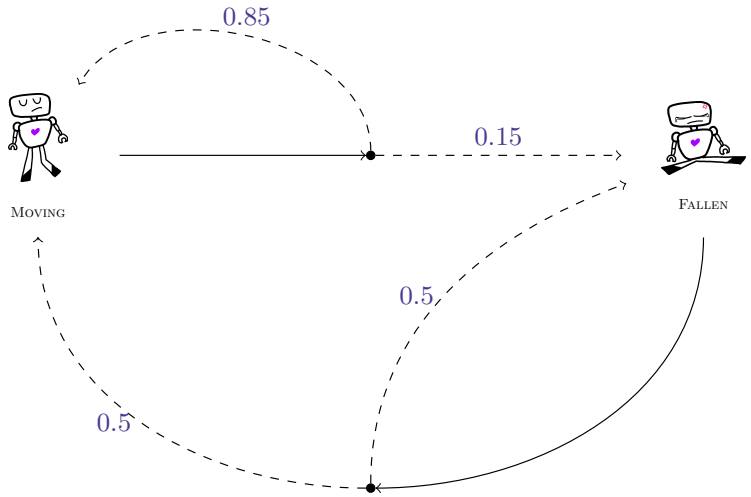

Figure 2.2: A Markov chain

Consider the finite paths $p_1 = \text{MOVING} \cdot \text{FALLEN}$ and $p_2 = \text{MOVING} \cdot \text{MOVING} \cdot \text{FALLEN}$. We have the following:

$$\mathbb{P}_{M,\text{MOVING}}(\mathsf{Paths}^{\omega}_{M}(p_1)) = 0.15$$
$$\mathbb{P}_{M,\text{MOVING}}(\mathsf{Paths}^{\omega}_{M}(p_2)) = 0.85 \times 0.15 = 0.1275$$

The probability of the robot falling either at one or two steps starting from MOVING can be calculated by $\mathbb{P}_{M,\text{MOVING}}(\mathsf{Paths}^{\omega}_{M}(p_1) \cup \mathsf{Paths}^{\omega}_{M}(p_2)) = 0.15 + 0.1275 = 0.2775$. This way probability of the robot eventually falling is

$$\sum_{i=1}^{\infty} Pr_{M,\text{MOVING}}(\mathsf{Paths}^{\omega}_{M}((\text{MOVING})^i \cdot \text{FALLEN}))$$

Which is equal to $\sum_{i=1}^{\infty}((0.85)^{i-1} \times 0.15) = 1$. □

We may omit the M from the subscript from the previous notations if the MC is clear from the context.

2.3.2 Probabilistic computation tree logic

Probabilistic computation tree logic or *PCTL* [HJ94] is a branching temporal logic which can be used to formulate conditions on a Markov chain. *PCTL state formulae* over a set of atomic propositions AP are defined according the following grammar:

$$\Phi := true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\varphi)$$

where $a \in AP$, Φ_1 and Φ_2 are state formulae, φ is a path formula and $J \subseteq [0,1]$ is an interval with rational bounds.

PCTL path formulae are defined according the following grammar:

$$\varphi := \bigcirc\Phi \mid \Phi_1 \mathcal{U} \Phi_2 \mid \Phi_1 \mathcal{U}^{\leq n} \Phi_2$$

where Φ_1 and Φ_2 are state formulae and $n \in \mathbb{N}$.

The satisfaction relation $\models$ between an infinite play $p = s_0 s_1 \ldots$ and a PCTL path formula is defined as follows:

- $p \models \bigcirc\Phi$ if $p[1] \models \Phi$.
- $p \models \Phi_1 \mathcal{U} \Phi_2$ if there exists $i \in \mathbb{N}$ such that $p[i] \models \Phi_2$ and for all $0 \leq j < i$, $p[j] \models \Phi_1$.
- $p \models \Phi_1 \mathcal{U}^{\leq n} \Phi_2$ if there exists $i \leq n$ such that $p[i] \models \Phi_2$ and for all $0 \leq j < i$, $p[j] \models \Phi_1$.

We define the probability of an PCTL path formula φ holding at a state $s \in S$ by

$$\mathbb{P}_M(s \models \varphi) = \mathbb{P}_{M,s}(\{p \in \mathsf{Paths}^{\omega}_M(s) \mid p \models \varphi\})$$

The satisfaction relation $\models$ between a state $s \in S$ and a PCTL state formula is defined as follows:

- $s \models true$.
- $s \models a$ if $a \in L(s)$.
- $s \models \Phi_1 \wedge \Phi_2$ if $s \models \Phi_1$ and $s \models \Phi_2$.
- $s \models \neg\Phi$ if $s \not\models \Phi$.
- $s \models \mathbb{P}_J(\varphi)$ if $\mathbb{P}_M(s \models \varphi) \in J$.

Using the Boolean connectives $\wedge$ and $\neg$, we can define other Boolean connectives such as $\vee$, $\rightarrow$, $\leftrightarrow$. The $\mathcal{U}$ operator (and its bounded version) also allows us to define some other useful operators, namely $\Diamond$ and $\Box$ as follows:

$$\Diamond\Phi = true\, \mathcal{U} \Phi \quad \text{and} \quad \Box\Phi = \neg\Diamond\neg\Phi$$

$$\Diamond^{\leq n}\Phi = true\,\mathcal{U}^{\leq n}\Phi \quad \text{and} \quad \Box^{\leq n}\Phi = \neg\Diamond^{\leq n}\neg\Phi$$

In other words,

- $p \models \Diamond\Phi$ if there exists $i \in \mathbb{N}$ such that $s_i \models \Phi$.
- $p \models \Diamond^{\leq n}\Phi$ if there exists $i \leq n$ such that $s_i \models \Phi$.
- $p \models \Box\Phi$ if for all $i \in \mathbb{N}$, $s_i \models \Phi$.
- $p \models \Box^{\leq n}\Phi$ if for all $i \leq n$, $s_i \models \Phi$.

Checking if a state satisfies a PCTL formula Φ is done by bottom-up traversal of the parse tree of Φ where the nodes of the parse tree represent the subformulae of Φ.

Theorem 2.3 ([BK08, Theorem 10.40])

For a state $s \in S$ in a Markov chain M, and PCTL state formula Φ, whether $s \models \Phi$ can be checked in time

$$\mathcal{O}(poly(|M|) \cdot n_{\max} \cdot |\Phi|)$$

where $n_{\max}$ is the maximal step bound that appears in a sub path formula $\psi_1\mathcal{U}^{\leq n}\psi_2$ of Φ ($n_{\max} = 1$ if Φ does not contain a $\mathcal{U}^{\leq n}$ operator).

2.3.3 Probabilistic bisimulation

Probabilistic bisimulation [SL94] relations are equivalence relations between states which requires two bisimilar states to be equally labelled and exhibit equivalent stepwise behaviour.

Definition 2.6 (Bisimulation for Markov chains)

Let $M = (S, P, AP, L)$ be a Markov chain. A (probabilistic) bisimulation is an equivalent relation $\sim$ on S such that for all states $s_1, s_2 \in S$ such that $s_1 \sim s_2$, we have:

- *$L(s_1) = L(s_2)$, and,*
- *For any state $t \in S$, $\sum_{t' \sim t} P(s_1, t') = \sum_{t' \sim t} P(s_1, t')$.*

If $s_1 \sim s_2$, we call s_1 and s_2 bisimilar equivalent or bisimilar. The relation $\sim$ partitions the set S into equivalent classes. The set $\{s' \mid s' \sim s\}$ is denoted by $[s]_\sim$. We denote the set of equivalent classes by $S_\sim$. The first condition in Definition 2.6 states that the bisimilar states are equally labelled. The last condition requires that for bisimilar states the

probability of moving by a single transition to some equivalence class is equal.

Bisimulation can be used to define smaller Markov chains, which satisfies similar properties as the original Markov chain.

Definition 2.7 (Bisimulation quotient of a Markov chain)

Let $M = (S, P, AP, L)$ be a Markov chain. The quotient Markov chain is defined by $M_\sim = (S_\sim, P', AP, L')$ where

- *For $s, t \in S$, $P'([s]_\sim, [t]_\sim) = \sum_{t' \sim t} P(s, t')$*
- *For $s \in S$, $L'([s]_\sim) = L(s)$*

Note that P' and L' are well-defined from the definition of bisimulation.

Bisimulation equivalent paths We can lift the notion of bisimulation equivalence to paths. The paths $p_1 = s_0 s_1 \dots$ and $p_2 = t_0 t_1 \dots$ are bisimulation equivalent, denoted by $p_1 \sim p_2$, if they are statewise bisimilar : $p_1 \sim p_2$ if and only if $s_i \sim t_i$ for all $i \geq 0$.

Bisimulation closed σ-algebra Let $M = (S, P, AP, L)$ be a Markov chain. For classes $S_0, S_1, \dots, S_n \in S_\sim$, we denote the set of all infinite paths $s_0 s_1 \dots$ such that $s_i \in S_i$ for $0 \leq i \leq N$ as $\mathsf{Paths}^\omega_{M_\sim}(S_0 S_1 \cdots S_n)$. The bisimulation closed σ-algebra is the smallest σ-algebra containing $\mathsf{Paths}^\omega_{M_\sim}(S_0 S_1 \cdots S_k)$ for any sequence of equivalent classes $S_0, S_1, \dots, S_k \in S_\sim$. We denote this σ-algebra by $\mathcal{F}^M_\sim$.

All events in $\mathcal{F}^M_\sim$ are measurable with respect to the standard sigma algebra $\mathcal{F}^M$ associated with the Markov chain M. In other words, $\mathcal{F}^M_\sim \subseteq \mathcal{F}^M$ as a basic element in $\mathcal{F}^M_\sim$ can be written as countable uninon of basic elements in $\mathcal{F}^M$:

$$\mathsf{Paths}^\omega_{M_\sim}(S_0 S_1 \cdots S_k) = \bigcup_{\substack{s_0 s_1 \cdots s_k \in \mathsf{Paths}_M \\ s_i \in S_i, 0 \leq i \leq k}} \mathsf{Paths}^\omega_M(s_0 s_1 \cdots s_k)$$

The following theorem states that bisimulation equivalence preserves PCTL-definable properties:

Theorem 2.4 ([BK08, Theorem 10.67])

Let $M = (S, P, AP, L)$ be a Markov chain and $s_1, s_2 \in S$. Then the following statements are equivalent:

1. $s_1 \sim s_2$.

2. *For a PCTL formula Φ, the following statements are equivalent:*
   - $s_1 \models \Phi$ *in* M
   - $s_2 \models \Phi$ *in* M
   - $[s_1]_\sim \models \Phi$ *in* $M_\sim$.

2.3.4 Markov decision process

A *Markov decision process* can be viewed as an extension of Markov chains and transition systems with finite state space which allows both probabilistic and nondeterministic transitions. It is also augmented with rewards corresponding to the states and the state-action pairs. A Markov decision process can be defined as follows:

Definition 2.8 (Markov decision process)

A Markov decision process or an MDP is a tuple $M = (S, A, P, R, R_T, AP, L)$, where

- *S is a finite set of states,*
- *A is a finite set of actions,*
- *P is a partial mapping from $S \times A$ to $\mathcal{D}(S)$ such that $P(s,a)(s')$ denotes the probability that action a in state s leads to state s',*
- *R is a partial mapping from $S \times A$ to $\mathbb{R}$ which defines the reward obtained for taking a given action from a state,*
- *R_T is a mapping from S to $\mathbb{R}$ that assigns a terminal reward to each state in S,*
- *AP is a finite set of atomic proposition,*
- *L is the labelling function from S to 2^{AP} such that $L(s)$ denotes the propositions in AP that is satisfied by the state s.*

Note that not all actions may be *legal* from a state as P is a partial function. Therefore, if an action a is legal from a state s, we will have $\sum_{s' \in S} P(s,a)(s') = 1$. Otherwise, we will have $P(s,a)(s')$ is undefined (denoted by $\perp$) for all $s' \in S$. Similarly, if an action a is legal from a state s, $R(s,a)$ will have a value in $\mathbb{R}$. Otherwise, $R(s,a) = \perp$. By abuse of notation, we can also use A to denote a mapping from S to 2^A, where $A(s) = \{a \in A \mid \forall s'\ P(s,a)(s') \neq \perp\}$, the set of legal actions from the state s.

Often the function R_T is not explicitly defined. In that case, it is assumed that $R_T(s) = 0$. For ease of notation, we denote the value $P(s,a)(s')$ as $P(s,a,s')$. The set

of propositions AP often is not explicitly defined. In that case, it is assumed that $AP = S$ with labelling function $L(s) = \{s\}$.

Paths The definitions and notations used for Markov chain can be extended in the case of MDPs. For a Markov decision process M, a *path* of length $i > 0$ is a sequence of i consecutive states and actions followed by a last state. We say that $p = s_0 a_0 s_1 \dots s_i$ is an i-length path in the MDP M if for all $t \in [0, i-1]$, $a_t \in A$ and $s_{t+1} \in \mathsf{Supp}(P(s_t, a_t))$. We also consider states to be paths of length 0. We respectively denote the last and first state of a path $p = s_0 a_0 s_1 \dots s_n$ by $\mathsf{last}(p) = s_n$ and $\mathsf{first}(p) = s_0$.

An *infinite path* is an infinite sequence $p = s_0 a_0 s_1 \dots$ of states and actions such that for all $t \in \mathbb{N}$, $a_t \in A$ and $s_{t+1} \in \mathsf{Supp}(P(s_t, a_t))$. We denote the finite prefix of length t of a finite or infinite path $p = s_0 a_0 s_1 \dots$ by $p_{|t} = s_0 a_0 \dots s_t$. Let $p = s_0 a_0 s_1 \dots s_i$ and $p' = s'_0 a'_0 s'_1 \dots s'_j$ be two paths such that $s_i = s'_0$. Then, $p \cdot p'$ denotes $s_0 a_0 s_1 \dots s_i a'_0 s'_1 \dots s'_j$. Let $p = s_0 a_0 s_1 \dots s_i$ be a path , a be an action and s be a state of M such that $s \in \mathsf{Supp}(P(s_i, a))$. Then, $p \cdot as$ denotes the path $s_0 a_0 s_1 \dots s_i a s$.

For an MDP M, the set of all finite paths of length i is denoted by Paths^i_M. We denote the set of all finite paths in M by Paths_M and the set of finite paths of length at most H by $\mathsf{Paths}^{\leq H}_M$. The set of all infinite paths is denoted by Paths^ω_M.

If $p \in \mathsf{Paths}^i_M$ and $i \leq j$, then let $\mathsf{Paths}^j_M(p)$ denote the set of paths p' in Paths^j_M such that there exists $p'' \in \mathsf{Paths}^{j-i}_M$ with $p' = p \cdot p''$. In particular, let $\mathsf{Paths}^i_M(s)$ denote the set of paths p in Paths^i_M such that $\mathsf{first}(p) = s$. Similarly, if $p \in \mathsf{Paths}_M$, then let $\mathsf{Paths}_M(p)$ denote the set of paths p' in Paths_M such that there exists $p'' \in \mathsf{Paths}_M$ with $p' = p \cdot p''$. For $p \in \mathsf{Paths}_M$, let $\mathsf{Paths}^\omega_M(p)$ denote the set of paths p' in Paths^ω_M such that there exists $p'' \in \mathsf{Paths}^\omega_M$ with $p' = p \cdot p''$.

Strategies A (probabilistic) *strategy* is a function $\sigma : \mathsf{Paths}_M \to \mathcal{D}(A)$ that maps a path p to a probability distribution in $\mathcal{D}(A)$. A strategy σ is *deterministic* if the support of the probability distributions $\sigma(p)$ has size 1. A strategy σ is *memoryless* if $\sigma(p)$ depends only on $\mathsf{last}(p)$, i.e. if σ satisfies that for all $p, p' \in \mathsf{Paths}_M$, $\mathsf{last}(p) = \mathsf{last}(p') \Rightarrow \sigma(p) = \sigma(p')$.

For a probabilistic strategy σ and $i \in \mathbb{N}$, let $\mathsf{Paths}^i_M(\sigma)$ denote the paths $p = s_0 a_0 \dots s_i$ in Paths^i_M such that for all $t \in [0, i-1]$, $a_t \in \mathsf{Supp}(\sigma(p_{|t}))$. Analogously, for a probabilistic strategy σ, let $\mathsf{Paths}^\omega_M(\sigma)$ denote the paths $p = s_0 a_0 \dots$ in Paths^ω_M such that for all $t \in \mathbb{N}$, $a_t \in \mathsf{Supp}(\sigma(p_{|t}))$. Also, for a probabilistic strategy σ, let $\mathsf{Paths}_M(\sigma)$ denote the set of

finite paths $p = s_0 a_0 \dots$ in Paths_M such that for all $t \in \mathbb{N}$, $a_t \in \mathsf{Supp}(\sigma(p_{|t}))$. For a finite path p of length $i \in \mathbb{N}$ and some $j \geq i$, let $\mathsf{Paths}^j_M(p, \sigma)$ denote $\mathsf{Paths}^j_M(\sigma) \cap \mathsf{Paths}^j_M(p)$. Similarly, for a finite path p, let $\mathsf{Paths}_M(p, \sigma)$ denote $\mathsf{Paths}_M(\sigma) \cap \mathsf{Paths}_M(p)$ and let $\mathsf{Paths}^\omega_M(p, \sigma)$ denote $\mathsf{Paths}^\omega_M(\sigma) \cap \mathsf{Paths}^\omega_M(p)$.

For ease of notation, for a stategy σ, we denote the value $\sigma(p, a)(s)$ as $\sigma(p, a, s)$.

A *nondeterministic strategy* is a function $\sigma : \mathsf{Paths}_M \to 2^A$ that maps a finite path p to a subset of A. For a strategy σ' and a nondeterministic strategy σ, we say $\sigma' \subseteq \sigma$ if for all p, $\mathsf{Supp}(\sigma'(p)) \subseteq \sigma(p)$. For a strategy σ, we can construct a non-deterministic strategy non-det(σ) by non-det(σ)$(p) = Supp(\sigma(p))$.

Markov chain defined by strategies An MDP M equipped with a strategy σ defines an MC M_σ. Intuitively, this is obtained by unfolding M, using the strategy σ and the probabilities in M to define the transition probabilities and ignoring the rewards.

Formally $M_\sigma = (\mathsf{Paths}_M(\sigma), P_\sigma, AP, L_\sigma)$ where for all finite paths $p \in \mathsf{Paths}_M(\sigma)$, $P_\sigma(p, p \cdot as) = \sigma(p, a) \cdot P(\mathsf{last}(p), a, s)$ and $L_\sigma(p) = L(\mathsf{last}(p))$. Thus a finite path p in $\mathsf{Paths}_M(\sigma)$ uniquely *matches* a finite path p' in M_σ when $\mathsf{last}(p') = p$. This way when a strategy σ and a state s is fixed, the probability measure defined in M_σ is also extended for paths in $\mathsf{Paths}_M(\sigma)$. In other words, the cylinder set of a path $p \in \mathsf{Paths}_M$ has probability $\mathbb{P}_{M_\sigma, s}(\mathsf{Paths}^\omega_M(p'))$, if p matches with p'.

For ease of notation, we write $\mathbb{P}_{M_\sigma, s}$ as $\mathbb{P}^\sigma_s$. We write the expected value of a random variable X with respect to the probability distribution $\mathbb{P}^\sigma_s$ as $\mathbb{E}^\sigma_s(X)$.

Example 2.4 Consider a simple Markov decision process M used in reinforcement learning to train a robot the most effective way to move by associating rewards to different actions. There are two possible ways for the robot to move: either it can walk or it can run. Running is more rewarding (it gives 2 reward instead of 1), but also if the robot tries to run, with probability 0.3, the robot falls. From the fallen state, the robot can stand again with probability 0.5. The robot gets a reward of -1 when it tries to stand again from the fallen state. This MDP is described in Figure 2.3. The terminal rewards are 0 for both states.

Consider the following memoryless strategy σ:

$$\sigma(p) = \begin{cases} [walk \mapsto 0.5, run \mapsto 0.5] & \text{if } last(p) = \textsc{Moving} \\ [stand \mapsto 1] & \text{if } last(p) = \textsc{Fallen} \end{cases}$$

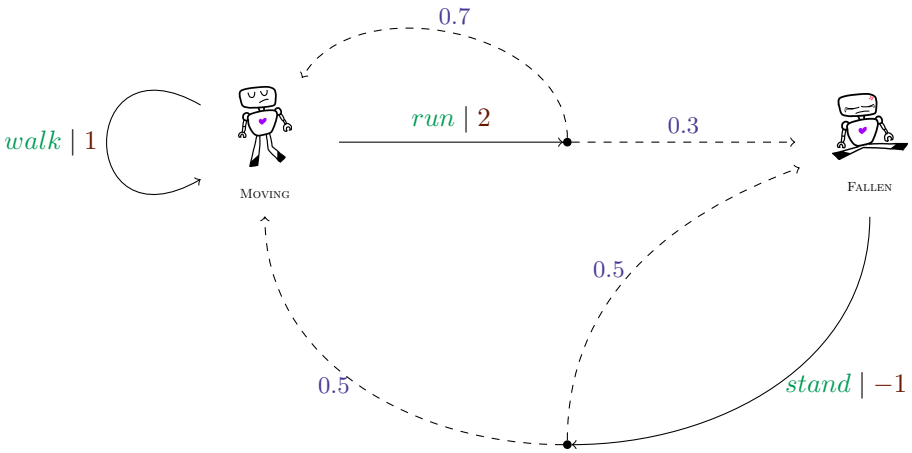

Figure 2.3: A Markov Decision Process

Then M_σ is the MC described in Example 2.3. □

Total reward The *total reward* of horizon n for a path $p = s_0 a_0 \dots$ in M is defined as $\mathsf{Reward}^n_M(p) = \sum_{i=0}^{n-1} R(s_i, a_i) + R_T(s_n)$.

The *expected total reward* of a probabilistic strategy σ in an MDP M, starting from state s and for a finite horizon $n \in \mathbb{N}$, is defined as $\mathsf{Val}^n_M(s, \sigma) = \mathbb{E}^\sigma_s [\mathsf{Reward}^n_M]$. The optimal expected total reward of horizon n starting from a state s in an MDP M is defined over all strategies σ in M as $\mathsf{Val}^n_M(s) = \sup_\sigma \mathsf{Val}^n_M(s, \sigma)$. One can restrict the supremum to deterministic strategies [Put94, Theorem 4.4.1.b].

Average reward The (long-term) *average reward* of an infinite path $p = s_0 a_0 s_1 \dots$ in M is defined as $\mathsf{AvgReward}_M(p) = \liminf_{n\to\infty} \frac{1}{n} \mathsf{Reward}^n_M(p)$.

The *expected average reward* of a probabilistic strategy σ in an MDP M, starting from state s, is defined as $\mathsf{Val}_M(s, \sigma) = \mathbb{E}^\sigma_s [\mathsf{AvgReward}_M]$. The optimal expected average reward starting from a state s in an MDP M is defined over all strategies σ in M as $\mathsf{Val}_M(s) = \sup_\sigma \mathsf{Val}_M(s, \sigma)$. One can restrict the supremum to deterministic memoryless strategies [Put94, Proposition 6.2.1]. A strategy σ is called *ϵ-optimal* for the expected average reward if $\mathsf{Val}_M(s, \sigma) \geq \mathsf{Val}_M(s) - \epsilon$ for all s.

Value iteration *Value iteration* is a *dynamic programming* [Put94, Section 4.5] technique that uses the fact that for all s in S and $i \in \mathbb{N}$:

$$\mathsf{Val}^0_M(s) = R_T(s)\,, \text{ and}$$
$$\mathsf{Val}^{i+1}_M(s) = \max_{a \in A}\left[R(s,a) + \sum_{s'} P(s,a,s')\mathsf{Val}^i_M(s')\right] \tag{2.1}$$

We define $\mathsf{opt}^{i+1}_M(s) = \arg\max_{a\in A}\left[R(s,a) + \sum_{s'} P(s,a,s')\mathsf{Val}^i_M(s')\right]$, i.e. the action that maximizes the value in the above equation[1].

For a state $s \in S$ and $i \in \mathbb{N}$, let $\sigma^{i,*}_{M,s}$ denote a deterministic strategy that maximizes $\mathsf{Val}^i_M(s,\sigma)$, and refer to it as an *optimal strategy* for the expected total reward of horizon i at state s. Then for a state s and a horizon i, $\sigma^{i,*}_{M,s}(p) = \mathsf{opt}^{i-|p|}_M(\mathsf{last}(p))$. We also define, $\mathsf{Val}^{i+1}_M(s,a) = \left[R(s,a) + \sum_{s'} P(s,a,s')\mathsf{Val}^i_M(s')\right]$ for a state s and an action a.

Moreover, $\frac{1}{n}\mathsf{Val}^n_M(s)$ approximates $\mathsf{Val}_M(s)$:

Theorem 2.5 ([Put94, Theorem 9.4.1.b])

For an MDP M, a state $s \in S$,

$$\lim_{n\to\infty} \frac{1}{n}\mathsf{Val}^n_M(s) = \mathsf{Val}_M(s)$$

This gives Algorithm 1 to approximate the long term average reward. This involves iterating Equation 2.1 till a suitable stopping condition is met to get an ϵ-close value.

Algorithm 1 Value iteration

Input: MDP M, a state $s_0 \in S$ and a precision $\epsilon > 0$
Output: a value v such that $|v - \mathsf{Val}_M(s)| \leq \epsilon$

1: $n \leftarrow 0$
2: $v_0(s) = R_T(s)$ for all $s \in S$
3: **while** *stopping criterion* is not met **do**
4: **for** $s \in S$ **do**
5: $v_{n+1}(s) = \max_{a\in A}\left[R(s,a) + \sum_{s'} P(s,a,s')v_n(s')\right]$
6: **end for**
7: $n \leftarrow n+1$
8: **end while**
9: **return** $\frac{1}{n}v_n(s_0)$

[1] There could be more than one action in the set that maximizes the value. In that case, we can take any one of the action. In other words, there could be multiple deterministic strategies that optimizes the expected reward.

Unfortunately, this has two problems:

- For a state s, even though $\frac{1}{n}\mathsf{Val}^n_M(s)$ converges, $\mathsf{opt}^n_M(s)$ may not converge; so it is not possible to find an optimal or ϵ-optimal strategy.
- For general MDPs, it is not always easy to define a good stopping condition.

The first problem does not arise in a specific class of MDPs called *strongly aperiodic MDP*. A Markov decision process is *strongly aperiodic* if $P(s, a, s) > 0$ for all $s \in S$ and $a \in A$. One can make an MDP strongly aperiodic without changing the optimal expected average reward and its optimal strategies [Put94, Section 8.5.4] with the following transformation:

Definition 2.9 (Aperiodic transformation)

For an MDP $M = (S, A, P, R, R_T, AP, L)$, we define a new MDP $M_\alpha = (S, A, P_\alpha, R, R_T, AP, L)$ for $0 < \alpha < 1$, where for all $s \in S$ and $a \in A$,

- $P_\alpha(s, a, s) = \alpha + (1 - \alpha)P(s, a, s)$ *and*
- $P_\alpha(s, a, s') = (1 - \alpha)P(s, a, s')$ *for all $s' \neq s$.*

Notice that M_α is strongly aperiodic.

Every finite path in M is also in M_α. Thus, for a strategy $\widehat{\sigma}$ in M_α, there is a σ in M whose domain is restricted to the paths in M. The following theorem states that aperiodic transformation does not change the optimal expected average reward and a strategy that optimizes the expected average reward in the transformed MDP is also a strategy that optimizes the expected average reward in the original MDP:

Theorem 2.6 ([Put94, Section 8.5.4])

Let M be an MDP. For $\alpha \in (0, 1)$, M_α is a new MDP generated by applying the aperiodic transformation mentioned above. Then the set of memoryless strategies that optimizes the expected average reward in M_α is the same as the set of memoryless strategies the optimizes the expected average reward in M. Also from any s, $\mathsf{Val}_M(s) = \mathsf{Val}_{M_\alpha}(s)$.

For $i \in \mathbb{N}$, let σ^i_M refer to a deterministic memoryless strategy that maps every state s in M to the first action of a corresponding optimal strategy for the expected total reward of horizon i, so that $\sigma^i_M(s) = \sigma^{i,*}_{M,s}(s)$. For a large enough n, in a strongly aperiodic Markov decision process, the strategy σ^n_M is ϵ-optimal for the expected average reward:

Theorem 2.7 ([Put94, Theorem 9.4.5])

For a strongly aperiodic Markov decision process M, it holds that

$$\mathsf{Val}_M(s) = \lim_{n\to\infty} [\mathsf{Val}_M^{n+1}(s) - \mathsf{Val}_M^n(s)]$$

Moreover, for any $\epsilon > 0$ there exists $N \in \mathbb{N}$ such that for all $n \geq N$, for all s,

$$Val_M(s, \sigma_M^n) \geq Val_M(s) - \epsilon\,.$$

End components For an MDP, an end component is a sub-MDP that is closed under probabilistic choices and that is strongly connected.

Definition 2.10 (End component of an MDP)

For an MDP $M = (S, A, P, R, R_T)$, an end-component *(EC) is a sub-MDP $M' = (T, A', P', R', R'_T)$ such that*

- *for all $s \in T$ and $a \in A$, $\mathsf{Supp}(P(s, a)) \subseteq T$, and,*
- *the underlying graph of M' is strongly connected, i.e., for any vertex q_1 and q_2 in M', there is a path from q_1 to q_2 and also from q_2 to q_1.*

A maximal end-component *(MEC) is an end component not included in another end-component.*

We may omit the M from the subscript from the previous notations if the MDP is clear from the context.

2.3.5 Underlying game graph of an MDP

The underlying structure of an MDP can be seen as a game graph where the nondeterminism is controlled by player 0 and the probabilistic transitions are controlled by player 1. Formally we define the game graph as follows:

Definition 2.11

For an MDP $M = (S, A, P, R, R_T, AP, L)$, we define the underlying game graph of M as $G_M = (Q_0 \uplus Q_1, E, Q_0, Q_1)$ where

- *$Q_0 = S$ and $Q_1 = S \times A$.*
- *For $q_0 = s_0 \in Q_0$ and $q_1 = (s_1, a_1) \in Q_1$, $(q_0, q_1) \in E$ if $s_0 = s_1$ and $a_1 \in A(s_0)$.*

- *For* $q_1 = (s_1, a_1) \in Q_1$ *and* $q_0 = s_0 \in Q_0$, $(q_1, q_0) \in E$ *if* $s_0 \in \mathsf{Supp}(P(s_1, a_1))$.

An infinite path $p = s_0 a_0 s_1 \dots$ in M uniquely *matches* with an infinite path $p' = q_0 q_1 \dots$ in G_M where for all $i \in \mathbb{N}$, $q_{2i} = s_i$ and $q_{2i+1} = (s_i, a_i)$. We can extract a deterministic strategy σ_M in M from a strategy σ_{G_M} for player 0 in G. Formally, $\sigma_M(p) = a$ if $\sigma_{G_M}(p') = (\mathsf{last}(p), a)$ where p matches with p'.

Example 2.5 The underlying game graph of the MDP in Example 2.4 is described in Figure 2.4. Player 0 vertices are denoted by circles and player 1 vertices are denoted by boxes.

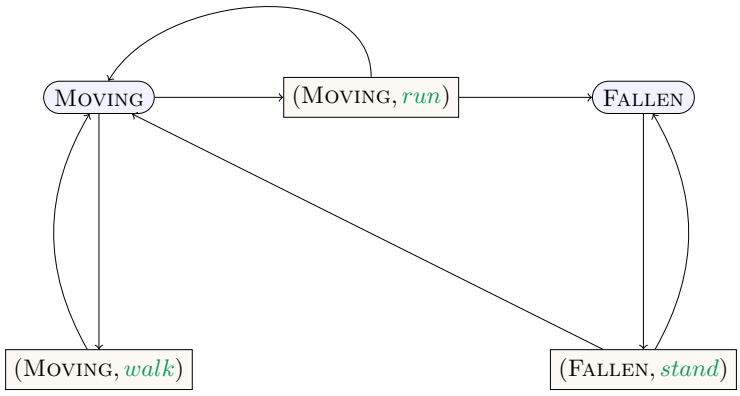

Figure 2.4: A 2-player game

Consider the safety game where player 0 needs to avoid the vertex FALLEN. A strategy $\sigma_{\mathcal{G}_M}$ for player 0 such that $\sigma_{\mathcal{G}_M}(p) = (\text{MOVING}, walk)$ if $\mathsf{last}(p) = \text{MOVING}$ is winning for player 0 from the state MOVING.

This gives a deterministic strategy σ_M in M to avoid the state FALLEN from the state MOVING where $\sigma_M(p) = [walk \mapsto 1]$ where $\mathsf{last}(p) = \text{MOVING}$. □

For ease of notation, for a deterministic strategy σ in M, we will also denote the corresponding strategy for player 0 in the underlying 2-player game as σ. For $a \in AP$, let $S_a = \{s \in S \mid L(s) = a\}$. For $a \in AP$, for a formula $\varphi \in \{\Diamond a, \Box a, \Diamond^{\leq H} a, \Box^{\leq H} a\}$, we will write $s \models \varphi$ if for all paths $p \in \mathsf{Paths}^{\omega}_M(s)$, $p \models \varphi$. Then for a state $s_0 \in S$, we have the following:

- The reachability game in $\mathcal{G}_M$ with target set S_a is winning with strategy $\sigma_{\mathcal{G}_M}$ if $s_0 \models \Diamond a$ in M_σ.
- The finite reachability game in $\mathcal{G}_M$ with target set S_a and horizon H is winning with strategy $\sigma_{\mathcal{G}_M}$ if $s_0 \models \Diamond^{\leq H} a$ in M_σ.
- The safety game in $\mathcal{G}_M$ with target set S_a is winning with strategy $\sigma_{\mathcal{G}_M}$ if $s_0 \models \Box a$ in M_σ.
- The finite safety game in $\mathcal{G}_M$ with target set S_a is winning with strategy $\sigma_{\mathcal{G}_M}$ if $s_0 \models \Box^{\leq H} a$ in M_σ.

2.3.6 Approximating MDPs

An MDP $M = (S, A, P, R, R_T, AP, L)$ is said to *structurally identical* to another MDP $M' = (S, A, P', R, R_T, AP, L)$ if for all $s \in S$ and $a \in A$, we have that $\mathsf{Supp}(P(s,a)) = \mathsf{Supp}(P'(s,a))$. For two structurally identical MDPs M and M with probability distribution P and P' respectively, we say that M is ϵ-approximate to M', denoted $M \approx^{\epsilon} M'$, if for all $s \in S$ and $a \in A$: $P(s,a) \sim^{\epsilon} P'(s,a)$.

The following Lemma captures the idea that optimal strategy in an MDP M' that approximate MDP M, would be a ϵ-optimal strategy in the original MDP M.

Lemma 2.2 (Adapted from [Cha12, Theorem 5])

Consider $\beta \in (0,1)$, and MDPs M and M' with set of state S such that $M \approx^{\eta_\beta} M'$ with $\eta_\beta \leq \frac{\beta \cdot p_{\min}}{8|S|}$, where $p_{\min}$ is the minimum probability appearing in M. For a state $s \in S$, let σ be a memoryless deterministic strategy such that $\mathsf{Val}_{M'}(s,\sigma) = \mathsf{Val}_{M'}(s)$. Then, it holds that $|\mathsf{Val}_M(s,\sigma) - \mathsf{Val}_M(s)| \leq \beta$.

2.4 Distance-optimal strategy for reachability

Given an MDP $M = (S, A, P, R, R_T, AP, L)$, let $T \subseteq S$. An objective we could be interested in is to find a strategy that maximizes the probability to reach the target. Formally, let $\mathbb{P}_{M_\sigma,s}(\Diamond T)$ denote the value $\mathbb{P}_{M_\sigma}(s \models \Diamond T)$, the probability to reach a state in T from the state s in the Markov chain M_σ. We want to find a strategy in

$$\Sigma_{M,s}(\Diamond T) = \arg\max_{\sigma} \mathbb{P}_{M_\sigma,s}(\Diamond T)\,.$$

Example 2.6 We can represent the game Frozen Lake as a Markov decision process. In this game, a robot moves in a slippery grid. It has to reach the target while avoiding holes in the grid. Each state in the MDP represents the current position of the robot in the grid. The states representing the target and the holes can be assumed to be sink state, i.e., the robot cannot move to any other positions from this state. Part of the grid contains walls and the robot cannot move into it. The frozen surface of the lake being slippery, when the robot tries to move by picking a cardinal direction, the next state is determined randomly over the four neighbouring positions of the robot, according to the following distribution weights: the intended direction gets a weight of 10, and other directions that are not a wall and not the reverse direction of the intended one get a weight of 1[2]. There are no rewards, and the terminal reward is 1 when the robot reaches the target and 0 otherwise. □

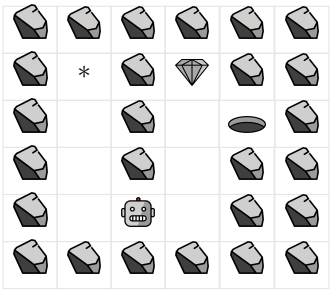

Figure 2.5: A 6×6 layout for Frozen-Lake

Figure 2.5 shows a grid of size 6×6. Note that there is no strategy to reach the target with probability 1. Consider a strategy σ_1 where the robot moves to the cell right of the starting cell and then move up to reach the target. We can have another strategy σ_2 where the robot instead moves left and then moves upwards till the cell marked with $*$ and then comes back to the initial position and then follows σ_1. Although both strategies are optimal in terms of probability to reach the target, σ_2 takes more time to reach the target in comparison to σ_1. Thus, a more practical approach would be to find a strategy that also minimizes the time needed to reach the target among the strategies that maximizes the probability to reach the target.

Given a path ρ in an MC $M = (S, P, AP, L)$ and $T \subseteq S$, we use $\mathsf{len}(\rho, T)$ to denote the length of the shortest prefix of ρ that reaches one of the states of T, that is, $\mathsf{len}(\rho, T) = i$ if $\rho[i] \in T$ and for all $j < i$, $\rho[j] \notin T$. Our objective is to find a strategy which minimizes the conditional expectation $\mathbb{E}_{M_\sigma,s}(\mathsf{len}(\rho, T) \mid \rho \models \Diamond T)$ among the strategies in $\Sigma_{M,s}(\Diamond T)$, the strategies which maximize $\mathbb{P}_{M_\sigma,s}(\Diamond T)$.

[2]The distribution is then normalized so that weights sum up to 1.

In this section, we will present how to get a such strategy. The algorithm works as follows: we remove actions that does not contribute to maximizing the probability to reach the target and then change the probabilities in the MDP as described in Definition 2.12. This way, using any strategy that maximizes the probability to reach the set T in original MDP M, the set T can be reached with probability 1 in the new MDP M'. Then, in the new MDP M', we calculate the strategy that minimizes the expected distance to the state T.

Before formally stating and proving the algorithm in Theorem 2.8, let us define some notation that we will use in this section. We denote the actions that, when taken from a state s, maximize the probability of reaching the goal states from s as the set

$$\mathsf{Opt}_M = \{(s,a) \in S \times A \mid \mathsf{Val}_M(s,T) = \sum_{s'} P(s,a,s') \cdot \mathsf{Val}_M(s',T)\}\,.$$

We use Σ_M^{Opt} to denote the strategies that takes actions according to Opt_M, that is,

$$\Sigma_M^{\mathsf{Opt}} = \{\sigma \mid \forall \rho, \forall a \in \mathsf{Supp}(\sigma(\rho)); (\mathsf{last}(\rho), a) \in \mathsf{Opt}_M\}\,.$$

Lemma 2.3

Given an MDP $M = (S, A, P, R, R_T, AP, L)$ and a set of states $T \subseteq S$, for every state $s \in S$ and for every action a, $\mathsf{Val}_M(s,T) \geq \sum_{s'} P(s,a,s') \cdot \mathsf{Val}_M(s',T)$.

Proof Suppose, there is a state $s \in S$ and an action $a \in A$ such that $\mathsf{Val}_M(s,T) < \sum_{s'} P(s,a,s') \cdot \mathsf{Val}_M(s',T)$. Now, consider the strategy σ' that takes action a from s and then from paths $s \cdot as'$ follows a strategy $\sigma_{s'} \in \Sigma_{M,s'}(\Diamond T)$ that maximizes the probability to reach states in T from s'. Formally,

$$\sigma'(\rho) = \begin{cases} a & \text{if } \rho = s \\ \sigma_{s'}(\rho') & \text{if } \rho = s \cdot as' \cdot \rho' \end{cases}$$

Then, $\mathbb{P}_{M_{\sigma'},s}(\Diamond T) = \sum_{s'} P(s,a,s') \cdot \mathsf{Val}_M(s,T) > \mathsf{Val}_M(s,T)$ which is a contradiction. □

The following lemma states that the strategies that maximize the probability of reaching the goal states always take 'locally optimal' actions.

Lemma 2.4

Given an MDP $M = (S, A, P)$ and a set of states $T \subseteq S$, for every state $s \in S$, $\Sigma_{M,s}(\Diamond T) \subseteq \Sigma_M^{\mathsf{Opt}}$.

Proof Suppose that there is a strategy $\sigma^* \in \Sigma_{M,s}(\Diamond T)$ where there exists a path ρ and an action $a \in \mathsf{Supp}(\sigma^*(\rho))$ such that $(\mathsf{last}(\rho), a) \notin \mathsf{Opt}_M$. Then, from Lemma 2.3 and the fact

that $(\mathsf{last}(\rho), a) \notin \mathsf{Opt}_M$,

$$\mathsf{Val}_M(\mathsf{last}(\rho), T) > \sum_{s'} P(\mathsf{last}(\rho), a, s') \cdot \mathsf{Val}_M(s', T)$$

and for every other action $a' \neq a$,

$$\mathsf{Val}_M(\mathsf{last}(\rho), T) \geq \sum_{s'} P(\mathsf{last}(\rho), a', s') \cdot \mathsf{Val}_M(s', T)\,.$$

Consider the strategy σ'' which differs from σ^* only on paths with ρ as prefix: on every path having ρ as a prefix, σ'' takes the next action according to a strategy $\sigma_{\mathsf{last}(\rho)} \in \Sigma_{M,\mathsf{last}(\rho)}(\Diamond T)$ that maximizes the probability to reach states in T from $\mathsf{last}(\rho)$, whereas, it takes action according to σ^* on every other path. Formally,

$$\sigma''(\rho') = \begin{cases} \sigma_{\mathsf{last}(\rho)}(\rho'') & \text{if } \rho' = \rho \cdot \rho'' \\ \sigma^*(\rho') & \text{otherwise.} \end{cases}$$

Note that, for every strategy σ,

$$\mathbb{P}_{M_\sigma,\rho}(\Diamond T) = \sum_{a'} \left(\sigma(\rho, a') \cdot \sum_{s'} \left(P(\mathsf{last}(\rho), a', s') \cdot \mathbb{P}_{M_\sigma,\rho\cdot a's'}(\Diamond T) \right) \right).$$

Also, $\mathbb{P}_{M_{\sigma^*},\rho\cdot a's'}(\Diamond T) \leq \mathsf{Val}_M(s', T)$ for all $a' \in A$. Therefore,

$$\begin{aligned} \mathbb{P}_{M_{\sigma^*},\rho}(\Diamond T) &= \sum_{a'} \left(\sigma^*(\rho, a') \cdot \sum_{s'} \left(P(\mathsf{last}(\rho), a', s') \cdot \mathbb{P}_{M_{\sigma^*},\rho\cdot a's'}(\Diamond T) \right) \right) \\ &\leq \sum_{a'} \left(\sigma^*(\rho, a') \cdot \sum_{s'} \left(P(\mathsf{last}(\rho), a', s') \cdot \mathsf{Val}_M(s', T) \right) \right) \\ &< \sum_{a'} \sigma^*(\rho, a') \cdot \mathsf{Val}_M(\mathsf{last}(\rho), T) \\ &= \mathsf{Val}_M(\mathsf{last}(\rho), T) \end{aligned}$$

So, $\mathbb{P}_{M_{\sigma''},\rho}(\Diamond T) = \mathbb{P}_{M_{\sigma_{\mathsf{last}(\rho)}},\mathsf{last}(\rho)}(\Diamond T) = \mathsf{Val}_M(\mathsf{last}(\rho), T) > \mathbb{P}_{M_{\sigma^*},\rho}(\Diamond T)$. Now note that, for any strategy σ,

$$\begin{aligned} \mathbb{P}_{M_\sigma,s}(\Diamond T) &= \mathbb{P}_{M_\sigma,s}(\rho' \models \Diamond T \wedge \rho \sqsubseteq \rho') + \mathbb{P}_{M_\sigma,s}(\rho' \models \Diamond T \wedge \rho \not\sqsubseteq \rho') \\ &= \mathbb{P}_{M_\sigma,s}(\mathsf{Paths}^\omega_{M_\sigma}(\rho)) \cdot \mathbb{P}_{M_\sigma,\rho}(\Diamond T) + \mathbb{P}_{M_\sigma,s}(\rho' \models \Diamond T \wedge \rho \not\sqsubseteq \rho') \end{aligned}$$

As $\sigma^*(\rho') = \sigma''(\rho')$ for any ρ' such that $\rho \sqsubseteq \rho'$,

$$\mathbb{P}_{M_{\sigma^*},s}(\mathsf{Paths}^\omega_{M_{\sigma^*}}(\rho)) = \mathbb{P}_{M_{\sigma''},s}(\mathsf{Paths}^\omega_{M_{\sigma''}}(\rho))\,.$$

and

$$\mathbb{P}_{M_{\sigma^*},s}(\rho' \models \Diamond T \wedge \rho \not\sqsubseteq \rho') = \mathbb{P}_{M_{\sigma''},s}(\rho' \models \Diamond T \wedge \rho \not\sqsubseteq \rho')\,.$$

Then $\mathbb{P}_{M_{\sigma^*},s}(\Diamond T) < \mathbb{P}_{M_{\sigma''},s}(\Diamond T)$, which cannot be true as σ^* is an optimal strategy. □

We define a transformation in the following definition where we create an MDP with

by removing states from which the target set T is not reachable. We also remove an action a from state s if (s, a) is not in Opt_M. Furthermore, we also redefine the probability distribution in a specific way.

Definition 2.12

We define a tranformation as follows: For an MDP $M = (S, A, P, R, R_T, AP, L)$ *and* $T \subseteq S$*, we define* $M' = (S', A, P', R, R_T, AP, L)$ *where*

$$S' = \{s \in S \mid \mathsf{Val}_M(s, T) > 0\}$$

and P' *is constructed from* P *in the following way:*

$$P'(s, a, s') = \begin{cases} P(s, a, s') \cdot \frac{\mathsf{Val}(s', T)}{\mathsf{Val}(s, T)} & \text{if } (s, a) \in \mathsf{Opt}_M \\ \bot & \text{otherwise.} \end{cases}$$

Note that $M' = (S', A, P')$ is well-defined as P' is indeed a probability distribution as $\sum_{s'} P'(s, a, s') = \sum_{s'} P(s, a, s') \cdot \frac{\mathsf{Val}(s', T)}{\mathsf{Val}(s, T)} = \frac{\mathsf{Val}(s, T)}{\mathsf{Val}(s, T)} = 1$.

From the construction of M', we have that the set of strategies in M' is Σ_M^{Opt}. For ease of notation, for a strategy $\sigma \in \Sigma_M^{\mathsf{Opt}}$, we write $\mathbb{P}_{M_\sigma, s}$ as $\mathbb{P}_{\sigma, s}$ and $\mathbb{P}_{M'_\sigma, s}$ as $\mathbb{P}'_{\sigma, s}$. Similarly, we write $\mathbb{E}_{M_\sigma, s}$ as $\mathbb{E}_{\sigma, s}$ and $\mathbb{E}_{M'_\sigma, s}$ as $\mathbb{E}'_{\sigma, s}$.

In Algorithm 2, we present how to get a strategy which minimizes the conditional expectation $\mathbb{E}_{M_\sigma, s}(\mathsf{len}(\rho, T) \mid \rho \models \Diamond T)$ among the strategies in $\Sigma_{M,s}(\Diamond T)$.

Algorithm 2 Calculating distance-optimal strategy for reachability

Require: $M = (S, A, P), T \subseteq S$

1: Create MDP $M' = (s', A, P')$ according to Definition 2.12.
2: Find a strategy σ^* that minimizes the expected distance in M':

$$\sigma^* \in \arg\min_\sigma \mathbb{E}'_{\sigma, s_0}(\mathsf{len}(\rho, T)) \,.$$

3: **return** σ^*.

To prove the correctness of Algorithm 2, we use following lemmata.

Lemma 2.5

For any strategy $\sigma \in \Sigma_M^{\mathsf{Opt}}$*,* $s_0 \in S'$ *and a path* $\rho = s_0 a_0 s_1 \dots s_n \in (S' \setminus T \cdot A)^* T \cap \mathsf{Paths}_M(s_0, \sigma)$*,*

$$\mathbb{P}'_{\sigma, s_0}(\mathsf{Paths}^\omega_{M'_\sigma}(\rho)) = \frac{\mathbb{P}_{\sigma, s_0}(\mathsf{Paths}^\omega_{M_\sigma}(\rho))}{\mathsf{Val}_M(s_0)} \,.$$

Proof As $s_n \in T$, $\mathsf{Val}_M(s_n, T) = 1$. So,

$$\begin{aligned}
\mathbb{P}'_{\sigma,s_0}(\mathsf{Paths}^{\omega}_{M'_\sigma}(\rho)) &= \prod_{i=0}^{n-1} \sigma(\rho_{|i}, a_i) \cdot P'(s_i, a_i, s_{i+1}) \\
&= \prod_{i=0}^{n-1} \sigma(\rho_{|i}, a_i) \cdot P(s_i, a_i, s_{i+1}) \cdot \frac{\mathsf{Val}_M(s_{i+1}, T)}{\mathsf{Val}_M(s_i, T)} \\
&= \mathbb{P}_{\sigma,s_0}(\mathsf{Paths}^{\omega}_{M_\sigma}(\rho)) \cdot \frac{\mathsf{Val}_M(s_n)}{\mathsf{Val}_M(s_0)} \\
&= \frac{\mathbb{P}_{\sigma,s_0}(\mathsf{Paths}^{\omega}_{M_\sigma}(\rho))}{\mathsf{Val}_M(s_0)}
\end{aligned}$$

□

Lemma 2.6

For any strategy $\sigma \in \Sigma_M^{\mathsf{Opt}}$ and $s_0 \in S'$, we have: $\mathbb{P}'_{\sigma,s_0}(\Diamond T) = \frac{\mathbb{P}_{\sigma,s_0}(\Diamond T)}{\mathsf{Val}_M(s_0)}$.

Proof Note that, $\mathsf{Paths}_{M'}(s_0) \cap (S' \setminus T)^* T = \mathsf{Paths}_M(s_0) \cap (S \setminus T)^* T$, as in M', we only removed states from which the states in T are not reachable. So, using the result from Lemma 2.5, we get:

$$\begin{aligned}
\mathbb{P}'_{\sigma,s_0}(\Diamond T) &= \sum_{p \in \mathsf{Paths}_{M'}(s_0) \cap (S' \setminus T)^* T} \mathbb{P}_{M',s_0}(\mathsf{Paths}^{\omega}_{M'_\sigma}(p)) \\
&= \sum_{p \in \mathsf{Paths}_{M}(s_0) \cap (S \setminus T)^* T} \frac{\mathbb{P}_{M,s_0}(\mathsf{Paths}^{\omega}_{M_\sigma}(p))}{\mathsf{Val}_M(s_0)} \\
&= \frac{\mathbb{P}_{\sigma,s_0}(\Diamond T)}{\mathsf{Val}_M(s_0)}
\end{aligned}$$

□

Corollary 2.1

For $s_0 \in S'$, $\sigma \in \Sigma_{M,s_0}(\Diamond T)$ if and only if $\mathbb{P}'_{\sigma,s_0}(\Diamond T) = 1$.

Thus, for $s_0 \in S'$, and any strategy $\sigma \in \Sigma_{M,s_0}(\Diamond T)$, $\mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho, T)) \neq \infty$ if and only if $\sigma \in \Sigma_{M,s_0}(\Diamond T)$. Now we show the relation between expected conditional length in M' and M:

Lemma 2.7

For any strategy $\sigma \in \Sigma_{M,s}(\Diamond T)$, $s_0 \in S'$,

$$\mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho, T)) = \frac{\mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho, T) \mid \rho \models \Diamond T)}{\mathsf{Val}_M(s_0)}\,.$$

Proof Note that, $\mathsf{Paths}_{M'}(s_0) \cap (S' \setminus T)^*T = \mathsf{Paths}_M(s_0) \cap (S \setminus T)^*T$ as in M', we only removed states from which the states in T are not reachable. Using the result from Lemma 2.5, we get:

$$\begin{aligned}\mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T)) &= \sum_{p \in \mathsf{Paths}_{M'}(s_0) \cap (S' \setminus T)^*T} \mathsf{len}(p,T) \cdot \mathbb{P}'_{\sigma,s_0}(\mathsf{Paths}^{\omega}_{M'_\sigma}(p)) \\ &= \sum_{p \in \mathsf{Paths}_{M'}(s_0) \cap (S' \setminus T)^*T} \mathsf{len}(p,T) \cdot \frac{\mathbb{P}_{\sigma,s_0}(\mathsf{Paths}^{\omega}_{M_\sigma}(p))}{\mathsf{Val}_M(s_0)} \\ &= \frac{\mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T)}{\mathsf{Val}_M(s_0)}\end{aligned}$$

□

Now, following theorem proves the correctness of Algorithm 2:

Theorem 2.8

For $s_0 \in S'$, let σ^* be a strategy in M' that minimizes $\mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T))$. Then,

1. $\mathbb{P}_{\sigma^*,s_0}(\Diamond T) = \mathsf{Val}_M(s_0)$.
2. $\mathbb{E}_{\sigma^*,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T) = \min_{\sigma \in \Sigma_{M,s_0}(\Diamond T)} \mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T)$

Proof We know that $\mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T)) \neq \infty$ if and only if $\sigma \in \Sigma_{M,s_0}(\Diamond T)$. So $\sigma^* \in \Sigma_{M,s_0}(\Diamond T)$. Therefore, we have $\mathbb{P}_{\sigma^*,s_0}(\Diamond T) = \mathsf{Val}_M(s_0)$.

From Lemma 2.7, we get $\mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T) = \mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T)) \cdot \mathsf{Val}_M(s_0)$ for any $\sigma \in \Sigma_{M,s_0}(\Diamond T)$. Then,

$$\begin{aligned}\arg\min_{\sigma \in \Sigma_{M,s_0}(\Diamond T)} \mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T)) &= \arg\min_{\sigma \in \Sigma_{M,s_0}(\Diamond T)} \mathbb{E}'_{\sigma,s_0}(\mathsf{len}(\rho,T)) \cdot \mathsf{Val}_M(s_0) \\ &= \arg\min_{\sigma \in \Sigma_{M,s_0}(\Diamond T)} \mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T)\end{aligned}$$

So, $\mathbb{E}_{\sigma^*,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T) = \min_{\sigma \in \Sigma_{M,s_0}(\Diamond T)} \mathbb{E}_{\sigma,s_0}(\mathsf{len}(\rho,T) \mid \rho \models \Diamond T)$. □

2.5 Multi-armed bandit problem

The name of the problem comes from imagining a scenario where a gambler is in a casino standing in front of a row of slot machines (sometimes known as "one-armed bandits"), who has to decide which machines to play in order to maximize their payoff. We are interested in this problem because it forms the basis of the theoretical analysis of

Monte Carlo tree search algorithms, an online heuristic approach to find good strategies in an MDP.

Let A denote a finite set of actions. For each $a \in A$, let $(x_{a,t})_{t\geq 1}$ be a sequence of random rewards associated to a. They correspond to successive plays of action a, and for every action a and every $t \geq 1$, let $x_{a,t}$ be drawn with respect to a probability distribution $\mathcal{D}_{a,t}$ over $[0, 1]$. We denote by $X_{a,t}$ the random variable associated to this drawing. In a *fixed distributions* setting (the classical bandit problem), every action is associated to a fixed probability distribution $\mathcal{D}_a$, so that $\mathcal{D}_{a,t} = \mathcal{D}_a$ for all $t \geq 1$.

The *bandit problem* [Mun14] consists of a succession of steps where the player selects an action and observes the associated reward, while trying to maximise the cumulative gains. For example, selecting action a, then b and then a again would yield the respective rewards $x_{a,1}$, $x_{b,1}$ and $x_{a,2}$ for the first three steps, drawn from their respective distributions.

Let the *regret* R_n denote the difference, after n steps, between the optimal expected reward $\max_{a\in A} \mathbb{E}[\sum_{t=1}^{n} X_{a,t}]$ and the expected reward associated to our action selection. The goal is to minimise the long-term regret when the number of steps n increases. As the underlying reward distributions are unknown, potential rewards must be estimated based on past observations. This leads to the exploitation-exploration dilemma: one needs to balance between exploiting the action currently believed to be optimal and exploring other actions that is appearing sub- optimal but may turn out to be better in the long run.

In [LR85], it was shown that for a large class of reward distributions, there exists no strategy for selecting the correct arm with a regret that grows slower than $O(\log n)$. A strategy is said to resolve the exploration- exploitation tradeoff if its regret growth rate is within a constant factor of this best possible regret rate.

The algorithm *UCB1* (upper confidence bound 1) of [ACF02] offers a practical solution to this problem for the stationary bandit case.For an action a and $n \geq 1$, let $\overline{x}_{a,n} = \frac{1}{n}\sum_{t=1}^{n} x_{a,t}$ denote the average payoff obtained from the first n plays of a. Moreover, for a given step number t let t_a denote how many times action a was selected in the first t steps. The algorithm UCB1 chooses, at step $t + 1$, the action a that maximizes $\overline{x}_{a,t_a} + c_{t,t_a}$, where c_{t,t_a} is defined as $\sqrt{\frac{2\ln t}{t_a}}$ if $t_a > 0$, and as $+\infty$ otherwise. This procedure enjoys the regret bound of $O(\log n)$ [ACF02].

2.5.1 Non-stationary Bandit Problem

We will make use of an extension of these results to the general setting of *non-stationary* bandit problems, where the distributions $\mathcal{D}_{a,t}$ are no longer fixed with respect to t. This problem has been studied in [KS06], and results were obtained for a class of distributions $\mathcal{D}_{a,t}$ that respect assumptions referred to as *drift conditions*.

For a fixed $n \geq 1$, let $\overline{X}_{a,n}$ denote the random variable obtained as the average of the random variables associated with the first n plays of a. Let $\mu_{a,n} = \mathbb{E}[\overline{X}_{a,n}]$. We assume that these expected means eventually converge, and let $\mu_a = \lim_{n\to\infty} \mu_{a,n}$. We also assume the following conditions on the sequences of random variables:

Definition 2.13 (Drift conditions)

- *For all $a \in A$, the sequence $(\mu_{a,n})_{n\geq 1}$ converges to some value μ_a.*
- *There exists a constant $C_p > 0$ and an integer N_p such that for $n \geq N_p$ and any $\delta > 0$, $\Delta_n(\delta) = C_p\sqrt{n\ln(1/\delta)}$, the following bounds hold:*

$$\mathbb{P}\Big[n\overline{X}_{a,n} \geq n\mu_{a,n} + \Delta_n(\delta)\Big] \leq \delta\,,$$
$$\mathbb{P}\Big[n\overline{X}_{a,n} \leq n\mu_{a,n} - \Delta_n(\delta)\Big] \leq \delta\,.$$

We define $\delta_{a,n} = \mu_{a,n} - \mu_a$. Then, μ^*, μ^*_n, δ^*_n are defined as μ_j, $\mu_{j,n}$, $\delta_{j,n}$ where j is the optimal action.[3] Moreover, let $\Delta_a = \mu^* - \mu_a$.

As $\delta_{a,n}$ converges to 0 by assumption, for all $\epsilon > 0$ there exists $N_0(\epsilon) \in \mathbb{N}$, such that for $t > N_0(\epsilon)$, then $2|\delta_{a,t}| \leq \epsilon\Delta_a$ and $2|\delta^*_t| \leq \epsilon\Delta_a$ for all all suboptimal actions $a \in A$.

We recall the results of [KS06], and provide an informal description of those results. Consider using the algorithm UCB1 on a non-stationary bandit problem satisfying the drift conditions, with $c_{t,t_a} = 2C_p\sqrt{\frac{\ln t}{t_a}}$ for some constant C_p.

Let $T_a(n)$ denote number of times action a has been played at time n. Let $\overline{X}_n = \sum_{a\in A}\frac{T_a(n)}{n}\overline{X}_{a,T_a(n)}$ denote the global average of payoffs received up to time n. Then, one can bound the difference between μ^* and $\overline{X}_n$:

[3] It is assumed for simplicity that a single action a is optimal, *i.e.* maximises $\mathbb{E}[X_{a,n}]$ for n large enough.

Theorem 2.9 ([KS06, Theorem 2])

Under the drift conditions of Definition 2.13, it holds that

$$|\mathbb{E}[\overline{X}_n] - \mu^*| \leq |\delta_n^*| + \mathcal{O}\left(\frac{|A|({C_p}^2 \ln n + N_0(1/2))}{n}\right).$$

This is the main theoretical guarantee obtained for the optimality of UCB1. Also for any action a, the authors state a lower bound for the number of times the action is played.

Theorem 2.10 ([KS06, Theorem 3])

Under the drift conditions of Definition 2.13, there exists some positive constant ρ such that after n iterations for all action a, $T_a(n) \geq \lceil \rho \ln(n) \rceil$.

The authors also prove a tail inequality similar to the one described in the drift conditions, but on the random variable $\overline{X}_n$ instead of $\overline{X}_{a,n}$. This will be useful for inductive proofs later on, when the usage of UCB1 is nested so that the global sequence $\overline{X}_n$ corresponds to a sequence $\overline{X}_{b,n}$ of an action b played from a previous state of the MDP.

Theorem 2.11 ([KS06, Theorem 4])

Fix an arbitrary $\delta > 0$ and let $\Delta_n = 9\sqrt{2n \ln(2/\delta)}$. Then for big enough n, the following holds true:

$$\mathbb{P}[n\overline{X}_n \geq n\mathbb{E}[\overline{X}_n] + \Delta_n(\delta)] \leq \delta$$
$$\mathbb{P}[n\overline{X}_n \leq n\mathbb{E}[\overline{X}_n] - \Delta_n(\delta)] \leq \delta$$

Finally, it is shown that the probability of making the wrong decision (choosing a suboptimal action) converges to 0 as the number of plays n grows large enough.

Theorem 2.12 ([KS06, Theorem 5])

Let I_t be the action chosen at time t, and let a^* be the optimal action. Then $\lim_{t\to\infty} Pr(I_t \neq a^*) = 0$.

2.6 Monte Carlo tree search

In *heuristic search*, from a state s, a tree of paths are created to approximate the expected total reward Val_M^H and corresponding optimal action opt_M^H. Conventional heuristic search

algorithms are based on Monte Carlo methods which estimate action values from current state by simulating many paths that start with each possible actions. When the action value estimates are accurate enough, the action enough or a specified budget (of number of iterations or time) is exhausted, the action with the highest estimate is returned.

We specifically focus on the *Monte Carlo Tree Search* (MCTS) algorithm [Bro+12]. Note that rewards in the MDP M are bounded as there are finitely many paths of length at most H, with rewards in $\mathbb{R}$. Thus, for the sake of simplicity we assume without loss of generality that for all paths p of length at most H the total reward $\mathsf{Reward}_M(p)$ belongs to $[0, 1]$.

Given an initial state s_0, MCTS is an iterative process that incrementally constructs a search tree rooted at s_0 describing paths of M and their associated values. An iteration constructs a path in M by following a decision strategy to *select* a sequence of nodes in the search tree. When a node that is not part of the current search tree is reached, the tree is expanded with this new node, whose expected reward is approximated by *simulation*. This value is then used to update the knowledge of all selected nodes in *backpropagation*.

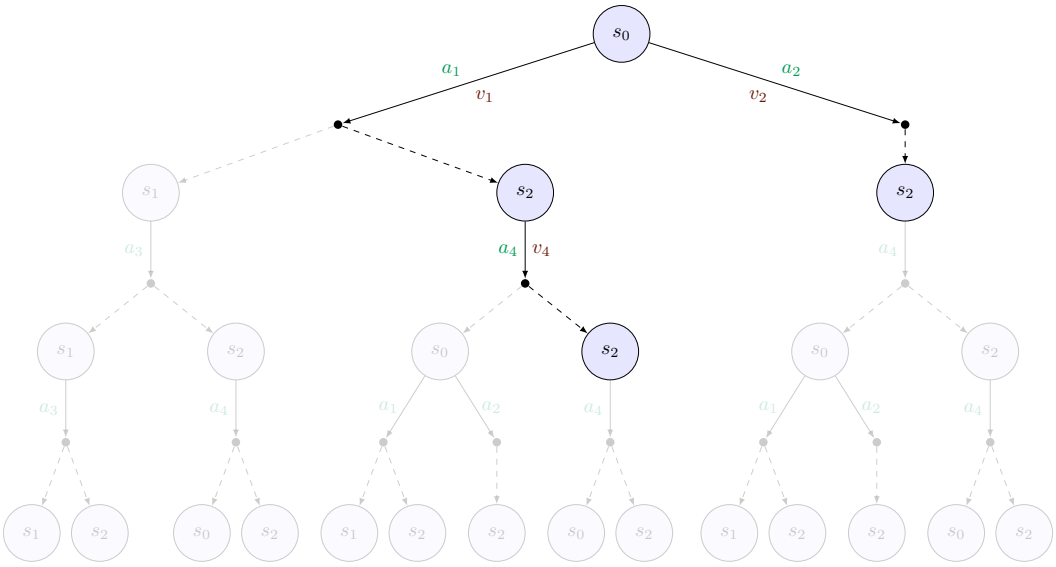

Figure 2.6: Already-built part of the search tree at the start of iteration

In the search tree, each node represents a path. For a node p and an action $a \in A$, let $\mathsf{children}(p, a)$ be a list of nodes representing paths of the form $p \cdot as'$ where $s' \in \mathsf{Supp}(P(\mathsf{last}(p), a))$. For each node we store a value $\mathsf{value}(p)$ computed for node p, meant to approximate $\mathsf{Val}_M^{H-|p|}(\mathsf{last}(p))$. For each node p and action a, we also store the value

$\mathsf{value}(p, a)$ which is meant to approximate the value

$$R(\mathsf{last}(p), a) + \sum_{s'} P(\mathsf{last}(p), a, s')\mathsf{Val}_M^{H-|p|-1}(s') \,.$$

We also store a counter $\mathsf{count}(p)$ that keeps track of the number of iterations that selected node p and $\mathsf{count}(p, a)$ that selected the action a from p. We add subscripts $i \geq 1$ to these notations to denote the number of previous iterations, so that $\mathsf{value}_i(p)$ is the value of p obtained after i iterations of MCTS, among which p was selected $\mathsf{count}_i(p)$ times. We also define $\mathsf{total}_i(p)$ and $\mathsf{total}_i(p, a)$ as shorthand for respectively $\mathsf{value}_i(p) \times \mathsf{count}_i(p)$ and $\mathsf{value}_i(p, a) \times \mathsf{count}_i(p, a)$.

Each iteration consists of three phases. Let us describe these phases at i^{th} iteration.

Selection phase Starting from the root node, MCTS descends through the existing search tree by choosing actions based on the current values and counters and by selecting next states stochastically according to the MDP. This continues until reaching a node q, either outside of the search tree or at depth H. In the former case, the simulation phase is called to obtain a value $\mathsf{value}_i(q)$ that will be backpropagated along the path q. In the latter case, we use the exact value $\mathsf{value}_i(q) = R_T(\mathsf{last}(q))$ instead.

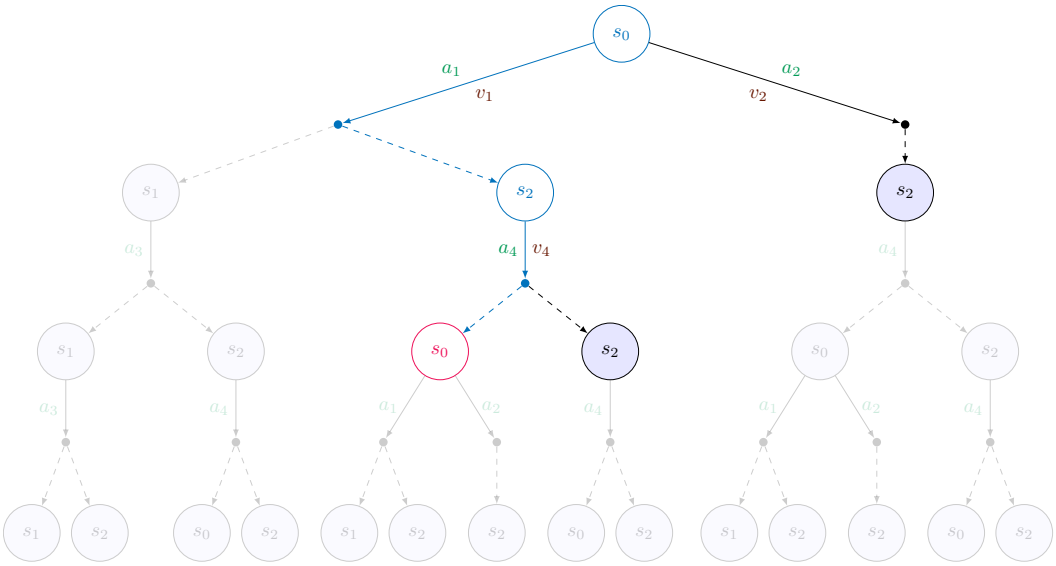

Figure 2.7: MCTS traverses through the existing search tree till it finds a new node and adds the new node to the tree during selection phase.

The action selection process needs to balance between the exploration of new paths

and the exploitation of known, promising paths. A popular way to balance both is the *upper confidence bound for trees* (UCT) algorithm [KS06], that interprets the action selection problem of each node of the MCTS tree as a bandit problem, and selects, for some constant C, an action a^* in the set

$$\arg\max_{a \in A} \left[\mathsf{value}_{i-1}(p, a) + C \sqrt{\frac{\ln\left(\mathsf{count}_{i-1}(p)\right)}{\mathsf{count}_{i-1}(p, a)}} \right] .$$

Simulation phase In the simulation phase, the goal is to get an *initial approximation* for the value of a node p, that will be refined in future iterations of MCTS.

Classically, a sampling-based approach can be used, where one computes a fixed number $c \in \mathbb{N}$ of paths $p \cdot p'$ in $\mathsf{Paths}_M^H(p)$. Then, one can compute the value of the node by averaging these values by

$$\mathsf{value}_i(p) = \frac{1}{c} \sum_{p'} \mathsf{Reward}_M(p') \,,$$

and fix $\mathsf{count}_i(p)$ to 1. Usually, the samples are derived by selecting actions uniformly at random in the MDP.

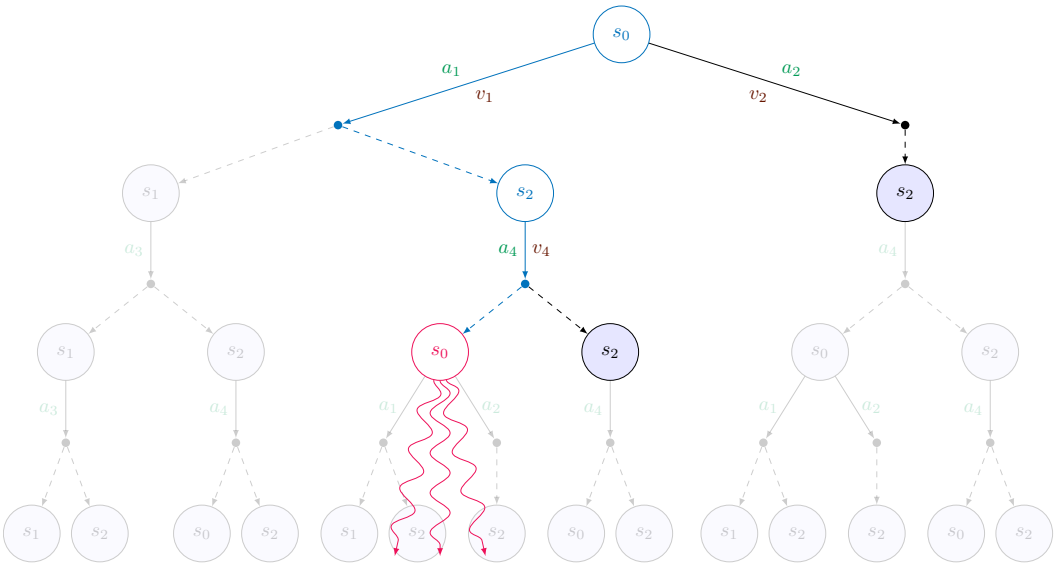

Figure 2.8: MCTS approximates the value of the newly-added node by sampling paths during simulation phase.

Backpropagation phase From the value $\mathsf{value}_i(p)$ obtained at a leaf node $p = s_0 a_0 s_1 \dots s_h$ at depth h in the search tree, let $\mathsf{reward}_i(p_{|k}) = \sum_{l=k}^{h-1} R(s_l, a_l) + \mathsf{value}_i(p)$ denote the re-

ward associated with the path from node $p_{|k}$ to p in the search tree. For k from 0 to $h-1$ we update the values according to

$$\mathsf{value}_i(p_{|k}) = \frac{\mathsf{total}_{i-1}(p_{|k}) + \mathsf{reward}_i(p_{|k})}{\mathsf{count}_i(p_{|k})}\,.$$

Similarly, the value $\mathsf{value}_i(p_{|k}, a_k)$ is updated based on $\mathsf{total}_{i-1}(p_{|k}, a_k)$, $\mathsf{reward}_i(p_{|k})$ and $\mathsf{count}_i(p_{|k}, a_k)$ with the similar formula:

$$\mathsf{value}_i(p_{|k}, a_k) = \frac{\mathsf{total}_{i-1}(p_{|k}, a_k) + \sum_{l=k}^{h-1} R(s_l, a_l) + \mathsf{value}_i(p)}{\mathsf{count}_i(p_{|k}, a_k)}\,.$$

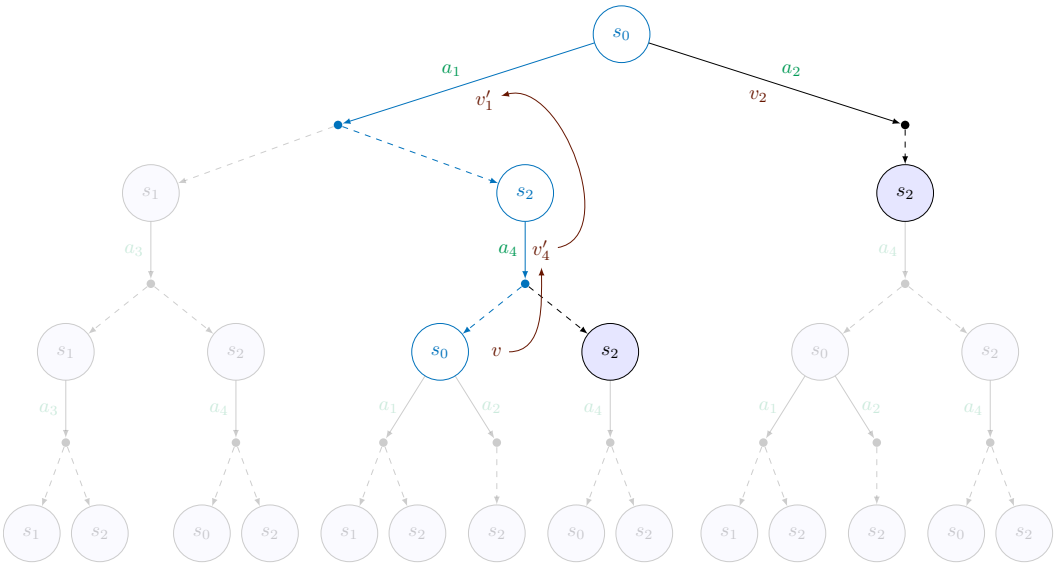

Figure 2.9: The value obtained at the new node is used to update the values in the nodes above in the search tree during backpropagation phase.

After n iterations of MCTS algorithms at state s_0, the action that maximizes $\mathsf{value}_n(s_0, a)$ is returned.

The MCTS algorithm proposed in [KS06] was presented for a version where MCTS was called recursively until leaves were reached, as opposed to the sampling-based approach that we discussed before. This algorithm enjoys the following guaranties:

Theorem 2.13

[KS06, Theorem 6] Consider an MDP M, a horizon H and a state s_0. Let $V_n(s_0)$ (resp. $V_n(s_0, a)$) be a random variable that represents the value $\mathsf{value}_n(s_0)$ (resp. $\mathsf{value}_n(s_0, a)$) at the root of the search tree after n iterations of the MCTS

algorithm on M. Then,

- $|\mathbb{E}[V_n(s_0)] - \mathsf{Val}_M^H(s_0)|$ *is bounded by* $\mathcal{O}((\ln n)/n)$.
- *The failure probability* $\mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}_M^H(s_0)]$ *converges to zero as* n *tends to infinity.*

The simulation-based approach has become more standard in practice and the same guaranties can be obtained[4] in the simulation-based approach as in the version mentioned in [KS06].

2.7 Task systems

A *task system* [GGR18] $\Upsilon = ((\tau_i)_{i\in[n]}, F, H)$ is composed of a set of n pre-emptible[5] tasks $(\tau_i)_{i\in[n]}$ partitioned into a set F of *soft tasks* and a set H of *hard tasks*. Time is assumed to be discrete and measured *e.g.* in CPU ticks.

Each task τ_i generates an infinite number of instances $\tau_{i,j}$, called *jobs* for $j \in \mathbb{N}$. Jobs generated by both hard and soft tasks are equipped with deadlines, which are relative to the respective arrival times of the jobs in the system. The computation time requirements of the jobs follow a discrete probability distribution, and are unknown to the scheduler, but upper bounded by their relative deadline. The tasks are assumed to be independent and generated stochastically: the occurrence of a new job of one task does not depend on the occurrences of jobs of other tasks, and both the inter-arrival and computation times of jobs are independent random variables.

Formally a *task* can be defined as follows:

Definition 2.14 (Task)

A task τ is a tuple $\langle C, D, A\rangle$, where:

- *C is a discrete probability distribution on the (finitely many) possible computation times of the jobs generated by τ;*
- *$D \in \mathbb{N}$ is the deadline of all jobs generated by τ which is relative to their arrival time; and*

[4] This can be observed as a direct implication of Theorem 5.1 which we will discuss in Section 5.1.

[5] This means the scheduler can temporarily interrupt a running task, and can resume it at a later time.

- *A is a discrete probability distribution on the (finitely many) possible inter-arrival times of the jobs generated by τ.*

Jobs generated by hard tasks must complete before their respective deadlines. For jobs generated by soft tasks, deadline misses result in a penalty. This is modelled by a cost function $cost : F \to \mathbb{Q}_{\geq 0}$ that associates to each soft task τ_j a cost $c(j)$ that is incurred every time a job of τ_j misses its deadline. The *scheduling problem* consists in finding a *scheduler*, i.e. a strategy for the scheduler that associates, to all CPU ticks, a task that must run at that moment, in order to avoid deadline misses by hard tasks and to minimize the mean cost of deadline misses by soft tasks.

It is assumed that, for a task $\tau = \langle C, D, A\rangle$, $\max(\mathsf{Supp}(C)) \leq D \leq \min(\mathsf{Supp}(A))$. Hence, at any point in time, there is at most one job per task in the system. Also, when a new job of some task arrives at the system, the deadline for the previous job of this task is already over. Finally, we assume that the task system is *schedulable for the hard tasks*, meaning that it is possible to guarantee that jobs associated to hard tasks never miss their deadlines.

Given a task system $\Upsilon = ((\tau_i)_{i\in[n]}, F, H)$ with n tasks, the *structure of the task system* Υ is $((\mathsf{struct}(\tau_i))_{i\in[n]}, F, H)$ where $\mathsf{struct}(\langle C, D, A\rangle) = (\langle \mathsf{Supp}(C), D, \mathsf{Supp}(A)\rangle)$.

We denote by $C_{\max}$ and $A_{\max}$ resp. the maximum computation time, and the maximum inter-arrival time of a task in Υ. Formally, $C_{\max} = \max(\bigcup_{i\in[n]} \mathsf{Supp}(C_i))$, and $A_{\max} = \max(\bigcup_{i\in[n]} \mathsf{Supp}(A_i))$. We also let $\mathbb{D} = \max_{i\in[n]}(|\mathsf{Supp}(A_i)|)$. We denote by $|\Upsilon|$ the number of tasks in the task system Υ.

Consider two task systems $\Upsilon_1 = ((\tau_i^1)_{i\in[n]}, F, H)$, and $\Upsilon_2 = ((\tau_i^2)_{i\in[n]}, F, H)$, with $|\Upsilon_1| = |\Upsilon_2|$, $\tau_i^j = \langle C_i^j, D_i^j, A_i^j\rangle$ for all $i \in [n]$ and $j \in [2]$. The two task systems Υ_1 and Υ_2 are said to be ϵ-*close*, denoted $\Upsilon_1 \approx^\epsilon \Upsilon_2$, if they have the same structure and the distributions are ϵ-close, *i.e.*,

- $\mathsf{struct}(\Upsilon^1) = \mathsf{struct}(\Upsilon^2)$,
- for all $i \in [n]$, we have $A_i^1 \sim^\epsilon A_i^2$, and
- for all $i \in [n]$, we have $C_i^1 \sim^\epsilon C_i^2$.

2.7.1 MDP for the scheduling problem

Given a system $\Upsilon = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of tasks, we describe the modelling of the scheduling problem by an MDP M_Υ. A *state* in the MDP encodes the following information about each task τ_i:

- a *distribution* c_i over the job's possible remaining computation times;
- the time d_i up to its deadline; and
- a distribution a_i over the possible times up to the next arrival of a new job.

For a state $s = \big((c_1, d_1, a_1) \ldots (c_n, d_n, a_n)\big)$, let $act(s) = \{i \mid c_i(0) \neq 1 \text{ and } d_i > 0\}$ be the tasks that have an active job in s. Let $mis(s) = \{i \mid c_i(0) = 0 \text{ and } d_i = 0\}$ be the tasks that have missed a deadline in s.

The possible *actions* from state s are to schedule an active task from $act(s)$ or to idle the CPU (represented by ε) for one CPU tick. The probability distribution arises from the valid moves available for the task generator:

- stay idle (ε),
- finish the current job without submitting a new one, (fin)
- submit a new job while the previous one is already finished (sub),
- submit a new job and kill the previous one, in the case of a soft task ($\mathrm{killANDsub}$).

Whenever a job by a soft task misses its deadline, it incurs a *cost*. This is represented as a negative reward. Whenever a hard task misses its deadline, the MDP reaches a sink state marked with $\perp$. Our *objective* is to find a strategy that

- avoids the state marked by $\perp$, and
- minimizes the average mean payoff reward

Example 2.7 Consider a system with the following tasks:

- one hard task $\tau_h = \langle C_h, 2, A_h \rangle$ such that $C_h(1) = 1$ and $A_h(3) = 1$; and
- one soft task $\tau_s = \langle C_s, 2, A_s \rangle$ such that $C_s(1) = 0.4$, $C_s(2) = 0.6$, and $A_s(3) = 1$; and the cost function c such that $c(\tau_s) = 10$.

This means that both tasks will submit their first job at time 0, both with deadlines at time $0 + 2 = 2$. Then, $\tau_{h,1}$ will have a computation time of 1, while $\tau_{s,1}$ will have a computation time which is either 1 (with probability 0.4) or 2 (with probability 0.6). Both tasks will

submit new jobs $\tau_{h,2}$ and $\tau_{s,2}$ at time $0 + 3 = 3$. Each time a job of τ_s misses its deadline, a penalty of 10 will be incurred.

Figure 2.10 presents part of the MDP Γ_Υ built from the set of tasks $\tau = \{\tau_h, \tau_s\}$.

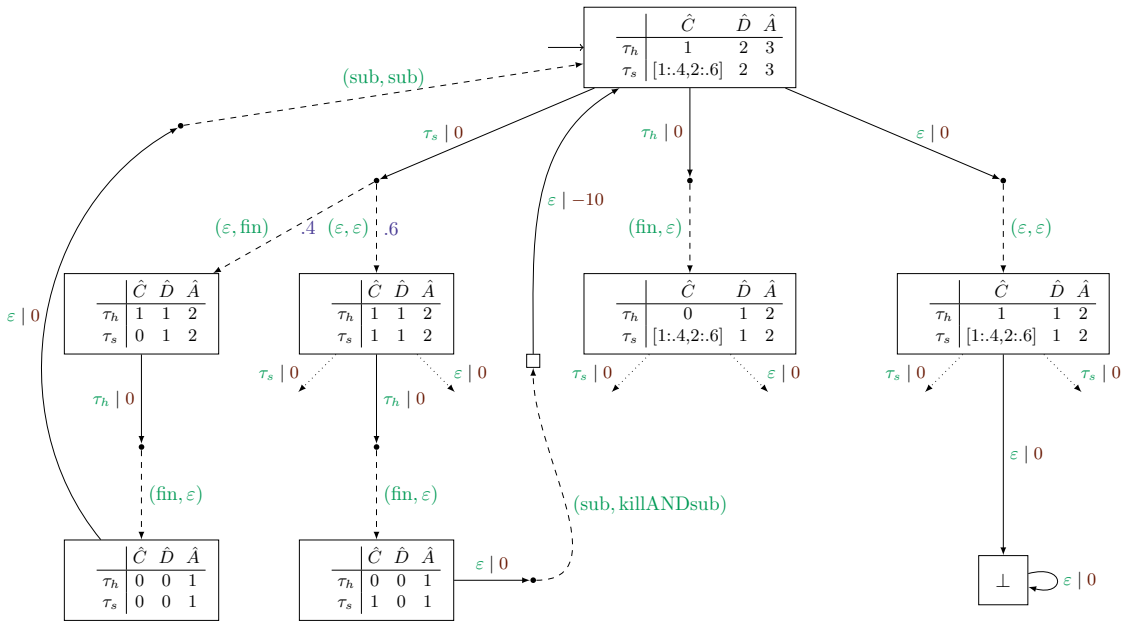

Figure 2.10: Part of the MDP for the task system in Example 2.7

The probabilistic transitions are labelled with a tuple denoting the task generators action corresponding to task τ_h and τ_s. From the initial state, the scheduler can schedule τ_s for one CPU tick, or schedule τ_h for one CPU tick or stay idle. If it schedules τ_s, then with probability 0.4, τ_s finishes and with probability 0.6, it would still have 1 CPU tick computation time left. If τ_s misses its deadline, a reward of -10 is added. When τ_h misses its deadline the MDP reaches a sink state marked with $\perp$. □

2.7.2 Safe region in a task system

We can consider the underlying game graph of the MDP M_Υ. There we have a safety game between Scheduler (player 0) and TaskGen (player 1) where the objective for Scheduler is to avoid the state marked with $\perp$. We say a state s is *safe* if Scheduler has a winning strategy from s in the safety game. For a state s, $\mathsf{safe}(s)$ is the set of actions a such that $(s, a) \in \sigma(s)$ where σ is the most general strategy for Scheduler in the safety game.

We consider the rest of the actions *unsafe*.

Given an initial state s_{init} in the MDP M_Υ, we let the *safe region* of M_Υ be the MDP $M_\Upsilon^{\mathsf{safe}}$ obtained from M_Υ by applying the following transformations:

- remove from M_Υ all unsafe actions;
- remove from M_Υ all vertices and edges that are not reachable from s_{init}.

Note that $M_\Upsilon^{\mathsf{safe}}$ is guaranteed to be non-empty as the task system Υ is guaranteed to be schedulable for its hard tasks by hypothesis. In the following lemma, we prove that this safe region is a single maximal end component:

Lemma 2.8

Let $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ be a task system and let $M_\Upsilon^{\mathsf{safe}}$ be the safe region of its MDP. Then $M_\Upsilon^{\mathsf{safe}}$ is a single maximal end-component.

Proof Observe that, since we want to prove that the whole MDP $M_\Upsilon^{\mathsf{safe}}$ corresponds to an MEC, we only need to show that the underlying graph of $M_\Upsilon^{\mathsf{safe}}$ is strongly connected. In order to show the strongly connected property, we fix a vertex v controlled by Scheduler, and show that there exists a path in $M_\Upsilon^{\mathsf{safe}}$ from v to v_{init}. By construction of $M_\Upsilon^{\mathsf{safe}}$, v is reachable from v_{init}, this entails that all vertices v' in the graph are also reachable from v.

Let $v_{init} = v_0, v'_0, v_1, v'_1, \cdots, v'_{n-1}, v_n = v$ be the path p leading to v, where all vertices v_j belong to Scheduler, and all v'_j are vertices that belong to TaskGen in the game graph. Then, from path p, we extract, for all tasks τ_i the sequence of *actual inter-arrival times* $\sigma_i = t^i(1), t^i(2), \ldots, t^i(k_i)$ defined as follows: for all $1 \leq j \leq k_i$, $t^i(j) \in \mathsf{Supp}(A_i)$ is the time elapsed (in CPU ticks) between the arrival of the $(j-1)^{th}$ job and the jth job of task i along p (assuming the initial release occurring in the initial state v_{init} is the 0-th release). In other words, letting $T^i(j) = \sum_{k=1}^{j} t^i(k)$, the jth job of τ_i is released along p on the transition between $v'_{T^i(j-1)}$ and $v_{T^i(j)}$. Observe thus that all tasks $i \in [n]$ are in the same state in vertex v_{init} and in vertex $v_{T^i(j)}$, i.e. the time to the deadline, and the probability distributions on the next arrival and computation times are the same in v_{init} and $v_{T^i(j)}$. However, the vertices $v_{T^i(j)}$ can be different for all the different tasks, since they depend on the sequence of job releases of τ_i along p. Nevertheless, we claim that p can be extended, by repeating the sequence of arrivals of all the tasks along p, in order to reach a vertex where all tasks have just submitted a job (i.e. v_{init}). To this aim, we first extend, for all tasks $i \in [i]$, σ_i into $\sigma'_i = \sigma_i, t^i(k_i + 1)$, where $t^i(k_i + 1) \in \mathsf{Supp}(A_i)$ ensures that the $k_i + 1$ arrival of a τ_i occurs *after* v.

For all $i \in [n]$, let Δ_i denote $\sum_{j=1}^{k_i+1} t^i(j)$, i.e. Δ_i is the total number of CPU ticks needed to reach the first state after v where task i has just submitted a job (following the sequence of arrival σ'_i defined above). Further, let $\Delta = \text{lcm}(\Delta_i)_{i\in[n]}$, the least common multiple of the Δ_i values. Now, let p' be a path in $M_\Upsilon^{\mathsf{safe}}$ that respects the following properties:

1. p is a prefix of p';
2. p' has a length of Δ CPU ticks;
3. p' ends in a Scheduler vertex v'; and
4. for all tasks $i \in [n]$: τ_i submits a job at time t along p' iff it submits a job at time $(t \mod \Delta_i)$ along p.

Observe that, in the definition of p', we do not constrain the decisions of Scheduler after the prefix p. First, let us explain why such a path exists. Observe that the sequence of task arrival times is legal, since it consists, for all tasks i, in repeating Δ/Δ_i times the sequence σ'_i of inter-arrival times which is legal since it is extracted from path p (remember that nothing that Scheduler player does can restrict the times at which TaskGen introduces new jobs in the system). Then, since Υ is schedulable, we have the guarantee that all Scheduler vertices in $M_\Upsilon^{\mathsf{safe}}$ have at least one outgoing edge. This is sufficient to ensure that p' indeed exists. Finally, we observe p' visits v (since p is a prefix of p'), and that the last vertex v' of p' is a Scheduler vertex obtained just after *all tasks* have submitted a job, by construction. Thus, $v' = v_{init}$, and we conclude that, from all Scheduler vertex v which is reachable from v_{init}, one can find a path in $M_\Upsilon^{\mathsf{safe}}$ that leads back to v_{init}.

This reasoning can be extended to account for the TaskGen vertices: one can simply select any successor $\overline{v}$ of v, and apply the above reasoning from $\overline{v}$ to find a path going back to v_{init}. Hence, the graph is strongly connected. □

2.8 Artificial neural networks

Artificial neural network models or *neural networks* or NNs are used to learn a function from a dataset of known inputs and outputs. This can be a *classification* problem, where the outputs are discrete labels : for example, predicting whether the input is picture of a cat or a dog given an image. This can also be a *regression* problem, where the outputs are continuous values: for example, predicting the temperature of tomorrow given today's meteorological data.

A neural network can be viewed as a *computational graph*[6] which is represented by a number of layers. The layer in the middle are called *hidden layers*. A neural network contains some parameters called *weights* which are initially assigned random values. It takes the output y of the previous layer and use the weights w_{ij} and b_j to compute the input x of the next layer where $x_j = \sum_i w_{ij} y_i + b_j$. Each nodes in the layer has a non-linear *activation function* that is applied on the input (otherwise the network would only be able to learn linear functions). Common non-linear functions that are used for activation are :

- *Sigmoid function* : $f(x) = \frac{1}{1+e^{-x}}$.
- *Rectified Linear Unit* or ReLU : $f(x) = \max(0, x)$.

The data is generally stored in multidimensional arrays called *tensors*. The process of transforming raw data into numerical features in tensor format so that it is easily processable by the network is called *feature extraction*. The data is divided into three disjoint sets: one is used solely for *training* the data, one for *validation*, i.e. to monitor the training procedure and another for *testing* the neural network.

Training a neural network would mean finding a set of values for the weights in this network such that it correctly maps inputs to the associated outputs. To measure how good the neural network is, functions called *loss functions* are used which computes a distance score based on the values predicted by the network Y and the expected output $\hat{Y}$. For example, the *mean squared error* is often used as a loss function:

$$\mathrm{MSE}(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i \right)^2$$

The *backpropagation* algorithm decides how much to update a weights w_{ij} by looking at the error E. This is done by following the *update rule* called *gradient descent* [Cau+47; Lem12]:

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{dE}{dw_{ij}}$$

where α is a positive constant called *learning rate*. Intuitively, it means if error decreases when the weight is increased i.e. $\frac{dE}{dw_{ij}} < 0$, the weight should increase. And otherwise, the weight should decrease. In practice, stochastic variants of gradience descent algorithm like *ADAM* [KB14] is used for faster computation.

At the end of the training, quality of the network is determined based on its error rate

[6] A computational graph is a directed graph where the nodes correspond to mathematical operations or variables.

on the test dataset. Sometimes, the network *overfits*, i.e., it shows low error rate on the training dataset but not on the test dataset. To avoid this, *dropout* [Hin+12] is often used where nodes in the neural network are omitted with a given probability called *dropout rate*.

2.8.1 Convolutional neural networks

In a *convolutional neural network* or CNN, each data-point in the dataset is divided in number of *channels* where each channel in a 2-dimensional matrix. Often the multidimensional tensors are *flattened* to create a 1-dimensional vector easily processable by the networks.

In a CNN, the weights are organized into sets of 3-dimensional structural units, known as *filters* or *kernels*. The *convolution* operation places the filter at each possible position in the input, and performs dot products between the weights in the filter and the values in the matching grid of the same size in the input. *Pooling* operations are used to reduce the size of the input by applying the maximum or average function in smaller regions of the input. For example, max-pooling of size 3 can take every 3×3 grid in the input and reduce them to cells containing the maximum value among these 9 values. CNN also contains layers performing linear transformations called *dense layers*.

2.8.2 Hyperparameter tuning

Hyperparameter tuning is often used to choose the optimal architecture for the neural network. This involves tuning for the following *hyperparameters*:

- number of layers,
- size of the hidden layers,
- dropout rate,
- activation functions,
- learning rate etc.

The optimal hyperparameters are searched in a given search space either by exhaustive searching or by randomized optimization [BB12].

Chapter 3

Formal methods in decision-time planning

When an MDP is too large to be analysed offline using formal algorithms, an online approach can be taken. This works as follows: after encountering a new state s, an action a that is "good" for reward optimization starting from the state is computed. This action a is taken from the state s and the state of the MDP evolves to state s' according to the probability distribution in the MDP. Then the same process is repeated from this new state s'. This is called *decision-time planning* [SB18, Chapter 8.8].

In this chapter, we consider approaches that can be used in decision-time planning, specifically using a *receding horizon* H. In this case, the controller tries to find an optimal action from a state by finding an optimal strategy in the MDP *unfolding* of depth H from the current state. Heuristic search algorithms like Monte Carlo tree search (described in Section 2.6), can be used during decision time planning. Another approach that can be used while planning is to model check in a smaller MDP which could be an abstraction of the larger one or could be created by pruning the larger MDP. For this, in Section 3.2, we define abstraction of MDPs using bisimulation. Also in Section 3.3, we define the notion of pruning an MDP using a non-deterministic strategy.

3.1 Receding horizon control

In *receding horizon control* or *model predictive control*, a horizon $H \in \mathbb{N}$ is fixed. From the current state s of the MDP, the algorithm finds $\mathsf{opt}^H_M(s)$, the first action of the strategy that maximizes the expected total reward in that horizon. When this action is identified, it is played from s and the state evolves to a new state s' according to the probability distribution in the MDP. Then, the same process is repeated from s'.

Formally, for a horizon H, it plays according to the strategy σ^H_M, the deterministic memoryless strategy that maps every state s in M to the first action of an optimal strategy for the expected total reward of horizon H starting from state s, which it calculates on-the-fly for the states that are visited. From Theorem 2.7, we get that if we have a large enough

horizon H, in a strongly aperiodic MDP, model predictive control would use an close to optimal strategy.

3.1.1 Unfolding of an MDP

Notice that when using model predictive control with a finite horizon H, when we are at state s, we are only looking at states that are within distance H from s. Thus, it is enough to look at a finite horizon unfolding of the MDP which ignores the states far from the state s.

We will use the notation $T(M, s_0, H)$ to refer to an MDP obtained as a tree-shaped *unfolding* of M from state s_0 and for a depth of H. In particular, the states of $T(M, s_0, H)$ correspond to paths in $\mathsf{Paths}_M^{\leq H}(s_0)$, i.e., the paths of length at most H starting from the state s_0.

Definition 3.1 (Finite horizon unfolding of an MDP)

For an MDP $M = (S, A, P, R, R_T, AP, L)$, a horizon depth $H \in \mathbb{N}$ and a state s_0, the unfolding of M from s_0 and with horizon H is the tree-like MDP defined as $T(M, s_0, H) = (S' = S_0 \cup \cdots \cup S_H, A, P', R', R'_T, AP, L')$, where for all $i \in [0, H]$, $S_i = \mathsf{Paths}^i(s_0)$. The mappings P', R', R'_T and L' are inherited from P, R, R_T and L naturally with additional self-loops at the leaves of the unfolding, so that for all $i \in [0, H]$, $p \in S_i$, $a \in A$ and $p' \in S'$,

$$P'(p, a, p') = \begin{cases} P(\mathsf{last}(p), a, \mathsf{last}(p')) & \text{if } i < H \text{ and } \exists s' \in S, p' = p \cdot as' \\ 1 & \text{if } i = H \text{ and } p' = p \\ 0 & \text{otherwise,} \end{cases}$$

$$R'(p, a) = \begin{cases} R(\mathsf{last}(p), a) & \text{if } i < H \\ 0 & \text{otherwise.} \end{cases}$$

$$R'_T(p) = R_T(\mathsf{last}(p))$$

$$L'_T(p) = L(\mathsf{last}(p))$$

Example 3.1 Figure 3.1 describes the unfolding of the MDP from Example 2.4 for horizon 1 starting from the state $Moving$. □

We will show that for a state s, the optimal expected total reward for horizon H in the

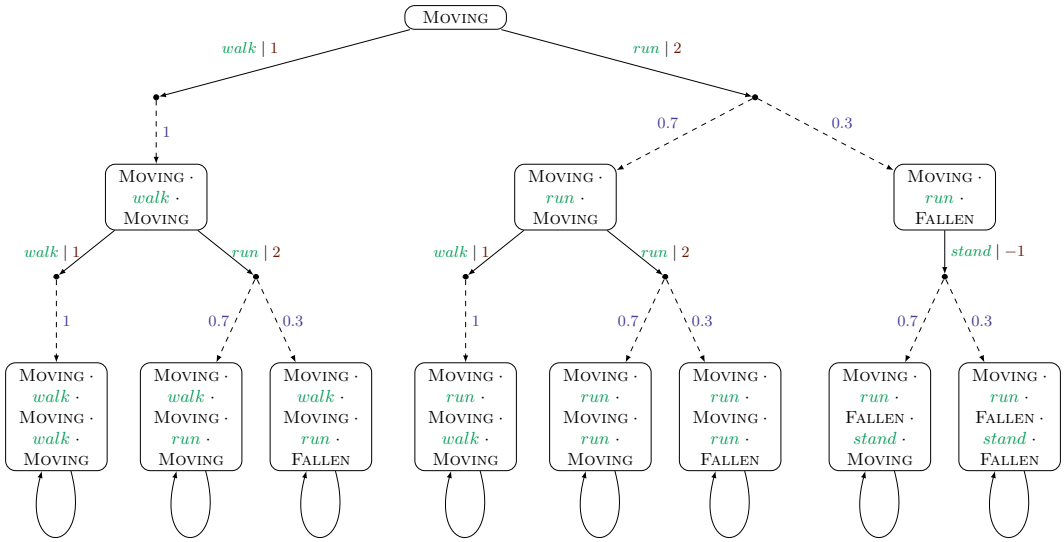

Figure 3.1: Unfolding of the MDP in Example 2.4 for horizon 2 starting from the state MOVING

MDP M (and corresponding strategy that optimizes it) is same as the optimal expected total reward for horizon H in the H-horizon unfolding of M from state s (and corresponding strategy):

Lemma 3.1

$\mathsf{Val}^H_M(s_0)$ is equal to $\mathsf{Val}^H_{T(M,s_0,H)}(s_0)$, and $\mathsf{opt}^H_M(s_0)$ is equal to $\mathsf{opt}^H_{T(M,s_0,H)}(s_0)$.

Proof Let us prove that for all $i \in [0, H]$ and all $p \in S_i$,

- $\mathsf{Val}^{H-i}_M(\mathsf{last}(p)) = \mathsf{Val}^{H-i}_{T(M,s_0,H)}(p)$, and
- $\mathsf{opt}^{H-i}_M(\mathsf{last}(p)) = \mathsf{opt}^{H-i}_{T(M,s_0,H)}(p)$.

We prove the first statement by induction on $H - i$. For $H - i = 0$, for all $p \in S_i$, $\mathsf{Val}^{H-i}_M(\mathsf{last}(p)) = \mathsf{Val}^{H-i}_{T(M,s_0,H)}(p) = R_T(\mathsf{last}(p))$. Assume the statement is true for $H-i = k$, so that for all $p \in S_{H-k}$, $\mathsf{Val}^k_M(\mathsf{last}(p)) = \mathsf{Val}^k_{T(M,s_0,H)}(p)$. Then for all $p \in S_{H-k-1}$, we have for all $a \in A$ and $s \in Supp(P(\mathsf{last}(p), a))$, $\mathsf{Val}^k_M(s) = \mathsf{Val}^k_{T(M,s_0,H)}(p \cdot as)$. It follows that

$$\begin{aligned}
\mathsf{Val}^{k+1}_M(\mathsf{last}(p)) &= \max_{a\in A}(R(\mathsf{last}(p), a) + \sum_s P(\mathsf{last}(p), a)\mathsf{Val}^k_M(s)) \\
&= \max_{a\in A}(R(\mathsf{last}(p), a) + \sum_s P(\mathsf{last}(p), a)\mathsf{Val}^k_{T(M,s_0,H)}(p \cdot as)) \\
&= \mathsf{Val}^{k+1}_{T(M,s_0,H)}(p)\,.
\end{aligned}$$

From $\mathsf{Val}_M^{H-i}(\mathsf{last}(p)) = \mathsf{Val}_{T(M,s_0,H)}^{H-i}(p)$ and $\mathsf{opt}_M^H(\mathsf{last}(p)) = \arg\max_{a\in A}(R(\mathsf{last}(p),a) + \sum_s P(\mathsf{last}(p),a)\mathsf{Val}_M^{H-1}(s))$, we derive $\mathsf{opt}_M^{H-i}(\mathsf{last}(p)) = \mathsf{opt}_{T(M,s_0,H)}^{H-i}(p)$. □

3.2 Bisimulation in MDPs

In Section 2.3.3, we described the notion of bisimulation for Markov chains. Here, we extend the notion of bisimulation for MDPs. The idea behind using bisimulation in decision-time planning is the following: instead of synthesizing an optimal strategy in the original MDP, we can search for an optimal strategy in the *bisimulation quotient*, which is a smaller MDP. Then from this strategy, we can generate an optimal strategy in the original MDP.

Definition 3.2

Let $M = (S, A, P, R, R_T, AP, L)$ be an MDP. A bisimulation relation is an equivalent relation $\sim$ on S such that for all state $s_1, s_2 \in S$ such that $s_1 \sim s_2$, we have:

- *$L(s) = L(s')$, and,*
- *for all $t \in T$ and $a \in A$, $\sum_{t'\sim t} P(s_1, a, t') = \sum_{t'\sim t} P(s_1, a, t')$*

If $s_1 \sim s_2$, we call s_1 and s_2 bisimilar equivalent or bisimilar. Similar to the case for Markov chains, we denote the set $\{s' \mid s' \sim s\}$ by $[s]_\sim$ and the set of equivalent classes by $S_\sim$. The first condition in Definition 3.2 states that the bisimilar states are equally labelled. The last condition requires that for bisimilar states the probability of moving to some equivalence class using same action is equal.

Definition 3.3 (Bisimulation quotient)

Let $M = (S, A, P, R, R_T, AP, L)$ be an MDP. The quotient MDP is defined by $M_\sim = (S_\sim, A, P', R', R'_T, AP, L')$ where

- *For $s, t \in S$ and $a \in A$, $P'([s]_\sim, a, [t]_\sim) = \sum_{t'\sim t} P(s, a, t')$,*
- *For $s \in S$ and $a \in A$, $R'(s, a) = 0$,*
- *For $s \in S$, $R'_T(s) = 0$,*
- *For $s \in S$, $L'([s]_\sim) = L(s)$.*

Note that P' and L' are well-defined from the definition of bisimulation. In our notion of bisimulation, we are only focusing on the case where the bisimilar states would simulate each other with respect to the probabilistic transitions, they are not simulating each other's

associated rewards. This is why in the bisimulation quotient we are ignoring the rewards.

Example 3.2 A state in the MDP for Pac-Man in Example 1.1 can be written as $s = (s^A, s^F)$ where s^A is the encoding of the position of the agents and s^F is the encoding of the position of the food pills. We define a relation $\sim$ where $s_1 \sim s_2$ iff $s_1^A = s_2^A$. In other words, two states are equivalent if the position of the agents are same in both states.

Suppose the states are labelled by $loss$ if it is losing i.e., Pac-Man has the same position as one of the ghosts. This only depend on the position of the agents, not on the food pills. Also, the probabilistic transition of the ghosts does not depend on the food positions. This makes $\sim$ a bisimulation. □

We can extend the notion of bisimilar equivalence to the paths. The paths $p_1 = s_0 a_0 s_1 \ldots$ and $p_2 = t_0 b_0 t_1 \ldots$ are bisimulation equivalent, denoted by $p_1 \sim p_2$, if and only if $s_i \sim t_i$ for all $i \geq 0$ and $a_i = b_i$ for all for all $i \geq 0$. we denote the set $\{p' \mid p' \sim p\}$ by $[p]_\sim$.

Let σ be a strategy in $M_\sim$. This defines a strategy $\gamma(\sigma)$ in M by $\gamma(\sigma)(p) = \sigma([p])$. For the ease of notation, for a strategy σ in $M_\sim$, we write denote the Markov chain $M_{\gamma(\sigma)}$ induced by $\gamma(\sigma)$ in M as M_σ. Out motivation is the following: instead of finding a strategy $\gamma(\sigma)$ in M, it is often easier to find a strategy σ using model-checking techniques in $M_\sim$ as it is a smaller MDP. We will show that a PCTL property will hold for a state s in the Markov chain M_σ if it holds for the state $[s]$ in the Markov chain $(M_\sim)_\sigma$.

We claim that $\sim$ is a bisimulation (for Markov chains) in the Markov chain M_σ:

Lemma 3.2

Let $\sim$ be a bisimulation in an MDP M. Let σ be a strategy in $M_\sim$. Then, $\sim$ is a bisimulation in M_σ.

Proof The states in M_σ are paths in $\mathsf{Paths}_M(\gamma(\sigma))$, i.e., the set of finite paths in M generated by following $\gamma(\sigma)$. Let $M_\sigma = (\mathsf{Paths}_M(\gamma(\sigma)), P_\sigma, AP, L_\sigma)$. For two paths $p_1, p_2 \in \mathsf{Paths}_M(\gamma(\sigma))$ such that $p_1 \sim p_2$, we need to prove that:

- $L_\sigma(p_1) = L_\sigma(p_2)$, and,
- For any state $p_3 \in \mathsf{Paths}_M(\gamma(\sigma))$, $\sum_{t' \sim t} P_\sigma(s_1, t') = \sum_{t' \sim t} P_\sigma(s_1, t')$.

Note that as $p_1 \sim p_2$ in M, $L(\mathsf{last}(p_1)) = L(\mathsf{last}(p_2))$. Hence,

$$L_\sigma(p_1) = L(\mathsf{last}(p_1)) = L(\mathsf{last}(p_2)) = L_\sigma(p_2).$$

This proves the first condition of bisimulation.

Also, for $p \in \mathsf{Paths}_M(\gamma(\sigma))$,

$$P_\sigma(p_1, p) = \begin{cases} \gamma(\sigma)(p_1, a) \cdot P(\mathsf{last}(p_1), a, s) & \text{if } p = p \cdot as, \\ 0 & \text{otherwise.} \end{cases}$$

Then, for a path p_3, if there is no $a \in A$ and $s \in S$ such that $p_3 \sim p_1 \cdot as$, we will have $\sum_{p \sim p_3} P_\sigma(p_1, p) = 0$. As $p_1 \sim p_2$, we will also have that there is no $a \in A$ and $s \in S$ such that $p_3 \sim p_2 \cdot as$. So, $\sum_{p \sim p_3} P_\sigma(p_2, p) = 0$.

If $p_3 \sim p_1 \cdot as$ for some $a \in A$ and $s \in S$, we have:

$$\begin{aligned} \sum_{p \sim p_3} P_\sigma(p_1, p) = \sum_{p \sim p_1 \cdot as} P_\sigma(p_1, p) &= \sum_{p_1 \cdot as' \sim p_1 \cdot as} P_\sigma(p_1, p_1 \cdot as') \\ &= \sum_{s' \sim s} P_\sigma(p_1, p_1 \cdot as') \\ &= \sum_{s' \sim s} \gamma(\sigma)(p_1, a) \cdot P(\mathsf{last}(p_1), a, s') \\ &= \sigma([p_1], a) \sum_{s' \sim s} P(\mathsf{last}(p_1), a, s') \end{aligned}$$

Now as $p_1 \sim p_2$, we also have $p_3 \sim p_2 \cdot as$. So, similarly:

$$\sum_{p \sim p_3} P_\sigma(p_2, p) = \sigma([p_2], a) \sum_{s' \sim s} P(\mathsf{last}(p_2), a, s')$$

Now $[p_1] = [p_2]$ and $\sum_{s' \sim s} P(\mathsf{last}(p_1), a, s') = \sum_{s' \sim s} P(\mathsf{last}(p_2), a, s')$. So

$$\sum_{p \sim p_3} P_\sigma(p_1, p) = \sum_{p \sim p_3} P_\sigma(p_2, p).$$

This proves the second condition on bisimulation. □

We want to show 'equivalence' between following two Markov chains:

1. $(M_\sim)_\sigma = (S_{\sim\sigma}, P_{\sim\sigma}, AP, L_{\sim\sigma})$ defined by the quotient MDP $M_\sim$ equipped with the strategy σ.
2. $(M_\sigma)_\sim = (S_{\sigma\sim}, P_{\sigma\sim}, AP, L_{\sigma\sim})$ which is the quotient of the Markov chain M_σ with respect to the bisimulation relation $\sim$.

Consider the function I between $S_{\sim\sigma}$ and $S_{\sigma\sim}$ where $I([s_0]a_0[s_1] \dots [s_n]) = [s_0 a_0 s_1 \dots s_n]$. We define the following relation $\sim_I$ on $S_{\sim\sigma} \uplus S_{\sigma\sim}$:

$$\sim_I = \bigcup_{p \in S_{\sim\sigma}} \{(p, p), (p, I(p)), (I(p), p), (I(p), I(p))$$

Lemma 3.3

$\sim_I$ is a bisimulation in the Markov chain $(M_\sim)_\sigma \uplus (M_\sigma)_\sim$.

Proof For any $p \in S_{\sim\sigma}$, $(p,p) \in \sim_I$. For any $[s_0 a_0 s_1 \dots s_n] \in S_{\sigma\sim}$, $[s_0 a_0 s_1 \dots s_n] = I([s_0]a_0[s_1]\dots[s_n])$. So, $([s_0 a_0 s_1 \dots s_n], [s_0 a_0 s_1 \dots s_n]) \in \sim_I$. Thus, $\sim_I$ is *reflexive*.

For any $p \in S_{\sim\sigma}$, we have $(p, I(p)) \in \sim_I$ and $(I(p), p) \in \sim_I$. So for any $(p,q) \in \sim_I$, $(q,p) \in \sim_I$. Thus, $\sim_I$ is *symmetric*.

Also, for any $(p, I(p)) \in \sim_I$, $(I(p), p) \in \sim_I$ and $(p,p) \in \sim_I$. So for any (p,q) and $(q,r) \in \sim_I$, $(p,r) \in \sim_I$. Thus, $\sim_I$ is *transitive*.

Thus, $\sim_I$ is an equivalence relation. Also, we have $L([s_0]a_0[s_1]\dots[s_n]) = L([s_n]) = L(s_0 a_0 s_1 \dots s_n) = L([s_0 a_0 s_1 \dots s_n])$.

Let $\rho = [s_0]a_0[s_1]\dots[s_n]$ and $\rho' = [s_0]a_0[s_1]\dots[s_n]a_{n_1}[s_{n+1}]$. Then,

$$
\begin{aligned}
P_{\sigma\sim}(I(\rho), I(\rho')) &= \sum_{s'_0 a_0 s'_1 \dots s'_n a_{n+1} s \sim s_0 a_0 s_1 \dots s_n a_{n+1} s_{n+1}} P_\sigma(s_0 a_0 s_1 \dots s_n, s'_0 a_0 s'_1 \dots s'_n a_{n+1} s) \\
&= \sum_{s \sim s_{n+1}} P_\sigma(s_0 a_0 s_1 \dots s_n, s_0 a_0 s_1 \dots s_n a_{n+1} s) \\
&= \sum_{s \sim s_{n+1}} \sigma(\rho, a_{n+1}) \cdot P(s_n, a_{n+1}, s) \\
&= \sigma(\rho, a_{n+1}) \cdot \sum_{s \sim s_{n+1}} P(s_n, a_{n+1}, s) \\
&= \sigma(\rho, a_{n+1}) \cdot P_\sim([s_n], a_{n+1}, [s_{n+1}]) \\
&= P_{\sim\sigma}(\rho, \rho')
\end{aligned}
$$

Each equivalence classes in $S_{\sim_I}$ contain only two elements ρ and $I(\rho)$ where $\rho \in S_{\sim\sigma}$. Also, $S_{\sim\sigma}$ and $S_{\sigma\sim}$ are disjoint. So from $P_{\sim\sigma}(\rho, \rho') = P_{\sigma\sim}(I(\rho), I(\rho'))$, we get that $\sim_I$ also obeys the second condition of bisimulation.

Thus, $\sim_I$ is a bisimulation relation in the Markov chain $(M_\sim)_\sigma \uplus (M_\sigma)_\sim$. □

Theorem 3.1

Let Φ be a PCTL state formula. Suppose there exists a strategy σ in $M_\sim$ such that $[s] \models \Phi$ in the Markov chain $(M_\sim)_\sigma$. Then there exists a strategy σ' in M such that $s \models \Phi$ in the Markov chain $M_{\sigma'}$.

Proof From Lemma 3.3, we get that there is a bisimulation relation $\sim_I$ such that the state

$[s]$ in $(M_\sim)_\sigma$ and the state $[s]$ in $(M_\sigma)_\sim$ are bisimilar. So $[s] \models \Phi$ in $(M_\sim)_\sigma$ implies $[s] \models \Phi$ in $(M_\sigma)_\sim$.

Hence, from Theorem 2.4, $s \models \Phi$ in M_σ, which is the Markov chain induced by the strategy $\gamma(\sigma)$ where for a path p in M, $\gamma(\sigma)(p) = \sigma([p])$. □

Example 3.3 In PAC-MAN, consider the set of states labelled with $loss$ where Pac-Man gets eaten by a ghost. Suppose from the current state s, we want to find a strategy σ such that Pac-Man stays safe for 10 steps with probability at least 0.9 following that strategy. In other words, we want to find a strategy such that in the Markov chain M_σ,

$$s \models \mathbb{P}_{\geq 0.9}(\Box^{\leq 10} \neg loss)\,.$$

We consider the quotient MDP described in Example 3.2 where two states with same position of the agents are put in same equivalence class. Suppose we find a strategy σ in the quotient MDP such that Pac-Man stays safe for 10 steps with probability at least 0.9 following that strategy. Then we can use this strategy in the original MDP by taking $\sigma(s^A)$ from any state (s^A, s^F). □

3.3 Pruning

A memoryless nondeterministic strategy can be used to *prune* an MDP to a sub-MDP. This can be useful during decision-time planning if we already know that some actions in the MDP is suboptimal and would not be part of the optimal strategy. This means that an optimal strategy will be contained in the nondeterministic strategy obtained by removing these actions.

We define the MDP pruned by a memoryless nondeterministic strategy as follows:

Definition 3.4 (Pruned MDP)

For an MDP $M = (S, A, P, R, R_T, AP, L)$ and a memoryless non-deterministic strategy σ, let the pruned MDP $M_\sigma = (S, A, P', R', R_T, AP, L)$ be the sub-MDP of M obtained by removing all action transitions that are not compatible with σ. In other words, we have,

$$P'(s, a, s') = \begin{cases} P(s, a, s') & \text{if } a \in \sigma(s) \\ \bot & \text{otherwise,} \end{cases}$$

$$R'(s,a) = \begin{cases} R(s,a) & \textit{if } a \in \sigma(s) \\ \bot & \textit{otherwise.} \end{cases}$$

Note that a nondeterministic strategy in an MDP would give a memoryless nondeterministic strategy in a finite horizon unfolding as states in the unfolding are finite paths. Thus, a nondeterministic strategy prunes the finite horizon unfolding:

Definition 3.5 (Pruned unfolding)

For an MDP $M = (S, A, P, R, R_T)$, a horizon $H \in \mathbb{N}$, a state s_0 and a nondeterministic strategy σ, let the pruned unfolding $T(M, s_0, H, \sigma)$ *be defined as a sub-MDP of $T(M, s_0, H)$ that contains exactly all paths in $\mathsf{Paths}^H_M(s_0) \cap \mathsf{Paths}^H_M(\sigma)$. It can be obtained by the following steps:*

1. *Construct the unfolding $T(M, s_0, H)$.*
2. *Prune the MDP $T(M, s_0, H)$ with σ.*

We argue that if a nondeterministic strategy contains an optimal strategy, the optimal strategy in the pruned unfolding using the nondeterministic strategy would give an optimal strategy in the original MDP.

Theorem 3.2

Suppose that there exists a strategy $\sigma' \subseteq \sigma$ such that σ' is an optimal strategy for expected total of horizon H at state s_0. Then, $\mathsf{Val}^H_M(s_0)$ equals $\mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)$. Moreover, $\mathsf{opt}^H_{T(M,s_0,H,\sigma)}(s_0) \subseteq \mathsf{opt}^H_M(s_0)$.

Proof As $\sigma' \subseteq \sigma$, σ' is a strategy that can be followed in $T(M, s_0, H, \sigma)$. So,

$$\mathsf{Val}^H_{T(M,s_0,H)}(s_0, \sigma') = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0, \sigma') \,.$$

Thus, from the definition of the optimal expected total reward,

$$\mathsf{Val}^H_{T(M,s_0,H)}(s_0, \sigma') = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0, \sigma') \leq \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)$$

Consider the strategy $\sigma^* \in \arg\max_{\sigma''} \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0, \sigma'')$. This is also a strategy in $T(M, s_0, H)$. So,

$$\mathsf{Val}^H_{T(M,s_0,H)}(s_0, \sigma^*) = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0, \sigma^*) \,.$$

Therefore, we have,

$$\mathsf{Val}^H_{T(M,s_0,H)}(s_0,\sigma') \geq \mathsf{Val}^H_{T(M,s_0,H)}(s_0,\sigma^*) = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0,\sigma^*) = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)$$

Thus $\mathsf{Val}^H_{T(M,s_0,H)}(s_0,\sigma') = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)$. Then, from Lemma 3.1, we get,

$$\mathsf{Val}^H_M(s_0) = \mathsf{Val}^H_{T(M,s_0,H)}(s_0,\sigma') = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)\,.$$

Let a be an action in $\mathsf{opt}^H_{T(M,s_0,H,\sigma)}(s_0)$. Then, there exists an optimal strategy $\sigma^* \in \arg\max_{\sigma''} \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0,\sigma'')$ so that $\sigma^*(s_0) = a$. It follows from $\mathsf{Val}^H_{T(M,s_0,H)}(s_0) = \mathsf{Val}^H_{T(M,s_0,H,\sigma)}(s_0)$ that σ^* is also an optimal strategy in $T(M,s_0,H)$. Then, by Lemma 3.1, we get, $a \in \mathsf{opt}^H_{T(M,s_0,H)}(s_0) = \mathsf{opt}^H_M(s_0)$. □

Chapter 4

Advice

In heuristic search, in every step of the planning, a large tree is constructed by sampling paths in the unfolding. We introduce the notion of *advice* which will be used to systematically inject domain-specific knowledge in heuristic searches to direct the search to the relevant part of the unfolding. Formally, an advice would denote a set of finite paths in an MDP which can be represented by a logical formula. It identifies the paths which would have an effect in the planning of the optimal strategy. This way an advice forces to simulate paths only at the relevant part of the unfolding.

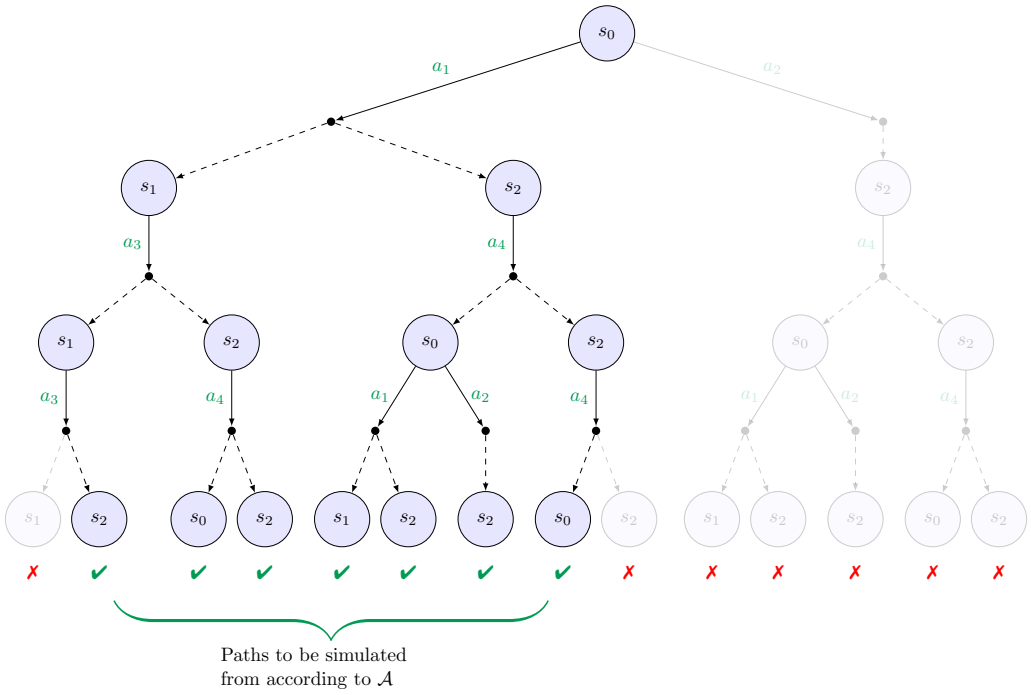

Figure 4.1: An advice forces the heuristic search to simulate from only relevant paths (marked by ✔ at the leaves) in the unfolding. A path marked by ✗ at the leaf does not satisfy the advice $\mathscr{A}$, so it will never be sampled.

Example 4.1 In the game of Pac-Man, consider the set of states labelled with $loss$ where Pac-Man gets eaten by a ghost. Since reaching such a state gives very bad reward, one advice could be to simulate among H-length paths containing no such states.

A naïve heuristic approach to find a good action in the root of the unfolding from the current would be to sample uniformly among these good paths. An advice could represent such paths. □

An advice can be represented *symbolically* by a logical formula from which the advice can be extracted on-the-fly. In this online computation, we will use formal method techniques but in a finite horizon unfolding of an abstraction of the MDP, which has much smaller state-space than the original MDP. In this chapter, we formally define advice and give examples how it can be encoded symbolically. We also discuss techniques to sample paths in an MDP based on an advice.

4.1 Symbolic advice

A *symbolic advice* $\mathscr{A}$ is a logical formula over finite paths whose truth value can be tested with an operator $\models$. Many standard notions can fit this framework. For example, a Boolean SAT formula, or an PCTL path formula can be used.

Example 4.2 In Pac-Man, consider the H-length paths containing no states labelled with $loss$ as described in Example 4.1. The set of all such paths can be written by an PCTL path formula $\varphi^H = \Box^{\leq H}(\neg loss)$. □

We denote by $\mathsf{Paths}_M^H(\mathscr{A})$ the set of paths $p \in \mathsf{Paths}_M^H$ such that $p \models \mathscr{A}$. For a path $p \in \mathsf{Paths}_M^{\leq H}$, we denote by $\mathsf{Paths}_M^H(p, \mathscr{A})$ the set of paths $p' \in \mathsf{Paths}_M^H(p)$ such that $p' \models \mathscr{A}$. In particular, for all $s \in S$, $\mathsf{Paths}_M^H(s, \mathscr{A})$ refers to the paths of length H that start from s and that satisfy $\mathscr{A}$.

Nondeterministic strategies for an advice For a symbolic advice $\mathscr{A}$ and a horizon H, we define a nondeterministic strategy $\sigma_{\mathscr{A}}^H$ and a nondeterministic strategy $\tau_{\mathscr{A}}^H$ for the environment such that for all paths p with $|p| < H$,

$$\sigma_{\mathscr{A}}^H(p) = \{a \in A \mid \exists s \in S, \exists p' \in \mathsf{Paths}_M^{H-|p|-1}(s), p \cdot as \cdot p' \models \mathscr{A}\},$$
$$\tau_{\mathscr{A}}^H(p, a) = \{s \in S \mid \exists p' \in \mathsf{Paths}_M^{H-|p|-1}(s), p \cdot as \cdot p' \models \mathscr{A}\}.$$

The strategies $\sigma_{\mathscr{A}}^H$ and $\tau_{\mathscr{A}}^H$ can be defined arbitrarily on paths p of length at least H, for example with $\sigma_{\mathscr{A}}^H(p) = A$ and $\tau_{\mathscr{A}}^H(p, a) = \mathsf{Supp}(P(\mathsf{last}(p), a))$ for all actions a. Note that by definition, $\mathsf{Paths}_M^H(s, \mathscr{A}) = \mathsf{Paths}_M^H(s, \sigma_{\mathscr{A}}^H) \cap \mathsf{Paths}_M^H(s, \tau_{\mathscr{A}}^H)$ for all states s.

Let $\top$ denote the *universal advice* satisfied by every finite path, and let $\sigma_\top$ and $\tau_\top$ be the associated nondeterministic strategies. Similarly, let $\bot$ denote the *empty advice* that is never satisfied, and let $\sigma_\bot$ and $\tau_\bot$ be the associated nondeterministic strategies. We define a class of advice that can be enforced against an adversarial environment by following a nondeterministic strategy, and that are minimal in the sense that paths that are not compatible with this strategy are not allowed.

Definition 4.1 (Enforceable advice)

A symbolic advice $\mathscr{A}$ is called an enforceable advice *from a state s_0 and for a horizon H if there exists a nondeterministic strategy σ such that $\mathsf{Paths}_M^H(s_0, \sigma) = \mathsf{Paths}_M^H(s_0, \mathscr{A})$, and such that $\sigma(p) \neq \emptyset$ for all paths $p \in \mathsf{Paths}_M^{\leq H-1}(s_0, \sigma)$.*

Note that Definition 4.1 ensures that paths that follow σ can always be extended into longer paths that follow σ. This is a reasonable assumption to make for a nondeterministic strategy meant to enforce a property. In particular, s_0 is a path of length 0 in $\mathsf{Paths}_M^0(s_0, \sigma)$, so that $\sigma(s_0) \neq \emptyset$ and so that by induction $\mathsf{Paths}_M^i(s_0, \sigma) \neq \emptyset$ for all $i \in [0, H]$.

Lemma 4.1

Let $\mathscr{A}$ be an enforceable advice from s_0 with horizon H. It holds that $\mathsf{Paths}_M^H(s_0, \sigma_\mathscr{A}^H) = \mathsf{Paths}_M^H(s_0, \mathscr{A})$.
Moreover, for all paths $p \in \mathsf{Paths}_M^{\leq H-1}(s_0)$ and all actions a, either $\tau_\mathscr{A}^H(p, a) = \tau_\top(p, a)$ or $\tau_\mathscr{A}^H(p, a) = \tau_\bot(p, a)$.
Finally, for all paths p in $\mathsf{Paths}_M^{\leq H-1}(s_0, \sigma_\mathscr{A}^H)$, $\sigma_\mathscr{A}^H(p) \neq \emptyset$ and $a \in \sigma_\mathscr{A}^H(p)$ if and only if $\tau_\mathscr{A}^H(p, a) = \tau_\top(p, a)$.

Proof We have $\mathsf{Paths}_M^H(s_0, \mathscr{A}) = \mathsf{Paths}_M^H(s_0, \sigma_\mathscr{A}^H) \cap \mathsf{Paths}_M^H(s_0, \tau_\mathscr{A}^H)$ for any advice $\mathscr{A}$. Let us prove that $\mathsf{Paths}_M^H(s_0, \sigma_\mathscr{A}^H) \subseteq \mathsf{Paths}_M^H(s_0, \mathscr{A})$ for an enforceable advice $\mathscr{A}$ of associated strategy σ. Let $p = p' \cdot as$ be a path in $\mathsf{Paths}_M^H(s_0, \sigma_\mathscr{A}^H)$. By definition of $\sigma_\mathscr{A}^H$, there exists $s' \in S$ such that $p' \cdot as' \models \mathscr{A}$, so that $p' \cdot as' \in \mathsf{Paths}_M^H(s_0, \mathscr{A}) = \mathsf{Paths}_M^H(s_0, \sigma)$. Since $s \in \mathsf{Supp}(P(\mathsf{last}(p'), a))$, $p = p' \cdot as$ must also belong to $\mathsf{Paths}_M^H(s_0, \sigma) = \mathsf{Paths}_M^H(s_0, \mathscr{A})$.

Consider a path p and an action a such that $|p| < H$. We want to prove that either all stochastic transitions starting from (p, a) are allowed by $\mathscr{A}$, or none of them are. By contradiction, let us assume that there exists s_1 and s_2 in $\mathsf{Supp}(P(\mathsf{last}(p), a))$ such that for all $p_1' \in \mathsf{Paths}_M^{H-|p|-1}(s_1)$, $p \cdot as_1 \cdot p_1' \not\models \mathscr{A}$, and such that there exists $p_2' \in \mathsf{Paths}_M^{H-|p|-1}(s_2)$ with $p \cdot as_2 \cdot p_2' \models \mathscr{A}$. From $p \cdot as_2 \cdot p_2' \models \mathscr{A}$, we obtain $p \cdot as_2 \cdot p_2' \in \mathsf{Paths}_M^H(\sigma)$, so that $p \cdot as_2$ is a path that follows σ. Then, $p \cdot as_1$ is a path that follows σ as well. It follows that

$\sigma(p \cdot as_1) \neq \emptyset$, and $p \cdot as_1$ can be extended in to a path $p \cdot as_1 p'_3 \in \mathsf{Paths}^H_M(\sigma)$. This implies the contradiction $p \cdot as_1 p'_3 \models \mathscr{A}$.

Finally, consider a path p in $\mathsf{Paths}^{\leq H-1}_M(s_0, \sigma^H_{\mathscr{A}})$. By the definitions of $\sigma^H_{\mathscr{A}}$ and $\tau^H_{\mathscr{A}}$, $a \in \sigma^H_{\mathscr{A}}(p)$ if and only if $\tau^H_{\mathscr{A}}(p, a) \neq \emptyset$, so that $\tau^H_{\mathscr{A}}(p, a) = \tau_\top(p, a)$. Then, let us write $p = p' \cdot as$. From $p \in \mathsf{Paths}^{\leq H-1}_M(s_0, \sigma^H_{\mathscr{A}})$ we get $a \in \sigma^H_{\mathscr{A}}(p')$, so that $s \in \tau^H_{\mathscr{A}}(p', a)$, and therefore $\sigma^H_{\mathscr{A}}(p) \neq \emptyset$. □

An enforceable advice is encoding a notion of guarantee, as $\sigma^H_{\mathscr{A}}$ is a winning strategy for the reachability objective on $T(M, s_0, H)$ defined by the set $\mathsf{Paths}^H_M(\mathscr{A})$.

We say that the enforceable advice $\mathscr{A}'$ is *extracted* from a symbolic advice $\mathscr{A}$ for a horizon H and a state s_0 if $\mathscr{A}'$ is the greatest part of $\mathscr{A}$ that can be guaranteed for the horizon H starting from s_0, *i.e.* if $\mathsf{Paths}^H_M(s_0, \mathscr{A}')$ is the greatest subset of $\mathsf{Paths}^H_M(s_0, \mathscr{A})$ such that $\sigma^H_{\mathscr{A}'}$ is a winning strategy for the reachability objective $\mathsf{Paths}^H_M(s_0, \mathscr{A})$ on $T(M, s_0, H)$. This greatest subset always exists because if $\mathscr{A}'_1$ and $\mathscr{A}'_2$ are strongly an enforceable advice in $\mathscr{A}$, then $\mathscr{A}'_1 \cup \mathscr{A}'_2$ is strongly enforceable by union of the nondeterministic strategies associated with $\mathscr{A}'_1$ and $\mathscr{A}'_2$. However, this greatest subset may be empty, and as $\bot$ is not an enforceable advice we say that in this case $\mathscr{A}$ cannot be enforced from s_0 with horizon H.

Example 4.3 Note that the advice in Example 4.1 is not enforceable, as there may be situations (p, a) where some stochastic transitions lead to bad states and some do not. For example, in Figure 4.2, the path of length $H = 1$ that corresponds to Pac-Man going left and the red ghost going up is allowed by the advice $\mathscr{A}$, but not by any safe strategy for Pac-Man as there is a possibility of losing by playing left.

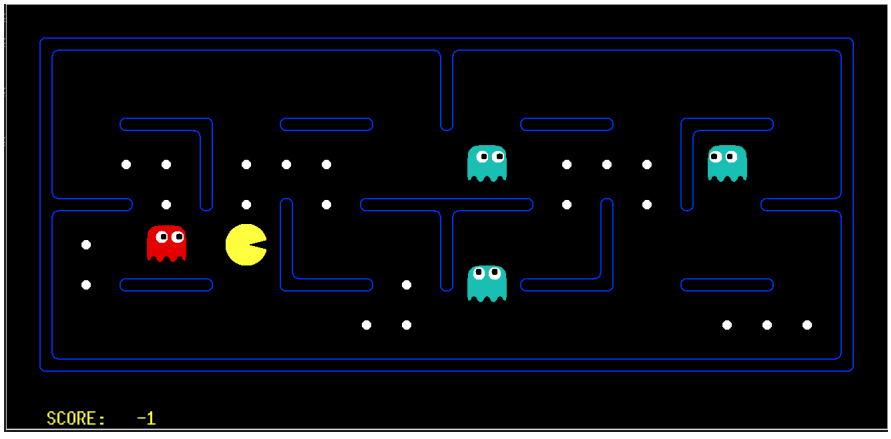

Figure 4.2: PAC-MAN with 4 ghosts in a 9×21 grid. The figure has been generated using the code from [DK].

But we can get an enforceable advice $\mathscr{A}$ by finding a strategy that ensures whatever actions the ghosts take, Pac-Man will not reach the states labelled with $loss$. We consider the safety game in the underlying game graph of the finite horizon unfolding of the MDP. From Theorem 2.1, we have a most general stategy to stay safe in this game. This nondeterministic strategy will enforce an advice $\mathscr{A}$. □

Pruning by an advice Note that for an enforceable advice $\mathscr{A}$, $\sigma_{\mathscr{A}}$ is a memoryless non-deterministic strategy in $T(M, s, H)$. This prunes the finite unfolding of an MDP from any state $s \in S$. We call this pruned unfolding $T(M, s, H, \mathscr{A})$.

Optimality assumption An enforceable advice $\mathscr{A}$ satisfies the *optimality assumption* for horizon H if for all $s \in S$, there exists a strategy $\sigma_s \subseteq \sigma_{\mathscr{A}}$ such that σ_s is an optimal strategy for expected total of horizon H at state s.

We argue that if an advice satisfies the optimality assumption, the optimal strategy in the pruned unfolding by this strategy gives an optimal strategy in the original MDP.

Theorem 4.1

Let $\mathscr{A}$ be a enforceable advice that satisfies the optimality assumption. Then, for an $s_0 \in S$, $\mathsf{Val}^H_M(s_0)$ equals $\mathsf{Val}^H_{T(M,s_0,H,\mathscr{A})}(s_0)$. Moreover, $\mathsf{opt}^H_{T(M,s_0,H,\mathscr{A})}(s_0) \subseteq \mathsf{opt}^H_M(s_0)$.

Proof By the optimality assumption, there exists a strategy $\sigma_{s_0} \subseteq \sigma_{\mathscr{A}}$ such that σ_{s_0} is an optimal strategy for expected total of horizon H at state s_0. Also, $T(M, S_0, H, \mathscr{A})$ is the pruned unfolding constructed from the state s_0 by the memoryless non-deterministic strategy $\sigma_{\mathscr{A}}$. So from Lemma 3.2, we get that $\mathsf{Val}^H_M(s_0) = \mathsf{Val}^H_{T(M,s_0,H,\mathscr{A})}(s_0)$ and $\mathsf{opt}^H_{T(M,s_0,H,\mathscr{A})}(s_0) \subseteq \mathsf{opt}^H_M(s_0)$. □

4.2 Sampling according to a symbolic advice

Given a symbolic advice $\mathscr{A}$ as a logical formula over paths of length H and a probability distribution w in $\mathcal{D}(\mathsf{Paths}^H_M)$, our goal is to sample paths of MDP M that satisfy $\mathscr{A}$ with

respect to w. This means the probability of a path p being sampled should be equal to

$$\frac{w(p)}{\sum_{p':p'\models\mathscr{A}} w(p')}\,.$$

We show two approaches for that:

Reject-based approach In this approach described in Algorithm 3, we keep on sampling paths of length H according to w until we found a path that is in support of $\mathscr{A}$.

Algorithm 3 Reject-based sampling approach

Input: A probability distribution $w \in \mathcal{D}(\mathsf{Paths}^H_M)$, a symbolic advice $\mathscr{A}$.
Output: a path p

1: **while** $p \not\models \mathscr{A}$ **do**
2: sample a path p of length H according to the probability distribution w
3: **end while**
4: **return** p

In this way, probability of sampling a path p at a single round is $w(p)$ and probability of sampling a path not satisfying $\mathscr{A}$ is $1 - \sum_{p':p'\models\mathscr{A}} w(p')$. So probability of sampling p which satisfies $\mathscr{A}$ at i^{th} round of this process is

$$w(p) \cdot \left[1 - \sum_{p':p'\models\mathscr{A}} w(p')\right]^{i-1}\,.$$

So the path p will be sampled with probability

$$\sum_{i=1}^{\infty} w(p) \times \left[1 - \sum_{p':p'\models\mathscr{A}} w(p')\right]^{i-1} = \frac{w(p)}{\sum_{p':p'\models\mathscr{A}} w(p')}\,.$$

SAT solver-based approach From the symbolic advice and a symbolic model of the MDP M, one can often encode $\mathscr{A}$ as a Boolean formula. For example, if $\mathscr{A}$ is described in Linear Temporal Logic, and a symbolic model of the MDP M is available, one can encode it as a Boolean formula of size linear in the size of the LTL formula and H [Bie+06].

Example 4.4 As in Example 4.1, given a horizon H and a state $s \in S$, we want to make sure that a state labelled with $loss$ is not reached within H steps. We can have Boolean variables to encode the states and the actions in the next H steps. For $i \in [0, H]$, let $\overline{s_i}$ be the Boolean variables that encodes the state at step i. For $i \in [0, H-1]$, let $\overline{a_i}$ be the Boolean variables that encodes the action at step i.

Then we can construct a SAT formula $\Phi(\overline{s_0}, \overline{a_0}, \ldots, \overline{s_H})$ with clauses that encode the game rules and clauses that enforce the advice. The former clauses are implications such as "if Pac-Man is in position (x, y) and plays the action a, then it must be in position (x', y') at the next game step", while the latter clauses state that the position of Pac-Man should never be equal to the position of one of the Ghosts. □

Let ω denote a weight function over Boolean assignments that matches w. This reduces our problem to the weighted sampling of satisfying assignments in a Boolean formula. An exact solver for this problem may not be efficient, but one can use the techniques of [Cha+14] to perform approximate sampling in polynomial time:

Theorem 4.2 ([Cha+14])

Given a CNF formula ψ, a tolerance $\epsilon > 0$ and a weight function ω, we can construct a probabilistic algorithm which outputs a satisfying assignment z such that for all y that satisfies $\mathscr{A}$:

$$\frac{\omega(y)}{(1+\epsilon)\sum_{x \models \psi} \omega(x)} \leq Pr[z = y] \leq \frac{(1+\epsilon)\omega(y)}{\sum_{x \models \psi} \omega(x)}$$

The above algorithm occasionally 'fails' (outputs no assignment even though there are satisfying assignments) but its failure probability can be bounded by any given δ. Given an oracle for SAT, the above algorithm runs in time polynomial in $\ln\left(\frac{1}{\delta}\right)$, $|\psi|$, $\frac{1}{\epsilon}$ and r where r is the ratio between highest and lowest weight according to ω.

In particular, this algorithm uses ω as a black-box, and thus does not require pre-computing the probabilities of all paths satisfying $\mathscr{A}$. In our particular application, the value r can be bounded by $\left(\frac{p_{\max}|A|}{p_{\min}}\right)^H$ where $p_{\min}$ and $p_{\max}$ are the smallest and greatest probabilities for stochastic transitions in M.

4.3 On-the-fly computation of an enforceable advice

A direct encoding of an enforceable advice may prove impractically large. We argue for an on-the-fly computation of $\sigma^H_{\mathscr{A}}$ instead, in the particular case where the enforceable advice is extracted from a symbolic advice $\mathscr{A}$ with respect to the initial state s_0 and with horizon H.

4.3.1 Computation using QBF solvers

Lemma 4.2

Let $\mathscr{A}'$ be an enforceable advice extracted from $\mathscr{A}$ for horizon H. Consider a node p at depth i in $T(M, s_0, H, \mathscr{A}')$, for all $a_0 \in A$, $a_0 \in \sigma^H_{\mathscr{A}'}(p)$ if and only if

$$\forall s_1 \exists a_1 \forall s_2 \dots \forall s_{H-i+1},\ p \cdot a_0 s_1 a_1 s_2 \dots s_{H-i+1} \models \mathscr{A}\ ,$$

where actions are quantified over A and every s_k is quantified over $\mathsf{Supp}(P(s_{k-1}, a_{k-1}))$.

Proof The proof is a reverse induction on the depth i of p. For the initialisation step with $i = H$, let us prove that $\forall s_1,\ p \cdot a_0 s_1 \models \mathscr{A}$ if and only if $a_0 \in \sigma^H_{\mathscr{A}'}(p)$. On the one hand, if $\mathscr{A}$ is guaranteed by playing a_0 from p, then a_0 must be allowed by the greatest enforceable subset of $\mathscr{A}$. On the other hand, $a_0 \in \sigma^H_{\mathscr{A}'}(p)$ implies $\forall s_1,\ p \cdot a_0 s_1 \models \mathscr{A}'$ as $\mathscr{A}'$ is enforceable, and finally $\mathscr{A}' \Rightarrow \mathscr{A}$. We now assume the property holds for $1 \leq i \leq H$, and prove it for $i - 1$. If $a_0 \in \sigma^H_{\mathscr{A}'}(p)$, then for all s_1 we have $s_1 \in \tau^H_{\mathscr{A}'}(p, a_0)$, so that there exists a_1 with $a_1 \in \sigma^H_{\mathscr{A}'}(p \cdot a_0 s_1)$. As $p \cdot a_0 s_1$ is at depth i we can conclude that $\forall s_1 \exists a_1 \forall s_2 \dots \forall s_{H-i+1},\ p \cdot a_0 s_1 a_1 s_2 \dots s_{H-i+1} \models \mathscr{A}$ by assumption. For the converse direction, the alternation of quantifiers states that $\mathscr{A}$ can be guaranteed from p by some deterministic strategy that starts by playing a_0, and therefore a_0 must be allowed by the enforceable advice extracted from $\mathscr{A}$. □

Therefore, given a Boolean formula encoding $\mathscr{A}$, one can use a Quantified Boolean Formula (QBF) solver to compute $\sigma^H_{\mathscr{A}'}$, the enforceable advice extracted from $\mathscr{A}$: this computation can be used whenever MCTS performs an action selection step under the advice $\mathscr{A}'$, as described in Section 5.2.

The performance of this approach will crucially depend on the number of alternating quantifiers, and in practice one may limit themselves to a smaller depth $h < H - i$ in this step, so that safety is only guaranteed for the next h steps.

Some properties can be inductively guaranteed, so that satisfying the QBF formula of Lemma 4.2 with a depth $H - i = 1$ is enough to guarantee the property globally. For example, if there always exists an action leading to states that are not bad, it is enough to check for safety locally with a depth of 1. This is the case in Pac-Man for a deadlock-free layout when there is only one ghost.

4.3.2 Computation using other formal method techniques

The QBF-based technique may often be too restrictive. In this case with model checking tools like Storm, we can take a more qualitative approach as follows:

Example 4.5 The set of all H-length paths where Pac-Man does not get eaten by a ghost can be written by an PCTL path formula $\varphi^H = \Box^{\leq H}(\neg loss)$. From the state s, for each action a, we calculate the minimum probability $\eta_{\varphi^H}(s,a)$ to satisfy φ^H when the action a is taken from the state s. Formally:

$$\eta_{\varphi^H}(s,a) = \max_{\sigma:\sigma(s)=a} \mathbb{P}_\sigma(s \models \varphi^H)\,.$$

Then, we can construct a nondeterministic strategy by selecting the action maximizing $\eta_{\varphi^H}(s,a)$. This strategy will enforce an advice. □

Note that this approach is similar to *probabilistic shielding* [Jan+14] computed to restrict unsafe action during reinforcement learning. But in our case, we perform computation on-the-fly based on the current position alone instead of constructing the entire MDP.

4.3.3 Computation using a function

We can compute a strategy from a function. Intuitively, this function could act as a 'score' $f(s,a)$ telling us how good an action a from s is. This function could be a heuristic function calculated from the description of an MDP, a function based on model-checking methods. For example, in Example 4.5, the function η_{φ^H} acts as such a function. Another interesting approach could be training a neural network to imitate the function f and use it as a *neural advice*.

For a function $f : S \times A \to \mathbb{R}$, we can create a strategy where from the state s, we choose the action a which maximizes the value $f(s,a)$. This is a nondeterministic strategy as there can be more than one maximizing action.

A more general approach could be using a threshold $\delta \in (0,1]$. In that case, instead of selecting an action from $\arg\max_a f(s,a)$, we can select from the set $\{a' \mid f(s,a') \geq \delta \cdot \max_a f(s,a)\}$.

Chapter 5

Monte Carlo tree search with advice

In Chapter 4, we discussed how we augment the conventional heuristic search algorithms with advice. In this chapter, we will discuss how the notion of advice fits in Monte Carlo tree search. The main idea is as follows: we have two possibly different advice, one is used during the selection phase to remove action non-compatible with it, and another is used during simulation phase to simulate paths according to it.

We will identify necessary conditions which ensure that the guarantees enjoyed by MCTS is still maintained after this augmentation. To do so we will argue that the results of MCTS are still maintained if we simulate restricted by any simulation advice ψ. Then, we will argue that using an enforceable selection advice φ during MCTS is equivalent to running MCTS in the unfolding pruned by the advice φ. Thus, from Theorem 4.1, we prove that we get the same optimal action as given by the MCTS algorithm without advice.

5.1 Generalized Monte Carlo tree search

We take a *more general approach* to the simulation phase of Monte Carlo tree search, defined by a finite domain $I \subseteq [0, 1]$ and a function $f : \mathsf{Paths}_M^{\leq H} \to \mathcal{D}(I)$ that maps every path p to a probability distribution on I. In this approach (see Figure 5.1), instead of sampling, the simulation phase simply draws a value $\mathsf{value}_i(p)$ at random according to the distribution $f(p)$, and sets $\mathsf{count}_i(p) = 1$.

To this end, every node p of the search tree is considered to be an instance of a bandit problem with non-stationary distributions. Every time a node is selected, a step is processed in the corresponding bandit problem.

Let $(\mathcal{I}_i(p))_{i\geq 1}$ be a sequence of iteration numbers for the MCTS algorithm that describes when the node p is selected. In other words, the simulation phase was used on p at iteration number $\mathcal{I}_1(p)$, and the i-th selection of node p happened on the iteration number $\mathcal{I}_i(p)$. We define sequences $(\mathcal{I}_i(p, a))_{i\in\mathbb{N}}$ similarly for node-action pairs.

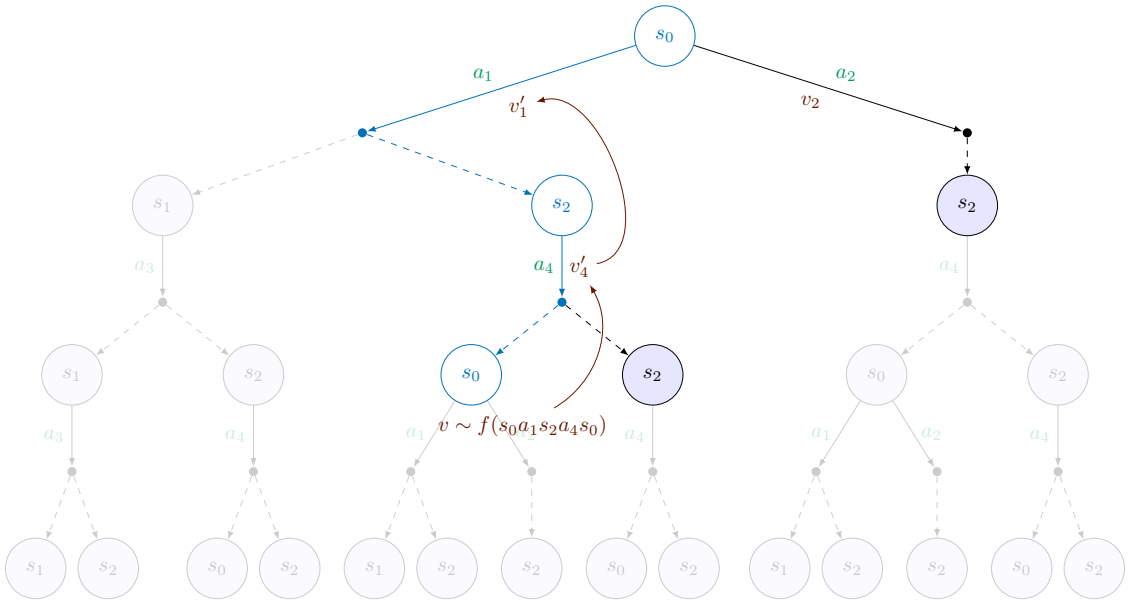

Figure 5.1: Instead of simulating, the value at node p is drawn according to a distribultion $f(p)$.

For all paths p and actions a, a payoff sequence $(x_{a,t})_{t\geq 1}$ of associated random variables $(X_{a,t})_{t\geq 1}$ is defined by $x_{a,t} = \mathsf{reward}_{\mathcal{I}_t(p,a)}(p)$. Note that in the selection phase at iteration number $\mathcal{I}_t(p,a)$, p must have been selected and must be a prefix of length k of the leaf node p' reached in this iteration, so that $\mathsf{reward}_{\mathcal{I}_t(p,a)}(p)$ is computed as $\mathsf{reward}_{\mathcal{I}_t(p,a)}(p'_{|k})$ in the backpropagation phase. According to the notations of Section 2.5, for all $t \geq 1$ we have $\mathsf{count}_{\mathcal{I}_t(p)}(p) = t$, $\mathsf{count}_{\mathcal{I}_t(p)}(p,a) = t_a$ and $\mathsf{value}_{\mathcal{I}_t(p)}(p,a) = \overline{x}_{a,t_a}$.

After n iterations of MCTS, we have:

$$\mathsf{total}_n(p) = \sum_{i|\mathcal{I}_i(p)\leq n} \mathsf{reward}_{\mathcal{I}_i(p)}(p)$$

$$\mathsf{total}(p,a) = \sum_{i|\mathcal{I}_i(p,a)\leq n} \mathsf{reward}_{\mathcal{I}_i(p,a)}(p,a)$$

From the structure of the MCTS algorithm, we can observe that, for all nodes p in the search tree, after n iterations, we have:

$$\mathsf{total}_n(p) = \mathsf{reward}_{\mathcal{I}_1(p)}(p) + \sum_{a\in A} \mathsf{total}_n(p,a)$$

$$\mathsf{total}_n(p,a) = \sum_{s\in \mathsf{Supp}(P(\mathsf{last}(p),a)} \mathsf{total}_n(p\cdot as) + R(\mathsf{last}(p),a)\cdot \mathsf{count}_n(p,a)$$

$$\mathsf{value}_n(p) = \frac{\mathsf{total}_n(p)}{\mathsf{count}_n(p)}$$

$$\mathsf{count}_n(p) = 1 + \sum_a \mathsf{count}_n(p, a)$$
$$\mathsf{count}_n(p, a) = \sum_s \mathsf{count}_n(p \cdot as)$$

We need to show that our payoff sequences $(x_{a,t})_{t\geq 1}$, still satisfy the drift conditions of Definition 2.13. We argue that this is true for all simulation phases defined by any I and f:

Lemma 5.1

For any MDP M, horizon H and state s_0, the sequences $(X_{a,t})_{t\geq 1}$ satisfy the drift conditions of Definition 2.13.

Proof In the following proof we will abuse notations slightly and conflate the variables and counters used in MCTS with their associated random variables, *e.g.* we write $\mathbb{E}[\mathsf{value}_n(s_0)]$ instead of $\mathbb{E}[V_n(s_0)]$ with $V_n(s_0)$ a random variable that represents the value $\mathsf{value}_n(s_0)$. We need to show that the following conditions hold:

1. $\lim_{\mathsf{count}_n(p)\to\infty} \mathbb{E}[\mathsf{value}_n(p, a)]$ exists for all a.
2. There exists a constant $C_p > 0$ such that for $\mathsf{count}_n(p, a)$ big enough and any $\delta > 0$, $\Delta_{\mathsf{count}_n(p,a)}(\delta) = C_p\sqrt{\mathsf{count}_n(p, a)\ln(1/\delta)}$, the following bounds hold:

$$\mathbb{P}\Big[\mathsf{total}_n(p, a) \geq \mathbb{E}[\mathsf{total}_n(p, a)] + \Delta_{\mathsf{count}_n(p,a)}(\delta)\Big] \leq \delta$$
$$\mathbb{P}\Big[\mathsf{total}_n(p, a) \leq \mathbb{E}[\mathsf{total}_n(p, a)] - \Delta_{\mathsf{count}_n(p,a)}(\delta)\Big] \leq \delta$$

We show it by induction on $H - |p|$. For a node at depth $H - 1$ of the tree, the sequences follow stationary distribution according to the probability distribution in the MDP. In the inductive step, we have, for a node p at depth k, all actions a_i and states s_j, that the sequences at node $p \cdot a_i s_j$ satisfy the drift conditions. We need to show that these conditions are also maintained one step above the tree, i.e., and the sequences at node p satisfy the drift conditions.

Base case of induction for Condition 1: For $|p| = H - 1$: $\mathsf{total}_n(p, a)$ is sum of $\mathsf{count}_n(p, a)$ independent variables

$$\mathsf{total}_n(p, a) = \sum_{i|I_i(p,a)\leq n} \mathsf{reward}_{I_i(p,a)}(p, a)$$

which follows same stationary distribution:

$$\mathsf{reward}_i(p,a) = R(\mathsf{last}(p),a) + R_T(s) \text{ with probability } P(\mathsf{last}(p),a,s).$$

Therefore we have,

$$\begin{aligned}
\mathbb{E}[\mathsf{value}_n(p,a)] &= \mathbb{E}\left[\frac{\mathsf{total}_n(p,a)}{\mathsf{count}_n(p,a)}\right] \\
&= \mathbb{E}\left[\frac{1}{\mathsf{count}_n(p,a)} \sum_{i|I_i(p,a)\le n} \mathsf{reward}_{I_i(p,a)}(p,a)\right] \\
&= \mathbb{E}\left[\frac{\mathsf{count}_n(p,a)}{\mathsf{count}_n(p,a)}\right] \left(\sum_s (R(\mathsf{last}(p),a) + R_T(s)) \cdot P(\mathsf{last}(p),a,s)\right) \\
&= \sum_s (R(\mathsf{last}(p),a) + R_T(s)) \cdot P(\mathsf{last}(p),a,s) \\
&= \sum_s R_T(s) P(\mathsf{last}(p),a,s) + R(\mathsf{last}(p),a)
\end{aligned}$$

So $\lim_{\mathsf{count}_n(p)\to\infty} \mathbb{E}[\mathsf{value}_n(p,a)]$ exists.

Base case of induction for Condition 2: From Theorem 2.2, for any $\delta > 0$,

$$\mathbb{P}\left[\sum_{i|I_i(p,a)\le n} \mathsf{reward}_{I_i(p,a)}(p,a) \ge \mathbb{E}\left[\sum_{i|I_i(p,a)\le n} \mathsf{reward}_{I_i(p,a)}(p,a)\right] + \sqrt{\frac{\mathsf{count}_n(p,a)}{2} \ln\frac{1}{\delta}}\right] \le \delta\,,$$

$$\mathbb{P}\left[\sum_{i|I_i(p,a)\le n} \mathsf{reward}_{I_i(p,a)}(p,a) \le \mathbb{E}\left[\sum_{i|I_i(p,a)\le n} \mathsf{reward}_{I_i(p,a)}(p,a)\right] - \sqrt{\frac{\mathsf{count}_n(p,a)}{2} \ln\frac{1}{\delta}}\right] \le \delta\,.$$

Which means,

$$\mathbb{P}\left[\mathsf{total}_n(p,a) \ge \mathbb{E}[\mathsf{total}_n(p,a)] + \sqrt{\frac{\mathsf{count}_n(p,a)}{2} \ln\frac{1}{\delta}}\right] \le \delta\,,$$

$$\mathbb{P}\left[\mathsf{total}_n(p,a) \le \mathbb{E}[\mathsf{total}_n(p,a)] - \sqrt{\frac{\mathsf{count}_n(p,a)}{2} \ln\frac{1}{\delta}}\right] \le \delta\,.$$

Therefore, condition 2. also holds with $C_p = \frac{1}{\sqrt{2}}$.

Inductive step for Condition 1: Assume that the drifting conditions are true for all $p \cdot as$. Then, from Theorem 2.9 we get:

$$\left| \mathbb{E}\left[\frac{\sum_{a'} \mathsf{total}_n(pas, a')}{\sum_{a'} \mathsf{count}_n(pas, a')} \right] - \lim_{\mathsf{count}_n(p \cdot as) \to \infty} \mathbb{E}\left[\mathsf{value}_n(p \cdot as, a^*) \right] \right|$$
$$\leq \left| \mathbb{E}\left[\mathsf{value}_n(p \cdot as, a^*) \right] - \lim_{\mathsf{count}_n(p \cdot as) \to \infty} \mathbb{E}\left[\mathsf{value}_n(p \cdot as, a^*) \right] \right| + \mathcal{O}\left(\frac{\ln(\mathsf{count}_n(p \cdot as) - 1)}{\mathsf{count}_n(p \cdot as) - 1} \right),$$

where a^* is the optimal action from $p \cdot as$. Now,

$$\begin{aligned}
\lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}[\mathsf{value}_n(p \cdot as)] &= \lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}\left[\frac{\mathsf{total}_n(p \cdot as)}{\mathsf{count}_n(p \cdot as)} \right] \\
&= \lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}\left[\frac{\mathsf{total}_n(p \cdot as) - \mathsf{reward}_{\mathcal{I}_1(p)}(p \cdot as)}{\mathsf{count}_n(p \cdot as) - 1} \right] \\
&= \lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}\left[\frac{\sum_{a'} \mathsf{total}_n(p \cdot as, a')}{\sum_{a'} \mathsf{count}_n(p \cdot as, a')} \right].
\end{aligned}$$

Let $\lim_{\mathsf{count}_n(p \cdot as) \to \infty} \mathbb{E}[\mathsf{value}_n(p \cdot as, a^*)]$ be denoted by $\mu_{p \cdot as}$ (we know that this limit exists from the induction hypothesis). From Theorem 2.10, $\mathsf{count}_n(p, a) \to \infty$ for all a when $\mathsf{count}_n(p) \to \infty$. And as for all states s, state s is chosen according to distribution $P(p, a, s)$, $\mathsf{count}_n(p \cdot as) \to \infty$ with probability 1 when $\mathsf{count}_n(p, a) \to \infty$. Then,

$$\begin{aligned}
\lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}[\mathsf{value}_n(p \cdot a)] &= \lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}\left[\sum_s \mathsf{value}_n(p \cdot as) \frac{\mathsf{count}_n(p \cdot as)}{\mathsf{count}_n(p, a)} + R(\mathsf{last}(p), a) \right] \\
&= R(\mathsf{last}(p), a) + \sum_s \mu_{p \cdot as} \cdot P(\mathsf{last}(p), a, s).
\end{aligned}$$

So $\lim_{\mathsf{count}_n(p) \to \infty} \mathbb{E}[\mathsf{value}_n(p \cdot a)]$ exists.

Inductive step for Condition 2: From Theorem 2.11, when $\mathsf{count}_n(p \cdot as)$ is big enough, for all $\delta > 0$, we have:

$$\mathbb{P}\Big[\sum_{a'} \mathsf{total}_n(pas, a') \geq \mathbb{E}[\sum_{a'} \mathsf{total}_n(pas, a')] + \Delta_1^s(\delta) \Big] \leq \frac{\delta}{2|S|} \tag{5.1}$$

where $\Delta_1^s(\delta) = 9\left(\sqrt{\mathsf{count}_n(p \cdot as) \ln\left(\frac{4|S|}{\delta} \right)} \right)$.

Also, the random variable associated to $\mathsf{reward}_{\mathcal{I}_1(p \cdot as)}(p \cdot as)$ follows a fixed stationary

distribution $f(p)$ in $[0,1]$. So from the Theorem 2.2,

$$\mathbb{P}\Big[\mathsf{reward}_{\mathcal{I}_1(p\cdot as)}(p\cdot as) \geq \mathbb{E}[\mathsf{reward}_{\mathcal{I}_1(p\cdot as)}(p\cdot as)] + \Delta_2(\delta)\Big] \leq \frac{\delta}{2|S|} \tag{5.2}$$

where $\Delta_2(\delta) = \frac{1}{\sqrt{2}}\left(\sqrt{\ln\left(\frac{2|S|}{\delta}\right)}\right)$.

We will use the fact that for n random variables $\{A_i\}_{i\leq n}$ and n random variables $\{B_i\}_{i\leq n}$,

$$\mathbb{P}\left[\sum_i A_i \geq \sum_i B_i\right] \leq \sum_i \mathbb{P}\left[A_i \geq B_i\right] .$$

Note that $\mathsf{total}_n(p\cdot as) = \sum_{a'} \mathsf{total}_n(pas,a') + \mathsf{reward}_{\mathcal{I}_1(p\cdot as)}(p\cdot as)$. From Equation 5.1 and Equation 5.2, we get:

$$\begin{aligned}
&\mathbb{P}\Big[\mathsf{total}_n(p\cdot as) \geq \mathbb{E}[\mathsf{total}_n(p\cdot as)] + \Delta_1^s(\delta) + \Delta_2(\delta)\Big] \\
&\leq \mathbb{P}\Big[\sum_{a'} \mathsf{total}_n(pas,a') \geq \mathbb{E}[\sum_{a'} \mathsf{total}_n(pas,a')] + \Delta_1^s(\delta)\Big] \\
&\qquad + \mathbb{P}\Big[\mathsf{reward}_{\mathcal{I}_1(p\cdot as)}(p\cdot as) \geq \mathbb{E}[\mathsf{reward}_{\mathcal{I}_1(p\cdot as)}(p\cdot as)] + \Delta_2(\delta)\Big] \\
&\leq \frac{\delta}{|S|} .
\end{aligned} \tag{5.3}$$

Recall that, $\mathsf{total}_n(p,a) = \sum_{s\in\mathsf{Supp}(P(\mathsf{last}(p),a)} \mathsf{total}_n(p\cdot as) + R(\mathsf{last}(p),a)\cdot\mathsf{count}_n(p,a)$.

Also, $\mathbb{E}[R(\mathsf{last}(p),a)\cdot\mathsf{count}_n(p,a)] = R(\mathsf{last}(p),a)\cdot\mathsf{count}_n(p,a)$ as $R(\mathsf{last}(p),a)$ and $\mathsf{count}_n(p,a)$ are constants.

Then we have, from 5.3:

$$\begin{aligned}
&\mathbb{P}\left[\mathsf{total}_n(p,a) \geq \mathbb{E}[\mathsf{total}_n(p,a)] + \sum_s(\Delta_1^s(\delta) + \Delta_2(\delta))\right] \\
&\leq \sum_s \mathbb{P}\Big[\mathsf{total}_n(p\cdot as) \geq \mathbb{E}[\mathsf{total}_n(p\cdot as)] + (\Delta_1^s(\delta) + \Delta_2(\delta))\Big] \\
&\qquad + \mathbb{P}\left[R(\mathsf{last}(p),a)\cdot\mathsf{count}_n(p,a) \geq \mathbb{E}[R(\mathsf{last}(p),a)\cdot\mathsf{count}_n(p,a)] + \Delta_2(\delta)\right] \\
&= \sum_s \mathbb{P}\Big[\mathsf{total}_n(p\cdot as) \geq \mathbb{E}[\mathsf{total}_n(p\cdot as)] + (\Delta_1^s(\delta) + \Delta_2(\delta))\Big] + 0 \\
&\leq \delta .
\end{aligned} \tag{5.4}$$

Similarly, when $\mathsf{count}_n(p \cdot as)$ is big enough, for all $\delta > 0$ it holds that

$$\mathbb{P}\Big[\mathsf{total}_n(p,a) \leq \mathbb{E}[\mathsf{total}_n(p,a)] - \sum_s \left(\Delta_1^s(\delta) + \Delta_2(\delta)\right)\Big] \leq \delta\,. \tag{5.5}$$

For all $s \in S$ and $\delta > 0$,

$$\Delta_1^s(\delta) + \Delta_2(\delta) = 9\left(\sqrt{\mathsf{count}_n(p \cdot as)\ln\left(\frac{4|S|}{\delta}\right)}\right) + \frac{1}{\sqrt{2}}\left(\sqrt{\ln\left(\frac{2|S|}{\delta}\right)}\right).$$

Then, for any $s \in S$, we can have a $C_s > 0$ big enough such that

$$\Delta_1^s(\delta) + \Delta_2(\delta) \leq C_s\left(\sqrt{\mathsf{count}_n(p \cdot as)\ln\left(\frac{4|S|}{\delta}\right)}\right).$$

Take $C = \max_s C_s$. Therefore, we have:

$$\begin{aligned}\sum_s (\Delta_1^s(\delta) + \Delta_2(\delta)) &\leq C\sum_s \sqrt{\mathsf{count}_n(p \cdot as)\ln\left(\frac{1}{\delta}\right)} \\ &\leq C\sum_s \sqrt{\mathsf{count}_n(p,a)\ln\left(\frac{1}{\delta}\right)} \\ &\leq C|S|\sqrt{\mathsf{count}_n(p,a)\ln\left(\frac{1}{\delta}\right)}\end{aligned}$$

So, there is a constant C_p such that for $\mathsf{count}_n(p,a)$ big enough and any $\delta > 0$, it holds that

$$\Delta_{\mathsf{count}_n(p,a)}(\delta) = C_p\sqrt{\mathsf{count}_n(p,a)\ln(1/\delta)} \geq \sum_s (\Delta_1^s(\delta) + \Delta_2(\delta))\,.$$

Therefore, from Equation 5.4 the following bound holds:

$$\begin{aligned}&\mathbb{P}\Big[\mathsf{total}_n(p,a) \geq \mathbb{E}[\mathsf{total}_n(p,a)] + \Delta_{\mathsf{count}_n(p,a)}(\delta)\Big] \\ &\leq \mathbb{P}\Big[\mathsf{total}_n(p,a) \geq \mathbb{E}[\mathsf{total}_n(p,a)] + \sum_s (\Delta_1^s(\delta) + \Delta_2(\delta))\Big] \\ &\leq \delta\end{aligned}$$

Similarly, from Equation 5.5, $\mathbb{P}\Big[\mathsf{total}_n(p,a) \leq \mathbb{E}[\mathsf{total}_n(p,a)] - \Delta_{\mathsf{count}_n(p,a)}(\delta)\Big] \leq \delta$.

This proves that for any p, the sequences $(x_{a,t})_{t\geq 1}$ associated with $\mathsf{reward}_{\mathcal{I}_t(p,a)}(p)$ satisfy the drift conditions. □

Then, from the properties of sequence of random variables satisfying drifting conditions, we get:

Theorem 5.1

Consider an MDP M, a horizon H and a state s_0. Let $V_n(s_0)$ (resp. $V_n(s_0, a)$) be a random variable that represents the value $\mathsf{value}_n(s_0)$ *(resp.* $\mathsf{value}_n(s_0, a)$*) at the root of the search tree after n iterations of the generalized MCTS algorithm on M. Then,*

- $|\mathbb{E}[V_n(s_0)] - \mathsf{Val}^H_M(s_0)|$ *is bounded by* $\mathcal{O}((\ln n)/n)$.
- *The failure probability* $\mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}^H_M(s_0)]$ *converges to zero as n tends to infinity.*

Proof From Lemma 5.1, we get that the sequences $(X_{a,t})_{t\geq 1}$ associated with the values $(\mathsf{reward}_{\mathcal{I}_t(p,a)}(p))_{t\geq 1}$ satisfy the drifting condition in Definition 2.13. Then from Theorem 2.9 and Theorem 2.12, we get the results. □

Note that sampling-based approaches mentioned in section 2.6 are captured by our general description of the simulation phase. Indeed, if the number of samples c is set to 1, let I be the set of rewards associated with paths of $\mathsf{Paths}^{\leq H}_M$, and let $f(p)$ be a probability distribution over I, such that for every reward $\mathsf{Reward}_M(p') \in I$, $f(p)(\mathsf{Reward}_M(p'))$ is the probability of path p' being selected with a uniform action selections in $T(M, s_0, H)$, starting from the node p. Then, the value $\mathsf{value}_i(p)$ drawn at random according to the distribution $f(p)$ corresponds to the reward of a random sample $p \cdot p'$ drawn in Paths^H_M. If the number of samples c is greater than 1, one simply needs to extend I to be the set of average rewards over c paths, while $f(p)$ becomes a distribution over average rewards.

5.2 MCTS with symbolic advice

We will augment the MCTS algorithm using two advice: a selection advice φ to guide the MCTS tree construction, and a simulation advice ψ to prune the sampling domain. We assume that the selection advice satisfies the optimality assumption, *i.e.*, φ is enforced by a nondeterministic strategy containing an optimal strategy. Notably, we make no such assumption for the simulation advice, so that any advice can be used.

Selection phase under advice We use the advice φ to prune the tree according to σ_φ. Therefore, from any node p our version of UCT selects, at iteration number i, an action in

$$\arg\max_{a \in \sigma_\varphi^H(p)} \left[\mathsf{value}_{i-1}(p,a) + C \sqrt{\frac{\ln\left(\mathsf{count}_{i-1}(p)\right)}{\mathsf{count}_{i-1}(p,a)}} \right] .$$

In other words, we are using UCT among actions allowed by the advice.

Simulation phase under advice For the simulation phase, from the node represented by the path p, a path $p \cdot p'$ is sampled according to the advice ψ. Different sampling procedures are discussed in Section 4.2. This can be interpreted as sampling paths according to a probability distribution over $\mathsf{Paths}_M^H(p, \psi)$. If there are no p' such that $p \cdot p' \models \psi$, the simulation phase outputs a value of 0 as it is not possible to satisfy ψ from p. We compute $\mathsf{value}_i(p)$ by averaging the rewards of these samples.

Theoretical analysis

We show that the theoretical guarantees of the MCTS algorithm are still maintained by the MCTS algorithm under symbolic advice.

Theorem 5.2

Consider an MDP M, a horizon H and a state s_0. Let $V_n(s_0)$ (resp. $V_n(s_0, a)$) be a random variable that represents the value $\mathsf{value}_n(s_0)$ *(resp.* $\mathsf{value}_n(s_0, a)$*) at the root of the search tree after n iterations of the MCTS algorithm under an enforceable advice φ satisfying the optimality assumption and a simulation advice ψ. Then, $|\mathbb{E}[V_n(s_0)] - \mathsf{Val}_M^H(s_0)| = \mathcal{O}((\ln n)/n)$. Moreover, the failure probability $\mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}_M^H(s_0)]$ converges to zero as n tends to infinity.*

Proof The simulation phase biased by ψ can be described in the formalism of generalized MCTS, with a domain

$$I = \left\{ \frac{1}{c} \sum_{i=1}^{c} \mathsf{Reward}_M(p_i) \mid p_1, \ldots, p_c \in \mathsf{Paths}_{T(M,s_0,H,\varphi)}^{\leq H} \right\} ,$$

and a mapping f_ψ from paths p in $\mathsf{Paths}_{T(M,s_0,H,\varphi)}^{\leq H}$ to a probability distribution on I describing the outcome of a sampling phase launched from the node p. Formally, the weight of $\frac{1}{c}\sum_{i=1}^{c} \mathsf{Reward}_M(p_i) \in I$ in $f(p)$ is the probability of sampling the sequence of paths $p_1, \ldots, p_c$ in the simulation phase under advice ψ launched from p. Then, from Theorem 5.1, using MCTS algorithm under an enforceable advice φ satisfying the optimality assumption

and a simulation advice ψ in MDP M is same as using generalized MCTS algorithm in the pruned MDP $T(M, s_0, H, \varphi)$.

Theorem 4.1 lets us conclude the proof as those values and strategies are maintained in M by the optimality assumption. In particular,

$$|\mathbb{E}[V_n(s_0)] - \mathsf{Val}^H_{T(M,s_0,H,\varphi)}(s_0)| = |\mathbb{E}[V_n(s_0)] - \mathsf{Val}^H_M(s_0)| = \mathcal{O}((\ln n)/n)\,.$$

Also, since $\mathsf{opt}^H_{T(M,s_0,H,\varphi)} \subseteq \mathsf{opt}^H_M(s_0)$, we will have the failure probability in M bounded by the failure probability in $T(M, s_0, H, \varphi)$:

$$\mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}^H_M(s_0)] \leq \mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}^H_{T(M,s_0,H,\varphi)}(s_0)]\,.$$

Then,the failure probability $\mathbb{P}[\arg\max_a V_n(s_0, a) \not\subseteq \mathsf{opt}^H_M(s_0)]$ converges to zero as n tends to infinity. □

Chapter 6

Applications of MCTS with advice

In this chapter, we give an informal description of our framework and then describe two applications of the MCTS algorithm with advice. The code of our implementation can be found here: https://debrajrc.github.io/MCTS-with-advice/

6.1 Description of the framework

Given an encoding of an MDP, a *simulator* for an MDP needs to do the following operations: report the current state, report available actions at the state, simulate a legal action from current state and report the new state and the reward gained. Such a simulator can be implemented depending on how the MDP is represented. For example, an MDP can be described in any file format supported by Storm model checker [Deh+17; Hen+22]. In this case, we simulate the MDP using the simulators implemented in Storm (version 1.7.0). Specifically, we use PRISM format [KNP11] to describe an MDP which can be simulated in Storm without constructing the explicit model. We use Stormpy [JV+22], a python wrapper on Storm providing several useful APIs.

6.1.1 Monte Carlo tree search

The implementation of MCTS uses such a simulator to find an optimal action during decision-time planning. The algorithm would have multiple parameters, for example, the receding horizon it is using, number of new nodes it is adding to the tree, number of simulations it is running each time a new node is added etc. as described in Section 2.6.

6.1.2 Selection and simulation strategy

The user has option to specify strategies that can be used to select the nodes in the search tree during the selection phase and to simulate paths during the simulation phase. By default, the UCT strategy (see Section 2.6) is used as the selection strategy. In that case, the user provides the constant C in the formula. For simulation, a strategy is used to choose

actions uniformly at random from the legal actions from a state. But the user can provide different probabilistic strategy that can be used during simulation.

6.1.3 Selection advice

We use an enforceable advice as a selection advice. The nondeterministic strategy σ enforcing the selection advice is represented by a function which given a state s of the MDP would return the set $\sigma(s)$. This can be implemented for different type of strategies:

Formal methods-based techniques The user gives a logical formula φ as an input and for each action a available in state s, we check whether $(s, a) \models \varphi$. This creates the non-deterministic strategy σ where $\sigma(s) = \{a \mid (s, a) \models \varphi\}$. Instead of the large MDP, a smaller abstraction of the original MDP can be defined by the user which can be used for the model checking.

Neural networks The user provides a function that, given a state of the MDP, creates a tensor (as a NumPy array [Har+20]). A pre-trained neural network (as a Keras [Cho+15] model saved in HDF5 format) can be used as an input in the implementation. For a neural network NN and a threshold value $\delta \in (0, 1)$, we calculate a non-deterministic strategy $\sigma_{\mathsf{NN},\delta}$ where

$$\sigma_{\mathsf{NN},\delta} = \{a \mid \mathsf{NN}(s, a) \geq \max_{a'} \mathsf{NN}(s, a')\}\,.$$

Other function-based techniques The user defines a function $f : S \times A \to \mathbb{R}$ that gives a value to any state-action pair in the MDP. For a threshold value $\delta \in (0, 1)$, we calculate a non-deterministic strategy $\sigma_{f,\delta}$ where

$$\sigma_{\mathsf{NN},\delta} = \{a \mid f(s, a) \geq \max_{a'} f(s, a')\}\,.$$

6.1.4 Simulation advice

Given an advice $\mathscr{A}$ as a logical formula $\varphi_{\mathscr{A}}$, we check if a path p satisfies the formula $\varphi_{\mathscr{A}}$. This is used for the reject-based approach defined in Section 4.2 where we only allow sampled paths according to the advice.

6.1.5 Terminal reward

The user provides a function $R_T : S \to \mathbb{R}$ that assigns a value to each states in the MDP. In general a neural network or an easy to calculate heuristic is used as such function.

6.2 Application 1: Pacman

We performed our experiments on the game Pac-Man. In this case, we automated the creation of a PRISM file from a text file describing the grid and the initial position of Pac-Man and the ghosts. In our experiments, the ghosts always choose an action uniformly at random from the legal actions available.

6.2.1 The game as an MDP

The game can be seen as a Markov decision process, where states encode a position for each agent[1] and for the food pills in the grid, where actions encode individual Pac-Man moves, and where stochastic transitions encode the moves of ghosts according to their probabilistic models. For each state and action pair, we define a reward based on the score gained or lost by this move, as explained in Example 1.1. We also assign a terminal reward to each state, so as to allow MCTS to compare paths of length H which would otherwise obtain the same score. Intuitively, better terminal rewards are given to states where Pac-Man is closer to the food pills and further away from the ghosts, so that terminal rewards play the role of a static evaluation of positions.

6.2.2 Advice

The *simulation advice* ψ that we consider is defined as a safety property satisfied by every path such that Pac-Man does not make contact with a ghost, as in Example 4.2. With a Boolean formula encoding ψ, one can use a SAT solver to obtain samples, or sampling tools as described in Proposition 4.2, such as WeightGen [Cha+14]. Alternatively, the tool UniGen [Cha+15] can be used to sample almost uniformly over the satisfying assignments of ψ.[2] Several techniques were used to reduce the state-space of the MDP in order to obtain smaller Boolean formulæ. For example, a ghost that is too far away with respect to H can be safely ignored, and the current positions of the food pills is not relevant for safety. But in our experiments, the reject based method for sampling is much faster than the SAT-based sampling and gave similar performance in terms of win-rate.

[1] The last action played by ghosts should be stored as well, as they are not able to reverse their direction.

[2] The distribution over path is slightly different than when sampling uniformly over actions in the pruned MDP $T(M, s_0, H, \psi)$, but UniGen enjoys better performances than WeightGen.

As a selection advice, during the exploration of the search tree, we restrict ourselves to actions a that maximize the probability to stay safe for the next 8 steps, *i.e.*, actions a such that $\eta_8(s,a) = \max_{a' \in A} \eta_8(s,a')$ as defined in Section 4.5. Since the online computation of the η_8 function is too expensive to be done at every node of the search tree, we only restrict the root node of the tree to ensure the safety of the immediate decisions.

6.2.3 Experiments

We experimented on the grid of size 9×21 described in Figure 1.1. We used a receding horizon $H = 10$. The baseline is given by a standard implementation of the algorithm described in Section 2.6. A search tree is constructed with a maximum depth H, for 40 iterations, so that the search tree constructed by the MCTS algorithm contains up to 40 nodes. At the first selection of every node, 20 samples are obtained by using a uniform policy. Overall, this represents a tiny portion of the tree unfolding of depth 10, which underlines the importance of properly guiding the search to the most interesting neighbourhoods. As a point of reference, we also had human players take control of Pac-Man, and computed the same statistics. The players had the ability to slow down the game as they saw fit, as we aimed for a comparison between the quality of the strategical decisions made by these approaches, and not of their reaction speeds.

We compare these baselines with the algorithm of Section 5.2, using the advice mentioned before.

Several techniques were used to reduce the state-space of the MDP in order to obtain smaller formulæ. For example, a ghost that is too far away with respect to H can be safely ignored, and the current positions of the food pills is not relevant for safety.

6.2.4 Results

A summary of our results is displayed in Table 6.1. We mainly use the number of games won out of 100 to evaluate the performance of our algorithms. The win, loss and draw columns denote win/loss/draw rates in percents (the game ends in a draw after 300 game steps). The food eaten column refers to the number of food pills eaten on average, out of 25 food pills in total. Score refers to the average score obtained over all runs.

Algorithm	win	loss	draw	food	score
MCTS	8	87	5	12.85	-383.63
MCTS + selection advice	36	25	39	20.08	28.82
MCTS + simulation advice	55	44	1	19.58	153.62
MCTS + both advice	90	9	1	24.39	512.13
Uniform	0	100	0	2.35	-498.25
Uniform + selection advice	5	30	65	15.64	-215.71
Human	44	56	0	18.87	57.76

Table 6.1: Summary of experiments with different ghost models, algorithms.

The baseline MCTS algorithm wins 8% of games. Adding the selection advice results in an increase of the win rate to 36%. The average score is improved as expected, but even if one ignores the ± 500 score associated with a win or a loss, we observe that more food pills were eaten on average as well. The simulation advice provides a increase in both win rate (achieving 55%) and average score. Using both advice at the same time gave the best results overall, with a win rate of 90%. Moreover, the simulation advice significantly reduces the number of game turns Pac-Man needs to win, resulting in fewer game draws.

In an older implementation described in [BCR20], from the simulation advice ψ, we extracted whenever possible a strongly enforceable *selection advice* φ that guarantees that Pac-Man will not make contact with a ghost, as described in Example 4.3. If safety cannot be enforced, the universal advice $\top$ is used as a selection advice, so that no pruning is performed. This is implemented by using the Boolean formula ψ in a QBF solver according to Lemma 4.2. For performance reasons, we could only guarantee safety for much smaller horizon than 10, that we fixed at 3 in our experiments. With this selection advice and the simulation advice ψ, we achieved 85% win-rate.

6.3 Application 2 : safe and optimal scheduling of tasks

We consider a setting in which we are given the structure of a task system $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ (described in Section 2.7) to schedule. While the structure of the system is known, the actual distributions that describe the behaviour of the tasks are unknown and need to be learnt to behave optimally or near optimally. The learning must be done only by observing the jobs that arrive along time. When the task system contains some hard tasks ($H \neq \emptyset$), all deadlines of such tasks must be enforced.

After the system is efficiently learnt, then we find and safe and (near) optimal strategy in the MDP of the Task system.

6.3.1 Model-Based Learning of task systems

For learning the inter-arrival time distribution of a task, a *sample* corresponds to observing the time difference between the arrivals of two consecutive jobs of that task. For learning the computation time distribution, a sample corresponds to observing the CPU time a job of the task has been assigned up to completion. Thus, if a job does not finish execution before its deadline, we do not obtain a valid sample for the computation time. Given a class of task systems, we say:

- The class is *probably approximately correct (PAC) learnable* if there is an algorithm $\mathbb{L}$ such that for all task systems Υ in this class, for all $\epsilon, \gamma \in (0,1)$: given $\mathsf{struct}(\Upsilon)$, the algorithm $\mathbb{L}$ can execute the task system Υ, and can compute Υ' such that $\Upsilon \approx^{\epsilon} \Upsilon'$, with probability at least $1-\gamma$.
- The class is *safely PAC learnable* if it is PAC learnable, and $\mathbb{L}$ can ensure safety for the hard tasks while computing Υ'.
- The class is (safely) *efficiently PAC learnable*[3] if it is (safely) PAC learnable, and there is a polynomial q in the size of the task system, in $\frac{1}{\epsilon}$, and in $\frac{1}{\gamma}$, s.t. $\mathbb{L}$ obtains enough samples to compute Υ' in a time bounded by q.

Efficient PAC learning for soft tasks Let $\Upsilon = ((\tau_i)_{i\in I}, F, \emptyset)$ be a task system with soft tasks only, and let $\epsilon, \gamma \in (0,1)$. We assume that for all distributions d occurring in the models of the tasks in Υ: $\min_{a\in\mathsf{Supp}(d)} d(a) > \epsilon$. To learn a model Υ' which is ϵ-close to Υ with probability at least $1-\gamma$, we apply Lemma 2.1 in the following algorithm:

1. for all tasks $i = 1, 2, \cdots \in F$, repeat the following learning phase:
   Always schedule task τ_i when a job of this task is active. This way, collect enough samples $\mathbb{S}(A_i)$ of A_i and $\mathbb{S}(C_i)$ of C_i as observed to apply Lemma 2.1 and obtain the desired accuracy as fixed by ϵ and γ.
2. the models of inter-arrival time distribution and computation time distribution for task τ_i are the relative frequencies $d(\mathbb{S}(A_i))$ and $d(\mathbb{S}(C_i))$ respectively.

[3] Note that our notion of efficient PAC learning is stronger than the definition used in classical PAC learning terminology [Val84] since we take into account the time that is needed to get samples and not only the number of samples needed.

It follows that task systems with only soft tasks are *efficiently* PAC learnable:

Theorem 6.1

There is a learning algorithm such that for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ *with* $H = \emptyset$*, for all* $\epsilon, \gamma \in (0,1)$*, the algorithm learns a model* Υ' *such that* $\Upsilon' \approx^{\epsilon} \Upsilon$ *with probability at least* $1-\gamma$ *after executing* Υ *for* $|F| \cdot A_{\max} \cdot \mathbb{D} \cdot \lceil \frac{1}{2\epsilon^2}(\ln 4\mathbb{D}|F| - \ln\gamma)\rceil$ *steps where* $\mathbb{D} = \max_{i\in[n]}(|\mathsf{Supp}(A_i)|)$.

Proof Using Lemma 2.1, given $\epsilon, \gamma' \in (0,1)$, for every distribution d of the task system, a sequence $\mathbb{S}$ of $\mathbb{D} \cdot \lceil \frac{1}{2\epsilon^2}(\ln 2\mathbb{D} - \ln\gamma')\rceil$ i.i.d. samples suffices to have $d(\mathbb{S}) \sim^{\epsilon} d$ with probability at least $1-\gamma'$. Since in the task system Υ, there are $2|F|$ distributions, with probability at least $1 - 2|F|\gamma'$, we have that the learnt model $\Upsilon' \approx^{\epsilon} \Upsilon$. Thus, for $\gamma' = \frac{\gamma}{2|F|}$, and using $2\exp(-2m\epsilon^2) \leq \frac{\gamma}{2|F|\mathbb{D}}$, we have that for each distribution, a sequence of $\mathbb{D} \cdot \lceil \frac{1}{2\epsilon^2}(\ln 4\mathbb{D}|F| - \ln\gamma)\rceil$ samples suffices so that $\Upsilon' \approx^{\epsilon} \Upsilon$ with probability at least $1-\gamma$.

We collect samples for $|F|$ soft task by scheduling one soft task after another. For each soft task, samples for computation time distribution and inter-arrival time distribution can be collected simultaneously, and observing each sample takes a maximum of $A_{\max}$ time steps. This proves the result. □

Safe learning with hard tasks We turn to task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with both hard and soft tasks. The learning algorithm must ensure that all the jobs of hard tasks meet their deadlines while learning the task distributions. The soft-task-only algorithm is clearly not valid for that more general case. Recall we have assumed schedulability of the task system for the hard tasks[4]. This is a necessary condition for safe learning but it is not a sufficient condition. Indeed, to apply Lemma 2.1, we need enough samples for all tasks $i \in H \cup F$.

First, we note that when executing any safe strategy for the hard tasks, we will observe enough samples for the hard tasks. Indeed, under a safe strategy for the hard tasks, any job of a hard task that enters the system will be executed to completion before its deadline. We then observe the value of the inter-arrival and computation times for all the jobs of hard tasks that enter the system. Unfortunately, this is not necessarily the case for soft tasks when they execute in the presence of hard tasks. Indeed, it is in general not possible to

[4]Note that safety synthesis already identifies task systems that violate this condition.

schedule all the jobs of soft tasks up to completion. We thus need stronger conditions in order to be able to learn the distributions of the soft tasks while ensuring safety.

PAC guarantees for safe learning Our condition to ensure safe PAC learnability relies on properties of the safe region $M_\Upsilon^{\mathsf{safe}}$ in the MDP M_Υ associated to the task system Υ.

Definition 6.1 (Good for sampling condition)

The safe region $M_\Upsilon^{\mathsf{safe}}$ of the task system $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ is good for sampling *if for all soft tasks $i \in F$, there exists a state $s_i \in M_\Upsilon^{\mathsf{safe}}$ such that:*

- *a new job of task i enters the system in s_i; and*
- *there exists a strategy σ_i of Scheduler that is compatible with the set of safe schedules for the hard tasks so that from s_i, under strategy σ_i, the new job associated to task τ_i is guaranteed to reach completion before its deadline.*

There is an algorithm that executes in polynomial time in the size of the MDP $M_\Upsilon^{\mathsf{safe}}$ which decides if $M_\Upsilon^{\mathsf{safe}}$ is good for sampling. Also, remember that only the knowledge of the structure of the task system is needed to compute $M_\Upsilon^{\mathsf{safe}}$.

Given a task system $M_\Upsilon^{\mathsf{safe}}$ that is good for sampling, given any $\epsilon, \gamma \in (0, 1)$, we safely learn a model Υ' which is ϵ-close to Υ with probability at least $1 - \gamma$ (PAC guarantees) by applying the following algorithm:

1. Choose any safe strategy σ_H for the hard tasks, and apply it until enough samples $(\mathbb{S}(A_i), \mathbb{S}(C_i))$ for each $i \in H$ have been collected according to Lemma 2.1. The models for tasks $i \in H$ are $d(\mathbb{S}(A_i))$ and $d(\mathbb{S}(C_i))$.
2. Then for each $i \in F$, apply the following phases:
   (a). from the current vertex s, schedule some task uniformly at random among the set of tasks that correspond to the safe edges in $\mathsf{safe}(s)$ up to reaching some s_i (while choosing tasks that do not violate safety uniformly at random, we reach some s_i with probability 1.[5] The existence of a s_i is guaranteed by the hypothesis that $M_\Upsilon^{\mathsf{safe}}$ is good for sampling).
   (b). from s_i, apply the strategy σ_i as defined by the second condition in the *good for sampling condition*. This way we are guaranteed to observe the computation time requested by the new job of task i that entered the system in vertex s_i, no matter how TaskGen behaves. At the completion of this job of task i, we have

[5] This follows from the fact that there is a single MEC in the MDP by Lemma 2.8.

collected a valid sample of task i.

(c). go back to (a) until enough samples $(\mathbb{S}(A_i), \mathbb{S}(C_i))$ have been collected for soft task i according to Lemma 2.1.

The properties of the learning algorithm above are used to prove that:

Theorem 6.2

There is a learning algorithm such that for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ with a safe region $M_\Upsilon^{\mathsf{safe}}$ that is good for sampling, for all $\epsilon, \gamma \in (0,1)$, the algorithm learns a model Υ' such that $\Upsilon' \approx^\epsilon \Upsilon$ with probability at least $1-\gamma$.

Proof For the hard tasks, as mentioned above, we can learn the distributions by applying the safe strategy σ_H to collect enough samples $(\mathbb{S}(A_i), \mathbb{S}(C_i))$ for each $i \in H$.

We assume an order on the set of soft tasks. First for all τ_i for $i \in F$, since $M_\Upsilon^{\mathsf{safe}}$ is good for sampling, we note that the set S_i of states s_i (as defined in the definition of good for sampling condition) is non-empty. Recall from Lemma 2.8 that $M_\Upsilon^{\mathsf{safe}}$ has a single MEC. Thus, from every state of $M_\Upsilon^{\mathsf{safe}}$, Scheduler by playing uniformly at random reaches some $s_i \in S_i$ with probability 1, and hence can visit the vertices of S_i infinitely often with probability 1. Now given ϵ and γ, using Theorem 6.1, we can compute an m, the number of samples corresponding to each distribution required for safe PAC learning of the task system. Since by playing uniformly at random, Scheduler has a strategy to visit the vertices of S_i infinitely often with probability 1, it is thus possible to visit these vertices at least m times with arbitrarily high probability.

Also, after we safely PAC learn the distributions for task τ_i, since there is a single MEC in $M_\Upsilon^{\mathsf{safe}}$, there exists a uniform memoryless strategy to visit a state s_{i+1} corresponding to task τ_{i+1} with probability 1. Hence, the result. □

In the algorithm above, to obtain one sample of a soft task, we need to reach a particular vertex s_i from which we can safely schedule a new job for the task i up to completion. As the underlying MDP $M_\Upsilon^{\mathsf{safe}}$ can be large (exponential in the description of the task system), we cannot bound by a polynomial the time needed to get the next sample in the learning algorithm. So, this algorithm does not guarantee efficient PAC learning. We develop in the next paragraph a stronger condition to guarantee efficient PAC learning.

Definition 6.2 (Good for efficient sampling)

The safe region $M_\Upsilon^{\mathsf{safe}}$ *of the task system* $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ *is* good for efficient sampling *if there exists* $K \in \mathbb{N}$ *which is bounded polynomially in the size of* $\Upsilon = ((\tau_i)_{i\in I}, F, H)$*, and if, for all soft tasks* $i \in F$ *the two following conditions hold:*

1. *let* S_0^{safe} *be the set of Scheduler vertices in* $M_\Upsilon^{\mathsf{safe}}$*. There is a non-empty subset* $\mathsf{Safe}_i \subseteq S_0^{\mathsf{safe}}$ *of states from which there is a strategy* σ_i *for Scheduler to schedule safely the tasks* $H \cup \{i\}$ *(i.e. all hard tasks* and *the task* i*); and*
2. *for all* $s \in S_0^{\mathsf{safe}}, i \in F$*, there is a uniform memoryless strategy* $\sigma_{\diamond\mathsf{Safe}_i}$ *s.t.:*
   (a). $\sigma_{\diamond\mathsf{Safe}_i}$ *is compatible with the safe strategies (for the hard tasks) of* $M_\Upsilon^{\mathsf{safe}}$*;*
   (b). *when* $\sigma_{\diamond\mathsf{Safe}_i}$ *is executed from any* $s \in S_0^{\mathsf{safe}}$*, then the set* Safe_i *is reached within* K *steps. By Lemma 2.8, since* $M_\Upsilon^{\mathsf{safe}}$ *has a single MEC, we have that* Safe_i *is reachable from every* $v \in S_0^{\mathsf{safe}}$*.*

Here again, the condition can be efficiently decided: there is a polynomial-time algorithm in the size of $M_\Upsilon^{\mathsf{safe}}$ that decides if $M_\Upsilon^{\mathsf{safe}}$ is good for efficient sampling.

Given a task system $M_\Upsilon^{\mathsf{safe}}$ that is *good for efficient sampling*, given $\epsilon, \gamma \in (0, 1)$, we safely and efficiently learn a model Υ' which is ϵ-close of Υ with probability at least than $1 - \gamma$ (efficient PAC guarantees) by applying the following algorithm:

1. Choose any safe strategy σ_H for the hard tasks, and apply this strategy until enough samples $(\mathbb{S}(A_i), \mathbb{S}(C_i))$ for each $i \in H$ have been collected according to Lemma 2.1. The models for tasks $i \in H$ are the relative frequencies $d(\mathbb{S}(A_i))$ and $d(\mathbb{S}(C_i))$.
2. Then for each $i \in F$, apply the following phase:
   (a). from the current state s, play $\sigma_{\diamond\mathsf{Safe}_i}$ to reach the set Safe_i.
   (b). from the current state in Safe_i, apply the strategy σ_i as defined above. This way we are guaranteed to observe the computation time requested by all the jobs of task i that enter the system.
   (c). go to (b) until enough samples $(\mathbb{S}(A_i), \mathbb{S}(C_i))$ are collected for task i as per Lemma 2.1. The models for task i are given by $d(\mathbb{S}(A_i))$ and $d(\mathbb{S}(C_i))$.

For a task system Υ, let $T = A_{\max} \cdot \mathbb{D} \cdot \lceil \frac{1}{2\epsilon^2}(\ln 4\mathbb{D}|\Upsilon| - \ln \gamma)\rceil$ where $A_{\max} = \max(\bigcup_{i\in[n]} \mathsf{Supp}(A_i))$ and $\mathbb{D} = \max_{i\in[n]}(|\mathsf{Supp}(A_i)|)$. The properties of the learning algorithm above are used to prove the following theorem:

Theorem 6.3

There exists a learning algorithm such that for all task systems $\Upsilon = ((\tau_i)_{i\in I}, F, H)$ *with a safe region* $M_\Upsilon^{\mathsf{safe}}$ *that is good for efficient sampling, for all* $\epsilon, \gamma \in (0,1)$*, the algorithm learns a model* Υ' *such that* $\Upsilon' \approx^\epsilon \Upsilon$ *with probability at least* $1-\gamma$ *after scheduling* Υ *for* $T + |F| \cdot (T + K)$ *steps.*

Proof Consider the algorithm described above. Since σ_H is a safe strategy for the hard tasks, we can observe the samples corresponding to the computation time distribution and the inter-arrival time distribution for all the hard tasks simultaneously while scheduling the system. Following the proof of Theorem 6.1, the samples required to learn the distributions of the hard tasks can be observed in time T.

Now consider an order on the set of tasks. Under the good for efficient sampling condition, again from the proof of Theorem 6.1, we need to execute the system for $|F|T$ time steps for collecting samples to PAC learn the computation time distributions and the inter-arrival time distributions for all soft tasks in F. Further, for every soft task τ_i with $i \in F$, from a state in S_0^{safe}, by using the strategy $\sigma_{\diamond\mathsf{Safe}_i}$, we reach Safe_i in at most K steps. Hence, the result. □

Example 6.1 Consider the following task system with one hard and one soft task. We want to learn the distributions associated to the tasks in the system.

- one hard task $\tau_h = \langle C_h, 2, A_h\rangle$ such that $C_h(2) = 1$ and $A_h(4) = 1$; and
- one soft task $\tau_s = \langle C_s, 2, A_s\rangle$ such that $\mathsf{Supp}(C_s) = \{1,2\}$ $C_s(1) = 0.4$, $C_s(2) = 0.6$, and $A_s(3) = 1$; and the cost function c such that $c(\tau_s) = 10$.

In particular, we want to learn computation time distribution for the soft task. We can see that during the execution of the task system, for every time t, Scheduler does not have a safe strategy from t that also ensures that the soft task will never miss a deadline. This implies that considering the good for efficient sampling condition, we have $\mathsf{Safe_i} = \emptyset$ for $i \in F$, and hence the good for efficient sampling condition is not satisfied by this task system. Thus, we cannot ensure safe and efficient PAC learning for this task system.

On the other hand, there exists a strategy such that for all the jobs of the soft task that arrive at time $\text{lcm}(4,3) \cdot n + 6 = 12n + 6$ (assuming that the system starts executing at time 0) for $n \geq 0$ can be scheduled to completion, and thus by Theorem 6.2, there exists an algorithm to safely PAC learn the task system. □

Using the learnt model Given a system Υ of tasks, and parameters $\epsilon, \gamma \in (0,1)$, once we have learnt a model Υ' such that $\Upsilon' \approx^{\epsilon} \Upsilon$, we construct the MDP $M^{\mathsf{safe}}_{\Upsilon'}$. From $M^{\mathsf{safe}}_{\Upsilon'}$, we can compute an optimal scheduling strategy that minimizes the expected mean-cost of missing deadlines of soft tasks. Such an algorithm is given in [GGR18]. Then, we execute the actual task system Υ under schedule σ. However, since σ has been computed using the model Υ', it might not be optimal in the original, unknown task system Υ. Nevertheless, we can bound the difference between the optimal values obtained in $M^{\mathsf{safe}}_{\Upsilon'}$ and $M^{\mathsf{safe}}_{\Upsilon}$.

The following lemma relates the model that is learnt with the approximate distribution that we have in the MDP corresponding to the learnt model. Let $\pi^{\Upsilon}_{\max}$ be the maximum probability appearing in Υ. Given $\epsilon \in (0,1)$, let $\beta = \min\{1, \pi^{\Upsilon}_{\max} + \epsilon\}$ and $\eta = \beta^{2n} - (\beta - \epsilon)^{2n}$, where $n = |\Upsilon|$.

Lemma 6.1

Let Υ be a task system, let $\epsilon, \gamma \in (0,1)$, let Υ' be the learnt model such that $\Upsilon' \approx^{\epsilon} \Upsilon$ with probability at least $1-\gamma$. Then we have that $\Gamma_{\Upsilon'} \approx^{\eta} \Gamma_{\Upsilon}$ with probability at least $1-\gamma$.

Proof Since we have that $\Upsilon' \approx^{\epsilon} \Upsilon$ with probability at least $1-\gamma$, by definition, we have that the probability that all the distributions of Υ' are ϵ-close to their corresponding distributions in Υ is at least $1-\gamma$. Let $|\Upsilon| = n$, and there are a total of $2n$ distributions. Let P and P' be the probability distributions in the MDPs M_{γ} and $M_{\gamma'}$ respectively. Thus corresponding to P in M_{γ}, if an edge has probability $p = p_1 p_2 \cdots p_{2n}$, and for P' we have the corresponding probability as p', then $|p' - p| \leq \prod_{i=1}^{2n}(p'_i) - \prod_{i=1}^{2n}(p_i)$, where p'_i is the estimation of p_i in P', and is such that $p'_i \leq \min\{1, p_i + \epsilon\}$, since each estimated probability in the distribution P' is also bounded above by 1. Now $p'_i \leq \beta$ for all $i \in [2n]$, and we have that

$$\prod_{i=1}^{2n} p'_i - \prod_{i=1}^{2n}(p_i) \leq \beta^{2n} - (\beta - \epsilon)^{2n}$$

Thus, $P' \sim^{\eta} P$ with probability at least $1-\gamma$. □

Using Lemma 6.1 and Lemma 2.2, we obtain the following guarantees on the quality of the strategy that our model-based learning algorithm outputs:

Theorem 6.4

Suppose we are given a task system Υ (with min probability $p_{\min}$) and a robustness precision $\beta \in (0,1)$. Let $\gamma, \epsilon \in (0,1)$ be s.t. $\epsilon \leq \frac{\beta p_{\min}}{8|S| + \beta \cdot p_{\min}}$. Let Υ^M be the model

that is learnt using the above algorithms s.t. $\Upsilon^M \approx^{\varepsilon} \Upsilon$ *with probability at least* $1-\gamma$*, and let* σ *be a memoryless deterministic expectation-optimal strategy of* Γ_{Υ^M}*. Then, with probability at least* $1-\gamma$*, the expected mean-cost of playing* σ *in* Γ_Υ *(i.e. in the task system* Υ*) is s.t. for all* $s \in S$*:* $|\mathsf{Val}_M(s,\sigma) - \mathsf{Val}_M(s)| \leq \beta$.

6.3.2 Experimental results

We first report experimental results on model-based learning and observe that the models are learnt efficiently with only a few samples. We compare the performance of our MCTS-based algorithms with a state-of-the-art deep Q-learning implementation from OpenAI [Dha+17] on a set of benchmarks of task systems of various sizes.

Models with only soft tasks In Figure 6.1, we show that the distributions of a task system with soft tasks can be learnt efficiently with a few samples. This is not the case in general for arbitrary MDPs where in order to collect samples, one may need to reach some specific states of the MDP, and it may take a considerable amount of time to reach such states. However, in this case of systems with only soft tasks, the number of samples increases linearly with time. As a representative task system, we display the learning curve for a system with six soft tasks in Figure 6.1.

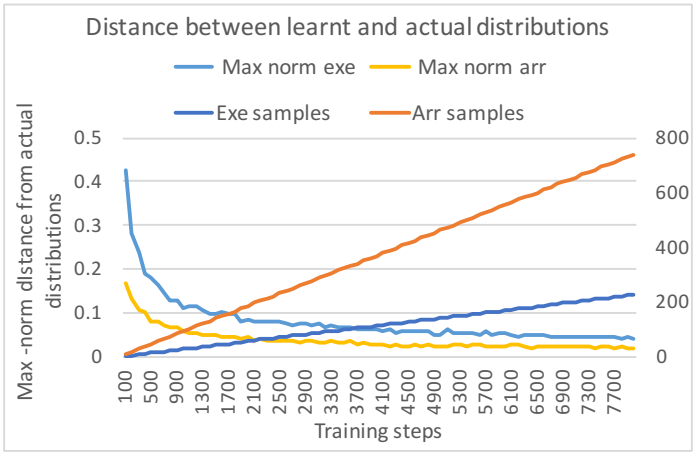

Figure 6.1: Learning distributions for a system with 6 soft tasks

Here "exe" and "arr" refer to the distributions of the computation times and the inter-arrival times respectively. The left y-axis is the max-norm distance between the probabilities

in the actual distributions and the learnt distributions across all soft tasks. The x-axis is the number of time steps over which the system is executed. For learning the computation time distribution, the soft tasks are scheduled in a round-robin manner. Once a job of a soft task is scheduled, it is executed until completion without being preempted. A sample for learning the computation time distribution of a soft task thus corresponds to a job of the task that is scheduled to execute until completion. Since the system has only soft tasks, a job can always be executed to finish its execution without safety being violated. On the other hand, the samples for learning the inter-arrival time distribution for each task correspond to all the jobs of the task that arrive in the system. Thus, over a time duration, for each task, the number of samples collected for learning the inter-arrival time distribution is larger than the number of samples collected for learning the computation time distribution. The number of samples of both kinds increases linearly with time. The y-axis on the right corresponds to the number of samples collected over a duration of time when the system executes. The plot "Exe samples" corresponds to the number of samples collected per task for learning the computation time distributions. Since the tasks are executed in a round-robin manner, the tasks have an equal number of samples for learning their computation time distributions. On the other hand, for learning inter-arrival time distributions, a task with larger inter-arrival time produces fewer samples than a task with smaller inter-arrival time. The plot "Arr samples" corresponds to the minimum of the number of jobs, over all the tasks, that arrived in the system. Each point in the graphs is obtained as a result of averaging over 50 simulations.

Safe model-based learning For safe model-based learning of systems with both hard and soft tasks, first, we verify that the task system satisfies the *good for efficient sampling* condition, and hence admits safe efficient PAC learning. We consider a small representative task system, and learn the distributions of the soft tasks in the system as shown in Figure 6.2. Using the learnt model in STORM, we also extracted the strategy that minimizes the expected mean cost. This strategy is then applied on the actual task system, and we report the observed value. In the X-axis, we have the number of discrete time steps over which samples corresponding to all soft tasks are collected. The result of each time point is an average over 10 experiments. We note that for samples collected over a few hundred discrete time steps, the optimal strategy for the learnt system is also optimal for the original task system. For fewer training steps than that shown in Figure 6.2, we observed that for some experiments the learnt task system (and hence the corresponding MDP) was not structurally similar to the original task system in the sense that some distributions in the learnt model did not have the same support as those in the original task system. The results show that

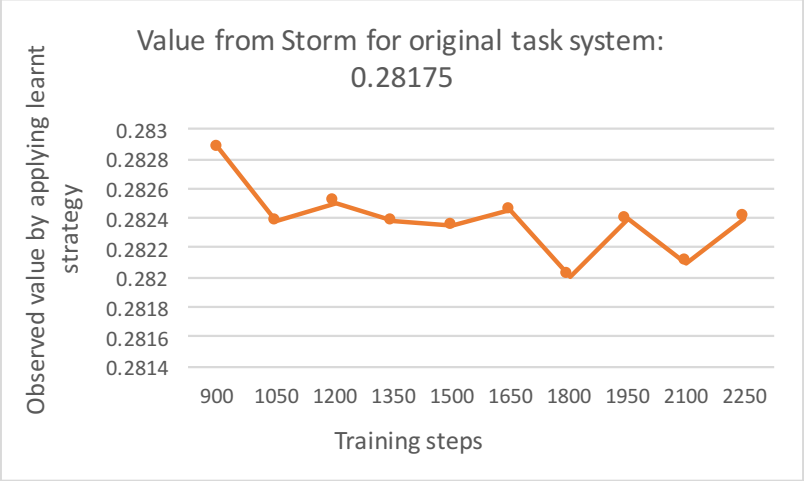

Figure 6.2: Model-based learning for 1 hard, 3 soft tasks

this approach is effective in terms of the quality of learning and the number of samples required.

Monte Carlo tree search In the above approach, the main bottleneck towards scalability is the extraction of an optimal strategy from the learnt model using probabilistic model-checkers like Storm. This is because the underlying MDP grows exponentially with the number of tasks. Therefore, we advocate the use of receding horizon techniques instead, that optimize the cost based on the next H steps for some horizon H. In our examples, the unfoldings have approximately 2^H states, so we use MCTS to explore them in a scalable way. We used two advice that are enforced by these following strategies:

1. Earliest deadline first strategy for hard tasks (EDF) The earliest deadline first (EDF) on hard task is a strategy for scheduler that ensures no hard tasks misses its deadline by always scheduling the job of a hard task with the closest deadline [But11, Section 4.4]. A job by soft task is scheduled only when no hard task is available. This ensures that all jobs of hard tasks are scheduled in time as the task system is schedulable for the hard tasks, but does not guarantee optimality with respect to cost. In spite of this drawback, it is a strategy that can be used for an advice because it is easy to calculate and therefore can be calculated on-the-fly while doing MCTS.

2. Most general strategy (MGS) We can consider the underlying game graph of the MDP. There we have a safety game where the objective for player 0 (the Scheduler) is

to avoid the state marked with $\bot$ denoting the states where a job generated by a hard task has missed its deadline. We calculated the most general strategy in this game using AbsSynthe [Bre+14]. This strategy is more complete, but on the other hand, it needs to be precomputed.

Deep Q-learning One of the most successful model-free learning algorithm is the *Q-learning* algorithm, due to Watkins and Dayan [WD92]. It aims at learning (near) optimal strategies in a (partially unknown) MDP for the *discounted sum* objective. In our scheduling problem, we search for (near) optimal strategies for the mean-cost and *not* for the discounted sum, as we want to minimize the limit average of the cost of missing deadlines of soft tasks. However, if the discount factor is close to 1, both values coincide [Sol03; MN81]. In our experiments, we use an implementation of deep Q-learning available in the OpenAI repository [Dha+17]. We make use of shielding [Cha+17a; Als+18; Avn+19], a technique that restricts actions in the learning process so that only those actions that are safe for the hard tasks can be used. More specifically, we used the earliest deadline first (EDF) strategy and the most general (MGS) strategy for safety to shield unsafe actions during deep Q-learning.

Experiments We compare some variants of model-based learning using MCTS and deep Q-learning in the context of scheduling. The first option is to set a very high penalty on missing the deadline of a hard task, and then to apply either MCTS or deep Q-learning. However, safety is not guaranteed in this case, and we report on whether a violation was observed or not. We call this variant unsafe MCTS and unsafe deep Q-learning respectively as a consequence. The second option is to enforce safety in MCTS and deep Q-learning by computing the most general safe scheduler for hard tasks, and then using it as advice for MCTS or the MGS shield for deep Q-learning. The third option is to use the earliest-deadline-first (EDF) scheme on hard tasks instead of MGS as an advice or a shield. Note that the second and the third options are required to ensure safety, and thus are applicable to systems that have at least one hard task, and hence are not applicable (NA) to systems with only soft tasks.

We used a heuristic terminal reward during MCTS where for each soft tasks, we check if its job can be finished before deadline if we allow all the hard tasks and all soft tasks with higher cost to finish first. We add the costs for all these tasks which may miss their deadline. This works as a good and easy to compute terminal reward.

Task	size	Storm output	MCTS unsafe	MCTS MGS	MCTS EDF	Deep-Q unsafe	Deep-Q MGS	Deep-Q EDF
4S	10^5	0.38	0.52	NA	NA	0.56	NA	NA
5S	10^6	T.O.	0	NA	NA	0.13	NA	NA
10S	10^{18}	T.O.	0	NA	NA	0.96	NA	NA
simple	10^2	0	0.72	0	0	1.08	0.1	0
1H, 2S	10^4	0.07	0.67	0.14	0.28	0.24	0.11	0.22
1H, 3S	10^5	0.28	1.13	0.45	0.49	∞	0.47	0.47
2H, 1S	10^4	0	0.92	0	0.2	∞	0.02	0.3
2H, 5S	10^{10}	T.O.	3.44	1.93	2.14	∞	2.39	2.48
3H, 6S	10^{14}	T.O.	4.17	2.88	2.97	∞	3.42	3.47
2H, 10S	10^{22}	T.O.	0.3	0.03	0.03	∞	1.42	1.6
4H, 12S	10^{30}	T.O.	2.1	1.2	1.3	∞	2.68	2.87

Table 6.2: Comparison of MCTS and reinforcement learning.

Results

In the first column of Table 6.2, we describe the task systems that we consider. A description 2H, 5S refers to a task system with two hard tasks and five soft tasks, while 4S refers to a task system with four soft tasks and no hard tasks. The output of STORM for the smaller task systems is given in the third column. We report sizes of the MDPs, computed with STORM whenever possible. Otherwise we report an approximation of the size of the state space obtained by taking the product of $(c_i + 1)(a_i + 1)$ over the set of tasks, where c_i and a_i are the greatest elements in the support of the distributions $\mathcal{C}_i$ and $\mathcal{A}_i$. Recall that the size of the state space is exponential in the number of tasks in the system. In the columns where safety is not guaranteed, ∞ denotes an observed violation (a missed deadline for a hard task).

For MCTS, at every step we explore 500 nodes of the unfolding of horizon 30, and the value of each node is initialized using 100 uniform simulations. This computation takes 1-4 minutes in our Python implementation for different benchmarks, running on a standard laptop. It is reasonable to believe that a substantial speedup could be obtained with well-optimised code and parallelism. For deep Q-learning, we train each task system for 10000 steps. The implementation of deep-Q learning in the OPENAI respository uses the Adam optimizer [KB15]. The size of the replay buffer is set to 2000 and the learning rate used is 10^{-3}. The probability ϵ of taking a random action is initially set to 1. This parameter reduces over the training steps, and becomes equal to 0.02 at the end of the training. The

network used is a multi-layer perceptron which, by default, uses two fully connected hidden layers, each with 64 nodes. Since we are interested in mean-cost objective, the discount factor γ is set to 1. We observed that reducing the value of γ leads to poorer results. The values reported for both MCTS and deep Q-learning are obtained as an average cost over 600 steps.

While deep Q-learning provides good results for small task systems with 3-4 tasks with several thousands of states, this method does not perform well for the benchmarks with large number of tasks. We trained the task system with 10 soft tasks with deep Q-learning for several million steps, but the state space was found to be too large to learn a good strategy, and the resulting output produced a cost that is much higher than that observed with MCTS.

Overall, our experimental results show that MCTS consistently provides better results, in particular when the task systems are large, with huge state spaces. This can be explained by the fact that MCTS optimizes locally using information about multiple possible "futures" while deep Q-learning rather optimizes globally using information about the uniquely observed trace. We observe that the performance of MCTS with EDF advice is only slightly worse than MCTS with MGS advice. EDF guarantees safety and does not require computing the most general safe strategy, therefore it forms a good heuristic for systems with many hard tasks, where MGS computation becomes too expensive.

Chapter 7

Imitation learning

In previous chapters, we have described different strategies based on a combination of formal methods and Monte Carlo tree search algorithm that aim for the optimal expected total reward. These *expert* receding horizon strategies may be too costly to compute online. In order to reach on-the-fly computing times low enough for real-time control, we propose the use of learning techniques to train learned strategies, encoded as neural networks, with the goal of *imitating* a given expert strategy.

This can take different forms depending on the expert strategy σ. In general, we define a function $f_\sigma : S \times A \to \mathbb{R}$ encoding the strategy σ so that from state s, the decision made by σ is equivalent to choosing an action from $\arg\max_{a \in A}(f_\sigma(s, a))$ uniformly at random. Intuitively, f_σ is a scoring function that rates how good every action is from the current state. To learn a memoryless strategy σ, this function can output the expected total reward under σ, or a heuristic score approximating it as returned by MCTS for example. In this case, the advice is seen as a (non-deterministic) expert strategy to be imitated. This way, we suggest that a symbolic advice can also be imitated by a neural network that can then be used as a *neural advice* in MCTS.

The plan is to train a neural network to learn the function f_σ as an offline step, then to use it to speed up the computations of decision-time planning. Depending on the size of the set of actions, we can either learn a neural network that takes a state-action pair (s, a) as input and outputs a single value $f_\sigma(s, a)$ or a neural network that takes a state s as input and outputs a vector in $\mathbb{R}^{|A|}$ containing values for each available actions.

We address the following challenges:

- representing a state s and its corresponding values $(f_\sigma(s, a))_{a \in A}$ so that it is easily processable by the neural network,
- generating data for f_σ representative of the state-space,
- choosing an architecture for the neural network,
- comparing the learnt strategy and the expert strategy σ.

7.1 Training a neural network

We propose the use of convolutional neural networks which would take a state in the MDP as a *tensor* with each channels of the tensor representing different features extracted from the state.

Example 7.1 In PAC-MAN, each tensor representing a state has 7 channels to denote respectively the distribution of walls, food pills, position of Pac-Man, and for each direction, positions of the ghosts who are moving towards that direction. For example the channel representing the distribution of walls would be a matrix w_{ij} of the size of the grid where $w_{ij} = 1$ if there is a wall at the co-ordinate (i, j), and otherwise $w_{ij} = 0$. □

Normalization of data We considered different approaches for normalization to make the outputs between 0 and 1. For example,

- *Globally* scaling the values between 0 and 1 so that $\min_{s,a} f(s, a)$ becomes 0 and $\max_{s,a} f(s, a)$ becomes 1 after normalization.
- *Locally* scaling the values so that for all state s, $\min_a f(s, a)$ becomes 0 and $\max_a f(s, a)$ becomes 1.

We argue that this local normalization is sufficient to learn the strategy as it captures the ordering of the actions. Experimentally, local normalization performed better than global normalization. We also experimented with non-linear transformations [BC64; YJ00] but they did not improve learning performances in our settings.

Description of the network Our neural networks contain a 2D convolution layer with 3×3 filters, a flattening layer, few hidden dense layers with the ReLU activation function and a final dense layer with the sigmoid activation function. Training is performed using ADAM optimizer with mean squared error as loss function. To choose the optimal hyperparameters, *e.g.* the exact number of layers and their size or the number of filters, we use hyperparameter tuning in each setting. In particular, we relied on the Python library KERASTUNER [OMa+19].

7.2 Dataset aggregration : Formally sharp DAgger

Let us detail how to construct a set of data of the shape (x, y), where $x \in S$ is the input of the neural network and $y \in \mathbb{R}^{|A|}$ is its output encoding $(f_\sigma(x, a))_{a \in A}$. We argue for the use of formal methods in order to answer the two following challenges: how to get a representative set of input values x, and how to get good y values for this set of input.

Perfect data Note that an expert strategy generated by an exact method (such as mode checking) is ensured an expected payoff higher than any expert strategy generated from a heuristic approach like Monte Carlo tree search. In a sense, if one sees a heuristic approach as an approximation of the optimal strategy, the data obtained from heuristic strategies can be seen as a noisy version of data that would otherwise be "perfect", *i.e.* pairs (x, y) where y is a vector encoding the decisions of a strategy σ that is optimal.

Representative set of inputs In order to generate a dataset to train on, a classical method is to pick states and actions uniformly at random within the state-space and to evaluate f_σ on these inputs. For example, one can consider Frozen Lake states obtained by placing the walls, the holes, the target and the robot at random empty positions. However, a neural network trained from such a dataset may perform poorly for states that play a key role in the expected payoff of a strategy (*i.e.* states that represent crucial decisions), as such states may not be likely to be selected at random within the state-space. The DAgger (Dataset Aggregation) algorithm [RGB11], in contrast, offers a dataset generation method based on running simulations in order to get a more realistic view of the states frequently encountered in real plays. While this approach can be part of the answer, it may not provide sufficiently many datapoints on the crucial decisions mentioned before, that may be few and far-between.

We propose an algorithm named *sharp DAgger* that would detect these states, refine the training set and retrain the network. This is done by simulating the strategy using the learnt neural network on the MDP and finding *counter-examples* where the neural network is performing poorly by comparing the value given by the network and the value $f_\sigma(s)$ associated with the exact method.

In Algorithm 4, we present a method to train the neural network by an iterative process that generates new data for the training set. In the first iteration, we train a neural

network NN_0 from an initial training dataset DATASET and in later iterations, we add more interesting data-points in that set. Initially, one could either randomly generate a small amount of data or simulate the MDP by following an uniform strategy. In iteration i, starting from an initial state s_0 in the MDP, we simulate a fixed number of paths until a given horizon H. We extract from these paths the states for which the current neural network NN_i trained from DATASET fails to predict the correct values. We add them to our dataset, then train the next iteration of the neural network. The decision on when to stop the sharp DAgger loop is taken based on evaluations of the quality of the neural network NN_i at each iteration i.

Algorithm 4 Sharp Dataset Aggregation (Sharp DAgger)

Input:
- A function $f_\sigma : S \to \mathbb{R}^{|A|}$ encoding an expert strategy σ,
- a state $s_0 \in S$,
- a distance function d,
- a precision value $\epsilon \in \mathbb{R}$,
- a horizon $h \in \mathbb{N}$.

Output: A strategy σ_i that imitates the strategy σ

$i \leftarrow 0$
DATASET = initial dataset
NN_0 = neural network trained using DATASET
σ_0 = strategy extracted from NN_0
while $i \leq iters$ **do**
    Paths_i = paths simulated following σ_i from s_0 for h steps
    for state s in paths $p \in \mathsf{Paths}_i$ **do**
        if $d(\mathsf{NN}_i(s), f(s)) \geq \epsilon$ **then**
            Add $(s, f(s))$ to DATASET
        end if
    end for
    $i \leftarrow i + 1$
    NN_i = neural network trained using DATASET
    σ_i = strategy extracted from NN_i
end while
return σ_i

7.3 Evaluating a learnt strategy

In order to evaluate the trained neural network, a traditional approach for machine learning can report on a loss function for a test dataset. Alternatively, one can measure

the accuracy of the network by reporting how many times the resulting learnt strategy has differed from the expert strategy as a classifier. But this may not be sufficient to evaluate how the learnt strategy is performing on the MDP. In Frozen Lake, consider a learned strategy that returns the same action as the expert strategy for all states in the MDP, except for one state where the learnt strategy gives a bad action that leads to a hole. Even though the learnt strategy has an almost perfect accuracy, it would perform badly compared to the expert strategy in real plays, and could lead to much worse rewards on expectation.

As such, we argue for the use of statistical model checking to evaluate the expected reward of a (neural) strategy. In particular, we can use the approximate probabilistic model checking method [Hér+04] where we simulate a set of paths following the expert strategy on the one hand and the neural strategy on the other, then compare their average rewards on these paths.

The following theorem gives a theoretical bound on the number of simulations needed to get a *probably approximately correct* approximation of the real expected reward.

Theorem 7.1

Suppose for MDP M, there exists $a < b$ such that $a \leq \mathsf{Reward}^h_M(p) \leq b$ for all paths p in M. Let $\delta \in (0, 1]$ and $\epsilon \in (0, b-a]$. Then for a strategy σ, suppose we sample $n \geq \frac{(b-a)^2}{2\epsilon^2} \ln(\frac{2}{\delta})$ paths $p_1, p_2 \ldots p_n$ independently at random from a state s in the MDP M following the strategy σ. Let $\overline{r} = \frac{1}{n}\sum_{i=1}^{n} \mathsf{Reward}^h_M(p_i)$. Then,

$$\mathbb{P}^\sigma_s(|\overline{r} - \mathsf{Val}^h_M(s,\sigma)| \geq \epsilon) \leq \delta\,.$$

Proof We have n independent identically distributed random variables $\mathsf{Reward}^h_M(p_i)$ with expected value $\mathsf{Val}^h_M(s,\sigma)$.

Then, $\mathbb{E}^\sigma_s(\overline{r}) = \mathsf{Val}^h_M(s,\sigma)$. So, from Theorem 2.2, we will have,

$$\mathbb{P}^\sigma_s(|\overline{r} - \mathsf{Val}^h_M(s,\sigma)| \geq \epsilon) \leq 2\exp\left(-\frac{2n\epsilon^2}{(b-a)^2}\right)\,.$$

As $n \geq \frac{(b-a)^2}{2\epsilon^2} \ln(\frac{2}{\delta})$, we get that $\mathbb{P}^\sigma_s(|\overline{r} - \mathsf{Val}^h_M(s,\sigma)| \geq \epsilon) \leq \delta$. □

But in practice, we usually need much less number of simulations to achieve a good approximation. For example, Consider the Frozen Lake layout in Figure 7.1.

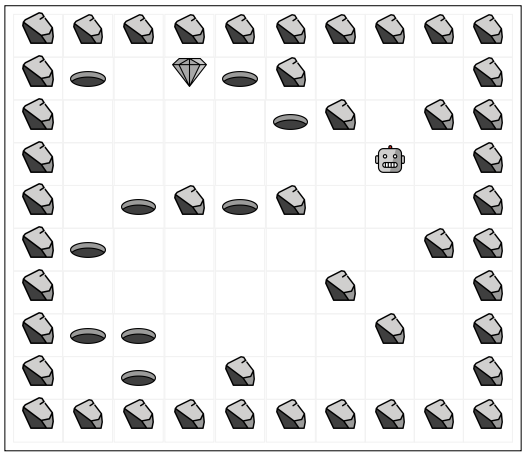

Figure 7.1: A 10×10 Frozen Lake layout.

Using exact methods we calculated the optimal expected reward in this grid to be 0.827. In Figure 7.2, for $n \in [1, 100]$, we independently simulated n paths using the optimal strategy and plotted the estimated reward obtained from statistical model checking. We see that we get a good approximation of the real expected reward with under 100 simulations.

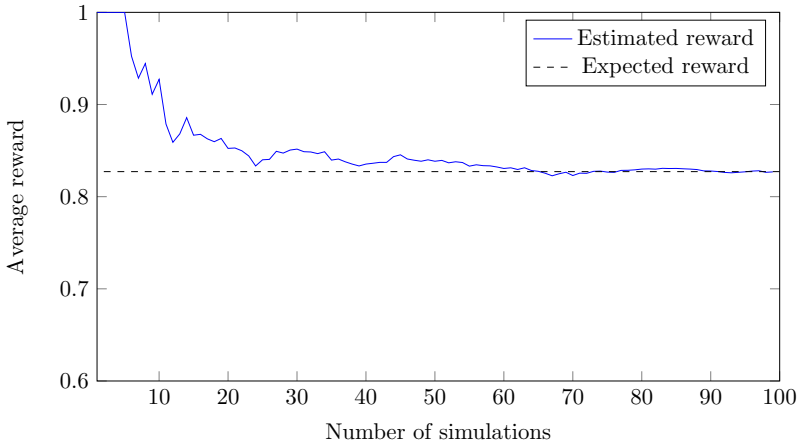

Figure 7.2: Statistical model checking for Frozen Lake

Chapter 8

Applications of imitation learning

In this Chapter, we report our experiments on two MDPs introduced in previous chapters. In Frozen Lake (Example 2.6), a robot moves in a slippery grid and has to reach the target while avoiding holes in the grid. This is an MDP that can be fully handled by model-checkers (using exact methods), and as such we use it to report on the benefits of using perfect data to train the surrogate strategy.

On the other hand, the PAC-MAN game (Example 1.1) provides more challenging MDPs to handle. There, we report on the performance of MCTS equipped with perfect or neural advice and on the performance of a surrogate strategy trained on data obtained from MCTS. The sharp DAgger algorithm (Algorithm 4) proves to be instrumental for learning efficiently in PAC-MAN.

8.1 Apllication 1: Frozen Lake

For the game described in Example 2.6, we randomly generated layouts of size 10x10 where we place walls at each cell in the border of the grid and with probability 0.1 at each of the other cells. Then we place holes in remaining cells with probability 0.1. Finally, we randomly place a target and an initial position in two of the remaining empty cells. If the game is neither won nor lost within 1000 steps, the game is considered a draw.

8.1.1 Expert strategies

Exact algorithm Consider the state in the MDP where the robot is on the target position. We label this state with $target$. Using the model checkers, we can compute the strategies $\mathsf{Opt}(s) = \arg\max_\sigma \mathbb{P}^\sigma_s(s \models \Diamond target)$ that maximizes the probability to reach the target t starting from state s. The practical strategy that we are interested in should not only maximize the probability to reach the target but also minimize the expected number of steps needed to reach the target. For a path ρ in MC M_σ, we define $\mathsf{len}(\rho, target) = i$ if $\rho[i]$ is

the target state and for all $j < i$, $\rho[j]$ is not the target set. Using formal methods techniques described in Algorithm 2 (Section 2.4), we can calculate a strategy in

$$\arg\min_{\sigma \in \mathsf{Opt}(s)} \mathbb{E}_s^\sigma(\mathsf{len}(\rho, target) \mid \rho \models \Diamond target)) .$$

This strategy is optimal for reaching the target with infinite horizon.

Monte Carlo tree search We compared the exact strategy with the strategy generated from Monte Carlo tree search with horizon $H = 30$. From state s, a search tree is constructed for 40 iterations. Thus, the search tree constructed by the MCTS algorithm contains up to 40 nodes. In each iteration of the MCTS algorithm, when a new node is added to the search tree, 10 samples are obtained by using a uniform strategy to estimate the value of the node.

8.1.2 Learnt strategies

Our training dataset contained $760k$ data-points which we used to imitate the expert strategies. Hyperparameter tuning resulted in neural networks containing a 2D convolution layer with 6 filters, a flattening layer and 2 dense layers.

8.1.3 Evaluation of the strategies

We randomly generated 1000 layouts and ran 100 games from each layout for 1000 steps using both expert strategies and the learnt strategies. If the robot does not reach the target or any hole within 1000 steps, we consider it as a drawn game. The average outcomes are reported in Figure 8.1. Using Storm, we calculate the optimal expected win rate to be 93% on average in the generated layouts. This value denotes the probability to reach the target eventually, using the optimal strategy. In practice, our statistical model checking approach requires fixing a finite horizon. Figure 8.1 confirms that horizon 1000 is sufficient as the expert strategy from Storm still reaches a win rate of 93%, same as the computed value with this horizon.

In comparison, our strategy learnt from Storm had a win rate of 81%. The expert strategy calculated using MCTS is suboptimal and showed a win rate of 77% while the strategy learnt from it has a win rate of 69%. This highlights the benefits of using exact

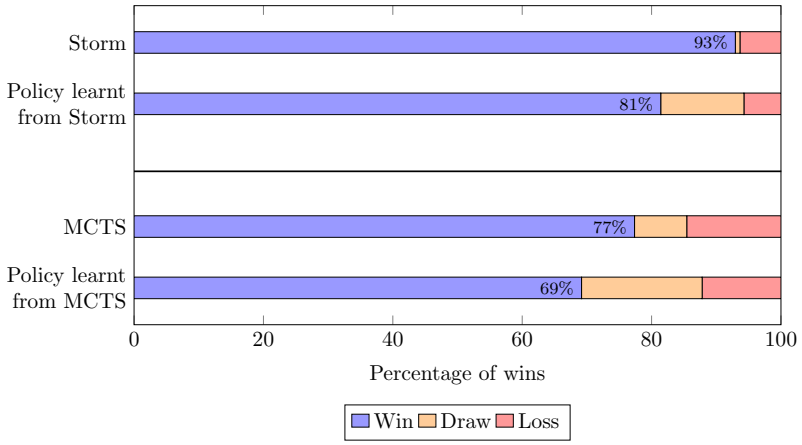

Figure 8.1: Perfect vs MCTS-based strategies for Frozen Lake.

methods to get noise-free data.

8.2 Application 2: Pac-Man

We performed our experiments on the game PAC-MAN in a grid of size 9×21 described in Figure 1.1. In our experiments, the ghosts always choose an action uniformly at random from the legal actions available. As explained in Section 6.2.1, we can view this as an MDP. Moreover, if Pac-Man does not win (eats all food pills) or lose (makes contact with a ghost) within 300 steps, we consider it a draw.

8.2.1 Expert strategies

Monte Carlo tree search The state-space of the MDP is too large to apply directly to find the optimal strategy. As a consequence, we decided to use Monte Carlo tree search with a receding horizon of $H = 10$. From state s, a search tree is constructed with a maximum depth of H for 40 iterations.We combined MCTS with the notion of advice as used in Section 6.2 in order to play Pac-Man. In each iteration of the MCTS algorithm, when a new node is added, 20 samples are obtained by using a uniform strategy to estimate the value of the node among the paths that are safe i.e. where Pac-Man is not eaten by a ghost. During the exploration of the search tree, we also restrict ourselves to actions a that maximize the probability to stay safe for the next 8 steps, *i.e.*, actions a such that $\eta_8(s, a) = \max_{a' \in A} \eta_8(s, a')$ as defined in Example 4.5. Since the online computation of

the η_8 function is too expensive to be done at every node of the search tree, we only restrict the root node of the tree to ensure the safety of the immediate decisions. We compare four different variants of MCTS here:

- a version without this expert (safety) advice,
- one where it is used at the root node of the tree,
- one where a neural advice is trained to imitate the safety advice and is used at the root node, and
- one where the neural advice is used at every node in the tree.

8.2.2 Neural advice

To speed up the MCTS procedure we train a neural network to imitate the safety advice. We used Algorithm 4 to create a dataset to train on. We use the L_∞ *metric* as the distance function with precision value $\epsilon = 0.2$ to find new data-points during the aggregation. In other words, we add the value $(s, (\eta_H(s,a))_{a\in A})$ to the dataset at the i^{th} iteration of sharp DAgger if

$$\max_{a\in A}(|\eta_H(s,a) - \mathsf{NN}_i(s,a)|) > 0.2\,.$$

In each iteration, we simulate 4000 games for 300 steps to generate Paths_i. We compare the safety status of the neural networks at each iteration of sharp DAgger in Figure 8.2.

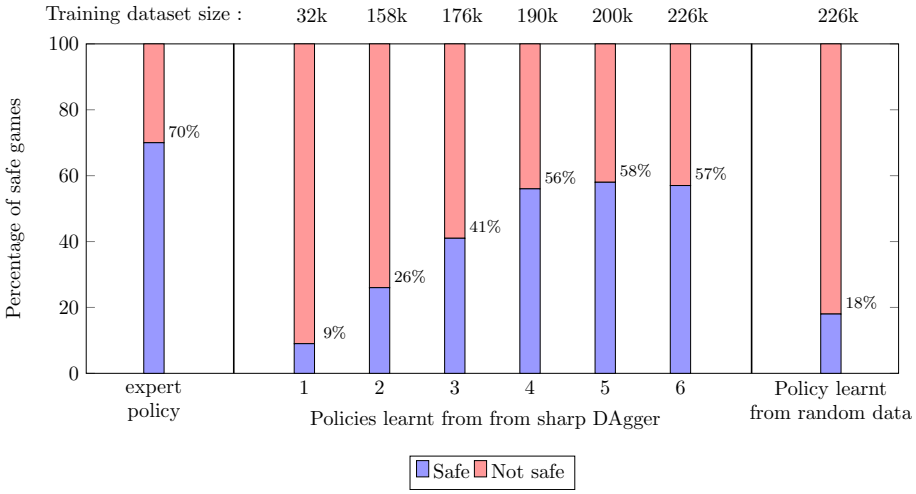

Figure 8.2: Sharp DAgger for Pac-Man neural advice

After 5 iterations, we observe that Pac-Man stays safe (for 300 steps) in 58% of games

when using the learnt strategy instead of staying safe in 70% of games with the strategy calculated from model checking. Hyperparameter tuning stabilized on neural networks using a 2D convolution layer with 6 filters, a flattening layer and 4 dense layers. The entire training dataset generated from sharp DAgger contains $226k$ data-points. To check the effectiveness of our method of data aggregation, we compare our learnt strategy with a strategy trained on $226k$ randomly generated data-points. We confirm that this learnt strategy performs much worse and stays safe in only 18% of games.

8.2.3 Using the neural advice in MCTS

To accommodate for the inherent noise in the output of the neural network NN, we fix a threshold $t = 0.9$ and consider the advice that allows almost-optimal actions with respect to t, *i.e.* the neural advice that restricts to actions a such that $\mathsf{NN}(s, a) \geq 0.9 \times \max_{a' \in A} NN(s, a')$.

We compare in Table 8.1 the performance of MCTS variants using expert or neural strategies as advice. We ran each setup on 100 games. The Python implementation of

Algorithm	win	loss	draw	food	score	time per step
MCTS	55	44	1	19.58	153.62	9.0 s
MCTS + expert selection advice	90	9	1	24.39	512.13	17.4 s
MCTS + neural selection advice (at root)	71	29	0	22.28	318.93	9.6 s
MCTS + neural selection advice (every node)	87	13	0	23.44	500.98	9.8 s

Table 8.1: Summary of experiments with different ghost models and algorithms. All the MCTS algorithm uses a simulation advice where we simulate only the safe paths.

MCTS that we rely on was not designed to optimize the performance in terms of computing time. In our case, the MCTS algorithm without any selection advice uses 9 seconds to decide on an action. Using the (formal methods based) expert advice at the root node of MCTS increases the time per decision by 8 extra seconds. While the 9 seconds spent in MCTS can be expected to be vastly lowered using code improvements,[1] the model checking done by Storm is already optimized. By replacing the expert advice with a neural advice, we can avoid this fixed cost of 8 seconds per decision, as the network can be consulted in 3 ms instead.

While the neural advice is not as good as the expert advice (it ensures win in 71% of

[1] MCTS and other simulation-based techniques are highly amenable to parallelism [CWH08].

games instead of 90% when used identically at the root node of the MCTS tree), we can afford to use it on every node of the search tree to dynamically prune the search. In this way, we can get an 87% win-rate that is the best of both worlds: we approach the win-rate of the expert advice with the computing time of the bare-bones version of MCTS.

8.2.3.1 Learning a surrogate strategy

We trained a surrogate neural network to imitate the expert strategy defined previously as MCTS with a neural advice at every node, that reached an 87% win-rate while keeping computing times as low as possible. To generate the dataset, we use our sharp DAgger algorithm and simulate 4000 games with horizon 300 in each iteration. To evaluate how well our strategies are performing, we compare the average number of wins obtained by following them in Figure 8.3.

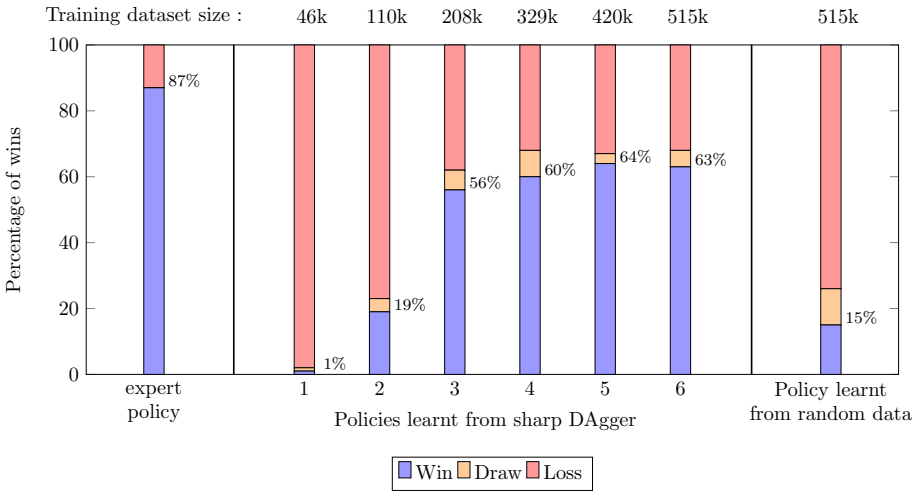

Figure 8.3: Sharp DAgger for Pac-Man surrogate strategies.

After 5 iterations, we reach a strategy with a win-rate of 64%, which is higher than the 55% of the "standard" version of MCTS, while having almost no need for online computing time as it is using a pre-trained neural network. Hyperparameter tuning stabilized on neural networks using a 2D convolution layer with 5 filters, a flattening layer, 5 dense layers. Finally, the training dataset generated with sharp DAgger contains $515k$ data-points. In comparison, a strategy learned from a randomly generated dataset of size $515k$ is only able to win in 15% of games, which confirms the importance of sharp DAgger in this setting.

Chapter 9

Conclusion

In this thesis, we study how to efficiently combine techniques from formal methods and learning for online computation of a strategy that aims at optimizing the expected long term reward in large systems modelled as Markov decision processes. This strategy is computed in a finite horizon unfolding using Monte Carlo tree search (MCTS) with a receding horizon. As a way to apply techniques used in formal methods during Monte Carlo tree search, we defined the notion of *advice* in Chapter 4. An advice, often symbolically written as a logical formula and computed on-the-fly, prunes part of the tree unfolding to differentiate 'good' and 'bad' part of the tree. A notable difference from related techniques like *shielding* is that building a shield requires one to construct the entire state-space of the MDP, whereas our approach performs its computations on-the-fly based on the current position alone. In Chapter 5, we augment the MCTS algorithm with advice and find sufficient conditions such that the guarantees regarding convergence rate and failure probability of classical Monte Carlo tree search are still maintained after this augmentation. Our version of MCTS uses two advice, one during the selection phase and another during the simulation phase.

We also report the implementations of these techniques using systems modelled as Markov decision precesses that may have 10^{20} states and beyond which cannot be handled with exact algorithms. In particular, in Section 6.2, we show how using advice based on formal methods significantly improves the performance of the Monte Carlo tree search algorithm in the game of Pac-Man, a game where Pac-Man has to eat food pills which avoiding contact with the ghosts.

We conclude that formal methods can provide good strategies and useful advice for MCTS, albeit at a high computational cost. For this reason, in Chapter 7, we show how to create a lower-latency neural advice by training a neural network to imitate an advice. This allows one to obtain the best of both worlds: the performance boost of the advice but without its computational cost. In Section 7.2, we propose a dataset-aggregation procedure called sharp DAgger (Algorithm 4) which leverages formal methods in order to obtain better quality data to train this neural network. We use statistical model checking to detect when additional samples are needed and generate these samples on demand when the performance of the learnt neural network does not match the quality of the strategy computed offline. In

Section 8.2, we report on the performance of MCTS equipped with neural advice and on the performance of a strategy trained on data obtained from MCTS using the sharp DAgger algorithm.

In Section 6.3, we showed how to safely and efficiently schedule tasks in the case where the structure of the system is known, but the exact distribution is not known. In this case, the system is first learnt by observing the jobs arriving in the system and then safe and (near)-optimal strategy is computed using MCTS with advice. Our experimental results show that MCTS consistently provides better results than deep Q-learning, in particular when the task systems are large, with huge state spaces.

The game Frozen Lake, where the robot moves in a slippery grid and has to reach the target which avoiding the holes, can be modelled using an MDP that can be fully handled by model-checkers using exact methods. The practical strategy that we are interested in should not only maximize the probability to reach the target but also minimize the expected number of steps needed to reach the target. This can be computed by exact algorithm (Algorithm 2) described in Section 2.4. In Section 8.1, we use this game to report on the benefits of using perfect data obtained from model checkers to train a neural network.

9.1 Future works

As future works, we are working on publishing our framework as a tool to apply the methods discussed on additional case studies. We consider looking at cases where we need to find safe and (near)-optimal strategies in larger systems, *e.g.*, cyber-physical systems where safety is highly crucial and hence use of advice based on formal methods is needed to enforce safety. These systems are often represented by hybrid models more complicated than discrete Markov decision processes and the Monte Carlo algorithms described in this thesis would need some modifications to work in these settings. For example, in [Jen+22], the authors discussed different simulation strategies that are needed for MCTS in priced timed automata.

Future work could be interesting to implement a framework to automate the extraction of SAT-based symbolic advice from the MDP. For that purpose, we have tried constructing an and-inverter graph (AIG) [BHW11] from the symbolic description of the MDP, from which, a SAT and a QBF formula can be generated. But in our approach, the graphs created were too big to process. Given an advice formulated as a Boolean formula, the

SAT sampling-based techniques are highly efficient in computing samples satisfying that formula. But in our experiments, the reject-based approach was faster at finding the samples we needed. Future work may involve cases where SAT sampling-based methods are essential as the paths satisfying the formula are too few to be found just by random sampling.

Bibliography

[Abr87] Bruce Abramson. *The expected-outcome model of two-player games*. 1987. URL: http://academiccommons.columbia.edu/download/fedora_content/download/ac:142327/CONTENT/CUCS-315-87.pdf.

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2-3 (2002), pp. 235–256. DOI: 10.1023/A:1013689704352.

[AD99] Robert B. Ash and Catherine A. Doleans-Dade. *Probability and Measure Theory*. 2nd edition. Harcourt Academic Press, 1999.

[Aga+22] Chaitanya Agarwal et al. "PAC Statistical Model Checking of Mean Payoff in Discrete- and Continuous-Time MDP". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 3–25. DOI: 10.1007/978-3-031-13188-2_1.

[Als+18] Mohammed Alshiekh et al. "Safe Reinforcement Learning via Shielding". In: *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI 2018)*. AAAI Press, 2018, pp. 2669–2678.

[Ash+17] Pranav Ashok et al. "Value iteration for long-run average reward in Markov decision processes". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 201–221.

[Ash+18] Pranav Ashok et al. "Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Vol. 11245. Lecture Notes in Computer Science. Springer, 2018, pp. 322–335. DOI: 10.1007/978-3-030-03421-4_21.

[Avn+19] G. Avni et al. "Run-Time Optimization for Learned Controllers Through Quantitative Games". In: *CAV*. 2019, pp. 630–649.

[BB12] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[BC64] George EP Box and David R Cox. "An analysis of transformations". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 26.2 (1964), pp. 211–243.

[BCJ18] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. "Graph Games and Reactive Synthesis". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 921–962. DOI: 10.1007/978-3-319-10575-8_27.

[BCR20] Damien Busatto-Gaston, Debraj Chakraborty, and Jean-François Raskin. "Monte Carlo Tree Search Guided by Symbolic Advice for MDPs". In: *31st International Conference on Concurrency Theory, CONCUR 2020,* ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 40:1–40:24. DOI: 10.4230/LIPIcs.CONCUR.2020.40.

[Bel57a] Richard Bellman. "A Markovian decision process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. URL: http://www.jstor.org/stable/24900506.

[Bel57b] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[BHW11] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 And Beyond*. Tech. rep. 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[Bie+06] Armin Biere et al. "Linear Encodings of Bounded LTL Model Checking". In: *Logical Methods in Computer Science* Volume 2, Issue 5 (Nov. 2006). DOI: 10.2168/LMCS-2(5:5)2006.

[BJW02] Julien Bernet, David Janin, and Igor Walukiewicz. "Permissive strategies: from parity games to safety games". In: *RAIRO-Theoretical Informatics and Applications* 36.3 (2002), pp. 261–275.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BL69] J. Richard Buchi and Lawrence H. Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311. ISSN: 00029947. URL: http://www.jstor.org/stable/1994916.

[Bla62] David Blackwell. "Discrete dynamic programming". In: *The Annals of Mathematical Statistics* (1962), pp. 719–726.

[Bla65] David Blackwell. “Discounted dynamic programming”. In: *The Annals of Mathematical Statistics* 36.1 (1965), pp. 226–235.

[Boh+12] Aaron Bohy et al. “Acacia+, a Tool for LTL Synthesis”. In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 652–657. ISBN: 978-3-642-31424-7.

[Brá+14] Tomás Brázdil et al. “Verification of Markov Decision Processes Using Learning Algorithms”. In: *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA 2014)*. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 98–114. DOI: 10.1007/978-3-319-11936-6_8.

[Bre+14] Romain Brenguier et al. “AbsSynthe: abstract synthesis from succinct safety specifications”. In: *Proceedings 3rd Workshop on Synthesis, SYNT 2014*. Ed. by Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha. Vol. 157. EPTCS. 2014, pp. 100–116. DOI: 10.4204/EPTCS.157.11.

[Bro+12] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[Bus+21] Damien Busatto-Gaston et al. “Safe Learning for Near-Optimal Scheduling”. In: *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021,* ed. by Alessandro Abate and Andrea Marin. Vol. 12846. Lecture Notes in Computer Science. Springer, 2021, pp. 235–254. DOI: 10.1007/978-3-030-85172-9_13.

[But11] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[BW15] Hendrik Baier and Mark H. M. Winands. “MCTS-Minimax Hybrids”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 7.2 (2015), pp. 167–179. DOI: 10.1109/TCIAIG.2014.2366555.

[Cau+47] Augustin Cauchy et al. “Méthode générale pour la résolution des systemes d’équations simultanées”. In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[CDS19] Arthur Clavière, Souradeep Dutta, and Sriram Sankaranarayanan. "Trajectory Tracking Control for Robotic Vehicles Using Counterexample Guided Training of Neural Networks". In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018*. Ed. by J. Benton et al. AAAI Press, 2019, pp. 680–688. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/3555.

[CE81] Edmund M Clarke and E Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on logic of programs*. Springer. 1981, pp. 52–71.

[Cha+05] Hyeong Soo Chang et al. "An adaptive sampling algorithm for solving Markov decision processes". In: *Operations Research* 53.1 (2005), pp. 126–139.

[Cha+14] Supratik Chakraborty et al. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *Proceedings of the 28th AAAI Conference on Artificial Intelligence, 2014 (AAAI 2014)*. AAAI Press, 2014, pp. 1722–1730. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364.

[Cha+15] Supratik Chakraborty et al. "On Parallel Scalable Uniform SAT Witness Generation". In: *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 304–319. DOI: 10.1007/978-3-662-46681-0_25.

[Cha+17a] K. Chatterjee et al. "Optimizing Expectation with Guarantees in POMDPs". In: *AAAI*. 2017, pp. 3725–3732.

[Cha+17b] Krishnendu Chatterjee et al. "Optimizing Expectation with Guarantees in POMDPs". In: *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI 2017)*. AAAI Press, 2017, pp. 3725–3732.

[Cha12] K. Chatterjee. "Robustness of Structurally Equivalent Concurrent Parity Games". In: *FOSSACS*. 2012, pp. 270–285.

[Cho+15] François Chollet et al. *Keras*. https://keras.io. 2015.

[Chu57] Alonzo Church. "Application of recursive arithmetic to the problem of circuit synthesis". In: *Journal of Symbolic Logic* 28.4 (1957).

[Cou06] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[CWH08] Guillaume M. J. -B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. "Parallel Monte-Carlo Tree Search". In: *Proceedings of the 6th International Conference on Computers and Games (CG 2008)*. Ed. by H. Jaap van den Herik et al. Vol. 5131. Lecture Notes in Computer Science. Springer, 2008, pp. 60–71. DOI: 10.1007/978-3-540-87608-3_6.

[Cyb89] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[Dac+16] Przemyslaw Daca et al. "Faster Statistical Model Checking for Unbounded Temporal Properties". In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 112–129. DOI: 10.1007/978-3-662-49674-9_7.

[Deh+17] Christian Dehnert et al. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Computer Aided Verification - 29th International Conference, CAV 2017,* ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 592–600. DOI: 10.1007/978-3-319-63390-9_31.

[Dha+17] P. Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[DK] John DeNero and Dan Klein. *CS 188 : Introduction to Artificial Intelligence*. URL: https://inst.eecs.berkeley.edu/~cs188.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GGR18] Gilles Geeraerts, Shibashis Guha, and Jean-François Raskin. "Safe and Optimal Scheduling for Hard and Soft Tasks". In: *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018,* ed. by Sumit Ganguly and Paritosh K. Pandya. Vol. 122. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 36:1–36:22. DOI: 10.4230/LIPIcs.FSTTCS.2018.36.

[GKW82] Kamal Golabi, Ram B Kulkarni, and George B Way. "A statewide pavement management system". In: *Interfaces* 12.6 (1982), pp. 5–21.

[HAK20] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. "Cautious Reinforcement Learning with Logical Constraints". In: *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, (AAMAS 2020)*. 2020, pp. 483–491. URL: https://dl.acm.org/doi/abs/10.5555/3398761.3398821.

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[Hen+22] Christian Hensel et al. "The probabilistic model checker Storm". In: *International Journal on Software Tools for Technology Transfer* 24.4 (2022), pp. 589–610.

[Hér+04] Thomas Hérault et al. "Approximate probabilistic model checking". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2004, pp. 73–84.

[Her+18] Michael Hertneck et al. "Learning an Approximate Model Predictive Controller With Guarantees". In: *IEEE Control. Syst. Lett.* 2.3 (2018), pp. 543–548. DOI: 10.1109/LCSYS.2018.2843682. URL: https://doi.org/10.1109/LCSYS.2018.2843682.

[Hin+12] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[HJ94] Hans Hansson and Bengt Jonsson. "A logic for reasoning about time and reliability". In: *Formal aspects of computing* 6.5 (1994), pp. 512–535.

[Hoe63] Wassily Hoeffding. "Probability Inequalities for Sums of Bounded Random Variables". In: *Journal of the American Statistical Association* 58.301 (1963), pp. 13–30. DOI: 10.1080/01621459.1963.10500830.

[How60] Ronald A Howard. "Dynamic programming and markov processes." In: (1960).

[Iva+19] Radoslav Ivanov et al. "Verisig: verifying safety properties of hybrid systems with neural network controllers". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 2019, pp. 169–178.

[Jan+14] Nils Jansen et al. "Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)". In: *31st International Conference on Concurrency Theory, CONCUR 2020*. Vol. 171. 2014, 3:1–3:16. DOI: 10.4230/LIPIcs.CONCUR.2020.3.

[Jen+22] Peter Gjøl Jensen et al. "Monte Carlo Tree Search for Priced Timed Automata". In: *Quantitative Evaluation of Systems*. Ed. by Erika Ábrahám and Marco Paolieri. Cham: Springer International Publishing, 2022, pp. 381–398. ISBN: 978-3-031-16336-4.

[JV+22] Sebastian Junges, Matthias Volk, et al. *Stormpy*. https://github.com/moves-rwth/stormpy. 2022.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KB15] Diederik P. Kingma and Jimmy Ba. "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*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[KH06] Wook Hyun Kwon and Soo Hee Han. *Receding horizon control: model predictive control for state models*. Springer Science & Business Media, 2006.

[KM17] Jan Křetínský and Tobias Meggendorfer. "Efficient strategy iteration for mean payoff in Markov decision processes". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2017, pp. 380–399.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[KPR18] Jan Křetínský, Guillermo A. Pérez, and Jean-François Raskin. "Learning-Based Mean-Payoff Optimization in an Unknown MDP under Omega-Regular Constraints". In: *Proceedings of the 29th International Conference on Concurrency Theory (CONCUR 2018)*. Vol. 118. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 8:1–8:18.

[KS06] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293. DOI: 10.1007/11871842_29.

[Lem12] Claude Lemaréchal. "Cauchy and the gradient method". In: *Doc Math Extra* 251.254 (2012), p. 10.

[LR85] T.L Lai and Herbert Robbins. “Asymptotically Efficient Adaptive Allocation Rules”. In: *Adv. Appl. Math.* 6.1 (Mar. 1985), pp. 4–22. ISSN: 0196-8858. DOI: 10.1016/0196-8858(85)90002-8.

[LT93] Nancy G Leveson and Clark S Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (1993), pp. 18–41.

[McN65] Robert McNaughton. “Finite-state infinite games”. In: *Project MAC Rep* (1965).

[MN81] J.-F. Mertens and A. Neyman. “Stochastic games”. In: *International Journal of Game Theory* 10.2 (1981), pp. 53–66.

[Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[MP43] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[Mun14] Rémi Munos. “From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning”. In: *Foundations and Trends in Machine Learning* 7.1 (2014), pp. 1–129. DOI: 10.1561/2200000038.

[OMa+19] Tom O’Malley et al. *KerasTuner*. https://github.com/keras-team/keras-tuner. 2019.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: 10.1002/9780470316887.

[QS82] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on programming*. Springer. 1982, pp. 337–351.

[RB10] Stéphane Ross and Drew Bagnell. “Efficient reductions for imitation learning”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 661–668.

[RGB11] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. “A reduction of imitation learning and structured prediction to no-regret online learning”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 627–635.

[Ros10] Sheldon M. Ross. *Introduction to probability models*. 10th edition. Amsterdam Heidelberg: Elsevier, 2010. ISBN: 978-0-12-375686-2.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sil+16] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[SL94] Roberto Segala and Nancy Lynch. "Probabilistic simulations for probabilistic processes". In: *International Conference on Concurrency Theory*. Springer. 1994, pp. 481–496.

[Sol03] E. Solan. "Continuity of the value of competitive Markov decision processes". In: *Journal of Theoretical Probability* 16 (2003), pp. 831–845.

[Val84] Leslie G. Valiant. "A Theory of the Learnable". In: *Commun. ACM* 27.11 (1984), pp. 1134–1142.

[WD92] C. J. C. H. Watkins and P. Dayan. "Technical Note Q-Learning". In: *Machine Learning* 8 (1992), pp. 279–292.

[YJ00] In-Kwon Yeo and Richard A Johnson. "A new family of power transformations to improve normality or symmetry". In: *Biometrika* 87.4 (2000), pp. 954–959.

[YS02] Håkan LS Younes and Reid G Simmons. "Probabilistic verification of discrete event systems using acceptance sampling". In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 223–235.

Index