Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems

Adrien Couetoux

▶ To cite this version:

Adrien Couetoux. Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems. Data Structures and Algorithms [cs.DS]. Université Paris Sud - Paris XI, 2013. English. NNT : . tel-00927252

HAL Id: tel-00927252
https://theses.hal.science/tel-00927252v1

Submitted on 12 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems

by

Adrien Couëtoux

Submitted to the Departement d'Informatique de l'Universite Paris Sud
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

UNIVERSITE PARIS SUD

September 2013

© Universite Paris Sud 2013. All rights reserved.

Author . . . .
Departement d'Informatique de l'Universite Paris Sud
September 30th, 2013

Certified by . . . .
Olivier Teytaud
Chargé de recherche
Thesis Supervisor

Accepted by . . . .
Nicole Bidoit
Directrice de l'école doctorale d'informatique de Paris Sud (EDIPS)

Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems

by

Adrien Couëtoux

Submitted to the Departement d'Informatique de l'Universite Paris Sud
on September 30th, 2013, in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

In this thesis, I studied sequential decision making problems, with a focus on the unit commitment problem. Traditionnaly solved by dynamic programming methods, this problem is still a challenge, due to its high dimension and to the sacrifices made on the accuracy of the model to apply state of the art methods. I investigated on the applicability of Monte Carlo Tree Search methods for this problem, and other problems that are single player, stochastic and continuous sequential decision making problems. In doing so, I obtained a consistent and anytime algorithm, that can easily be combined with existing strong heuristic solvers.

Thesis Supervisor: Olivier Teytaud
Title: Chargé de recherche

Acknowledgments

I am immensely grateful for the support I have received during these past three years. I have been lucky to be surrounded by a loving family, true friends, brilliant colleagues, and an exceptional advisor.

I want to thank Olivier Teytaud for having introduced me to the wonderful and intimidating world of research. The more time passes, the more I realize how great he is at mentoring students. I have had the pleasure to work *with* him, and not *for* him. By treating me as an equal and listening to all my ideas and questions, Olivier helped me develop curiosity and critical thinking, without the fear of making mistakes.

I thank Louis Wehenkel and Philippe Preux for their thorough review of my manuscript and their comments; they made this manuscript much more readable than it originally was. I also thank Nicolas Sabouret, Tristan Cazenave, Olivier Buffet, Simon Lucas and Marc Schoenauer for being members of the comity, and for their questions.

I want to thank all the member of the TAO team, at INRIA Saclay, for the wonderful work environment they all contributed to build. Thank you Fabien, Romaric, and Jean-Baptiste for having taken good care of me when I first joined the team. Many thanks to Hassen, who helped me many, many times to perform experiments on more cores than I count, and helping the rookie I was to deal with terminal command lines. Thanks also to Anne and Nikolaus for training me during my initial internship. Thank you Marc and Michèle for the insightful suggestions and advice; in particular for the preparation of my defense. Thanks to everyone who made the coffee race a real challenge: Jialin, Marie-Liesse, Sandra, Jean-Marc, and Philippe. I will also never thank Marie Carol enough for her outstanding work as the secretary of the team. I owe most of the experimental results here to Grid5000.

Special thanks also to Damien Ernst, who through the years, despite the distance, has always been there for me, whether I needed scientific advice or professional guidance.

This work would also not have been what it is if I had not visited National University of Tainan and Dong Hwa University, for two semesters. The welcome I received there was beyond any expectations, thanks to Prof. Lee, Prof. Yen, the OASE lab, Ada, Cherry, Tom, Yuki, Orlando, and Meng-Zheng. I also owe my survival there to Maggie, my amazing Chinese teacher.

I would not have kept my sanity without my friends, who magnificently dealt with the person I was at all times, including the unpredictable, stressed and needy deadline-minus-two-days me. In alphabetical order, thanks to: Agnès G., Alice G., Armel P., Benjamin B., Erika P., Fabien D., François S., Grégory L., Guillaume L., Julie P., Julien P., Julien L., Marc H., Marjorie W., Mica P., Olivier B., Pauline G., Pierre G., Sébastien M., Solène G., Solenne B., Thibault V., Thomas B., Vincent G., Viviane N., Yixin C. Thanks to Sarah for joining me on this ride and supporting me through its most difficult times.

Enfin, je remercie ma famille, qui m'a toujours soutenu durant mes études, psychologiquement et financièrement. Merci à mes parents Franceline et Loïc, mon frère Basile.

Contents

1 Introduction **11**

1.1 Outline of the manuscript . . . 11

1.2 Motivation: energy stock management . . . 12

1.2.1 Hydrothermal Scheduling (HS) . . . 12

1.2.2 Challenging features . . . 14

1.2.3 Multi-stock formulation . . . 16

1.3 Modelling sequential decision making problems . . . 17

1.3.1 Decision epochs . . . 18

1.3.2 State variables . . . 19

1.3.3 Action variables . . . 20

1.3.4 Random processes . . . 20

1.3.5 Transition function . . . 21

1.3.6 Reward function . . . 22

2 Algorithms for Sequential Decision Making **23**

2.1 Decision making under uncertainty . . . 23

2.1.1 Policies . . . 24

2.1.2 Value function . . . 25

2.1.3 Action-Value function. . . . 26

2.2 The finite case . . . 26

2.2.1 Bellman equations . . . 26

2.2.2 Basic Dynamic Programming algorithm . . . 27

2.2.3 Value iteration . . . 28

2.2.4 Policy iteration . . . 29

2.2.5 Q learning . . . 30

2.2.6 SARSA . . . 30

2.3 The large scale case: RL and approximate DP . . . 32

2.3.1 Approximate Value Iteration . . . 33

2.3.2 Approximate Policy Iteration . . . 36

2.3.3 Policy search . . . 39

2.3.4 Actor critic methods . . . 39

2.3.5 Mathematical programming for unit commitment . . . 40

3 Continuous MCTS **43**
3.1 Finite MCTS 43
3.1.1 Algorithm description 44
3.1.2 Improvements in MCTS applied to games 49
3.2 Infinite domain: continuous MCTS 50
3.2.1 MCTS-SPW for infinite action space 51
3.2.2 Why MCTS-SPW fails on infinite state space 53
3.2.3 Proposed solution: DPW 54
3.2.4 The Trap problem 56
3.2.5 Conclusion 57
3.3 Exploring infinite action spaces 58
3.3.1 Existing methods 58
3.3.2 Our contribution: Blind Value (BV) 59
3.3.3 Experimental comparison 61
3.3.4 Conclusion 63
3.4 Generalization: continuous RAVE 64
3.4.1 Rapid Action Value Estimation: the finite case 64
3.4.2 Continuous Rapid Action Value based Estimation 65
3.4.3 Experimental Validation 66

4 Theoretically consistent MCTS **73**
4.1 Specification of the Markov Decision Tree setting 73
4.2 Specification of the Polynomial Upper Confidence Tree algorithm . . 74
4.3 Main result 76
4.4 From Decision Nodes to Random Nodes 77
4.4.1 Children of Random Nodes are selected almost the same number of times 77
4.4.2 Consistency of Random Nodes 78
4.5 From Random Nodes to Decision nodes 80
4.5.1 Children of decision nodes are selected infinitely often 80
4.5.2 Decision nodes are consistent 81
4.6 Base step, initialization and conclusion of the proof 85
4.7 Experimental validation 86
4.8 Conclusion 87

5 Hybridization of MCTS **88**
5.1 Custom default policies 88
5.1.1 Introduction 88
5.1.2 Algorithms 89
5.1.3 Experiments 90
5.1.4 Conclusion 95
5.2 Mixing MCTS with heuristics to solve POMPD 96
5.2.1 Belief state estimation, from Mines to mathematics 96
5.2.2 MCTS with belief state estimation 100
5.2.3 Experimental results 101

5.2.4 Conclusions 102
5.3 A Meta-bandit framework 103
5.3.1 Review of multi-armed bandit formulas 103
5.3.2 Energy management: comparing different master plans 105
5.3.3 Discussion 106

6 Backup operators for SDM 108
6.1 State of the art 108
6.1.1 Related work and how it relates with MCTS 108
6.1.2 When asymptotic results are not enough 109
6.2 Finding the right balance between optimism and conservatism 111
6.2.1 Alternatives to UCB 111
6.2.2 Empirical analysis 114
6.3 Conclusions 117

7 Conclusion 120

List of Figures

1-1 Hydroelectric Scheduling problem, with one stock and one thermal power plant. Random processes are shown in orange. G represents the power grid, where produced electricity is dispatched to satisfy the demand. 14

1-2 Hydroelectric Scheduling problem, with multiple stocks and one thermal power plant. Random processes are shown in orange. G represents the power grid, where produced electricity is dispatched to satisfy the demand. 17

3-1 Generic tree structure for sequential decision making under uncertainty. In this figure, x is the initial state, the root of the tree. One of the explored feasible actions from x is a. One of the explored possible outcome of the pair (x, a) is x'. 45

3-2 An example of the four phases of MCTS, from (i) to (iv). The orange color is used to indicate the parts of the tree that are used/modified at each phase. 48

3-3 Shape of the reward function: Trap problem. 57

3-4 Mean of the reward, for the trap problem with $a = 70$, $h = 100$, $l = 1$, $w = 0.7$, $R = 0.01$. The estimated standard deviations of the rewards are $STD_{DPW} = [13.06, 12.88, 12.88, 12.06, 14.70, 0, 0]$ for Double PW and $STD_{SPW} = [7.16, 7.16, 8.63, 9.05, 0, 0, 0]$ for Simple PW - the differences are clearly significant, where STD means standard deviation. 58

3-5 Median of the reward, for the trap problem with $a = 70$, $h = 100$, $l = 1$, $w = 0.7$, $R = 0.01$ 59

3-6 Reward, as a function of the computation time. Problem settings: 12 stocks, 16 time steps. MTCS with BV is 10 times faster than MCTS, for budgets up to 10 seconds per decision. 62

3-7 Reward, as a function of the number of simulations per decision. Problem settings: 80 stocks, 6 time steps. MTCS with BV is 10 times faster than MCTS, for all budgets. 63

3-8 The treasure hunt benchmark problem involves two options: the presence of a hole in the middle of the arena (left) and a probabilistic transition setting (right). 68

3-9 Treasure hunt with 15×15 arena, without hole (top: deterministic transitions; middle and bottom: probabilistic transitions with respectively $\epsilon = .5$ and 1). The speed up is only significant in the case of $\epsilon = 1$, where it makes the convergence about 3 times faster. 69
3-10 Treasure hunt with 5×5 arena, with hole (top: deterministic transitions; middle and bottom: probabilistic transitions with respectively $\epsilon = .5$ and 1). In this case, the speed up is quite significant, with MCTS+RAVE being about 10 times faster than vanilla MCTS. . . . 70
3-11 Comparative performances of UCT,discrete RAVE, $cRAVE_{state}$, $cRAVE_{action}$ and $cRAVE_{action,state}$ on the energy management problem, versus the computational budget (number of simulations). The upper the better. 71

5-1 Performances of different variants of MCTS on the 1 river unit commitment problem (left) and the binary rivers (right), with 7 stocks, 24 time steps. Y axis shows the reward (the higher the better). 93
5-2 Performances of different variants of MCTS on the randomly connected unit commitment problem (7 stocks, 24 time steps).Y axis shows the reward (the higher the better). 94
5-3 Results on the energy investment problem. The five DPS curves (curves 1 to 5) are very close to each other; results are better than for the heuristic alone, and versions without the heuristic are almost the same as versions with the heuristic. The MCTS+DPS+neural network was the most efficient strategy, outperforming MCTS+DPS+neural network+heuristic. The sums of Gaussians require more time for learning, hence the poor results for moderate budgets. 95
5-4 A case in which choosing the next move is non-trivial. 97
5-5 Left: here, you can deduce that one of the remaining locations is a mine. If you play in the middle location, you have an expected number of (unique) moves before losing which is, if you play perfectly, 1 (the three outcomes, losing immediately, or losing after 1 move, or completely solving the game, are equally likely) - whereas playing the top or bottom unknown location gives an expected number of (unique) moves 4/3 (with probability $\frac{1}{3}$, it's an immediate loss - otherwise, it's a complete solving). Middle: a situation with 50% probability of winning. Right: this situation is difficult to analyze mathematically; our program immediately sees that the top-right location (0,6) is a mine. However, this could easily (and faster) be found by a branch-and-bound optimization. More importantly, it also sees that the location just below (1,6) is good; but the real good point is that it can say which locations are more likely to lead to a long-term win than others. . . . 98

6-1 Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. 110

6-2 Reward function shape for the trap problem with a crash probability. x is the state variable; $0 < d < 1$ is the difficulty parameter; H is the number of time steps; $d(H-1)$ is the length between the initial state 0 and the gap; d is the width of the gap; the reward is obtained at the final time step, can be equal to a, $-g$, h, or $-K$; if the final state x_f is larger than $dH-1$, there is a probability p of 'crashing' and getting a reward of $-K$. 115
6-3 MCTS-DPW set to 0.25. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1. 117
6-4 MCTS-DPW set to 0.50. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1. 118
6-5 MCTS-DPW set to 0.75. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1. 119

List of Tables

4.1 **Definition of coefficients and convergence rates** 76

4.2 Left: Cart Pole results; episodes are 200 time steps long. Right: Unit Commitment results, with 2 stocks, 5 plants, and 6 time steps. 86

5.1 Results of various implementations on the MineSweeper games. Results are averaged over 10^5 games except 16x30 which is averaged over 10^4 games. For OH, results are obtained with 10000 simulations per move, except expert mode and intermediate mode (99 mines on 16x30 and 40 mines on 16x16) which use 100 simulations per move. BSSUCT is not documented for cases in which it was too slow for being operational in [107]. 102

5.2 Left: "real" value of each arm; computed by giving MCTS a large budget (100s). Right: expected simple regret of different bandit algorithms, as a function of the budget allowed to arm plays and to T. . . 107

Chapter 1

Introduction

In this chapter, we first present the outline of the manuscript. We then introduce the kind of real world application that has been the initial motivation driving our research work: energy stock management problems. Then, we show how one can model such Sequential Decision Making (SDM) problems, and the methods developed to solve them. The way people define SDM problems can vary from one community to the other (eg. from the Operations Research (OR) community to the Reinforcement Learning (RL) community), and we will actually use sources from both.

1.1 Outline of the manuscript

This first chapter presents the framework and the motivation of our work, along with essential definitions for sequential decision making, like state variables.

The second chapter is our attempt at explaining the state of the art in sequential decision making. We describe some of the most famous algorithms, for finite and infinite problems. We also mention the most popular methods used for the application that initially motivated this work, the management of energy stocks.

The third chapter covers our main contributions to Monte Carlo Tree Search (MCTS), after presenting it in its vanilla form (i.e. for finite and deterministic problems). Our contributions can be summarized as follows: we extend MCTS to continuous domains with a trick called Double Progressive Widening (DPW)[39] (I implemented the algorithm, made experiments and participated in the writing of the paper); we show a method called Blind Value (BV)[38] that is intended to improve the exploration of an unknown and large set of actions (I designed the BV heuristic, did the experimental validation, and participated in the writing of the paper); we extend the RAVE heuristic to continuous setting[40] (I did part of the experimental validation and writing of the paper).

The fourth chapter contains the formal proof of consistency of continuous MCTS with polynomial exploration[12]. I participated in the proof of the main result, and modified the paper after an initial rejection (by writing a comprehensive state of the art section, adding experimental validation, and shortening and clarifying the paper for a conference format).

The fifth chapter is about the possible hybridisations of MCTS with other methods or applications. First, we show a promising framework to use Direct Policy Search (DPS) to improve the default policy of MCTS in an agnostic way[42]. I participated in the design of the framework, writing of the paper and experimental validation. Then, we show how MCTS can be used to solve Partially Observable Markov Decision Processes (POMDP), with an application to the game of Mine Sweeper[41]. I participated to the adaptation of the code of MCTS to the POMDP setting. Some more experimental results have been obtained later by [108], and are included in this chapter. Finally, we present a meta-bandit framework. We compare various bandit algorithms, to choose among a set of investments (i.e. arms), where each arm is evaluated by running MCTS to measure the value of the corresponding investment. This work has been submitted and is currently under review.

The sixth chapter presents a study of alternatives to the classical Upper Confidence Bound (UCB) formula, in the context of MCTS. We include alternative ways to back up the rewards in the tree; some of them have been explored in the finite setting, and we study them in the continuous setting. We obtain promising preliminary results, and insights about the importance of a meta-parameter of the DPW trick in MCTS. This work is currently under review.

Finally, the seventh chapter concludes, and suggests some directions for future work.

1.2 Motivation: energy stock management

Energy management problems have received an enormous attention in the past few decades. Among the likely reasons behind this rise of interest, we could mention the economic weight of the energy market, the foreseeable scarcity of our energy resources, the increasing share of unpredictable renewable energy sources, and possibly the simple fact that these problems are very challenging.

According to a growing number of reports, the share of renewable sources in our energy mix is going to keep growing, with estimates for 2050 going from 35% to 80%[86]. A recent report, by the International Energy Agency (IEA) found that renewable sources would likely surpass natural gas by 2016[68].

First, we will present the energy management problem, and more specifically the version that involves hydroelectric stocks. Then, we will explain our goals and motivation, i.e. what we think are some of the current weaknesses in the state of the art methods.

1.2.1 Hydrothermal Scheduling (HS)

What we are specifically interested in is the hydrothermal scheduling. This problem is the one faced by an agent trying to maximize his utility function, given a certain number of energy production facilities (some hydroelectric, other thermoelectric, possibly also some renewable energy sources), and some model of the energy market (energy demand, fuel prices...) and of the weather conditions (temperature, inflows...). This

optimization is happening at the present time, but will impact future time steps. It is thus called planning, even though there may be many opportunities to change one's decisions over time, as more information becomes available. This problem is well described for a single hydroelectric stock, in [111]. We borrow their formulation and rewrite it here:

$$\min \sum_{t=1}^{T-1} \Psi_t(d_t - h_t) \tag{1.1}$$

$$h_t = k.\left[\phi(v_t) - \theta(u_t)\right] r_t \tag{1.2}$$

$$u_t = r_t + s_t \tag{1.3}$$

$$v_t = v_{t-1} + (i_t - u_t)\beta \tag{1.4}$$

$$v_t \in V_t \tag{1.5}$$

$$u_t \in U_t \tag{1.6}$$

$$r_t \in R_t \tag{1.7}$$

$$s_t \geq 0 \tag{1.8}$$

where

T : planning period;
t : index of planning stages;
$\Psi_t(.)(*)$: thermal cost function at stage t;
$d_t(*)$: energy demand at stage t;
h_t : hydro generation at stage t;
k : average specific efficiency;
$\phi(v_t)$: forebay elevation at stage t;
$\theta(u_t)$: tailrace elevation at stage t;
r_t : water discharged at stage t;
$i_t(*)$: inflow at stage t;
v_t : reservoir volume at the end of stage t;
u_t : water released at stage t;
s_t : water spilled at stage t;
β : conversion factor;
V_t , U_t and R_t : feasible sets representing bounds for the variables v_t , u_t , and r_t , respectively.

The decision variables, or actions, are the water discharged and water spilled at each time step (the latter can be seen as a slack variable).

Variables followed by a "(*)" are very likely to be the origin of the randomness in the process. This is to separate them from deterministic variables, or from variables that are deterministic functions of random variables.

The thermal cost $\Psi_t(.)$ is likely to be a random function, as prices of fuel tend to fluctuate over time. The demand d_t is also known for being random, following some underlying seasonal trend (eg. more heating is needed in northern countries during winter, many people turn on the heaters when coming back from work around 6 or

7pm, week days are different from week ends, etc). Finally, the inflows i_t depend on the weather conditions, and are thus stochastic.

In this setting, the agent is the person (or entity) controlling the energy production facilities: the hydroelectric plant (HP) and the thermal power plant (TPP). Every time it is doable, the agent readjusts how these facilities are used, in order to satisfy a random and time varying demand. Using HP is virtually free, but using TPP has a cost. Depending on the model, this cost function may be linear, quadratic, non continuous, etc. It also randomly changes over time, influenced by fuel price variations. Failure to satisfy the demand incurs a prohibitive cost. A simple visual representation of this problem is shown in Fig. 1-1.

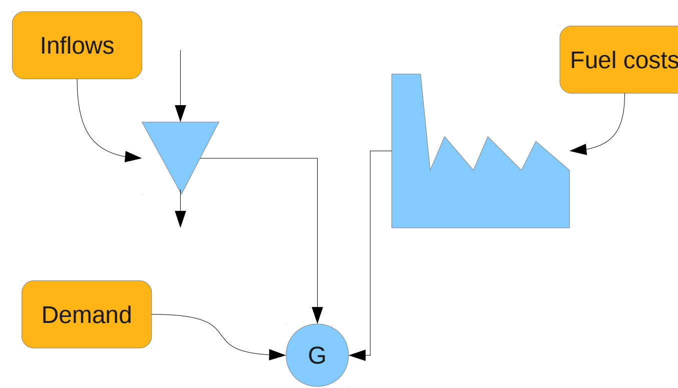


Figure 1-1: Hydroelectric Scheduling problem, with one stock and one thermal power plant. Random processes are shown in orange. G represents the power grid, where produced electricity is dispatched to satisfy the demand.

1.2.2 Challenging features

Here are four big challenges of HS problems (this list is non exhaustive):

- high dimension, both in state and action spaces that can be continuous, and long term time horizon;
- representation of randomness;
- non convexity of the cost function;
- non convexity of the action space.

Because of the first challenge, the industry has been attracted to mathematical programming more than to anything else. Other very convincing approaches have been approximation methods (see next chapter). The second challenge has been mostly tackled by simplifying the real stochastic models, representing them as small

samples of possible trajectories, or even by reducing the problem to certainty equivalents (i.e. replacing the randomness by some arbitrary predetermined sequence of events). The third challenge, as far as we know, has not truly been addressed. All methods we have seen so far consider that the cost function is convex. Sadly, even though the models of hydroelectric stocks are never perfect, their cost function is most likely not convex, because their efficiency decreases with the water elevation (i.e. with the amount of water in the stock). The last challenge can make classical convex optimization algorithms ineffective. The alternative methods (that include the use of Monte Carlo simulations) are usually less powerful (i.e. slower) than convex optimization methods.

Non convexity of energy problems. We briefly focus on this feature of the hydroelectric scheduling problem. We are interested in the cost function's behaviour when the state value changes. Indeed, what most methods assume, in order to solve this problem, is that the value associated with a given state of the system is a convex function of that state[93]. Our goal here is not to prove that every hydroelectric system ever operated is fundamentally non convex. It is simply to show that the assumption that it is indeed convex has very good reasons not to hold in reality. Making such an assumption should be, in our opinion, backed by solid evidence of it being true, or it should be shown that it introduces an acceptable model error. What follows is one example among the probably numerous reasons why the cost function might be non convex.

Considering the model previously introduced, we notice that the hydroelectric production h_t is an increasing function of the difference between the forebay and the tailrace elevation. This makes h_t an increasing function of the current forebay elevation, i.e. an increasing function of the current volume in the stock, v_t. This corresponds to the well known fact that the efficiency of an hydroelectric plant increases with its stock, due to the laws of physics. Let us write this increasing efficiency function $f(v_t)$, so that $h_t = \int_{v_t - r_t}^{v_t} f(u)du$. Note that we use a more precise expression of the hydroelectric production, where we integrate the elementary contribution per unit of volume. We also assume the absence of inflow and spillage, for clarity's purposes.

To look at the value of being in a given state v_t, we will consider only the last time step (the value computed at the last time step would then back propagate through time, as it will become more clear in the next chapter that covers dynamic programming). Assuming we neglect what happens after the final time step, the best decision is to use all the water we have, i.e. $r_t = v_t$. The demand having to be satisfied by thermal production is then $p_{th} = \max(0, d_t - \int_0^{v_t} f(u)du)$. Then, the last time step's cost is $\mathcal{C}_t = \Psi_t(p_{th})$.

The question now is to see if that function is convex or not, with respect to the state v_t. It is straightforward to compute its second order derivative

$$\frac{\partial^2 \mathcal{C}_t}{\partial v_t^2} = \begin{cases} -f'(v_t)\Psi_t'(p_{th}) + f(v_t)^2 \Psi_t''(p_{th}), & \text{if } p_{th} > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{1.9}$$

Since Ψ_t is an increasing function of p_{th}, and $-f'(v_t) < 0$ (increasing efficiency of a hydroelectric plant with v_t), the first term is negative. The second term, depending on the shape of the thermal costs, can be negative, null, or positive. The conclusion is that when we include the non linearity of the electricity production from water plants, it brings a source of non convexity in the cost function.

We did make a few reasonable assumptions to obtain this result (neglecting water salvage costs, inflows at the last time step...), but they were mostly done for the sake of clarity. Obviously, each power plant has its specificities, and we are not trying to prove a general result holding for all hydroelectric plants.

This provides us with one of our motivations: to work on optimization methods for SDM, that do not require the convexity assumption. Doing so may provide a way to reduce the model error when dealing with HS problems. Evaluating the size of this model error in real world instances is a challenging task. But it is an essential step to improve the state of the art of hydroelectric scheduling.

1.2.3 Multi-stock formulation

The above formulation corresponds to a single hydroelectric stock. We write here the straightforward extension to N stocks. We will use a N by N *transfer matrix* M to model the flows between stocks. For $1 \leq i < j \leq N$, $M_{i,j}$ is the portion of the water coming out of stock i that goes into stock j. To simplify the notations, we assume that all the water transits in one time step. All the notations associated to one stock, like v_t (or u_t, r_t) are now noted $v_{i,t}$ for each stock i. In this case, v_t will now refer to the column vector of size N containing each stock's variable.

$$\min \sum_{t=1}^{T-1} \Psi_t(d_t - h_t) \tag{1.10}$$

$$h_t = k. \sum_{j=1}^{N} \left[\phi(v_{j,t}) - \theta(u_{j,t})\right] r_{j,t} \tag{1.11}$$

$$u_t = r_t + s_t \tag{1.12}$$

$$v_t = v_{t-1} + (i_t - u_t + M \cdot u_{t-1})\beta \tag{1.13}$$

$$v_t \in V_t \tag{1.14}$$

$$u_t \in U_t \tag{1.15}$$

$$r_t \in R_t \tag{1.16}$$

$$s_t \geq 0 \tag{1.17}$$

In practice, water transfer between stocks can be subject to delays or losses. There might also be operational constraints on how much water can transit on each connection, to avoid overflows, for example. We show a simple illustration of the problem with multiple stocks in Fig. 1-2.

This version of the HS problem is essentially the version we have used throughout

this work. One can easily create small variations of this problem, by changing the probability distribution of the inflows, the connections between the stocks, the number of power plants, etc. We will try to make it clear when using this problem in the following chapters.

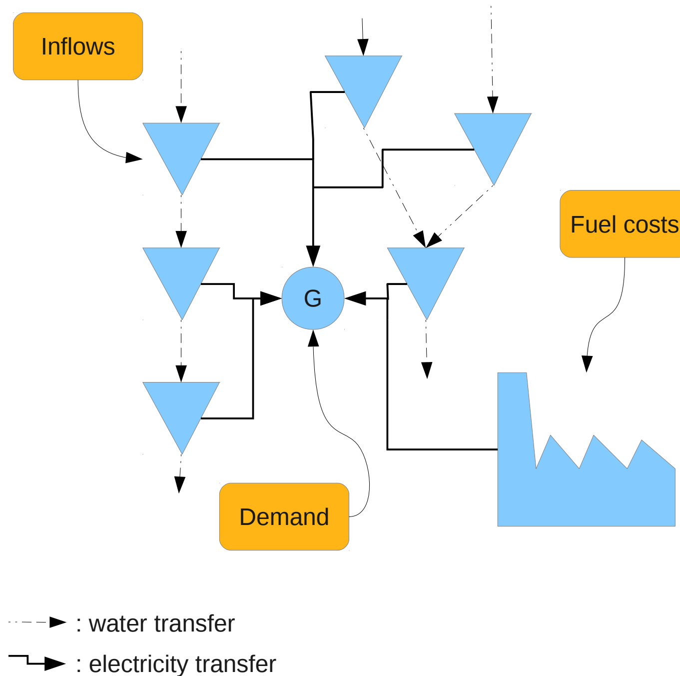


Figure 1-2: Hydroelectric Scheduling problem, with multiple stocks and one thermal power plant. Random processes are shown in orange. G represents the power grid, where produced electricity is dispatched to satisfy the demand.

1.3 Modelling sequential decision making problems

In SDM problems, one often considers the following framework: there is an *agent* (also known as controller) who can interact with a given *environment* (also know as process), for a certain number of sequential time steps. At each time step, the environment sends its current state to the agent, who in return, sends an action to the environment. The environment then changes its state accordingly, and sends the corresponding reward to the agent. The goal for the agent is to take actions such that the expected total long term reward is as high as possible.

If we call information everything the agent obtains from the environment (state changes, rewards...), then the problem we are dealing with unfolds as follows : initial information - decision - new information - decision - ..., until a final state is reached

(i.e. where no further action is needed or taken into account). In the case of infinite time horizons (where there is no guarantee to ever reach a final state), this process can have no end, and the problem is usually not solved by simulation, but by applying a discount factor to effectively neglect events that happen very far into the future.

What we call a SDM problem is also called a Dynamic Program in [96], or a multistage decision making problem, or also decision making under uncertainty. We choose Sequential Decision Making as it sounds the most transparent and self evident, in our opinion. Unless specified otherwise, the reader should assume that this is all done 'under uncertainty', as this is one of the key features of our original motivation (HS problems).

In an attempt to make our definitions easier to understand, we will often make a parallel with a simple real life example of SDM: a driver, in his car, wants to reach a certain destination, using the shortest usable path. The agent would be the driver, the environment would be everything that is relevant to the driver's task: the behaviour of his car, how roads are connected, the weather conditions, the traffic, random accidents, etc. The time steps may be of one second each. The state, i.e. the information available to the driver in his environment, would be his position, the current weather condition, and possibly partial information about traffic when listening to a radio station. The reward might be minus the time taken by the driver to reach his destination (so that we are indeed trying to maximize said reward).

We now proceed to give definitions of the different elements composing a SDM problem:

– decision epochs

– states

– actions

– random process

– transition function

– reward function

The definitions and notations we use in this chapter are taken from [19] and [96] for the control and OR community, and from [97] and [112] for the RL community.

1.3.1 Decision epochs

The decision epochs are the times at which a decision should be made, i.e. at which the agent should choose an action. In this work, we only consider the case of discrete decision epochs (thus called time steps). The reader interested in the continuous time settings, where decisions are made continuously, can read [19]. Discrete time steps do not necessarily mean equally spaced decision epochs. One might define a time step to occur each time the agent has to make a decision. In our example, this might happen every time the driver encounters a crossroad.

When relevant, we will index variables with their corresponding time step t, using the convention from [96]: a variable X_t indexed by t is deterministic at time t. However, while at time t, variable X_{t+1} could be both deterministic or stochastic, depending on the context. In the case of the driver, if X_{t+1} is the age of the driver at $t+1$, it is deterministic, but if it represents the traffic conditions at $t+1$, one can assume some uncertainty about X_{t+1}. Note that in the latter case, X_t represents the traffic conditions from $t-1$ to t, and is deterministic at t. This convention can seem a little counter-intuitive at first but avoids some confusion about what is random and what is not.

1.3.2 State variables

The name state variable being self explanatory, it is not always clearly defined. It can contain information about the physical state of the agent, pure information signal received from the environment, or also information about the history. To make things as clear as possible, we borrow the following definition from [96]:

Definition 1 (State variable)**.** *A state variable is the minimally dimensioned function of history that is necessary and sufficient to compute the decision function, the transition function, and the contribution function.*

Using this definition makes the distinction between history dependent processes and history independent processes useless. If the history information is needed to make better decision, to compute the next transition, or the costs, then it has to be included in the state variable. Otherwise, it should not be included (unless one wants to add computational complexity for free).

Similarly, if the time is important to compute the decision function (non stationary policy), or the transition and contribution functions (non stationary processes), then it should be included in the state. Thus, we only index variables with time, like x_t when it serves the purpose of making things clearer for the reader, or for information that becomes available and deterministic at time t.

Also, it should be noted that according to this definition, the state variable includes enough of the observable information to compute the optimal decision (if this one exists, and with respect to the observable information).

This definition of the state variables is maybe the most important part of this chapter. We will come back to it in the following sections, to explain its implications.

We will generally refer to state variables as $x \in \mathcal{X}$, with $\mathcal{X}$ referring to the state space. $\mathcal{X}$ can be finite, infinite but countable, or infinite and uncountable. It is also common to use s as a notation for a state variable, but it tends to imply that the state space is finite, as it is often used by in the framework of finite Markov Decision Processes. The nature of this space is both determined by the nature of the problem and by the choices of the model. Some methods require to have a discrete state space, and will thus need to first discretize a continuous state space before doing anything else. In our example, if the state only contains information about the position of the driver, it should be a continuous space. But, if the expected accuracy of the results

allow it, one may discretize the state space, by rounding up all states to integer values in meters.

1.3.3 Action variables

Actions are the result of the agent's decisions. They represent the lever available to the optimizer. The goal is to try to find the best ones. We will generally denote an action $a \in A$, A being an action space. As for states, an action space can be finite, discrete and countable, or uncountable.

It is common to talk about *feasible actions*, or *legal actions*, to talk about actions that are possible from a certain state x, at some time step t. This set depends on the environment. When this set depends on the state x and the time t, we will write it $A_t(x)$. Many problems have one unique feasible action space, for all t and x; in these cases we will simply write this space A for clarity.

As for state spaces, a continuous action space is sometimes discretized. For example, if one dimension of the action vector is the acceleration of the car, this coordinate could be continuous or discretized. In the latter case, the accuracy of the discretization becomes a meta parameter of the method.

Looking at our definition of the state variables, it means that if at time t, the past actions $(a_{t'})_{0 \leq t' \leq t-1}$ matter to the choice of a_t, or to the transition, or to the random process, then it should be included in the state variable. Otherwise, it should not.

Post decision state variable. We mention here the notion of post decision state variable. This is the state variable *after* the decision has been chosen, but *before* the random process and transition actually happened. We note the variable x_t^a. In some cases, this variable would simply be the pre decision state variable x_t, augmented by a_t, i.e. $x_t^a = (x_t, a_t)$. But, in some cases, it can be smaller than this. Indeed, the only information the post decision state variable needs to contain is the information necessary to compute the random process and the transition function, but does not require the information needed to compute the action a_t. In some applications, this can be used to reduce the size of the space where the search happens.

1.3.4 Random processes

The random processes refers to the variables that take random values, influencing the outcome of the decision making process. We consider that this flow of information arrives continuously to the agent, and we denote by w_t the information available at time t. This random variable follows a generally unknown random distribution, defined by a probability density function $p_{w_t}(w_t = w)$, sometimes written $p_{w_t}(w)$ for short. We write Ω_t the set of possible realizations at t.

Depending on the problem, this process, the set of possible realization, and the related probability distributions may depend on the time, the state (this could include the "history"), and the actions made. In this case, one would then write the density function $p_{w_t}(w_t = w|s, a)$, and the set of realizations $\Omega_t(s, a)$.

Note that, according to our definition, if w_t depends on the history of the process, $(w'_t)_{1\leq t'\leq t-1}$, then this history should be included in the state variable x_t. Then, a process will be Markovian if and only if the history is not included in the state variable.

The set Ω can be finite, discrete and countable, or uncountable. Much of the challenges posed by the random processes relies in how one can estimate its distribution.

The way the knowledge about this random process is made available can vary from one application to the other. In data driven applications, one might be given a fixed amount of past realizations, and try to infer the hidden distribution. In model based approaches, one needs to have a generative model of the random process, able to generate new realizations whenever prompted. Our work focuses on the latter case.

We will see later that the way the optimizer deals with the random process of the problem is an extremely important choice. Some methods choose to bypass the randomness entirely, optimizing their actions in the case of one fixed series of realizations. Others choose to optimize their action as if the random process would always take values equal to the expectation of this process. And others are trying to make decisions in order to minimize a given risk criterion, like the costs incurred in the 5% worst cases. Finally, some methods are simply trying to optimize the expected cost, computed with respect to the full distribution of the random process. All these approaches have their advantages and drawbacks. Generally speaking, one needs to make a trade off between being theoretically consistent and optimal, and having a fast and scalable method.

As many random processes in real applications are not Markovian, people like to consider series of random variables rather than isolated ones. It is then convenient to talk about random scenarios to refer to the realizations of these series of random variables.

In our driver's example, the random process could be the traffic conditions. This information becomes available step by step to the driver, as he takes new roads and sees by himself how busy they are. Given a model of the traffic conditions, one may want to optimize the journey assuming that one fixed scenario will happen (eg. the scenario "congested roads"). Another optimizer would consider the average traffic conditions, and look for the best actions to choose if these were the real conditions. One could also minimize the duration of the trip if the traffic conditions are among the 5% worst. Finally, one could try to optimize the decision making so that the expected time of the trip, with respect to the traffic distribution, would be minimal.

1.3.5 Transition function

The transition function models the dynamics of the problem. At time t, given a state x_t, an action a_t, and a random process variable w_{t+1} (this is the random information that will become available between t ant $t+1$), the next state is given by the transition function f, so that $x_{t+1} = f(x_t, a_t, w_{t+1})$.

In the stochastic case, we will often need to write the probability density function of x_{t+1}, and we will write it $p(.|x_t, a_t)$. Formally this notation means that for all $x \in \mathcal{X}$, $p(x|x_t, a_t) = p(f(x_t, a_t, w_{t+1}) = x)$.

At this point, we can illustrate the definition of state variables introduced earlier. If the transition function f requires any observable information that is not contained in a_t or in the possible realization of w_{t+1}, then it must be included in x_t. This means that if all past actions $(a_{t'})_{0 \leq t' \leq t-1}$ are needed to compute f, then they should be included in x_t. Similarly, if the probability distribution of w_{t+1} depends on $(w_{t'})_{0 \leq t' \leq t}$, then it should be included in x_t. Of course, if for some computational reasons, one does not want to include all that information in x_t, it is possible to truncate the state to a more manageable size. But then, one needs to be aware that the sampled realizations of w_{t+1} are very likely to be biased, and that the result of f may be far from the true model's result.

Non stationary transitions. Using our definition of state variables, unless we want to make the dependence on time of f as obvious as possible, we can simply rely on the fact that in this case, the time will be included in the state variable. When convenient, we may still overload the notations and use $f_t(x_t, a_t, w_{t+1})$, to specifically refer to the transition function at time t.

1.3.6 Reward function

The reward function g is how one quantifies the preferences of the optimizer. In financial applications, a common utility measure would be to minimize the cost (in dollars, or in any other currency), or to minimize a risk, or even the cost incurred in the worst case scenario. In the case of a driver trying to reach a destination, the most common measure is to minimize the expected duration of the trip, thus being called *shortest path problem*.

The reward function is computed as each time step by the environment, and is noted $r_t = g(x_t, a_t, w_{t+1})$. Using this notation, the objective of the agent is to maximize the expected long term reward $\mathbb{E}(R_N) = \mathbb{E}(\sum_{0 \leq t \leq N} \gamma^t r_t)$. For the problem to be properly defined, we need to have $\mathbb{E}(R_N) < \infty$.

When R_N is a random quantity, the questions mentioned before arises, namely: what are we trying to achieve? We can try to minimize the expected costs, minimize the "worst case scenario" costs (when this even makes sense), or minimize the costs assuming one given scenario will happen.

These are all legitimate ways of assessing the performance of a particular methods. However, as it is the most common way to proceed, we will use, as our objective, the maximization of the expected reward.

Non stationary reward functions. Just like with transition functions, reward functions may depend on the time t. In this case, following our definition of state variables, x_t will include the time.

Chapter 2

Algorithms for Sequential Decision Making

In this chapter, we introduce the notions of policies and value functions. We then show how they have been used to develop methods to solve SDM problems, first in small spaces, then in larger spaces.

2.1 Decision making under uncertainty

Using the notations previously introduced, the objective of the optimizer is, given an initial state x_0, to choose the best action possible a_0^*, then receive some new information, be in a new state x_1, choose the best possible action a_1, and so on. Unlike planning in a deterministic setting where one could say that the objective is to find one series of actions $(a_i)_{i\geq 0}$ that would be optimal, SDM under uncertainty adds randomness, and the possibility to react to it.

This makes even formulating the problem and the objective function difficult, using only traditional notations from the mathematical programming literature.

For instance, let us look at the following description of the problem:

$$(a_i)_{i\geq 0} = \underset{a_i \in A_i(x_i), i\geq 0}{\operatorname{argmax}} \sum_{0\leq t\leq N} \gamma^t g(x_t, a_t, w_{t+1}) \tag{2.1}$$

$$s.t. x_{t+1} = f(x_t, a_t, w_{t+1}), t \geq 0. \tag{2.2}$$

This formulation is problematic, as it suggests that one chooses all the future actions at once. Doing so, there is no guarantee that the first new state reached, x_1, even allows a_1 as a legal action. Indeed, x_1, even with a_0 fixed, is random. Using the formulation above is unfit for SDM under uncertainty, and should not be used.

Intuitively, when dealing with uncertainty, one wants to choose an action now (a_0) while considering many possible futures. This means considering many possible reachable states, and the actions that would be taken, would one be in these states. This is what motivates the notion of policies, introduced in this section, followed by

the notions of value and action-value functions.

2.1.1 Policies

Policies define how, given some information, the agent makes his decisions. Once again, we borrow the definition from [96]:

Definition 2 (Policy)**.** *A policy is a rule (or function) that determines a decision given the available information in state* x_t*.*

It is important to notice that, given our definition of a state variable, x_t contains all the observable information useful to make decisions. For example, if only the current physical state of the system is necessary, x_t will not contain any of the $(w_{t'})_{t' \geq 1}$. On the other hand, if the knowledge of the history of the random process helps to make better decisions, then it should be included in the state variable, and would thus be part of the policy's arguments.

This broad definition covers many kinds of policies. A policy can be an arbitrary rule, a parametric or non parametric function, or a combination of both. In the case of the driver's problem, a policy could be "always turn right when possible". It could be a polynomial function of the geographical position of the driver.

Given a policy π, a time t, and a state $x_t \in \mathcal{X}$, the action chosen is $a_t \in A$ with probability $\pi(x_t, a_t)$. For convenience, we write $\pi(x_t)$ the random variable that takes value a_t with probability $\pi(x_t, a_t)$.

In the case of a deterministic policy, there is one unique $a_{x_t} \in A$ such that $\pi(x_t, a_{x_t}) = 1$, and we can write, for short, $\pi(x_t) = a_{x_t}$. In this case, $\pi(x_t)$ is simply a deterministic function of x_t.

Non stationary policies. Again, thanks to the definition of state variables introduced before, if choosing an action (i.e. calling the policy) requires the knowledge of time, then the time should be included in the state variable. However, in some cases, like for SDP, it is also useful to define a non stationary policy π as a sequence of N stationary policies $(\pi_i)_{0 \leq i \leq N-1} = \pi$.

Non anticipativity. An important notion when trying to find a good policy is the non anticipativity constraint. One may or may not respect it, and it is essential to be aware of it.

The non anticipativity constraint is respected if at time t, the optimizer makes no assumption about what will be the actual realization of the random variable series $(w_{t'})_{t' \geq t+1}$. He can only use information about the distribution followed by the future random process if available, or sample possible futures if a generative model is available.

A common way to violate this constraint is to assume that a specific series of event will apply, in a deterministic way. This means that the action chosen at time t is chosen assuming a specific future is known. It is easy to see that this cannot be consistent, i.e. this cannot lead to an optimal decision (unless based purely on luck).

Moreover, proceeding that way, one will easily act in an overly optimistic manner, i.e. overconfident in the fact that the future will unfold in a certain way.

Our definition of a policy defines non anticipative policies. When referring to policies that violate the non anticipativity constraint, we will use the term *anticipative policies*.

2.1.2 Value function

Using all the previous notations, it is now convenient to define the value function of a state x at time t, $V^\pi(x)$. Informally, this value corresponds to the reward obtained by an agent starting in state x and applying the policy π until he reaches a final state. It can also be seen as the "value of being in state x at time t if we follow policy π".

Formally, $V^\pi : \mathcal{X} \mapsto \mathbb{R}$ is defined by:

$$V^\pi(x_{t_0}) = \mathbb{E}_\pi\left(\sum_{t_0 \le t \le N} \gamma^t r_t | x_{t_0}\right) \tag{2.3}$$

with $\mathbb{E}_\pi$ meaning the expectation under the assumption that at each time step, the agent applies policy π.

Hence, if we write Π the set of all possible policies π, then given an initial state x_0, an optimal policy π^* verifies:

$$V^{\pi^*}(x_{t_0}) = \sup_{\pi \in \Pi} V^\pi(x_{t_0}) = V^*(x_{t_0}) \tag{2.4}$$

$V^*(x_{t_0})$ is called the optimal value of x_{t_0}, the highest achievable expected reward that one can achieve when starting in x_{t_0}.

In the context of sequential decision making, it is especially useful because it allows the optimizer to avoid searching forward in time. When the time horizon is long, the computational savings can be enormous. Intuitively, when in time t, suppose we have access, for all possible states $x' \in \mathcal{X}$, to $V^*(x')$. Then, one could try all possible actions a from x, look at the states x' they lead to, and compare these states by using the function $V^*(x')$. This function would act like a fitness function, our problem would become a classical black box noisy optimization problem, and we would only need to choose our favourite noisy optimization algorithm (Fabian, Evolutionary Strategies...).

This is why a many approaches to solve SDM problems try to approximate the value function. This notion is also a cornerstone of the fundamental Dynamic Programming approach, a term chosen by Bellman. We will cover this method in the next section.

2.1.3 Action-Value function.

A very similar notion is the action-value function, or Q value $Q^\pi : \mathcal{X} \times A \mapsto \mathbb{R}$, defined by:

$$Q^\pi(x_{t_0}, a_{t_0}) = \mathbb{E}_\pi \left(\sum_{t_0 \leq t \leq N} \gamma^t r_t | x_{t_0}, a_{t_0} \right) \tag{2.5}$$

Similarly, we define the optimal action-value $Q^*(x_{t_0}, a_{t_0}) = \sup_{\pi \in \Pi} Q^\pi(x_{t_0}, a_{t_0})$

The two optimal value functions defined above are connected by the following equations:

$$V^*(x) = \sup_{a \in A} Q^*(x, a) \tag{2.6}$$

$$Q^*(x, a) = \mathbb{E}_w(g(s, a, w)) + \gamma \sum_{x' \in \mathcal{X}} p(x'|s, a) V^*(x') \tag{2.7}$$

The notion of Q value is especially intuitive in the sense that, if the agent is in state x at time t, and if some oracle gives him, for all possible actions $a \in A$, the corresponding Q values, the problem would essentially be a one state optimization problem. The agent could search the action space A once, looking for $a^* = \text{argsup}_{a \in A} Q^*(s, a)$. As seen in the above equations, the knowledge of optimal values can be used to derive the optimal action value functions, making them just as useful to solve the entire problem.

In a sense, being given the Q values is more powerful than being given the value function, because then there is no need to even use the generative model to find the optimal action. However, it comes at a computational cost, because Q values need to be computed for each couple (x, a), whereas value functions are only function of the state variables.

2.2 The finite case

In this section, we investigate the case where both the state and action spaces are finite. We present the fundamental and intuitive principle of optimality, described by Richard Bellman in 1957, on which is based the Dynamic Programming algorithm.

2.2.1 Bellman equations

Principle of Optimality. The principle of optimality is defined as follow by [16]:

Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

It is also more formally defined in [19]:

Let $\pi^* = \pi_0^*, ..., \pi_{N-1}^*$ be an optimal policy for the basic problem, and assume that when using π^*, a given state x_i occurs at time i with positive probability. Consider the subproblem whereby we are at x_i at time i and wish to minimize the cost-to-go

from time i to time N

$$\mathbb{E}\left(g_N\left(x_N\right)+\sum_{t}^{N-1} g\left(x_t, \pi_t\left(x_t\right), w_t\right)\right) \tag{2.8}$$

Then the truncated policy $\pi_i^*, ..., \pi_{N-1}^*$ is optimal for this subproblem.

Back to the case of a driver, suppose an oracle gives the agent the exact time needed to reach the destination if time step is a fixed t, and for any position $x \in \mathcal{X}$. Then, it is quite intuitive that, given his position x_{t-1} at time $t-1$, the driver could simply search all of his reachable positions x', and select the one that minimizes the time given by the oracle.

Given this principle, we know that if we slowly solve all the tail problems (starting with the easiest one), we will eventually solve the full problem. What we need is to make sure the initialization works, and to formalize the iterations of the process.

Bellman equation for Q-values. The Bellman equations for the Q-value function of a policy π is obtained by decomposing the expected value of choosing action a in state x as the expected reward of (x, a) plus the expected reward obtained from the resulting state, if one follows π thereafter. More formally, the Q value of (x, a) with respect to the policy π verifies:

$$Q^\pi(x,a) = \mathbb{E}_w\left\{g(x,a,w) + \gamma Q^\pi(f(x,a,w), \pi(f(x,a,w))\right\} \tag{2.9}$$

Similarly, the optimal Q value of (x, a), i.e. the Q value of (x, a) with respect to an optimal policy if there exists one, verifies:

$$Q^*(x,a) = \mathbb{E}_w\left\{g(x,a,w) + \gamma \sup_{a'} Q^*(f(x,a,w), a')\right\} \tag{2.10}$$

Bellman equation for Value function. The Bellman equations for the value function of a policy π is obtained by decomposing the total expected reward one would obtained starting in x and following π as the expected instant reward of following π in state x plus the expected reward obtained from the resulting state, if one follows π thereafter. More formally, the value function with respect to π verifies the following equation:

$$V^\pi(x) = \mathbb{E}_w\left\{g(x,\pi(x),w) + \gamma V^\pi(f(x,\pi(x),w))\right\} \tag{2.11}$$

And, the optimal value function verifies the following equation:

$$V^*(x) = \sup_{a\in A}\left(\mathbb{E}_w\left\{g(x,a,w) + \gamma V^*(f(x,a,w))\right\}\right) \tag{2.12}$$

2.2.2 Basic Dynamic Programming algorithm

In this section, we suppose that the horizon N, the state and action spaces $\mathcal{X}$ and A are all finite. Given such assumptions, one can turn the Bellman equation for value

function into an update rule, and be guaranteed to obtain the optimal value of all initial states, in a finite number of steps.

This basic algorithm is formally described in [19]:

For every initial state x_{t_0}, the optimal reward $V^{\pi^*}(x_{t_0})$ of the basic problem is equal to $V(x_{t_0})$, given by the last step of the following algorithm, which proceeds backward in time from period $N-1$ to period 0:

$$V(x_N) = g(x_N) \tag{2.13}$$

$$V(x_t) = max_{a_t \in A(x_t)} \mathbb{E}_{w_{t+1}} \left(g(x_t, a_t, w_{t+1}) + \gamma V(f(x_t, a_t, w_{t+1})) \right) \tag{2.14}$$

for $t = 0, 1, ..., N-1$, where the expectation is taken with respect to the probability distribution of w_{t+1}, which depends on x_t and a_t. Furthermore, if $a_t^* = \pi_t^*(x_t)$ minimizes the right hand side of Second Equation for each x_t and t, the policy $\pi^* = \pi_0^*, ..., \pi_{N-1}^*$ is optimal.

Note that we suppose here that we are able to compute the exact expectation with respect to the w's, and that this is only possible in the finite case (and only practical in small scale problems).

In the remaining of this section, we present two different ways to implement the dynamic programming idea, in the more general case of an infinite horizon, with the assumption that $0 < \gamma < 1$, and that the generative model is available to the agent.

2.2.3 Value iteration

This form of DP is the closest to the intuitive basic DP introduced in the previous subsection. The idea is to iteratively derive the Q-value function, so that eventually we have a function arbitrarily close to the optimal one. It is then straightforward to act greedily with respect to Q^*.

The update formula is simply the right hand side of the Bellman equation. At this point, it is convenient to introduce the Bellman operator T, a mapping from $\mathcal{Q}$ to $\mathcal{Q}$, with $\mathcal{Q} = \mathbb{R}^{A \times \mathcal{X}}$ being the space of all Q functions:

$$T(Q)_t(x, a) = \mathbb{E}_w \left\{ g(x, a, w) + \gamma \max_{a'} Q(f(x, a, w), a') \right\} \tag{2.15}$$

There are many formalization of this algorithm, we chose to show here the one written in [29].

Theoretical justification. This algorithm works, i.e. Q^l converges asymptotically to Q^*, when $\gamma < 1$. Under this condition, the mapping T is a contraction with respect to the infinity norm, and has a unique fixed point, $Q^* = T(Q^*)$ [69]. As the convergence is only asymptotic in the general case, one needs to choose as stopping criterion $\epsilon > 0$, or alternatively a maximum number of iterations L to ensure the desired accuracy.

Algorithm 1 Q value iteration for finite MDP

Input: $\gamma < 1$, f, g, and stopping criterion $\epsilon > 0$
Output: $\|Q^* - Q^l\| < \Delta(\epsilon)$
  Initialize $Q^0 = 0_{\mathcal{Q}}$ and $l = 0$
  repeat
    $l \leftarrow l + 1$
    for all (x, a, t), do $Q^l(x, a) \leftarrow T(Q^{l-1})_t(x, a)$
  until $\|Q^l - Q^{l-1}\| < \epsilon$
  return Q^l

Computational cost. The total cost of L iterations for a stochastic finite MDP is

$$L|\mathcal{X}|^2|A|(2 + |A|) \tag{2.16}$$

2.2.4 Policy iteration

Policy iteration works as follows: first, one needs an initial policy π_0. Then, one evaluates this policy by computing its corresponding Q value function, i.e. one needs to solve the Bellman equation corresponding to it. Then, one modifies the current policy, making it choose actions greedily with respect to the Q value function obtained at the previous step. This is repeated until a stopping criterion, usually when to successive policies are almost identical. As for value iteration, policy iteration has been described in many papers, and we chose the description made in [29].

Policy evaluation. We detail here the crucial step of policy evaluation. Given a policy π, we want to return an Q value function that is as close as possible to the real Q value function underlying this policy, Q^π.

Here, it is convenient to introduce the Bellman operator for a policy π, T^π, a mapping from $\mathcal{Q}$ to $\mathcal{Q}$, with $\mathcal{Q} = \mathbb{R}^{A \times \mathcal{X}}$ being the space of all Q functions:

$$T^\pi(Q)_t(x, a) = \mathbb{E}_w \left\{ g(x, a, w) + \gamma \max_{a'} Q^\pi(f(x, a, w), a') \right\} \tag{2.17}$$

Algorithm 2 Policy evaluation for Q value functions in stochastic finite MDPs

Input: $\gamma < 1$, f, g, and stopping criterion $\epsilon > 0$, a policy π
Output: a Q function verifying $\|Q^* - Q\| < \Delta(\epsilon)$
  Initialize $Q^{\pi,0} = 0_{\mathcal{Q}}$ and $l = 0$
  repeat
    $l \leftarrow l + 1$
    for all (x, a, t), do $Q^{\pi,l}(x, a) \leftarrow T^\pi(Q^{\pi,l-1})_t(x, a)$
  until $\|Q^{\pi,l} - Q^{\pi,l-1}\| < \epsilon$
  return $Q^{\pi,l}$

Algorithm 3 Policy iteration for Q value functions in stochastic finite MDPs

Input: $\gamma < 1$, f, g, and stopping criterion $\epsilon > 0$, a policy π
Output: a policy π verifying $\|\pi^* - \pi\| < \Delta(\epsilon)$
  Initialize $\pi_0 = 0_\Pi$ and $l = 0$
  repeat
    evaluate π_l, i.e. derive Q^{π_l}
    improve π_l into π_{l+1} by doing $\pi_{l+1}(x) \in \text{argmax}_a Q^{\pi_l}(x, a)$ for all $x \in \mathcal{X}$
  until $\|\pi_l - \pi_{l-1}\| < \epsilon$
  return π_l

Full algorithm and efficiency. A big advantage of policy iteration over value iteration is that the Bellman operator used is the one associated with a specific policy π, which means the equation to solve is a linear one. Assuming the size of the state and action space is not too big, this can be solved quite efficiently.

Each call to policy evaluation takes about $4|\mathcal{X}|^2|A|$.

2.2.5 Q learning

Q-learning [121] is an off policy algorithm that approximates the optimal Q-value function. Although initially meant for small scale (e.g. finite space) problems, it is possible to make it efficient for larger scale problems, via function approximations for example.

This algorithm is essentially made of two parts:

– an update function that, given (x, a, x', r, Q) updates Q according to the transition (x, a, x') and instant reward obtained r.

– a way to choose x and a; this can be simply chosen from pre-generated data, or obtained online by following some policy π chosen by the user.

Alg. 4 provides a formal description of the Q-learning algorithm (from [112] and [113]), in the case where a generative model is available. Note that the policy π could be anything. Common choices include the greedy policy with respect to the current Q-value function, possibly combined with ϵ-greedy exploration, or the Boltzmann scheme. The authors of [110] propose some sampling strategies with theoretical guarantees.

One can still apply Q-learning from data, without a generative model. In this case, for every (x, a, x', r) available in the data, we can apply the update function. The theoretical convergence of this algorithm was proven in [121], with the assumption that the sampling strategy samples all actions infinitely often (asymptotically), and that states and actions are discrete.

2.2.6 SARSA

Described in [112], and first introduced in [105], SARSA is an on policy TD algorithm, very similar to Q-learning. The difference lies in the update function. Instead of

Algorithm 4 Q-learning with generative model

Input: a generative model of the problem, a policy π, a step size $\alpha > 0$
Output: an approximation of Q^*
  Initialize Q:$(\mathcal{X} \times A) \mapsto \mathbb{R}$ arbitrarily for all $(x, a) \in (\mathcal{X} \times A)$
  repeat
    Pick an initial state $x \in \mathcal{X}$
    repeat
      choose action a according to a policy π
      use the generative model to obtain resulting state x' and reward r
      $Q(x, a) \leftarrow Q(x, a) + \alpha \left(r + \gamma \max_{a' \in A} Q(x', a') - Q(x, a)\right)$
      $x \leftarrow x'$
    until x is a final state
  until no more computing time
  return Q

taking the maximum over all actions of the current approximation of Q, we simply choose a' according to the policy π chosen by the user. As before, this policy can be a greedy policy with respect to Q, with additional exploration, or anything that can provide a feasible action.

Alg. 5 provides a formal description of SARSA.

Algorithm 5 SARSA algorithm

Input: a generative model of the problem, a policy π, a step size $\alpha > 0$
Output: an approximation of Q^*
  Initialize Q:$(\mathcal{X} \times A) \mapsto \mathbb{R}$ arbitrarily for all $(x, a) \in (\mathcal{X} \times A)$
  repeat
    Pick an initial state $x \in \mathcal{X}$
    repeat
      choose action a according to a policy π
      use the generative model to obtain resulting state x' and reward r
      $Q(x, a) \leftarrow Q(x, a) + \alpha \left(r + \gamma Q(x', \pi(x')) - Q(x, a)\right)$
      $x \leftarrow x'$
    until x is a final state
  until no more computing time
  return Q

This algorithm is guaranteed to converge to the real Q-value function Q^*, as long as exploration is assured, and as long as the exploratory policy slowly becomes greedy with respect to the Q-value function[110]. Becoming greedy too quickly would hurt exploration, and never becoming greedy would prevent convergence.

The essential difference between Q-learning and SARSA is what the reinforcement learning community calls *on-policy* and *off-policy* algorithms. Q-learning, because it has two distinct policies, one to explore actions, and one that is evaluated, is called off-policy. SARSA, on the other hand, is call on-policy because it uses the policy that

is being evaluated as its exploratory policy.

2.3 The large scale case: RL and approximate DP

What all the algorithms of the previous section, as they are presented, share, is that they need to store the Q values (or V values, depending on the implementation), for every single state and action (or just every state if using V values). For problems where the state and action spaces are both finite and small, this is not a problem. However, for large spaces, and in particular for continuous (thus infinite) spaces, this proves highly impractical, if not outright infeasible.

There are a number of workarounds to solve this issue. The first one is the approximate dynamic programming approach, that we first introduce in its generic form. We will then present function approximations methods, that suppress the need to store a table of infinite size in memory. The approximating function can be of fixed dimension (like in the case of parametric function approximations), or of data driven size (non parametric function approximations).

In DP, one recursively travels through all reachable states, backward in time, to compute the exact value function $V^*(.)$. To do so, one needs to use the final states values as initial values, and the following Bellman equation as an iteration rule:

$$V^*(x) = \sup_{a \in A} \left(\mathbb{E}_w \left\{ g(x, a, w) + \gamma V^*(f(x, a, w)) \right\} \right) \tag{2.18}$$

In ADP, we aknowledge that solving this equation exactly is intractable. We can divide the challenges of this equation in three categories:

– the state space is too large: we should approximate V (value function approximations)

– we cannot compute the expectation exactly (sampling methods, post decision state)

– the action space is too large: we should discretize the action space, or find efficient ways to divide it.

These challenges correspond to the *curses of dimensionality*, and are the focus of most of the literature following Bellman's work. We will divide this section as follows: section 2.3.1 deals with the methods that simply try to approximate the value function itself, section 2.3.2 presents methods that aim at iteratively approximate an optimal policy, section 2.3.3 will show how some methods translate the whole problem into a noisy optimization problem, by doing a search in the policy space, section 2.3.4 will present the famous actor-critic methods, also known as general policy iteration methods. Finally, we will present some of the most popular methods for solving unit commitment problems, developed by the mathematical programming community, in section 2.3.5.

2.3.1 Approximate Value Iteration

In this part, we look at the various methods that revolve around the approximation of the value function V^* or Q value function Q^*. Because the state and action spaces are large, one cannot represent these functions exactly, let alone compute them exactly.

There are two main approximation approaches: parametric and non parametric. The former uses a predetermined parameter θ to characterize an approximation function $\bar{V}_\theta$. The latter uses data driven characterizations of the approximation function.

Parametric approximations

One way to approximate V^* is to project it on a user defined set of *basis functions* ϕ_f, where $f \in \mathcal{F}$ represent features of the problem, and $\mathcal{F}$ is the feature set. Using this idea, and some parameter vector $\theta \in \mathbb{R}^{\mathcal{F}}$ we can write the approximate value function as follows:

$$\bar{V}_\theta(x) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(x) \tag{2.19}$$

Note that this approximation is linear in the parameter vector θ. The complexity of this function can be hidden in the basis functions. The simplest basis functions that we could think of are the powers of the state variable, so that one could write:

$$\bar{V}_\theta(x) = \sum_{i \geq 0} \theta_i s^i \tag{2.20}$$

The size of the feature space is up to the user. May one choose to have only two features, the previous simple approximation could be written:

$$\bar{V}_\theta(x) = \theta_0 + \theta_1 s \tag{2.21}$$

The aim is then to find a parameter θ that makes the approximation as close as possible to the real value function. Of course, one does not have access to this real value function. One can only sample trajectories with a generative model of the problem, or use existing samples. The generic algorithm could be described as follows:

Algorithm 6 Generic Approximate Parametric Value Iteration

Input: a generative model or sampled trajectories, a feature function basis $\mathcal{F}$
Output: an approximation of V^*, $\bar{V}_0$
  Initialize θ_0, $\bar{V}_{\theta_0}$:$\mathcal{X} \mapsto \mathbb{R}$ and $n = 0$ arbitrarily
  repeat
    Evaluate current approximation, and obtain either one score $\hat{v_n}$
    Update θ_n according to that evaluation
    $n \leftarrow n + 1$
  until no more computing time
  return $\bar{V}_n$

The two main challenges of value function approximation are:

– how to choose the set of features $\mathcal{F}$ (and thus the basis functions) ?

– how to optimize θ ?

The choice of basis functions is mostly an art, and problem dependent. However, there are many papers dealing with this issue. First, some people focus on optimizing a given set of basis functions, like in [126] where the authors use gradient-based optimization to adapt the basis functions. Another way to proceed is to work on the choice of basis functions. It can be done by starting with a very small set of functions, and by slowly adding new basis functions to it, like in [118, 91, 64]. Alternatively, one can start with a very large set of basis function, and slowly selecting a smaller subset of it, as in [76].

There are essentially two ways to optimize θ: in an incremental manner, or from batches of sampled trajectories. Depending on this choice, there may be different options to both evaluate a given approximation $\bar{V}_{\theta_n}$, and to update it according to the evaluation. To illustrate both cases, we will present the stochastic gradient descent and the least square fitting method.

Stochastic gradient descent. In this case, we suppose that we want to update the value of θ_n after each function transition. These transition are either simulated online or picked from a pre-sampled database.

At a given step n, and in some state x, we evaluate the current approximation $\bar{V}_{\theta_n}$, and obtain some value $\hat{v}(x)$. The way we do the evaluation would be, typically, by selecting an action a greedily with respect to the current approximation, observing one transition (x, a, x', r), and using the current approximation to give a value to x'. That way, the evaluation returns the score:

$$\hat{v}(x) = r + \gamma \bar{V}_{\theta_n}(x') \tag{2.22}$$

Using the same notation as in the Bellman error minimization, we update θ as follows:

$$\theta_{n+1} = \theta_n - \alpha_n(\bar{V}_n(x) - \hat{v}(x))\phi(x) \tag{2.23}$$

As presented, this method has a few difficulties:

The choice of a is done as follows:

$$a = \operatorname{argmax} \mathbb{E}_w(g(x, a, w) + \gamma \bar{V}_{\theta_n}(f(x, a, w)) \tag{2.24}$$

This is the most intuitive way to select an action, but it is easy to see that this does not favour exploration at all. In addition, it might be extremely difficult to compute the expectation, and one would probably need to use another approximation for it. Finally, this method requires the choice of a step size, that can be critical. A small step size may lead to stalling, a large step size to instability. Many works revolve around adapting this step size to avoid both dangers, and a comprehensive review of these can be found in [58].

Least square fitted Q-iteration. Here, we show a method that uses a sample of transitions $\mathcal{D} = (x_m, a_m, x'_m, r_m)_{1 \leq m \leq M}$. It can be either a given sample, or be augmented online if a generative model is available. Such a method is used in [50], where the authors use the results of each iteration of the following algorithm to update the sample (as explained in [29]), in order to maximize the utility of the samples used inside the value iteration step:

Algorithm 7 Least-squares fitted Q-iteration with parametric approximation

---

Input: a sample $\mathcal{D}$ of transitions , initial parameter θ_0, initial value function approximation $\bar{Q}^0_{\theta_0}$
Output: an approximation of Q^*
  Initialize $n = 0$, $\bar{Q}_{\theta_n}$
  repeat
    for $m = 1$ to M **do**
      $\hat{Q}_n(x_m, a_m) \leftarrow r_m + \gamma \max_a \bar{Q}_{\theta_n}(x'_m, a)$
    end for
    $\theta_{n+1} \leftarrow \text{argmin}_\theta \sum_m (\bar{Q}_\theta(x_m, a_m) - \hat{Q}_n(x_m, a_m))^2$
    $n \leftarrow n + 1$
  until no more computing time
  return $\bar{Q}_{\theta_n}$

In some way, the choice of the samples drives how the evaluation of a given approximate Q-value function is done. Taking many samples sharing the same initial state and action means that one is trying to obtain a very accurate expectation with respect to the random process w, at the expense of not visiting many different states and actions. At the opposite, putting only different states and actions in the sample means that the evaluation step is very noisy, but that one explores as many states and actions as possible.

Non Parametric

These methods rely on a given set of samples $\mathcal{D} = (x_i, v_i)_{1 \leq i \leq m}$, and use that information to infer the value of any new state x. A typical example of non parametric approximation are the kernel-based approximators. In that case, given some kernel $K_\mathcal{D}$, the nearest neighbor method for regression would give the following value to $x \in \mathcal{X}$:

$$\bar{V}_\mathcal{D}(x) = \sum_{1 \leq i \leq m} v_i \frac{K_\mathcal{D}(x, x_i)}{\sum_{1 \leq i \leq m} K_\mathcal{D}(x, x_i)} \tag{2.25}$$

Despite their name, non parametric methods still involve a choice of parameters, like the choice of the kernel, and the choice of some parameters of that kernel. Some tuning, even automatic, might still be required.

2.3.2 Approximate Policy Iteration

Approximate Policy Iteration is similar to approximate VI. Except than instead of focusing on approximating the real Value function, to eventually act greedily with respect to that approximation, one tries to directly approximate the optimal policy. A major consequence of this difference is that a policy can be almost optimal, even though the matching value function is far from the optimal one. The intuition is that what matters to make a policy good is how it ranks the possible actions relatively to one another, not the absolute value given to each state and action. Even if the absolute values are wrong, the ranking can still be good (the inverse cannot be true).

An approximate PI algorithm is repeating two steps until the policy at hand is satisfactory (or until we run out of time): the policy evaluation step, and the policy improvement step. The generic approximate PI algorithm based on Q-values is formulated in [29] as follows:

Algorithm 8 Approximate PI with Q-values

Input: some sample trajectories, and/or a generative model
Output: $\bar{\pi}_n$, an approximation of π^*
  Initialize $n = 0$, policy $\bar{\pi}_n$
  repeat
    Policy evaluation : find $\bar{Q}^{\bar{\pi}_n}$, an approximate of the true Q-value function of $\bar{\pi}_n$
    Policy improvement : find a policy verifying $\bar{\pi}_{n+1}(x) \approx \mathrm{argmax}_a \bar{Q}^{\bar{\pi}_n}(x, a)$ for all x
    $n \leftarrow n + 1$
  until $\bar{\pi}_n$ is satisfactory or no more computing time
  return $\bar{Q}_{\theta_n}$

The policy evaluation can be a lot like approximate value iteration. Given some policy, one is merely trying to find its real value function, possibly by iteratively updating a parametric approximation of this value function, for example. The policy improvement step can be exact, when the state and action spaces are small enough to allow a simple enumeration. But of course, as we are here interested in large spaces, one needs to rely on approximations, in a similar fashion to how value functions are approximated.

Policy evaluation

We will focus here on simple policy evaluation methods, but there exist several other options, like Least Square Temporal Difference (LSTD, seen in [24] and [23]), and Least Square Policy Evaluation (LSPE, as seen in [92] and [29]). We show one example of policy evaluation based on a parametric Q-value approximation, and one example of a non parametric one.

Least square fitted policy evaluation. This method is extremely similar to the least square fitted Q-value iteration presented in Section 6. The main difference

is that the update of the approximate value function is done following the current approximate policy, instead of acting greedily with respect to the value function. The formal algorithm is shown below:

Algorithm 9 Least-squares fitted policy evaluation with parametric approximation

---

Input: a policy π to evaluate, a sample $\mathcal{D}$
Output: an approximation of Q^h
  Initialize $n = 0$, $\bar{Q}^{\pi}_{\theta_n}$
  repeat
    for $m = 1$ to M **do**
      $T(\bar{Q}^{\pi}_{\theta_n})(x_m, a_m) \leftarrow r_m + \gamma \bar{Q}^{\pi}_{\theta_n}(x'_m, \pi(x'_m))$
      $\hat{Q}_n(x_m, a_m) \leftarrow T(\bar{Q}_{\theta_n})(x_m, a_m)$
    end for
    $\theta_{n+1} \leftarrow \text{argmin}_{\theta} \sum_m (\bar{Q}_{\theta}(x_m, a_m) - \hat{Q}_n(x_m, a_m))^2$
    $n \leftarrow n + 1$
  until no more computing time
  return $\bar{Q}_{\theta_n}$

As for methods presented before, this algorithm is sensitive to how the samples are chosen (one needs to strike a balance between noisy estimations of each (x, a), and exploration of many (x, a)). Also, as it is a parametric method, one needs to first chose a set of basis functions, to find a balance between over fitting (too many basis functions) and under fitting (too few).

Policy evaluation with rollouts. To circumvent the difficulty of choosing basis functions, one can use a non parametric method. One example is to evaluate the policy π with rollouts, also known as Monte Carlo simulations. It is required to have a generative model. This approach has been proposed in [47]. The idea of this method is that from each pair (x, a), we simulate N trajectories of length K, where each action chosen after a follows policy π. The final estimation of the value of each pair (x, a) is the average of the cumulated return of the N trajectories:

$$\hat{Q}^{\pi}(x, a) = \frac{1}{N} \sum_{i=1}^{N} \left[g(x, a, w_{(i,1)}) + \sum_{k=1}^{K} \gamma^k g(x_{(i,k)}, \pi(x_{(i,k)}), w_{(i,k+1)}) \right] \tag{2.26}$$

where, for each trajectory i, and each step k, given a state $x_{(i,k)}$, the following state is generated using the policy π and the generative model as follows: $x_{(i,k+1)} = f(x_{(i,k)}, \pi(x_{(i,k)}), w_{(i,k+1)})$.

Essential parameters of this method include the horizon K and the number of simulated trajectories N. If the problem and the policy are deterministic, then $N = 1$ is sufficient. However, if the problem and/or the policy have a high volatility, one would probably need a very high value for N to obtain an estimate accurate enough to discriminate different pairs (x, a). The length of the horizon should aim at balancing

computational cost and estimation error. A very short horizon would be computationally cheap, but would likely give a very biased estimate (missing important events in the future). A very long horizon would probably allow for a smaller estimation error, but could become computationally too expensive.

Finally, note that this evaluation method becomes highly impractical when there are a large number of pairs (x, a) where one wants to evaluate the policy.

Policy improvement.

At this step, we use the new approximate value function $\bar{Q}^{\pi_l}$ associated to the current policy π_l, to update the policy. The first and straight forward idea is to visit each state x and update the policy accordingly:

$$\pi_{l+1}(x) = argmax_a \bar{Q}^{\pi_l}(x, a) \tag{2.27}$$

This only works if the state space is small enough, and if finding the maximizing action is tractable. Depending on the action space size and on the shape of $\bar{Q}^{\pi_l}$, this may very well be computationally too expensive.

The alternative is, of course, to parametrize the policy π, using a set of basic functions ψ_i, $1 \leq i \leq \mathcal{N}$. The basis functions are actually policies, rules that output an action, from the information contained in state variables. We will still use the notation θ for the parameter vector. Using these notations, a linearly parametrized policy would, from a state x, return the action:

$$a = \pi(x) = \sum_{i=1}^{\mathcal{N}} \psi_i(x)\theta_i = \psi^T(x)\theta \tag{2.28}$$

We then need a sample of states $x_1, ..., x_M$ on which we want to fit the parametric policy to the value function obtained at the policy evaluation step, $\hat{Q}^{\pi_l}$.

Given this setting, there have been two ways to improve the policy π_l, i.e. to update θ_l (we are only considering the parametric case).

First option, separate the improvement in two steps:

– for each state x_m, find the greedy action $a_m \in \operatorname{argmax}_a \hat{Q}^{\pi_l}(x_m, a)$

– $\theta_{l+1} = \operatorname{argmin}_\theta \sum_{m=1}^{M} \left(\psi^T(x_m)\theta - a_m\right)^2$

Notice how the second step is merely a convex optimization problem, and will usually not be the biggest challenge. However, one first needs to find all greedy actions, something that is not guaranteed to be easy.

The second option is to directly look for the θ that maximizes the Q-value function, i.e.

$$\theta_{l+1} = \operatorname*{argmax}_\theta \sum_{m=1}^{M} \hat{Q}^{\pi_l}(x_m, \psi^T(x_m)\theta) \tag{2.29}$$

This avoids having to search the entire action space, but it creates an optimization problem that will generally be non linear.

2.3.3 Policy search

Policy search methods, as the name indicates, search in the space of policies Π for an optimal policy π^*. To our knowledge, all these methods use parametric policies $\pi_\theta \in \Pi$, where θ is some parameter to optimize. Then one can essentially divide policy search methods into two groups: gradient based policy search when π_θ is differentiable with respect to θ, and gradient free policy search when it is not (or when one chooses not to use any gradient for other reasons).

Gradient based policy search. These methods, like policy iteration, are divided in two steps: policy evaluation, and policy improvement. First one initialize the parameter (and thus the policy) to some θ_0. Then, one evaluates the resulting policy, either exactly, or by sampling many possible returns of this policy. One also needs to evaluate the gradient of the reward of this policy. This is the core of the method, and there has been a great deal of attention on this issue. One of the main difficulties is to keep the variance of the estimate of the gradient small enough to make it relevant. This what the authors of [85] focus on, in the case of finite state MDPs.

Once the gradient is properly estimated, one can apply a classic gradient ascent to the parameter θ, and repeat.

This type of policy search offers the comforting use of gradient ascent, but suffers from local optimum traps, and variance reduction issues.

Gradient free policy search. These methods do not use the gradient (or its estimate) of the average reward. Instead, they consider the problem of finding a good parameter θ as a noisy (gradient free) optimization problem. Doing so, they can pick amongst the existing techniques, such as evolutionary optimization[14, 61] or cross-entropy methods[83]. In [123], the authors use neuroevolutionary optimization to automatically find function approximations.

The main advantage of gradient free policy search is that it is less vulnerable to local optima than the previously described methods. It also dodges the problem of gradient estimation.

2.3.4 Actor critic methods

The term *actor-critic*, was first used in the context of computer science in [15]. The idea came naturally to distinguish the two types of process interacting in the value and policy iteration methods. In the latter, one iteratively evaluates a policy by approximating its associated value function, and improves that policy based on that evaluation. One keeps evaluating a policy (hence the term 'critic'), and improving this policy by using the feedback (hence the term 'actor'). Actor-critic methods are doing almost the same thing, with one big difference: the improvement step is not done by making the policy acting greedily with respect to the new value function. Instead, one simply moves in the direction of these greedy actions. It is a less radical step, intended to provide additional stability to the improvement step.

Having this in mind, one refers to the evaluation part, i.e. the part that tries to approximate the value function of a given policy, as the *critic*. On the other side, the *actor* receives these values from the critic, and tries to improves his current policy: it is the policy improvement step.

Using this terminology, policy gradient methods that do not use any value function are called *actor only methods* (see William's REINFORCE algorithm [124], and [90] [100]). Conversely, value iteration algorithms, like SARSA, can be called *critic only methods*.

Most of the actor-critic algorithms so far have relied on policy gradient, or on natural policy gradient ([95]) for the actor part. A more comprehensive survey on actor-critic algorithms can be found in [63].

2.3.5 Mathematical programming for unit commitment

When working on unit commitment problems, as described in [93], the sizes of both the action space becomes so large that the community working on them has often relied on mathematical programming, and more specifically stochastic programming [49] to handle them. Typically, these methods are extremely powerful to deal with very large action spaces, but often rely on convexity assumptions for the reward functions, or on other simplifications of the real problem to run correctly.

In this section, we cover quickly some of the approaches coming from the mathematical programming community that have been at the center of the attention of the practitioners.

Benders cuts

In [93], the authors use a decomposition approach, based on Benders cuts [17], to tackle a unit commitment with multiple hydroelectric reservoirs.

This method requires a model of the problem that has a convex objective function and linear constraints. The stochasticity can be handled as a set of possible realizations, or scenarios.

The main idea is to separate the problem into the two usual blocks: on one side, the immediate cost function, and on the other, a value function to represent everything that would happen later. As before, the goal is to slowly approximate the value function, because finding the exact one is intractable. In this case, the trick is to exploit the (assumed) convexity of the value function, to approximate it with a relaxed linear program. Then, the dual variables of the solution are used to derive cuts (the so called Benders cuts), that are lower bounds to the real value function. This method is thus often called Stochastic Dual Dynamic Programming (SDDP).

For the sake of clarity, we will explain the process for a two time steps problem. One is trying to minimize the costs, so formally, given an initial state x_0:

$$a_0^* = \operatorname{argmin} \mathbb{E}_w g(x_0, a_0, w_1) + \gamma V^*(f(x_0, a_0, w1)) \tag{2.30}$$

with

$$V^*(f(x_0, a_0, w1)) = \min_{a_1} \mathbb{E}_{w_2} g(f(x_0, a_0, w1), a_1, w_2) \tag{2.31}$$

subject to linear constraints on a_0, a_1.

As this method can work on scenario trees, we will write p_i, $i = 1 \ldots K_1$ the probability of having the ith as a realization for the first time step. Similarly, we will write $p_{i,j}$, $j = 1 \ldots K_2$ the probability of, after observing the ith realization, having the jth realization for the second time step.

We can then rewrite the objective function as:

$$\min_{a_0} \sum_{i=1}^{K_1} p_i \left(g(x_0, a_0, w_{1,i}) + \gamma \alpha_i^*(x_0, a_0) \right) \tag{2.32}$$

subject to linear constraints on a_0, and with

$$\alpha_i^*(x_0, a_0) = \min_{a_1} \sum_{j=1}^{K_2} p_{i,j} g(f(x_0, a_0, w_{1,i}), a_1, w_{2,j}) \tag{2.33}$$

subject to linear constraints on a_1.

Using this decomposition, one can proceed as follows:

- initialize $\bar{\alpha}_i(x_0, .)$ for $i = 1 \ldots K_1$ as a piecewise linear function that is a lower bound to $\alpha_i^*(x_0, .)$
- solve 2.32 with some linear programming solver; the resulting $\bar{a_0^*}$ gives a cost z that is a lower bound to the real optimum
- solve 2.33, with $a_0 = \bar{a_0^*}$; the resulting cost $\hat{z}$ is an upper bound to the real optimum
- if $z - \hat{z}$ is smaller than some given tolerance, then we stop and consider the solution satisfying; otherwise we use the second stage problem's dual variable to generate new approximations $\alpha_i^*(x_0, .)$ (this is the part that uses Benders cuts), and go back to the first stage problem.

This method has been quite popular to solve hydro-thermal energy management problems [94], because of a few reasons. First, it uses powerful mathematical programming techniques, that easily scale up with very high dimensional problems. Second, each step of the algorithm gives a tightening pair of lower and upper bounds to the optimal cost, which gives an idea of how far the approximate solution is from the real optimum. Finally, it pairs well with a tree based model of randomness, that allows for non Markovian processes, commonly encountered in these problems (weather is known to be highly non Markovian).

That being said, this method suffers from a major drawback: it requires the value functions (on which are based the α functions above) to be convex. If this assumption does not hold, then the Benders cuts cannot be applied to derive approximations of the real value functions. Using this method on non convex value functions can lead to arbitrarily large errors in the value function approximations.

Certainty Equivalent Control

Certainty Equivalent Control (CEC) is the term employed to cover the methods transforming an initially stochastic problem into a deterministic problem, by replacing the random variables by one fixed trajectory.

One variant of this, often applied to unit commitment problems, is *Model Predictive Control (MPC)* [98, 52]. The idea is to represent the system so that the all variables are the variation from the desired stable state. Then, using a rolling horizon, one tries to keep the system stable (i.e. the variables close to zero) over that short horizon, chooses an action, receives a feedback, and moves forward in time.

Chapter 3

Continuous MCTS

In this chapter, we introduce Monte Carlo Tree Search algorithm for finite deterministic MDPs, in Section 3.1. We then present our contributions on this topic, namely its extension to continuous and stochastic setting. This includes the Double Progressive Widening trick in Section 3.2, a way to improve the exploration of vast action spaces in Section 3.3, and a technique to transport information from one branch of the tree to the other in Section 3.4.

3.1 Finite MCTS

MCTS algorithm became popular when applied to the game of Go [54] [74, 43]. Before its application to this game, most algorithms failed to either explore all possible games by brute force (impossible with our current computing power) or accurately estimate the optimal value function of any given state (i.e. a given board configuration). Today, most, if not all algorithms of computer Go are based on MCTS.

In addition to the notations previously introduced, we will need the two following definitions.

Generative model of the environment. We write $M : \mathcal{X} \times A \mapsto \mathcal{X} \times \mathbb{R}$ the function that models the environment. More precisely, given a state x and a legal action a, it returns a couple (x', r), x' being the next state and r the instantaneous reward obtained. Both x' and r can be stochastic, and follow random distributions unknown to the agent.

Random action sampler. We write $s : \mathcal{X} \mapsto A$ the function that, given a state x, returns a legal action $s(x) = a$. This function can be stochastic, and follow an unknown random distribution.

We call *random policy* the policy that is to chose, in any state x, the first action returned by s.

If, for any $x \in \mathcal{X}$, the legal action space is $[0, 1]$, then a natural candidate for $s(x)$ would be the uniform distribution over $[0, 1]$. This specific case actually allows many other methods : biased sampling, equally spaced discretization, etc. But it

is important to note that our assumption (having a random action sampler) is less strong than knowing exactly what the action space is.

We will talk about this again in Section 3.3.1, where we mention some of the state of the art work that make use of the strong assumption that the action domain is known. In that section, we will also explain why, for real world applications like energy unit commitment problems, that assumption might not hold.

The state-action tree structure. We specify here what tree structure will be used throughout this work, to represent reachable states and feasible actions. We purposely use a general structure, that can be used for both deterministic and stochastic environment. This will simplify the explanations about the transition from finite and deterministic to infinite and stochastic settings.

To pave the way for a clear tree structure in stochastic environments, we differentiate two distinct and alternated kinds of nodes:

- *decision nodes*, where the agent needs to decide what action to take. These nodes also correspond to the agent being in a certain state $x \in \mathcal{X}$, and are then labelled by x.
- *random nodes*, where the generative model is used to simulate the transition associated to a certain couple state action (x, a), with a possibly random outcome. They will naturally be labelled as (x, a).

The root of the tree is made of the initial decision node $r \in \mathcal{X}$, of depth $d(r) = 0$. We define the depth of a node as half the distance between this node and the root. This way, the first layer of random nodes following the root are at depth 0.5, followed by a layer of decision nodes at depth 1, and so on.

For any node z (whether it is a decision or a random node), we write $n(z)$ the number of times this node has been simulated.

We show a generic representation of this tree structure, where decision nodes and random nodes appear in successive layers, until the time horizon is reached, in Fig.

Remarks:

- in the case of a deterministic environment, each random node has one and only one child.
- all final nodes are degenerated decision nodes, where no decision can me made

3.1.1 Algorithm description

In a nutshell, MCTS relies on the idea that, given a state x, if we have access to $Q^*(x, a)$ for all $a \in A$, acting greedily with respect to Q^* will give an optimal action, and the problem becomes a "trivial" optimization problem. The way MCTS approximates Q^* is that for all $a \in A$, it simulates random trajectories starting with (x, a). As the number of simulation grows, one can hope that the average reward obtained over these simulation will give a good estimate of $Q^*(x, a)$. This can only work if, at

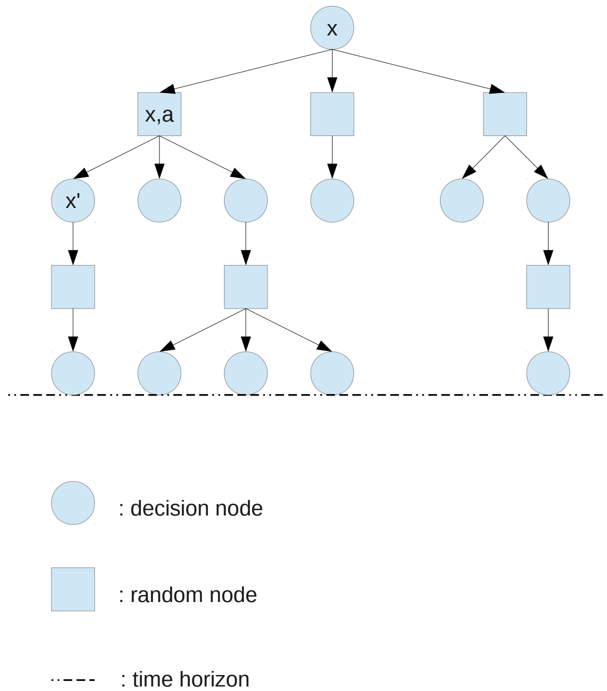


Figure 3-1: Generic tree structure for sequential decision making under uncertainty. In this figure, x is the initial state, the root of the tree. One of the explored feasible actions from x is a. One of the explored possible outcome of the pair (x, a) is x'.

some point, the simulations are not purely random. Otherwise, the algorithm would asymptotically act greedily with respect to V^s, the value function underlying the random policy:

$$\tilde{a^*} = \underset{a \in A}{\operatorname{argmax}} \left(r + V^s(f(x, a, w))\right) \tag{3.1}$$

This is where the *Tree* comes from, in order to actually converge to Q^*, one wants to bias the trajectories toward an optimal behaviour. This is achieved by slowly constructing an unbalanced tree, where nodes contain visited states, and branches represent actions. That way, as more simulations are made, one can spend most of the simulations on promising actions, thus giving them more weight in the average reward computation. Under certain assumptions, we will see that one can guarantee the consistency of MCTS, i.e. that the action chosen will be an optimal one.

We divide the formal description of MCTS into 5 algorithms. There are many other ways to describe it that can be found in the literature [35, 55]. This one is just the way that is the most convenient to explain our contributions in the later sections.

First, there is the main loop of MCTS, Alg. 10. Given some time budget and an initial state x, if slowly builds a tree of possible trajectories, and eventually returns the

best action, according to some pre-defined criterion, described in Alg. 14 (although there exist a few options to design this part of the algorithm [106], we just chose the most popular one, which is to pick the most simulated action). Then, we detail the way the tree is grown in Alg. 11. During one run through the tree, we need a function to dictate how to navigate in this tree, that we show in Alg. 12. We also need a function that evaluates a state, in order to avoid having to add every single visited state in memory. This last function is shown in Alg. 13.

One way to think of MCTS is to divide each iteration into *four phases*:

- (i) selection phase: we travel inside the tree (corresponds to a part of the GROWTREE function, and to the SELECT function);
- (ii) expansion phase: we decide to add a new node to the tree (part of the GROWTREE function);
- (iii) simulation phase: we use the default policy to simulate a sequence of actions until a final state is met (EVAL function);
- (iv) back-propagation phase: we use the information gathered during the simulation to update the tree (part of GROWTREE).

Algorithm 10 MCTS algorithm for finite states and actions

Input: initial state r, time budget B, generative model $M(x,a) = [x', r]$, action sampler $s : \mathcal{X} \mapsto A$, constant $K > 0$, and a default policy ϕ
Output: a chosen action $\widetilde{a^*}$
  Initialize $t_0 \leftarrow t$, and $\mathcal{T} \leftarrow r$
  while $t < t_0 + B$ **do**
    GROWTREE$(r, M(.,.), \alpha, K)$
  end while
  return BESTACTION$(\mathcal{T})$

Important features of the algorithm. This algorithm makes a few key assumptions in order to work properly:

- both the action space and the state space are finite. Otherwise, the SELECT function will keep selecting unexplored actions, and no node of depth higher than 1 would be added.
- the agent has access to a generative model of the problem, and to a random action sampler.
- the SDM problem has a finite horizon, at which an exact reward can be computed.

Algorithm 11 GROWTREE

Input: current tree $\mathcal{T}$, generative model $M(x,a) = [x', r]$, action sampler $s : \mathcal{X} \mapsto A$
Output: none, just updates the tree and possibly makes it grow

Initialize $x \leftarrow r(\mathcal{T})$, and $CR \leftarrow 0$
repeat
  $a \leftarrow$ SELECT(x, K)
  Children(x) ←Children$(x) \cup (x,a)$
  $[x', r] \leftarrow M(x,a)$ {M() generates a random possible outcome, and a reward}
  Children(x,a) ←Children$(x,a) \cup x'$
  $r(x,a) \leftarrow r$
  $x \leftarrow x'$
until $n(x) == 0$ or x is a final state
$CR \leftarrow$ EVAL(x, M, ϕ)
while x not root **do**
  $n(x) \leftarrow n(x) + 1$
  $(x,a) \leftarrow$ Father(x)
  $CR \leftarrow CR + r(x,a)$
  $CR(x,a) \leftarrow CR(x,a) + CR$
  $n(x,a) \leftarrow n(x,a) + 1$
end while

Algorithm 12 SELECT

Input: state-node x, $K > 0$ action sampler s
Output: an action a

$a_x = \text{argmax}_{a \in Children(x)} \widehat{Q}(x,a)$
with $\widehat{Q}(x,a) = \frac{CR(x,a)}{n(x,a)} + K \left(\frac{ln(n(x))}{n(x,a)}\right)^{\frac{1}{2}}$ if $n(x,a) > 0$, $+\infty$ otherwise
return a_x

Algorithm 13 EVAL

Input: state-node x, default policy φ, generative model M
Output: a real number R

initialize $R \leftarrow 0$
while x not final state **do**
  $a \leftarrow \varphi(x)$
  $[x', r] \leftarrow M(x,a)$
  $R \leftarrow R + r$
  $x \leftarrow x'$
end while
return R

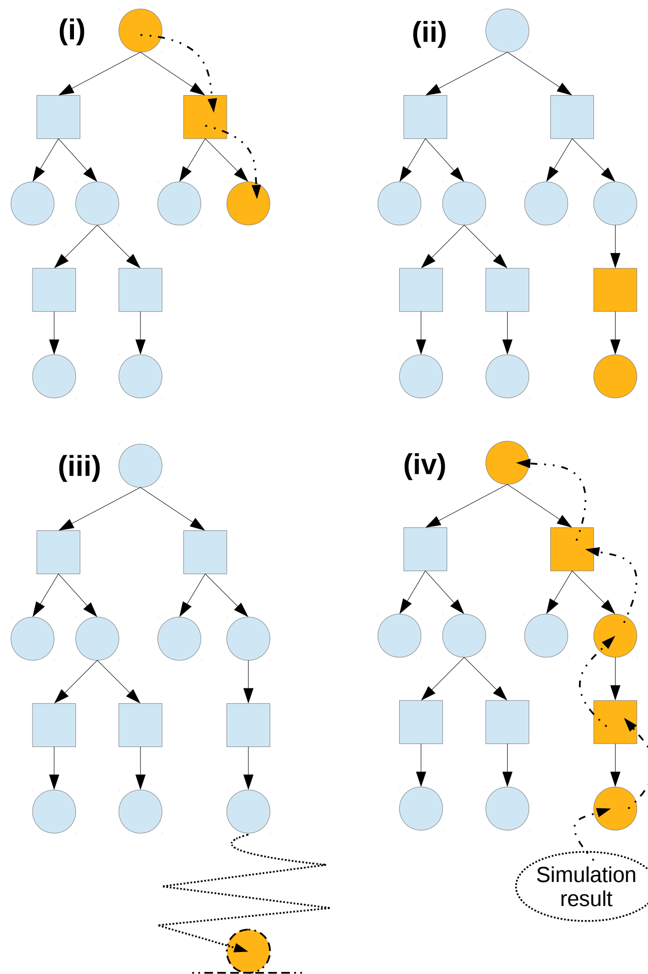


Figure 3-2: An example of the four phases of MCTS, from (i) to (iv). The orange color is used to indicate the parts of the tree that are used/modified at each phase.

These assumptions all hold true in the case of the game of Go, for example. They also hold true for many classical benchmark problems from the reinforcement learning community [112].

An essential part of the algorithm is in the function SELECT. After initially exploring all possible actions from a given state x, the agent has a very noisy estimation of the corresponding Q values. At this point, he needs to decide how to allocate simulations between the different actions. To allocate simulations uniformly between all actions would be a way to make sure no branch of the tree is neglected. To allocate all the simulations into the most promising action would be a risky attempt to invest computing power only in the optimal action. None of these extreme choices is perfect, and this is known as the *Exploration/Exploitation dilemma* (E/E dilemma), well studied by the multi armed bandit community [77, 99, 8], and applied to MCTS algorithms [74, 75].

Algorithm 14 BESTACTION

Input: a tree $\mathcal{T}$

Output: an action a

return $\text{argmax}_{a \in Children(r(\mathcal{T}))} n(r,a)$

In our case, investing one simulation into one action can be seen as "playing one arm". The true expected return of this arm would be $Q^*(x,a)$, and the sampled return would be the result of one sampled trajectory all the way to a final state. Under the strong assumption that the reward of all arms has a support equal to $[0,1]$, the UCB1 formula [8] provides optimal cumulated regret, i.e. an optimal compromise between exploration and exploitation. Even though these assumption do not hold, in most cases, in MCTS, the use of UCB1 formula in the selection function have shown very good result, and theoretical consistency [74] for finite state spaces.

One essential property of MCTS, that is sometimes overlooked, is its *any-time property*. MCTS can be interrupted at any time, and will return its best current recommended action. Of course, the longer it runs, the better the recommendation will be. When using one of the traditional methods described in the previous chapter, like SDDP, the running time of the algorithm is unknown, and interrupting it before it has finished will provide a worthless solution.

On the contrary, with MCTS, there is no minimal (and unknown) time budget. This is an especially interesting property for problems in very large dimension, like unit commitment. One might want an answer, the best available, before a fixed time.

Another interesting feature of MCTS is that it does not require the random process to be Markovian to work properly. Or, more precisely, the tree structure naturally embeds the history of a state. In any given node, if one wants to access its history, one can just travel from that node, upwards toward the root. This is as opposed to the traditional value approximation type of methods, that need to store that information in the state variable at all time, making it a challenge because of memory limitations.

Finally, one should note that MCTS uses an acyclic tree structure. This makes it fairly straightforward to parallelize. This feature is particularly attractive because most of the time, it is easier to obtain more machines than it is to increase the computing power of one single machine.

3.1.2 Improvements in MCTS applied to games

MCTS has been successfully applied to games, but this was often the result of problem specific improvements (even though the ideas themselves can be used on different applications). We will cover some of the tricks that made MCTS successful in the game of Go, and then mention some of the other accomplishments related to MCTS.

MCTS applied to computer Go. One of the biggest improvements to MCTS applied to the game of Go came from the customization of the rollouts by using domain

knowledge [57]. Some work has been done to learn patterns from past games, using ELO ratings, and to use these patterns both in the rollouts and to apply pruning in the tree [45]. In [54], the authors add offline knowledge (value function approximates) to guide exploration at the early stages of MCTS, when little online knowledge is available. We show an extension of this method, called Rapid Value Estimation (RAVE), to continuous domains in Section 3.4. The addition of automatically generated opening books (from past professional games or hand made opening books), in [6, 70], is another way of adding human knowledge in MCTS, considerably improving its performances. In [56], the authors use a tuned constant to adjust the balance between exploration and exploitation in the game of Go, thus stepping away from the default way that consist in exploring all actions first before starting exploitation.

Finally, MCTS is very well suited for parallelization[31, 53, 34, 71]. As an any time algorithm, the benefits of more computing power are almost always beneficial, even though it can be hard to predict by how much.

The very basic version of MCTS uses almost no knowledge of the problem: no convexity hypothesis, no explicit knowledge of the feasible action space, no assumption of continuity, etc. This means that in its simplest form, MCTS is essentially a guided blind search, using a tree structure and exploration/exploitation balance to slowly direct the search in the good direction. Most of the significant improvements in computer Go have then been about adding offline knowledge, hand coded or generated, into the algorithm. The topic of where such heuristics should be included in MCTS applied to computer Go is reviewed in more details in [48].

Application to other games. The idea of decisive move has been applied to MCTS in the game of Havannah in [115] (a decisive move is a move that guarantees a player a win, an anti-decisive move is a move required to avoid a loss). Using this idea can lead to huge gains in computational efficiency by pruning large parts of the tree.

To deal with large branching factors, one option is to aggregate successive moves in move groups, also known as macro actions. This has been investigated and shown to be successful in the game of Go in [36].

Opening books were also applied to the game of Amazons in [73].

3.2 Infinite domain: continuous MCTS

In this section, we investigate the consequences of an infinite support of the probability distribution underlying the transition, and infinite action spaces, on MCTS. Here and in the rest of this work, what we mean by *infinite set* is a set with infinite cardinal. This set can be bounded or unbounded. Infinite action spaces are encountered in many situation, like in problems where the action is the acceleration of a vehicle, the amount of dollars to invest in a stock, or how much water should be stored in a hydro electric dam. In practice, results covering the infinite action space case can be used to handle problems where the action space is simply too large relatively to the number of iterations.

Problems with infinitely many possible random outcomes in the transition function are also easily found. One can encounter such an environment when the randomness comes from oil price fluctuations, daily precipitations, or noisy robotic control, for instance. In these cases, the random variables w can take infinitely many different values. Formally, this means that for any couple $(x, a) \in \mathcal{X} \times A$, the support of the random variables $f(x, a, w)$ and $g(x, a, w)$ are infinite. This means that it is impossible to explore every single possible outcome. One can only approximate the real distribution.

In the first part, we present some of the existing work made to extend MCTS to continuous actions, on which some of our contributions are based. The second part will show a simple example where this state of the art version of MCTS does not suffice when the possible random outcomes to the transition are also infinite. Finally, we present our contribution, a working extension of MCTS to fully continuous MDPs, namely the Double Progressive Widening trick (DPW)[39].

3.2.1 MCTS-SPW for infinite action space

Whereas in a finite action setting, one can explore all legal actions from a state at least once very quickly, it is not doable when the number of actions gets too big (in particular when this number is infinite). Since one cannot explore all actions at once, the idea is instead to progressively explore new actions. What remains to decide is the rate at which new actions are explored.

Progressive strategies have been proposed in [45, 33] for tackling problems with big action spaces; they have been theoretically analyzed in [119], and used for continuous spaces in [103, 104]. We will here define a variant of progressive widening.

Simple Progressive Widening (SPW). Let us focus on the E/E dilemma mentioned above. The goal is for the agent to decide how to allocate simulations to the different actions feasible from its current state x. Many papers have been published on such problems, in particular around upper confidence bounds[78, 7]. Given some state x, some information from past simulations, Upper Confidence Bounds, in its simplest version, recommends to choose any unexplored action if there is still at least one, and a_{UCB} maximizing $UCB(a)$ otherwise, with:

$$UCB(a) = \frac{CR(x, a)}{n(x, a)} + K \left(\frac{ln(n(x))}{n(x, a)} \right)^{\frac{1}{2}} \tag{3.2}$$

Variants of the score function are termed ‘bandit algorithms’; there are plenty of variants of the score formula.

A trouble in many mathematical works around such problems is that the set $A(x)$ is usually assumed small in front of the number of iterations. More precisely, the behaviour of the algorithm above is trivial for $t \leq \#A(x)$. [120] proposed the use of a constant x such that $n(x, a) = 0 \Rightarrow UCB(a) = s$; this is the so-called First Play Urgency algorithm. There are other specialized efficient tools for bandits used in “trees” such as rapid action value estimates [54, 51]; however these tools assume

some sort of homogeneity between the actions at various time steps. [44, 119, 33] proposed progressive strategies for big/infinite sets of arms.

The principle is as follows for some constants $C > 0$ and $\alpha \in]0,1[$ (as it is independent of the algorithm used for choosing an action, within a given pool of possible actions, we do not explicitly write a score function as above). We write x the state we are currently in, $A(x)$ the entire set of feasible actions in x, and $\overline{A(x)}_t \subset A(x)$ the set of explored actions at iteration t. We initialize $\overline{A(x)}_0 = \emptyset$. For each $t \geq 0$, we apply the following simple algorithm:

Algorithm 15 Simple Progressive Widening method

Input: state x, $t \geq 0$, $\overline{A(x)}_t \subset A(x)$
Output: an action a
  if $n(x)^\alpha > \#\overline{A(x)}_t$ **then**
    $a \leftarrow s(x)$
    $\overline{A(x)}_{t+1} \leftarrow \overline{A(x)}_t \cup \{a\}$
  else
    $a \leftarrow \text{argmax}_{a \in \overline{A(x)}_t}\, score(x, a)$
  end if
  return a

The key point is that the chosen action is restricted to a finite subset of $A(x)$. And, progressively, $\overline{A(x)}$ increases in size, by adding to it randomly sampled actions are increasingly rare occasions.

This algorithm has the advantage that it is any-time: we do not have to know in advance at which value of t the algorithm will be stopped. [44] applied it successfully in the very efficient CrazyStone implementation of Monte-Carlo Tree Search [45]. Upper Confidence Tree (or Monte-Carlo Tree Search) is not a simple setting as above: when choosing an action, we reach a new state; one can think of Monte-Carlo Tree Search (or UCT) as having one bandit in each possible state s of the reinforcement learning problem, for choosing between (infinitely many) actions.

Applying the SPW technique to MCTS means that we modify the SELECT function as follows (other functions remaining unchanged):

It is important to keep in mind that the progressive widening algorithm is applied in each visited state; some states might be visited only once, or never, and some other states are visited very often. MCTS with progressive widening or progressive strategies is the only version of MCTS which works in continuous action spaces [104, 103]; however, it was applied only with the property that applying a given action a in a given state s can lead to finitely many states only. We will see that this methodology (the algorithm above) does not work as is in the case in which there is a null probability of reaching twice the same state when applying the same action in the same state (i.e. typically it does not work for stochastic transitions with continuous support).

Algorithm 16 SELECT with SPW

Input: state-node x, $K > 0$ action sampler s, parameter $1 > \alpha > 0$
Output: an action a
  if $n(x)^\alpha > \#\text{Children}(x)$ **then**
    $a_x \leftarrow s(x)$
    $\text{Children}(x) \leftarrow \text{Children}(x) \cup \{(x, a)\}$
  else
    $a_x = \text{argmax}_{a \in Children(x)} \widehat{Q}(x, a)$
    with $\widehat{Q}(x, a) = \frac{CR(x,a)}{n(x,a)} + K \left(\frac{ln(n(x))}{n(x,a)}\right)^{\frac{1}{2}}$
  end if
  return a_x

3.2.2 Why MCTS-SPW fails on infinite state space

The key part of MCTS that is affected by the stochasticity of the environment is Alg. 11, in the first "while loop", right after picking one action a. We take a look at two simplistic ways to deal with this part of MCTS in a stochastic setting with infinite outcomes, and explain why it is probably a bad idea.

The worst idea: apply deterministic MCTS. In the case of a deterministic environment, the way to handle what follows the choice of an action a is quite straight forward, and can be divided in two cases:

– the action has been explored at least once in the past, in which case (x, a, x') is in the tree, where x' is the only possible outcome of this transition, and $n(x') > 0$.

– the action has never been explored before. We pass (x, a) to the model, and obtain a new state x', and (x, a, x') is added to the tree.

It is relatively straightforward to see that in this case, any action a in the tree will have only one child node. This means that each series of actions $a_0, ...a_H$ represented in the tree is only simulated once. The resulting Q values are then biased in favor of lucky trajectories. Formally, the recursive equation implemented by this version of MCTS is:

$$Q(x, a) = g(x, a, w) + \max_{a \in A} \gamma Q(f(x, a, w), a) \tag{3.3}$$

where w follows the distribution $p_w(.|x, a)$. There is no expectation in the equation because only the first sampled disturbance w is taken into account for each couple (x, a) in the tree.

The false good idea: SPW. In the case of a stochastic environment with a finite support for the distribution followed by the disturbance, it makes sense to always call the generative model when travelling through a random node (x, a) in the tree.

Indeed, one will eventually sample all possible outcomes (x, a, x'), and start building long branches (i.e. trajectories made of more than one action).

However, when the support $X(x, a)$ of $f(x, a, w)$ is infinite, we have, for any $x' \in X(x, a)$, $P(f(x, a, w) = x') = 0$. This means that no decision node besides the root node gets more than 1 visit. In other words, the tree gets wider and wider, never growing longer than a depth 1. Any data obtained from sampled actions after depth 1 is only the result of actions chosen by the default policy. The estimate of the value function of actions chosen from the root can be, at best, as good as this default policy.

In the case where the default policy is to chose a random action uniformly over the feasible action set, this would mean that we evaluate all immediately reachable states as if all actions taken after were random. Formally, the update function implemented by such an algorithm is:

$$Q(x, a) = g(x, a, w) + \max_{a \in A} \gamma Q^{\varphi}(f(x, a, w), a) \tag{3.4}$$

In the case of our car driving example, if acting randomly has a high chance of crashing the car quickly, one can see how this method of evaluating states would give almost no information about optimal actions. This would, however, give information about the value of the default policy, for each (x, a) added to the tree.

3.2.3 Proposed solution: DPW

In the previous section, we presented SPW, and explained why it is not sufficient to guarantee the consistency (i.e. convergence to the optimal action) in the case where the support of the disturbance is infinite. In this section, we provide a solution, namely the Double Progressive Widening method. Applied to MCTS as described above, it requires the modification of the GROWTREE function and the addition of a function to select decision nodes from random nodes, SELECT OUTCOME.

The idea remains similar to the one of SPW: as more simulations are allocated to a random node (x, a), we want to maintain a balance between exploration and exploitation. To do so, some of the simulations are spent widening the tree (adding new random outcomes from (x, a)), and some are spent going deeper in the tree (exploiting known occurrences (x, a, x')).

At this point, we need to introduce an important notation: given a random node (x, a), and a reachable state x', we call number of occurrences of (x, a, x') the number of times x' was obtained from the generative model applied to (x, a). We write this number $n(x, a, x')$. Note that the following equations hold, for any random node (x, a), and for any reachable state x' from that node:

$$n(x, a) \geq \sum_{x' \in C(x,a)} n(x, a, x') \text{ and } n(x') \geq n(x, a, x') \tag{3.5}$$

We now formally described the modified GROWTREE function and the added function SELECT OUTCOME below:

This algorithm is not so intuitive, for the second progressive widening part. The

Algorithm 17 SELECT OUTCOME

Input: random node (x, a), $K > 0$ action sampler s, $1 > \beta > 0$
Output: a reachable state x' and its immediate reward r
  if $n(x, a)^\beta > \#\text{Children}(x, a)$ **then**
    $[x', r] \leftarrow M(x, a)$
    $\text{Children}(x, a) \leftarrow \text{Children}(x, a) \cup \{x'\}$
  else
    Choose a decision node $x' \in C(x, a)$ with probability $\frac{n(x,a,x')}{\sum_{x_i \in C(x,a)} n(x,a,x_i)}$
    $r \leftarrow r(x')$
  end if
  return $[x', r]$

Algorithm 18 GROWTREE with DPW

Input: current tree $\mathcal{T}$, generative model $M(x, a) = [x', r]$, action sampler $s : \mathcal{X} \mapsto A$
Output: none, just updates the tree and possibly makes it grow
  Initialize $x \leftarrow r(\mathcal{T})$, and $CR \leftarrow 0$
  repeat
    $a \leftarrow \text{SELECT}(x, K)$
    $\text{Children}(x) \leftarrow \text{Children}(x) \cup (x, a)$
    $[x', r] \leftarrow SELECTOUTCOME(x, a)$
    $\text{Children}(x, a) \leftarrow \text{Children}(x, a) \cup x'$
    $r(x, a) \leftarrow r$
    $x \leftarrow x'$
  until $n(x) == 0$ or x is a final state
  $CR \leftarrow \text{EVAL}(x, M, \phi)$
  while x not root **do**
    $n(x) \leftarrow n(x) + 1$
    $(x, a) \leftarrow \text{Father}(x)$
    $CR \leftarrow CR + r(x, a)$
    $CR(x, a) \leftarrow CR(x, a) + CR$
    $n(x, a) \leftarrow n(x, a) + 1$
  end while

idea is as follows:

- If $n(x,a)^\beta$ is large enough, we consider adding one more child to the pool of visited children: we simulate a transition and get a state x'. If we get an already visited child, then we go to this child; otherwise, we create a new child.
- If $n(x,a)^\beta$ is not large enough, then we sample one of the previously seen children. As they are not necessarily equally likely, we select a child proportionally to the number of times it has been generated.

The algorithm has been designed with a "consistency" objective in mind, which is twofold:

- Infinite visiting: we want that if a node is visited infinitely often, then we generate infinitely many children, and each of these children is itself visited infinitely often. By induction, this property ensures that all created nodes are visited infinitely often. Progressive widening and the UCB formula (or many other formulas in fact) ensure this property.
- Propagation: the average reward of any node visited infinitely often converges to a limit and this limit (for a non-terminal node) is the average reward corresponding to its children which have best asymptotic average reward. This property is ensured by the careful sampling in the progressive widening.

3.2.4 The Trap problem

We provide here some simple empirical results showing the difference between the standard MCTS (with SPW) and our modified version, MCTS-DPW. The domain used to compare the two algorithms is as simple as possible, but still shows the fundamental weakness of MCTS-SPW.

Problem description.

This problem has been designed to clearly illustrate the weakness of the simple progressive widening. In this problem, one has to make two successive decisions, in order to maximize the reward. As we will see, the optimal policy is to make a risky move at the first step, in order to be able to obtain the maximum reward on the second (and last) step. The state will be denoted x, and is initialized at $x_0 = 0$. At each time step t the decision is denoted $d_t \in [0,1]$. Let $R > 0$ be the noise amplitude at each time step. At a time step t, given the current state x_t and a decision d_t, we have:

$$x_{t+1} = x_t + d_t + R \times Y,$$

Y being a random variable following a uniform distribution on $[0,1]$.

The trap problem relies on five positive real numbers: the high reward h, the average reward a, the initial ramp length l, and the trap width w. The high reward will be given if and only if we cross the trap, otherwise we obtain 0. If we stay on the

initial ramp, we get the average reward. We thus define the reward function $r(\cdot)$ as follows:

$$r(x) = \begin{cases} a & \text{if} \quad x < l \\ 0 & \text{if} \quad l < x < l + w \\ h & \text{if} \quad x > l + w \end{cases}$$

The objective is to maximize $r(x_0) + r(x_1)$, the cumulated reward.

The shape of the reward function is shown in Fig.3-3.

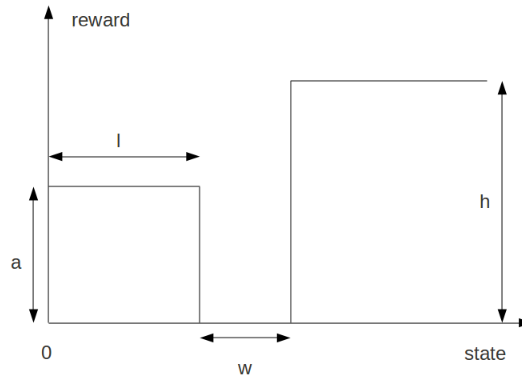


Figure 3-3: Shape of the reward function: Trap problem.

Experimental results.

We compare simple progressive widening and double progressive widening on the trap problem. In our experiments, we used the following settings: $a = 70$, $h = 100$, $l = 1$, $w = 0.7$, $R = 0.01$. With these parameters, the optimal behaviour is to have the first decision $d_0 \in [0.7, 1]$ and $d_1 \geq 1.7 - d_0$. If one makes optimal decisions, one has an expected reward of $r^* = 170$. That is the reward toward which the Double progressive widening version of Monte Carlo Tree Search converges. However, the Simple progressive widening version does not reach this optimal reward. Worse, as we increase the computation time, it becomes less efficient, converging toward a local optimum, 140.

The mean values of the rewards are shown in Fig. 3-4 and the medians of the rewards are shown in Fig. 3-5. Each point is computed according to 100 simulations.

3.2.5 Conclusion

We introduced *Double Progressive Widening*, in order to make MCTS work on continuous domains. As our simple experiment shows, it is necessary to change the basic version of MCTS (i.e. the simple widening version) in order to make it converge on continuous domain. The main reason behind this is the infinite support of the random process of the SDM. Because of it, when the probability of generating the same state

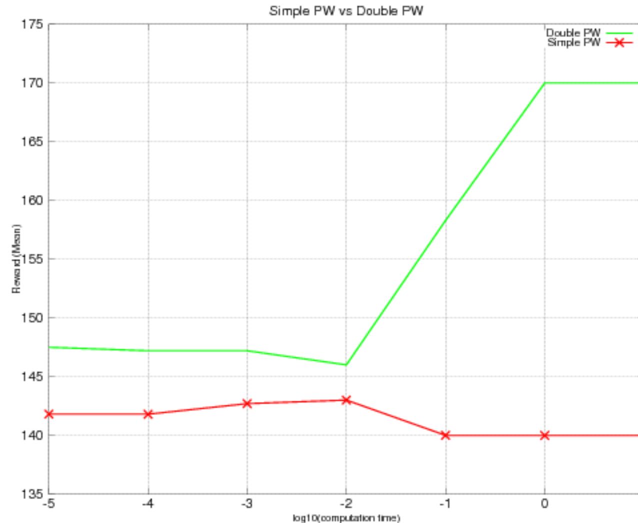


Figure 3-4: Mean of the reward, for the trap problem with $a = 70$, $h = 100$, $l = 1$, $w = 0.7$, $R = 0.01$. The estimated standard deviations of the rewards are $STD_{DPW} = [13.06, 12.88, 12.88, 12.06, 14.70, 0, 0]$ for Double PW and $STD_{SPW} = [7.16, 7.16, 8.63, 9.05, 0, 0, 0]$ for Simple PW - the differences are clearly significant, where STD means standard deviation.

twice is equal to zero, SPW-MCTS fails. In Chapter 4, we will prove the consistency of a version of MCTS based on DPW.

3.3 Exploring infinite action spaces

In this section, we consider the challenge posed by infinite action spaces. One obvious problem is that all actions cannot be explored. A number of solutions have been proposed: discretization, SPW, and others that we will review in Section 3.3.1. Another interesting problem is how to use the information gathered from several data points in a large action space to guide the search in the rest of that space.

First, we will look at the state of the art methods trying to tackle infinite action spaces, and to sometimes use it at their advantage. Then, we will present our contribution on this topic, namely the Blind Value function[38].

3.3.1 Existing methods

Recent impressive results in the field of planning with MCTS variants in continuous MDP have been published; most of them, as far as we know, rely on a discretization of the action space. This the case of HOOT [84] and HOLOP [122] that both rely on the HOO algorithm, introduced in [26]. HOO is a bandit algorithm that deals with continuous arms by using a tree of coverings of the action space, which requires,

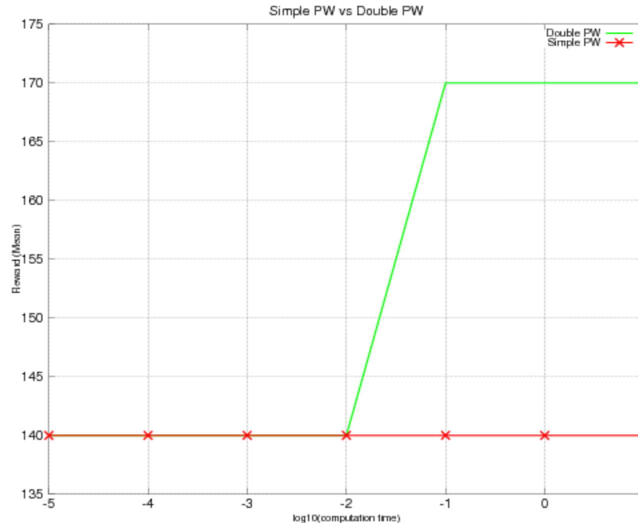


Figure 3-5: Median of the reward, for the trap problem with $a = 70$, $h = 100$, $l = 1$, $w = 0.7$, $R = 0.01$

in their work, the action space to be compact, with known bounds. Other notable contributions using a discretization of the action space are [72] and [11]. What these methods have in common is the assumption that the action space is continuous, but that we have enough knowledge about it to divide it in a certain number of equally spaced actions. Or, in the case of HOO, it is required to have a compact action space with known bounds. In toy benchmark problems like inverted pendulum, this is straightforward. However, in more realistic applications, this can be difficult. This is the case of the unit commitment problem, as described in [20] , where the agent needs to decide at each time step how to use a wide array of energy production facilities: water stocks, thermal plants, nuclear plants, etc. This problem has an action space that cannot be easily discretized. First, it has both discrete and continuous components (some power plants having a minimal energy output). Second, there are many operational constraints, making the action space non convex, and the bounds hard to find. In practice, finding feasible actions can come down to adding noise to the objective function of a simplified version of the problem, applying a Linear Programming method on said simplified problem, and using the result as a feasible action. There are many other options to sample a feasible action, but raw discretization is not one of them.

3.3.2 Our contribution: Blind Value (BV)

The principle of Blind Value is to help the exploration of new decisions. One can want to explore a new decision from any state already in the tree. Although in our case, the optimizer cannot bias the sampling of new decisions, we propose a method that does use the information available in the tree. More precisely, we use the information

about the children of the current node. In terms of states and decisions, it means: when we want to explore a new decision from state x, we use information about all the decisions explored from this state x in the past simulations to select a new decision $a \in A(x)$ to be explored from state x.

Note that one could use any other information in the tree: brother nodes, grand children nodes, father node, etc. However, even the exploitation of the direct children of a node only is computationally costly. And, the more distant in the tree some information is, the more likely it is to be irrelevant to the node we are currently in (states might be very different, and this type of problem is also highly time step dependant). [54] has proposed the use of Rapid Action Value Estimates (RAVE), which are an interesting other possibility; we will consider the mixing of blind value with RAVE values in a further work. [54] also proposed the use of information from related nodes; after preliminary positive results, this was later removed from the corresponding implementations (for the game of Go) for correctly tuned implementations.

The idea of BV is to try to explore decisions that are far away from known decisions during the first simulations, and then to focus on areas that have a lot of decisions with high UCB values. This is done by sampling a number of new decisions, and by selecting one of them according to a combination of these two criterions (explore unknown regions and explore regions with many decisions with high UCB values in it).

More precisely, we sample a number $M \geq 1$ of random decisions, and we pick the one that is the most interesting to explore. The way we measure the interest of a decision is through a function from the decision space to $\mathbb{R}$, denoted $BV(.)$ (Blind Value). This function can be defined in many different ways. In our proposed method, at an iteration n, given a state x and a decision $a \in A(x)$, we chose to define $BV(a)$ as the minimum over $D_n(x)$ of the sum of two parts. The first part is the UCB value of $a_i \in A_n(x)$, the second is the distance between $a \in A(x)$ and $a_i \in A_n(x)$, multiplied by an adaptation coefficient. We use the standard euclidean distance, but any other distance could be used instead.

More precisely, the Blind Value of a in state x with actions $a_i \in A_n(x)$ already explored is

$$BV(a) = \min_{a_i \in A_n(x)} \{UCB(a_i) + \rho dist(a_i, a)\} \tag{3.6}$$

What follows is the detailed blind value algorithm:

Exploration of new decisions
Input: a state x, a set D of already explored decisions, an integer M, and a distance function over the decision space, $dist$
Output: an unexplored decision a.
Generate M random decisions. Let A_{pool} be the set composed of these decisions.
Compute $\sigma_{known} = UnbiasedStandardDeviation_{a \in D}(UCB(a))$
Compute $\sigma_{pool} = UnbiasedStandardDeviation_{a \in A_{pool}}(dist(a, 0))$, 0 being the center of the domain
Compute $\rho = \frac{\sigma_{known}}{\sigma_{pool}}$
return $a = argmax_{y \in A_{pool}} BV(y, \rho, D)$

Computing BV (Blind Value)
Input: an unexplored decision y, a real number ρ, and D the set of explored actions
Output: a real number $BV(y, \rho, D)$.
return $min_{d \in D}(\rho \times dist(d, y) + UCB(d))$

3.3.3 Experimental comparison

Our test case is an energy management problem, as described in Section 1.2.3. There are N energy stocks, H time steps, and a thermal power plant with a given maximum capacity and production cost function. In our experiments, we used a quadratic cost function. At each time step, each stock also receives an inflow. Each inflow follows its own independent random distribution.

At each time step, the decision maker has to decide how much to produce from each stock, and how much to produce from the thermal plant. His goal is to satisfy a time varying demand at the lowest possible cost.

We ran two algorithms on this problem: the continuous version of MCTS, as introduced in [39], and the same algorithm with the addition of Blind Value (MCTS-BV), with the sample size parameter set to 20. This experiment was run with 12 stocks and 16 time steps. The results are shown in fig 3-6. In this experiment, as in the following ones, each point is computed from 10000 runs of the algorithm on one problem instance. The 95% confidence intervals are plotted as blue segments around the points, even though their small size can make them very hard to see in some cases.

This figure shows that even in dimension 12, BV already gives an edge of magnitude 10 to MCTS, in terms of computation time (to reach a certain level of performance, MCTS requires 10 times as many simulations as MCTS-BV). The problem

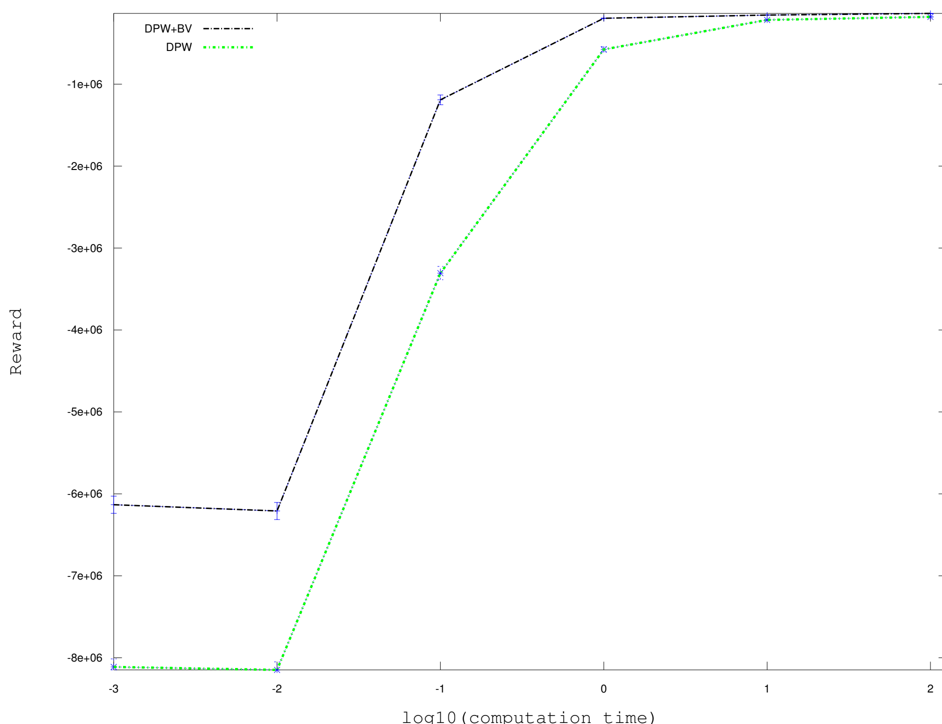


Figure 3-6: Reward, as a function of the computation time. Problem settings: 12 stocks, 16 time steps. MTCS with BV is 10 times faster than MCTS, for budgets up to 10 seconds per decision.

being reasonably easy, we also see that this edge decreases when the computation time increases (but only for a computation time of about 3 minutes). This is due to the fact that, as the budget gets bigger, both algorithms get very close to the optimum, in terms of reward.

This is why we made a second experiment, on the same problem, with a much higher dimension. In this experiment, there are 80 stocks, 6 time steps, and $M = 640$. With the information given to the algorithms (just the transition and the sampling functions), this problem is incredibly difficult, and naturally has very low reward (the highest average possible reward being around -2.5×10^7). Given its dimension, there is no way of exploring the entire decision space, even with a very low density. The results are shown in Fig. 3-7.

This figure shows that BV still gives an edge of magnitude 10 to MCTS in terms of computation time, even though we were not able to approach the optimum with our computing capacities. One can also note that in this setting, the difference seems to be increasing as the budget increases. This leads to think that on very difficult problems, BV can divide by ten, or even more, the computing time necessary to reach a certain level of performance.

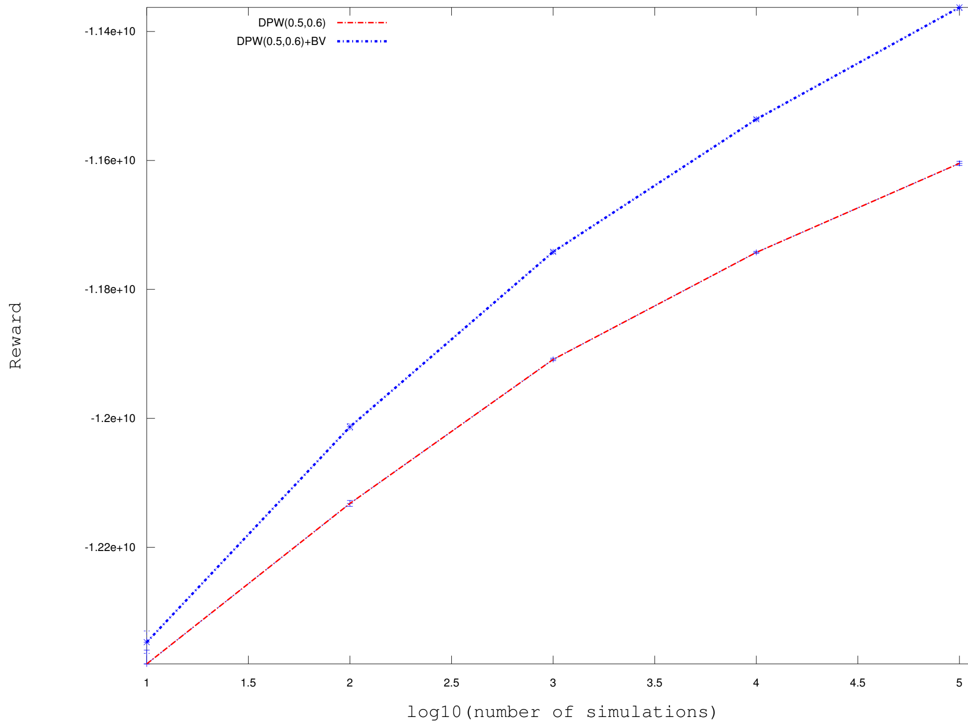


Figure 3-7: Reward, as a function of the number of simulations per decision. Problem settings: 80 stocks, 6 time steps. MTCS with BV is 10 times faster than MCTS, for all budgets.

3.3.4 Conclusion

Our contribution, *Blind Value*, only requires a random action sampler to work. No assumptions on the continuity of the reward function with respect to the action variables are made. Also, we do not use the shape of the feasible action space. Although we do not have theoretical guarantees, it improved the convergence speed of MCTS by a factor 10 in our experiments. It can be seen as a way to approximately discretize the action space progressively, and slowly focus the computing effort in the regions that return good rewards.

As a future work, we would like to work on a way to improve BV's meta-parameters online, as more information is gathered during simulations.

We should also investigate the case where the action space is not only of infinite cardinal, but also *unbounded*. Our experiments were made on a case where actions are bounded, and we do not know how our heuristic would perform if this was not true.

3.4 Generalization: continuous RAVE

This section's content can be, for the most part, be found in [40].

The main two heuristics combined with UCT aim at guiding the exploration strategy, through limiting the number of considered actions with *Progressive Widening* (PW) [45, 39, 119], and selecting the most promising actions with *Rapid Action Value Estimate* (RAVE).

While RAVE is acknowledged to be a key factor of MCTS efficiency, to our best knowledge it has been limited until now to discrete action and state spaces. Motivated by applications in management and robotics, this contribution focuses on extending RAVE to continuous action and state spaces using a Gaussian convolution-based smoothing (section 3.4.2). The proposed approach is experimentally validated on two problems, the artificial treasure hunt benchmark, and a real-world energy management problem (section 3.4.3). The paper concludes with a discussion and some perspectives for further research.

3.4.1 Rapid Action Value Estimation: the finite case

First pioneered in the context of computer-Go [54], Rapid Action Value Estimation (RAVE) aims at a more robust assessment of actions, through sharing the rewards gathered along different subtrees of the game tree. Formally, let $Q_{RAVE}(x, a)$ denote the empirical reward averaged over all tree-walks where action a has been selected after visiting state x, and let $m(x, a)$ be the number of such tree-walks. A variant of the UCT-policy is defined as follows:

$$\pi_{RAVE}(x) = \text{ argmax } \left\{ Q^{\oplus}_{RAVE}(x, a) = Q_{RAVE}(x, a) + C' \sqrt{\frac{\log m(x)}{m(x, a)}}, a \in \mathcal{A} \right\} \quad (3.7)$$

with $m(x)$ being the sum of $m(x, a)$ over all actions a.

Although taking more tree-walks into account contributes to a faster convergence of the action value estimate, $Q_{RAVE}(x, a)$ is a biased estimate of $Q(x, a)$, and should therefore be replaced by the true estimate $Q(x, a)$ whenever $n(x, a)$ permits to do so with reasonable confidence. It thus comes naturally to consider a dynamic weighted average of $Q_{RAVE}(x, a)$ and $Q(x, a)$, defining

$$\pi_{UR}(x) = \text{ argmax } \left\{ Q^{\oplus}_{UR}(x, a), a \in \mathcal{A} \right\} \quad (3.8)$$

with

$$\begin{aligned} Q^{\oplus}_{UR}(x, a) &= \beta(x, a) Q^{\oplus}_{RAVE}(x, a) + (1 - \beta(x, a)) Q^{\oplus}_{UCT}(x, a) \\ \beta(x, a) &= \sqrt{\tfrac{k}{3n(x,a)+k}} \end{aligned} \quad (3.9)$$

where the *equivalence parameter* k represents the (domain-dependent) number of tree-walks required for the unbiased $Q_{UCT}(x, a)$ to provide as reliable an estimate as $Q_{RAVE}(x, a)$.

3.4.2 Continuous Rapid Action Value based Estimation

This section presents the proposed extension of RAVE to the case of continuous action spaces and continuous space states.

Continuous action spaces

While the presented discrete RAVE approach supports the fast estimation of action values, its reliability decreases as the number of actions which can be taken into account increases everything else being equal. Indeed in a continuous action space $\mathcal{A}$, the number of times a given action is tried is 0 in expectation, which renders RAVE useless.

It thus comes naturally to consider a smooth estimate of action values, e.g. using Gaussian convolution. Formally, given a training set $\mathcal{D} = \{(x_i, y_i), i = 1 \dots n, x_i \in \mathbb{R}^d, y_i \in \mathbb{R}\}$, a Gaussian estimate of the value y associated to some $x \in \mathbb{R}^d$ is defined as

$$\hat{y}_\sigma(x) = \frac{1}{\sum_{i=1}^{n} e^{-\frac{1}{\sigma^2} d(x,x_i)^2}} \sum_{i=1}^{n} e^{-\frac{1}{\sigma^2} d(x,x_i)^2} \times y_i$$

where σ is a smoothing parameter weighting the relative importance of the nearest neighbors of x and $d(x, x')$ stands for the chosen distance on the space. In the remainder of this section, only the Euclidean distance on $\mathbb{R}^d$ will be considered. In applications, prior knowledge about the application domain is provided through the choice of the distance.

Along this line, let $l_x = x.a_0 \dots x_i.a_i \dots$ denote a tree walk starting in x and let $\mathcal{R}(l_x)$ denote the associated cumulative empirical reward. $Q_{RAVE,a}(x, a)$ is defined as:

$$Q_{RAVE,a}(x,a) = \frac{1}{\sum_{l_x,\ a_i\ in\ l_x} e^{-\log N_a \frac{d(a,a_i)^2}{\alpha_{action}}}} \sum_{l_x,\ a_i\ in\ l_x} e^{-\log N_a \frac{d(a,a_i)^2}{\alpha_{action}}} \times \mathcal{R}(l_x) \quad (3.10)$$

where α_{action} is a problem dependent parameter (proportional to the square dimension of the action space for the sake of homogeneity); N_a denotes the overall number of actions involved in all l_x, and the $\log N_a$ term is meant to peak the Gaussian convolution as the available empirical evidence increases. Counter $n(x, a)$ is likewise estimated using Gaussian convolutions and $\beta(x, a)$ is computed from $n(x, a)$ (Eq. (3.9)).

Both Q_{RAVE} and $Q_{RAVE,a}$ consider all tree-walks visiting state s and the cumulative reward gathered thereafter. The difference is that Q_{RAVE} only considers those tree-walks which have executed action a, whereas$Q_{RAVE,a}$ considers them all with a weight which decreases exponentially depending on the distance between the executed actions and the considered action a. As $Q_{RAVE,a}$ is even more biased than Q_{RAVE} (since it takes all actions into account, though weighted), one considers also the dynamic combination of Q_{RAVE} and $Q_{RAVE,a}$ as in Eq. (3.9)). One defines:

$$Q^{\oplus}_{UR_a}(x,a) = \beta(x,a)\, Q^{\oplus}_{RAVE,a}(x,a) + (1-\beta(x,a)) Q^{\oplus}_{UCT}(x,a)$$

and $\pi_{UR_a}(x)$ selects the action maximizing $Q^{\oplus}_{UR_a}(x, a)$.

Note that $Q_{RAVE,a}(x, a)$ is computed for a finite subset of $\mathcal{A}$ only, due to the progressive widening effects: only a finite number of actions is considered in each state node. The associated continuous rapid action value estimate is updated after each tree-walk.

Continuous state spaces

As already said, $Q_{RAVE,a}$ and Q_{RAVE} alike are strongly biased as they take into account every tree-walk conditionally to their visiting s and executing a or some similar action thereafter, although this action might be executed in a state s' very different from s.

In the case of continuous state spaces, it thus comes naturally to weight the contribution related to some state-action pair (x_i, a_i) depending on the distance between s and s_i. Formally, let us define $Q_{RAVE,a,x}(x, a) =$

$$\frac{1}{\sum_{l_x,\ s_i.a_i\ in\ l_x} e^{-\log N_{a,x}\left\{\frac{d(x,s_i)^2}{\alpha_{state}}+\frac{d(a,a_i)^2}{\alpha_{action}}\right\}}} \sum_{l_x,\ s_i.a_i\ in\ l_x} e^{-\log N_{a,x}\left\{\frac{d(x,s_i)^2}{\alpha_{state}}+\frac{d(a,a_i)^2}{\alpha_{action}}\right\}} \times \mathcal{R}(l_x) \tag{3.11}$$

As in Eq. 3.10, constant α_{state} is problem-dependent and proportional to the square dimension of state space, and $N_{a,x}$ is used to peak the Gaussian convolution as the available evidence to estimate $Q_{RAVE,a,x}$ increases.

Discussion

The proposed Continuous RAVE (cRAVE) heuristics involves two additional problem dependent parameters α_{action} and α_{space}, respectively involved in Eqs. (3.10) and (3.11). Note that $Q_{RAVE,a,x}$ can be viewed as a generalization of $Q_{RAVE,a}$(by taking $\alpha_{space} = \infty$), which itself generalizes Q_{RAVE} ($\alpha_{action} = \infty$).

Continuous RAVE can encapsulate prior knowledge on the action and space states, through using some informed dissimilarity function on the state and/or action spaces.

3.4.3 Experimental Validation

This section reports on the empirical validation of the Continuous RAVE heuristics, considering an artificial benchmark and a real-world problem. First, we describe the goals of experiments, and the experimental setting.

Goals of experiment and experimental setting

The primary goal of experiments is to assess the efficiency of the action and (state, action) cRAVE heuristics, comparatively to the MCTS/UCT baseline. Both heuristics are plugged in the same MCTS/UCT algorithm with double progressive widening and default parameters [39]. After a few preliminary experiments, the value of

the problem-dependent parameters α_{action} and α_{state} are set to d_{action} and $10^{-3}d_{state}$ where d_{action} and d_{state} respectively correspond to the dimension of the action and state spaces. The chosen distance in both action and state spaces is the Euclidean distance.

The equivalence parameter k is set to 50.

In both problems the policy value is compared to the baseline approach for the same computational budget (number of tree-walks used to select an action). Each value, averaged over independent runs, is reported together with the standard deviation.

The second goal of experiments is to study the sensitivity of the cRAVE heuristics with respect to the time horizon and size of the state space.

The TreasureHunt benchmark

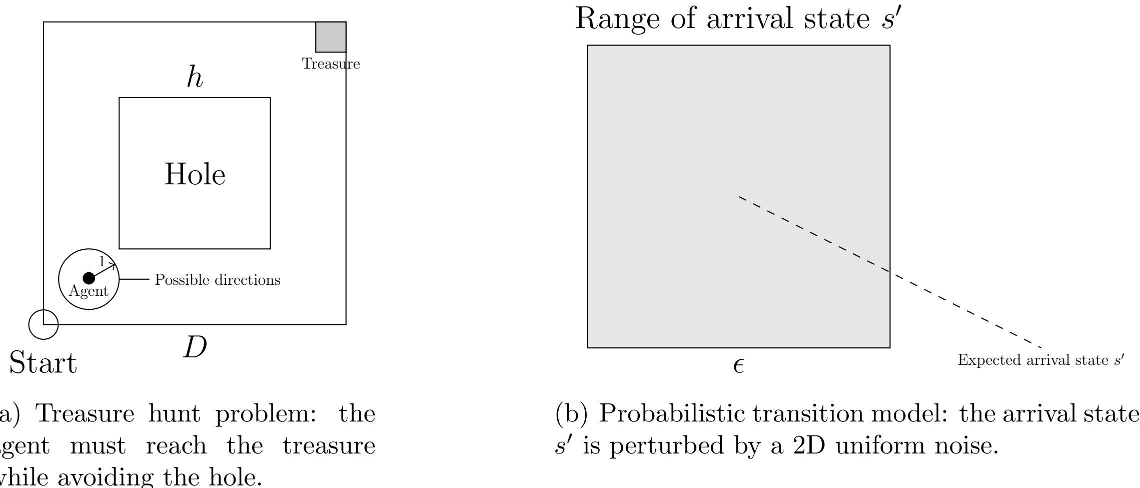
The artificial treasure hunt problem involves a squared arena of size D (Fig. 3-8(a), left). The state space is $\mathcal{X} = [0, D]^2$. The goal of the agent, initially located in the lower left corner, is to reach the treasure in the upper right corner. The agent speed is fixed; its direction a varies in $\mathcal{A} = [0, 2\pi]$. In each time step, the agent gets an instant reward of -1; reaching the treasure location gets an instant reward of 1,000. Two options are considered: with deterministic and probabilistic transition probabilities; with and without hole (the square hole with size h is located in the center of the arena). Transition probabilities $P^a_{xx'}$ are defined as follows: upon selecting action (direction) a in state $x \in \mathbb{R}^2$, the agent arrives in state $x' = x + (cosa, sina) + (U[-\epsilon/2, \epsilon/2], U[-\epsilon/2, \epsilon/2])$, where $U[a, b]$ denotes a random variable uniformly drawn in $[a, b]$ ($\epsilon = 0$ in the deterministic case; Fig. 3-8(a), right). Being in the hole yields an instant reward of -500.

A tree-walk stops when the agent reaches the treasure, or falls in the hole, or after traveling a distance $10D$. In the deterministic setting, the optimal reward thus is 1,000 minus the shortest path between the starting location and the treasure (conditionally to avoiding the hole). Note that the optimal strategy in the probabilistic transition setting is not straightforward.

The motivations for the treasure hunt problem is to study the scalability of the cRAVE heuristics with respect to the size of the arena. It is worth mentioning that quite a few planning problems (path planning) can be formulated as treasure hunt problems in high dimensional spaces involving many holes (see e.g. [117]).

Figure 3-9(a) (top) displays the comparative results obtained by cRAVE$_{action,state}$, cRAVE$_{action}$ and UCT in the deterministic transition setting with no hole. In this most simple setting, there is no significant difference although cRAVE$_{action,state}$ significantly improves on UCT for small time budgets. Interestingly, cRAVE$_{action,state}$ does not much improve on cRAVE$_{action}$. This is explained as the optimal trajectory is the straight line from the initial state to the treasure location: the optimal action does not depend on the current state in this simple problem. The advantage of cRAVE$_{action,state}$ will become significant in more complex settings when the optimal decision depends on the current state.

Figure 3-9(b) (medium and bottom) reports on the results in the probabilistic



(a) Treasure hunt problem: the agent must reach the treasure while avoiding the hole.

(b) Probabilistic transition model: the arrival state s' is perturbed by a 2D uniform noise.

Figure 3-8: The treasure hunt benchmark problem involves two options: the presence of a hole in the middle of the arena (left) and a probabilistic transition setting (right).

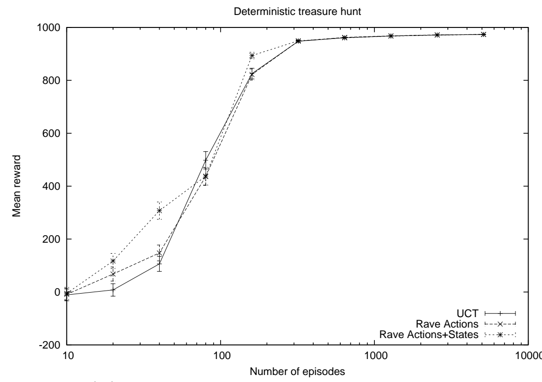
setting (respectively $\epsilon = .5$ and 1), where the optimal action $\pi(x)$ now depends on x. In the probabilistic cases, both cRAVE$_{action,state}$ and cRAVE$_{action}$ clearly improve on UCT. Unexpectedly, cRAVE$_{action}$ outperforms cRAVE$_{action,state}$, all the more so as the noise is moderate. The proposed interpretation for this finding goes as follows: on the one hand, the estimate variance is lower when the state is not taken into account; on the other hand, the optimal decision only slightly depends on state s; overall, cRAVE$_{action}$ thus enforces a faster convergence of the estimate while its bias remains moderate. This interpretation is confirmed as the gap between cRAVE$_{action,state}$ and cRAVE$_{action}$ decreases with the noise amplitude ϵ.

The results obtained for the treasure hunt with a hole are reported in Figs. 3-10(a), 3-10(b) and 3-10(c). Clearly, the optimal move here depends on the current state, even in the deterministic transition setting. As expected, cRAVE$_{action,state}$ significantly improves on cRAVE$_{action}$ in all deterministic and probabilistic transition settings with the hole, although the gap decreases with the noise amplitude increasing. Further, both cRAVE$_{action}$ and cRAVE$_{action,state}$ improve on the baseline UCT.

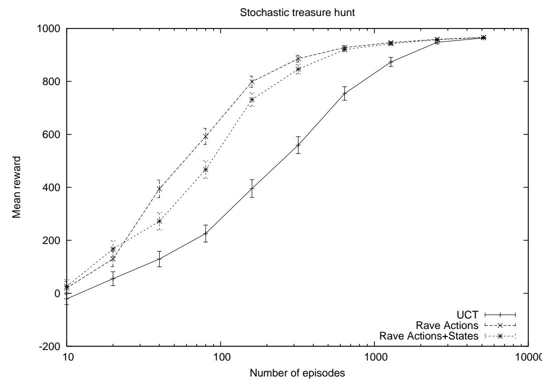
Energy Management Problem

This real-world problem describes a power plant involving S stocks of energy (e.g. hydro-electric stocks); the time horizon is T. In each time step, the possible action is to produce a (continuous) quantity of electricity using any of the S stocks. Overall, the energy demand is supplied with i) the energy produced from the hydro-electric stocks; ii) if needed, the energy produced from the thermal power stations. In the latter case, an additional super-linear cost is incurred. Historically this real-world problem is the applicative motivation of [87]'s and [16]'s seminal works on decomposition by dynamic programming. Refer to Section 1.2.3 for a more detailed description of the problem.

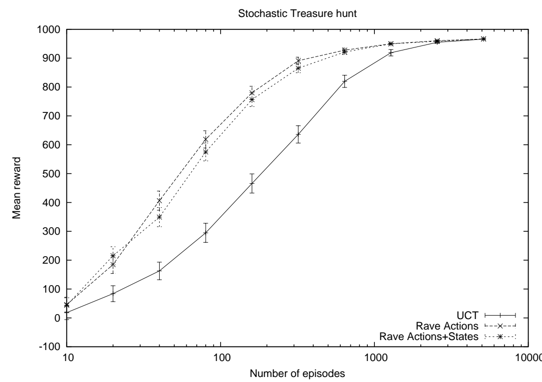
MCTS was investigated to find an optimal energy management policy within this



(a) Deterministic case, no trap.

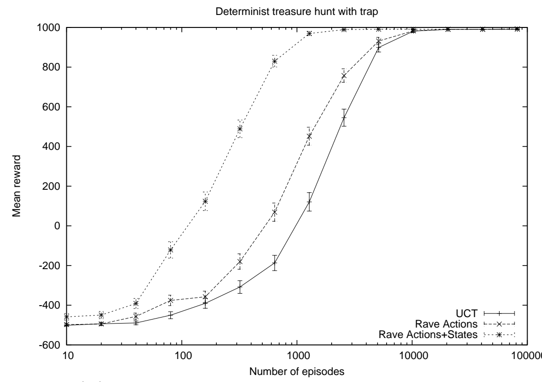


(b) Stochastic case with $\epsilon = 0.5$, no trap.

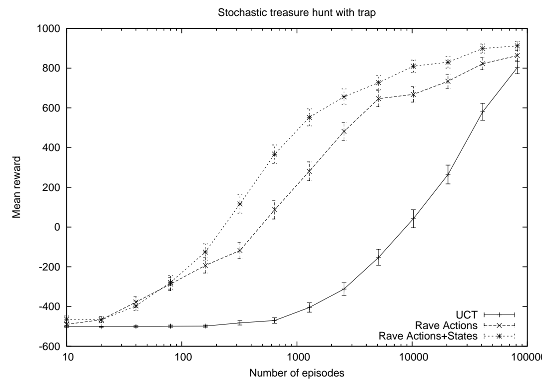


(c) Stochastic case with $\epsilon = 1$, no trap.

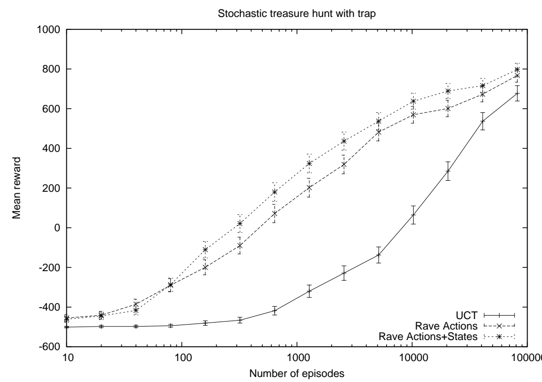
Figure 3-9: Treasure hunt with 15×15 arena, without hole (top: deterministic transitions; middle and bottom: probabilistic transitions with respectively $\epsilon = .5$ and 1). The speed up is only significant in the case of $\epsilon = 1$, where it makes the convergence about 3 times faster.



(a) Deterministic case, with trap.



(b) Stochastic with $\epsilon = 0.5$, with trap.



(c) Stochastic with $\epsilon = 1.$, with trap.

Figure 3-10: Treasure hunt with 5×5 arena, with hole (top: deterministic transitions; middle and bottom: probabilistic transitions with respectively $\epsilon = .5$ and 1). In this case, the speed up is quite significant, with MCTS+RAVE being about 10 times faster than vanilla MCTS.

setting, motivated by the fact that the underlying model of the power plants is non-linear and non-deterministic.

The experimental setting considers $S = 6$ stocks and $T = 12$ time horizon. The problem-dependent constants α_{action} and α_{state} are set to 10 (by consistency with the former treasure hunt problem, considering the range and dimension of action and state spaces, and to enforce the discrimination between different states and actions).

Figure 3-11 comparatively displays the results obtained by UCT, RAVE cRAVE$_{action}$ and cRAVE$_{action,state}$ on this problem. Interestingly, UCT is dominated by all other variants, including the RAVE variant devised to deal with discrete action spaces. Furthermore, cRAVE$_{action,state}$ significantly outperforms cRAVE$_{action}$; this finding was expected as two pairs (x, a) and (x', a') can only be considered similar if similar actions a and a' are applied on similar stock positions x and x'. For instance, the decision of using a minimal amount of water in order to use it later on, makes sense *if and only if* the stock positions are low, which is described through the current state. Overall, the merits of the cRAVE heuristics are fully empirically demonstrated on this simplified energy management problem.

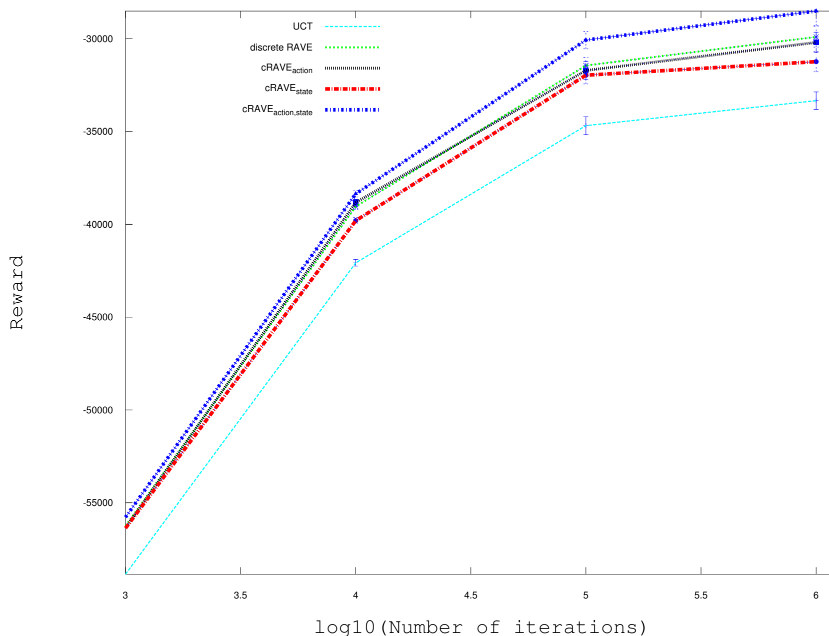


Figure 3-11: Comparative performances of UCT,discrete RAVE, cRAVE$_{state}$, cRAVE$_{action}$ and cRAVE$_{action,state}$ on the energy management problem, versus the computational budget (number of simulations). The upper the better.

Conclusion

The contribution proposed in this section concerns the extension of the Rapid Action Value Estimate heuristics, originally proposed to prevent misleading exploration in large action spaces. RAVE has been extended to continuous action spaces

(cRAVE$_{action}$, Eq. 3.10) using a Gaussian convolution; this approach was itself extended to the case of a continuous action and state spaces (cRAVE$_{action,state}$, Eq. 3.11). While these extensions can be easily plugged on the top of an UCT/RAVE algorithm, they only involve two additional hyper-parameters. The experimental validation of the approach on an artificial and a real-world problems fully demonstrates its potentialities, and its robustness w.r.t. some changes to the hyper-parameters.

A primary perspective for further work is to apply cRAVE in discrete domains where some distance/dissimilarity function can be defined using expert priors, e.g. classical game test beds like Go [80], Hex [1] or Havannah [114].

A longer-term perspective concerns the coupling of cRAVE$_{action,state}$ with the progressive widening (PW) heuristics. As already mentioned, PW introduces a new action in each state node from time to time, when the number of times this state has been visited reaches a given threshold. An interesting possibility would be to use $Q_{RAVE,a}$ and $Q_{RAVE,a,x}$ as value expectation, and select the *continuous* action a^* maximizing e.g. $Q_{RAVE,a,x}(x,a)$ over the whole action space $\mathcal{A}$.

Chapter 4

Theoretically consistent MCTS

In this section we present one version of MCTS, for which we have proved the consistency, under some assumptions. This section is almost entirely published at ECML 2013 [12].

4.1 Specification of the Markov Decision Tree setting

We use the classical terminology of Markov Decision Processes. In this framework, a *player* has to make sequential decisions until the process stops: he is then given a *reward*. As usual, the goal of the player is to maximize the expected reward. This section considers the general case where the process, also called transition, is a fully observable MDP, with finite horizon, and no cycles. *In this setting, the only things available to the agent are a simulator, or transition function, and an action sampler.*

As per usual in this setting, there is a state space and an action space. To build a tree in the stochastic setting, we choose to build it with two distinct and alternated types of nodes:

- decision nodes, where a decision needs to be made, are generally noted z. The intuition is that they correspond to a certain state where the agent might be.

- random nodes, where the transition can be called, are noted $w = (z, a)$. They correspond to the case where the agent was in state z and decided to take action a (sometimes called post-decision state).

The tree will have a unique root decision node r, the initial state where the agent starts. We define the depth of a node as half the distance from this node to the root in the tree. Hence decision nodes have integer depth while random nodes have semi-integer depth, e.g. to access a node of depth 2 we have the sequence of nodes root=decision(depth 0) - random (0.5) - decision (1) - random (1.5) - decision (2). Leaves are assumed to all have the same integer depth, denoted $d_{\max}$, and bear some deterministic *reward* $r(z)$.

It is well known [16] that for each node z, there exists a value $V^*(z)$, termed optimal Bellman value, frequently used as a criterion to select the best action in sequential decision making problems. We will use this value as a measure of optimality for actions. Given our distinction between decision nodes and random nodes, we use a natural notation for optimal Bellman values for both categories of nodes.

Let $w = (z, a)$ be a random node, and $P(z'|z, a)$ be the probability of being in node z' after taking action a in node z. Then, its optimal value is:

$$V^*(z, a) = \int_{z'} dP(z'|z, a) V^*(z') \tag{4.1}$$

Let z be a decision node. Then, its optimal value is defined as follows:

$$V^*(z) = \begin{cases} sup_a V^*(z, a) & \text{if } z \text{ is not a leaf,} \\ r(z) & \text{if } z \text{ is a leaf} \end{cases} \tag{4.2}$$

In particular, we formally define optimality of actions as follows:

Definition 3. *Let z be a non-leaf decision node, $w = (z, a)$ be a child of z, and $\epsilon > 0$. We say that the action a, i.e. the selection of node w, is optimal with precision ϵ if and only if $V(w) \geq V^*(z) - \epsilon$.*

There may be no optimal action since the number of children may be infinite.

Regularity hypothesis for decision nodes

This is the assumption that for any $\Delta > 0$, there is a non zero probability to sample an action that is optimal with precision Δ. More precisely, there is a $\theta > 0$ and a $p > 1$ (which remain the same during the whole simulation) such that for all $\Delta > 0$,

$$V(w = (z, a)) \geq V^*(z) - \Delta \text{ with probability at least } \min(1, \theta \Delta^p). \tag{4.3}$$

4.2 Specification of the Polynomial Upper Confidence Tree algorithm

We refer to [74] for the detailed specification of Upper Confidence Tree; we here define our variant PUCT (Polynomial Upper Confidence Trees).

In PUCT, we sequentially repeat episodes of the MDP and use information from previous episodes in order to explore and find optimal actions in the subsequent episodes. We denote by $n(z)$, for any decision node z, the total number of times that node z has been visited after the nth episode. Hence a node z has been encountered at episode n if $n(z) \geq 1$, and we always have $n = n(r)$. The notation is identical for random nodes.

We denote by $\hat{V}(z)$ the empirical average of a decision node z and $\hat{V}(z, a)$ the empirical average of a random node $w = (z, a)$. Note that if PUCT works properly, $\hat{V}(z)$ should converge to $V^*(z)$ when $n(z)$ goes to infinity.

How we select and construct children of a given node depends on two sequences of coefficients: α_d, the *progressive widening coefficient*, defined for all integer and semi-integer depths d, and e^d, the *exploration coefficient*, defined only for integer depths (i.e. decision nodes). These coefficients are defined according to Table 4.1. We sometimes indicate, as on Table 4.1, by a small "R" or "D" if a coefficient corresponds to a random or decision node, but otherwise it should be clear from the context.

PUCT algorithm

Input: a root node r, a transition function, an action sampler, a time budget, a depth d_{max}, parameters α and e for each layer

Output: an action a

while time budget not exhausted **do**
  while current node is not final **do**
    if current node is a decision node z **then**
      if $\lfloor n(z)^\alpha \rfloor > \lfloor (n(z)-1)^\alpha \rfloor$ **then**
        we call the action sampler and add a child $w = (z, a)$ to z
      else
        we choose as an action among the already visited children (z, a) of z, the one that maximizes its score, defined by:

$$\hat{V}(z,a) + \sqrt{\frac{n(z)^{e(d)}}{n(z,a)}}. \tag{4.4}$$

      end if
    else
      if $\lfloor n(w)^\alpha \rfloor = \lfloor (n(w)-1)^\alpha \rfloor$ **then**
        we select the child of z that was least visited during the simulation
      else
        we construct a new child (i.e. we call the transition function with argument w)
      end if
    end if
  end while
  we reached a final node z with reward $r(z)$; we back propagate all the information in the constructed nodes, and we go back to the root node r.
end while
Return the most simulated child of r.

With this algorithm, we see that if a decision node z at depth d has been visited n times, then we have visited during the simulation exactly $\lfloor n^{\alpha_d^D} \rfloor$ of its children, a number which depends on the *progressive widening constant* α_d^D. This is the so-called progressive widening trick [45].

For a random node z, we actually have the same property, depending on the *double progressive widening constant* α_d^R: this is the so-called double progressive widening trick ([39]; see also [59]).

Decision Node (d integer)	Random Node (d semi-integer)
$\alpha_d^D := \frac{1}{10(d_{\max}-d)-3}$ for $d \leq d_{\max}-1$ $e^d := \frac{1}{2p}\left(1-\frac{3}{10(d_{\max}-d)}\right)$ for $d \leq d_{\max}-1$	$\alpha_d^R := \begin{cases} \frac{3}{10(d_{\max}-d)-3} & \text{for } d \leq d_{\max}-\frac{3}{2} \\ 1 & \text{for } d = d_{\max}-\frac{1}{2} \end{cases}$
$\gamma_d^D := \frac{1}{10(d_{\max}-d)}$ for $d \leq d_{\max}-1$	$\gamma_d^R := \frac{1}{10(d_{\max}-d)-2}$ for $d \leq d_{\max}-\frac{1}{2}$

Table 4.1: **Definition of coefficients and convergence rates**

4.3 Main result

Definition 4 (Exponentially sure in n)**.** *We say that some property (P) depending on an integer n is* exponentially sure in n *(denoted e.s.) if there exists positive constants C, h, η such that the probability that (P) holds is at least*

$$1 - C\exp(-hn^{\eta}).$$

Theorem 1. *Define all exploration coefficients e_d and all progressive widening coefficients α_d as in Table 4.1. There is a constant $C > 0$, only depending on $d_{\max}$, such that after n episodes of PUCT, for every node z at depth d we have*

$$|\hat{V}(z) - V^*(z)| \leq \frac{C}{n(z)^{\gamma_d}} \ \text{e.s. in } n(z) \tag{4.5}$$

Additionally, for every node $w = (z, a)$ at depth $d + \frac{1}{2}$ we have

$$|\hat{V}(w) - V^*(w)| \leq \frac{C}{n(w)^{\gamma_{d+\frac{1}{2}}}} \ \text{e.s. in } n(w) \tag{4.6}$$

Corollary 1. *After n episodes, let $w_n(r)$ be the most simulated child node of r. Then,*

$$w_n(r) \text{ is optimal with precision } O\left(n^{-\frac{1}{10d_{\max}}}\right) \ \text{e.s. in } n \tag{4.7}$$

The proof is based on an induction on the following property and is detailed in the following three sections. Let us define this property.

Definition 5 (Induction property $Cons(\gamma_d, d)$)**.** *There is a $C_d > 0$ such that for all nodes at integer depth d,*

$$|\hat{V}(z) - V^*(z)| \leq C_d n(z)^{-\gamma_d} \ \text{e.s. in } n(z)$$

and for all nodes w at semi integer depths $d + \frac{1}{2}$,

$$|\hat{V}(w) - V^*(w)| \leq C_{d+\frac{1}{2}} n(w)^{-\gamma_{d+\frac{1}{2}}} \ \text{e.s. in } n(w)$$

In Section 4.4, we show that if $Cons(\gamma_d, d)$ holds for $d \geq 1$, i.e. for decision nodes in one given layer, then $Cons(\gamma_{d-\frac{1}{2}}, d - \frac{1}{2})$ holds, i.e. holds for the random nodes in

the above layer. In Section 4.5, we show that if $Cons(\gamma_{d+\frac{1}{2}}, d+\frac{1}{2})$ holds for $d \geq 0$, i.e. for random nodes in one given layer, then $Cons(\gamma_d, d)$ holds, i.e. holds for the decision nodes in the above layer. Finally, we establish in Section 4.6 that $Cons(\gamma, d)$ holds for maximal depth d_{max}, which will settle the proof of Theorem 4.5.

4.4 From Decision Nodes to Random Nodes

In this section we consider a random node w with semi-integer depth $d - \frac{1}{2} \geq 0$. We suppose that there exist a $\gamma_d^D > 0$ such that $Cons(\gamma_d^D, d)$ holds for any child node z of w. Recall that all nodes at this depth have $\lfloor n^{\alpha^R_{d-\frac{1}{2}}} \rfloor$ constructed children when they have been visited n times. We will show that we can define $\alpha^R_{d-\frac{1}{2}}$ so that $Cons(\gamma^R_{d-\frac{1}{2}}, d-\frac{1}{2})$ holds. For convenience, if w is a random node, we will refer to the ith child z_i of w by its index i directly. Then, the number of visits in z_i after the nth iteration of PUCT will be simply called $n(i)$ instead of $n(z_i)$. Similarly, the empirical value of this node will be noted $\hat{V}(i)$ instead of $\hat{V}(z_i)$.

4.4.1 Children of Random Nodes are selected almost the same number of times

With our politics for dealing with random nodes, described in section 4.2, the kth child of a random node w is constructed at episode $\lceil k^{\frac{1}{\alpha}} \rceil$. We now show that all constructed children of w but the last one are visited almost the same number of times.

Lemma 1. *Let w be a random node with progressive widening coefficient $\alpha \in]0;1[$. Then after the nth visit of w in the simulation, all children z_i, z_j of w with $1 \leq i, j < \lfloor n^\alpha \rfloor$ satisfy*

$$|n(i) - n(j)| \leq 1. \tag{4.8}$$

In fact, in the next section, we will only use the following consequence of Lemma 1.

Corollary 2. *When a random node z is visited for the nth time, all children of z have been selected at most $\frac{n}{\lfloor n^\alpha \rfloor - 1}$ times, and all children of z but the last one have been selected at least $\frac{n}{\lfloor n^\alpha \rfloor} - 1$ times.*

Proof. Let us simply call k the kth child of w for all $k \geq 1$, and denote n_k the number of visits (hereafter simply called timesteps) in w when child k was introduced. By definition, $n_k = \lceil k^{\frac{1}{\alpha}} \rceil$. Now remark the statement of Lemma 1 is equivalent to

$$\text{(4.8) is satisfied } \textit{for all children} \text{ of } w \text{ at every timestep of the form } n_k - 1 \text{ for } k \geq 2\ . \tag{4.9}$$

Indeed, considering the statement of Lemma 1 at time n_k, where child k is automatically selected, gives (4.9). On the other hand, consider what happens if (4.9) is true: at time n_k a new child k of w appears, and will be repeatedly selected until

it reaches the minimal number of selection among other children, or until n_{k+1} is reached. So if $n_{k+1} - n_k$ is large enough, (4.8) will hold for all children but k in a first period, and then for all children until n_{k+1}.

We proceed by induction: for $k = 1$, statement (4.9) is trivial, and then we will show

$$n_{k+1} - n_k \geq \lfloor \frac{n_k - 1}{k - 1} - 1 \rfloor. \tag{4.10}$$

Indeed, if (4.9) is true, then at time $n_k - 1$ the average number of selections among the $k - 1$ first children is $\frac{n_k - 1}{k-1}$, so that the number of selections that the new child must reach is the quantity on the right.

Would the delay between to new children $n_{k+1} - n_k$ be non-decreasing, the result would follow very easily. Unfortunately here, this is not the case.

Let $p = \lfloor \frac{\lceil k^{\frac{1}{\alpha}} \rceil - 1}{k-1} \rfloor$. This implies $\lceil k^{\frac{1}{\alpha}} \rceil \geq p(k-1) + 1$, so

$$k^{\frac{1}{\alpha}} > p(k-1). \tag{4.11}$$

Also, by definition, the left hand side of (4.10) is equal to

$$\begin{aligned}
\lceil (k+1)^{\frac{1}{\alpha}} \rceil - \lceil k^{\frac{1}{\alpha}} \rceil &\geq (k+1)^{\frac{1}{\alpha}} - k^{\frac{1}{\alpha}} - 1 \\
&\geq \frac{1}{\alpha} k^{\frac{1}{\alpha} - 1} - 1, \text{ by using the fact that } 0 < \alpha < 1 \\
&\geq \frac{p(k-1)}{\alpha k} - 1,
\end{aligned}$$

where we use (4.11) for the last inequality.

So (4.10) is settled if

$$\frac{p(k-1)}{\alpha k} - 1 \geq p - 1,$$

i.e. $\alpha \leq \frac{k-1}{k}$. So suppose now that we are in the case $\alpha > \frac{k-1}{k}$: then by (4.11) we deduce

$$p < \frac{k^{\frac{k}{k-1}}}{k-1}. \tag{4.12}$$

It is easily checked that the quantity on the right-hand side of (4.12) is less than 3 and so yields $p \leq 2$. Fortunately in this case Eq. 4.10 is trivial. □

4.4.2 Consistency of Random Nodes

Lemma 2 (Random nodes are consistent)**.** . *If there is a* $1 \geq \gamma_d > 0$ *such that for any child* z *of the random node* w *we have* $Cons(\gamma_d, d)$*, then we have* $Cons(\gamma_{d-\frac{1}{2}}, d - \frac{1}{2})$*, with* $\gamma_{d-\frac{1}{2}} = \frac{\gamma_d}{1+3\gamma_d}$ *if we define the progressive widening coefficient* $\alpha^R_{d-\frac{1}{2}}$ *by* $\alpha^R_{d-\frac{1}{2}} = \frac{3\gamma_d}{1+3\gamma_d}$.

Proof. From now on, w is fixed in order to simplify notation; therefore, we simply denote $\alpha^R_{d-\frac{1}{2}}$ by α, and $n(w)$ by n.

Fix n such that $n^\alpha \geq 3$. Define $i_0 = \lfloor n^\alpha \rfloor$ as the last constructed child of node w, and $r = \lfloor n^\alpha \rfloor - 1 = i_0 - 1$. To prove the result, we need to prove an upper bound on the following quantity, that holds exponentially surely in n:

$$|\hat{V}(w) - V^*(w)| \;=\; |\left(\sum_{1\leq i<i_0} \frac{n(i)}{n}\hat{V}(i) + \frac{n(i_0)}{n}\hat{V}(i_0)\right) - V^*(w)|$$

Decompose this as

$$\begin{aligned} |\hat{V}(w) - V^*(w)| \;&\leq\; |\sum_{1\leq i<i_0} \left(\frac{n(i)}{n} - \frac{1}{r}\right)\hat{V}(i)| && (4.13)\\ &+\; |\sum_{1\leq i<i_0} \frac{1}{r}\left(\hat{V}(i) - V^*(i)\right)| && (4.14)\\ &+\; |\sum_{1\leq i<i_0} \frac{1}{r}\left(V^*(i) - V^*(w)\right)| && (4.15)\\ &+\; |\frac{n(i_0)}{n}\hat{V}(i_0)| && (4.16)\end{aligned}$$

First consider (4.13). By Lemma 1, there is a integer p such that all children $i = 1, \cdots, i_0 - 1$ have been selected p or $p+1$ times, with $p = O\left(n^{1-\alpha}\right)$. So, we have for all $i = 1, 2, \cdots, i_0 - 1$,

$$|\frac{n(i)}{n} - \frac{1}{\lfloor n^\alpha \rfloor - 1}| \;\leq\; |\frac{p}{n} - \frac{1}{\lfloor n^\alpha \rfloor - 1}| + \frac{1}{n}$$

The definition of p gives $(i_0 - 1)p \leq n \leq i_0(p+1)$, so that

$$|\frac{p}{n} - \frac{1}{i_0 - 1}| \leq \frac{i_0 + p}{(i_0 - 1)n} = O\left(\frac{1}{n} + \frac{1}{n^{2\alpha}}\right)$$

so that in the end for (4.13) we have

$$|\sum_{1\leq i<i_0} \left(\frac{n(i)}{n} - \frac{1}{r}\right)\hat{V}(i)| = O\left(n^\alpha\left(\frac{1}{n} + \frac{1}{n^{2\alpha}} + \frac{1}{n}\right)\right) = O\left(\frac{1}{n^{1-\alpha}} + \frac{1}{n^\alpha}\right)$$

Consider now (4.14). $Cons(\gamma_d, d)$ holds, so for each child $i = 1, 2, \cdots, \lfloor n^\alpha \rfloor - 1$ of w, Lemma 1 leads to:

$$|\hat{V}(i) - V(i)| \leq C_d p^{-\gamma_d} \leq C_d \frac{1}{\lfloor n^{1-\alpha} \rfloor^{\gamma_d}} \text{ e.s. in } n^{1-\alpha}$$

Finally for (4.14) it is exponentially sure in n that

$$| \sum_{0 \le i < i_0} \frac{1}{\lfloor n^\alpha \rfloor - 1} \left(\hat{V}(i) - V^*(i) \right) | = O\left(\frac{1}{n^{(1-\alpha)\gamma_d}} \right). \tag{4.17}$$

Now we turn to (4.15). Since w is a random node, the value $V^*(i)$ of each new child i of w constructed by the algorithm is given by a random law whose mean is $V^*(w)$. Thus we can apply Hoeffding's inequality to the sum in (4.15) and we obtain that for $t > 0$,

$$| \sum_{0 \le i < i_0} \frac{1}{\lfloor n^\alpha \rfloor - 1} \left(V^*(i) - V^*(w) \right) | \le t \tag{4.18}$$

with probability at least $1 - 2\exp\left(-2t^2 \left(\lfloor n^\alpha \rfloor - 1\right)\right) = 1 - 2\exp(-Cn^{\frac{\gamma_d}{1+3\gamma_d}})$

with $t := n^{-\frac{\gamma'}{1+3\gamma_d}}, \alpha = \frac{3\gamma_d}{1+3\gamma_d}$, and $C > 0$. This proves that (4.18) is e.s. in n.

Finally consider (4.16): since the last child of w has been selected at most p times, we have

$$\left| \frac{n(i_0)}{n} \hat{V}(i_0) \right| = \frac{1}{n} \times O\left(\frac{n}{n^\alpha} \right) = O\left(\frac{1}{n^\alpha} \right).$$

All in all, we have have shown that it is exponentially sure in $n = n(w)$ that

$$|\hat{V}(w) - V^*(w)| = O\left(\underbrace{\frac{1}{n^{1-\alpha}} + \frac{1}{n^\alpha}}_{(4.13)} + \underbrace{\frac{1}{n^{(1-\alpha)\gamma_d}}}_{(4.14)} + \underbrace{\frac{1}{n^{\frac{\gamma_d}{1+3\gamma_d}}}}_{(4.15)} + \underbrace{\frac{1}{n^\alpha}}_{(4.16)} \right). \tag{4.19}$$

With $\alpha = \frac{3\gamma_d}{1+3\gamma_d}$ and $\gamma_d \le 1$, it is straightforward to check that the smallest exponent is $\frac{\gamma_d}{1+3\gamma_d}$, so that $Cons(\gamma_{d-\frac{1}{2}}, d - \frac{1}{2})$ is true with $\gamma_{d-\frac{1}{2}} = \frac{\gamma_d}{1+3\gamma_d}$□ □

4.5 From Random Nodes to Decision nodes

Let z be a non leaf decision node at depth d. In this section, we will show that if the induction property holds for all random nodes at depth $d + \frac{1}{2}$, it will hold for z.

4.5.1 Children of decision nodes are selected infinitely often

Lemma 3. *Let f be a non-decreasing map from $\mathbb{N}$ to $\mathbb{N}$. Consider a stochastic bandit setting with a countable set of children, progressive widening coefficient α and exploration function f, i.e. the score at time n of a child i is computed by*

$$sc_n(i) = \hat{V}_n(i) + \sqrt{\frac{f(n)}{n(i)}}.$$

Then if i denotes the ith constructed child, for all $n \geq i^{\frac{1}{\alpha(1-\alpha)}}$ we have

$$n(i) \geq \frac{1}{4}\min(f(n^{1-\alpha}), n^{1-\alpha}).$$

In particular, all constructed children are selected infinitely often provided that $\lim_{+\infty} f = +\infty$.

Proof. Fix n and consider the child i_0 maximizing $n(i_0)$, i.e. the most selected child at time n. Let n' be the last time i_0 has been selected. Since there are at most n^α children at time n we have

$$n'(i_0) = n(i_0) \geq \frac{n}{n^\alpha} = n^{1-\alpha} \tag{4.20}$$

where (i) $n'(i_0)$ is the number of times i_0 has been drawn before time n'; (ii) $n(i_0)$ is the number of times i_0 has been drawn before time n. Thus we also have

$$n' \geq n'(i_0) \geq n^{1-\alpha}. \tag{4.21}$$

Consider now any child i already constructed at time n'. Since i_0 was selected at time n' we must have

$$\sqrt{\frac{f(n')}{n'(i)}} \leq sc_{n'}(i) \leq sc_{n'}(i_0) \leq 1 + \sqrt{\frac{f(n')}{n'(i_0)}}. \tag{4.22}$$

Rewriting 4.22 and using 4.20 leads to

$$\frac{1}{\sqrt{n'(i)}} \leq \frac{1}{\sqrt{n^{1-\alpha}}} + \frac{1}{\sqrt{f(n')}} \leq \frac{2}{\sqrt{\min(f(n'), n^{1-\alpha})}} \tag{4.23}$$

so that for all children i at time n existing at time n' we have

$$n(i) \geq n'(i) \geq \frac{1}{4}\min\left(f(n^{1-\alpha}), n^{1-\alpha}\right)$$

as announced. Finally, note that a child i existed at time n' if $i \leq (n^{1-\alpha})^\alpha \leq n'^\alpha$, which leads to the prescribed condition. □ □

Corollary 3. *For the exploration function $f(n) = n^e$ with $0 < e < 1$ we obtain*

$$n(i) \geq \frac{1}{4}n^{e(1-\alpha)} \text{ if } i \leq n^{\alpha(1-\alpha)}.$$

4.5.2 Decision nodes are consistent

Lemma 4 (Decision nodes are consistent)**.** . *If there is a $\frac{1}{2} > \gamma_{d+\frac{1}{2}} > 0$ such that for any child w of the decision node z we have $Cons(\gamma_{d+\frac{1}{2}}, d+\frac{1}{2})$, then we have $Cons(\gamma_d, d)$ with $\gamma_d = \frac{\gamma_{d+\frac{1}{2}}}{1+7\gamma_{d+\frac{1}{2}}}$ if we define the progressive widening coefficient α_d^D by*

$\alpha_d^D = \frac{\gamma_{d+\frac{1}{2}}}{1+4\gamma_{d+\frac{1}{2}}}$.

Proof. Let z be a decision node at depth $d \geq 0$. For simplicity, we note $\alpha_d = \alpha$ and $e_d = e$. Suppose that there is a $\frac{1}{2} > \gamma_{d+\frac{1}{2}} > 0$ such that for all random nodes w at depth $d + \frac{1}{2}$, $Cons(\gamma_{d+\frac{1}{2}}, d + \frac{1}{2})$ is true. To show $Cons(\gamma_d, d)$, we will proceed in two steps: first we establish an upper bound on $\hat{V}(z) - V^*(z)$, and then a lower bound.

Upper bound. First we obtain an upper bound on $\hat{V}(z) - V^*(z)$. Let $\epsilon < 1 - \alpha$ to be fixed later. We partition the children of z in two classes:

- class I : children i such that $n(i) \leq n(z)^{1-\alpha-\epsilon}$;
- class II : other children;

$$\begin{aligned}
\hat{V}(z) - V^*(z) &= \sum_{i \text{ in class } I} \frac{n(i)}{n(z)}(\hat{V(i)} - V^*(z)) + \sum_{i \text{ in class } II} \frac{n(i)}{n(z)}(\hat{V(i)} - V^*(z)) \\
&\leq \sum_{i \text{ in class } I} \frac{n(i)}{n(z)} + \sum_{i \text{ in class } II} \frac{n(i)}{n(z)}(\hat{V}(i) - V^*(i)) \\
&\leq \frac{n^\alpha \times n^{1-\alpha-\epsilon}}{n} + C_{d+\frac{1}{2}}(n)^{-\gamma_{d+\frac{1}{2}}(1-\alpha-\epsilon)} \text{e.s. in } n^{-\gamma_{d+\frac{1}{2}}(1-\alpha-\epsilon)} \text{ by induction} \\
&\leq n^{-\epsilon} + C_{d+\frac{1}{2}} n^{-\gamma_{d+\frac{1}{2}}(1-\alpha-\epsilon)}.
\end{aligned}$$

We now choose $\epsilon = \frac{\gamma_{d+\frac{1}{2}}(1-\alpha)}{1+\gamma_{d+\frac{1}{2}}}$ and obtain

$$\hat{V}(z) - V(z) \leq (1 + C_{d+\frac{1}{2}}) n^{-\gamma_{d+\frac{1}{2}} \frac{1-\alpha}{1+\gamma_{d+\frac{1}{2}}}} \text{ e.s. in } n \tag{4.24}$$

Lower bound.

We assumed that there exists a constant θ such that when we pick a new child for z, it has a value satisfying $V(i) \geq V^*(z) - \Delta$ with probability at least $\min(1, \theta\Delta^p)$.

The induction hypothesis on the next level gives us a fixed coefficient $\gamma_{d+\frac{1}{2}} \in]0; 0.5[$ such that all children w of z verify e.s. in $n(w)$:

$$\left| V^*(w) - \hat{V}(w) \right| \leq C_{d+\frac{1}{2}} n(w)^{-\gamma_{d+\frac{1}{2}}}.$$

The parameters to be fixed on this level are

- the progressive widening coefficient $\alpha := \frac{\gamma_{d+\frac{1}{2}}}{1+4\gamma_{d+\frac{1}{2}}}$;
- the exploration coefficient $e := \frac{1}{1+4\gamma_{d+\frac{1}{2}}} - \frac{1}{\gamma_{d+\frac{1}{2}}}(1 - \frac{1}{2p})\alpha = \frac{1}{2p(1+4\gamma_{d+\frac{1}{2}})}$.

To these coefficients we add a parameter ξ which we define by

$$\xi := \frac{1}{1 + e\gamma_{d+\frac{1}{2}}(1-\alpha)} \tag{4.25}$$

$$\text{and let} \quad \Delta := \left(\frac{1}{4}n^{\xi e(1-\alpha)}\right)^{-\gamma_{d+\frac{1}{2}}}. \tag{4.26}$$

First step : exponentially surely in n there exists at time $\lceil n^{\xi(1-\alpha)}\rceil$ a child i_0 of z such that

$$V(i_0) \geq V(z) - \Delta \text{ and } i_0 \leq n^{\xi(1-\alpha)\alpha}. \tag{4.27}$$

At time step $\lceil n^{\xi(1-\alpha)}\rceil$, the number of children of z is a at least $\lfloor n^{\xi(1-\alpha)\alpha}\rfloor$. The (true hidden optimal) values of these children being given randomly and independently, the probability there is not a single child i_0 with $V(i_0) \geq V^*(z) - \Delta$ at time $\lceil n^{\xi(1-\alpha)}\rceil$ is at most

$$p_n := (1 - \theta\Delta^p)^{\lfloor n^{\xi(1-\alpha)\alpha}\rfloor}$$

$$\begin{aligned}
\log p_n &\sim_n n^{\xi(1-\alpha)\alpha}\log(1 - \theta\Delta^p) \\
&\sim_n -n^{\xi(1-\alpha)\alpha}\theta\left(\frac{1}{4}n^{\xi e(1-\alpha)}\right)^{-\gamma_{d+\frac{1}{2}}p} \\
&\sim_n -4^{\gamma_{d+\frac{1}{2}}p}\theta n^{\xi(1-\alpha)(\alpha - e\gamma_{d+\frac{1}{2}}p)} \\
&\sim_n -4^{\gamma_{d+\frac{1}{2}}p}\theta n^{\xi(1-\alpha)0.5\alpha}.
\end{aligned}$$

The exponent of n in this quantity being positive, we deduce that the existence of i_0 is exponentially sure in n.

Second step: e.s. in n, all children selected at a time n' between n^ξ and n have a high score.

Let n' be such that $n^\xi \leq n' \leq n$. Then $n'^{\alpha(1-\alpha)} \geq n^{\xi(1-\alpha)\alpha} \geq i_0$. And, by Corollary 3,

$$n'(i_0) \geq \frac{1}{4}n'^{e(1-\alpha)} \geq \frac{1}{4}n^{\xi e(1-\alpha)}.$$

Hence there exists a $C' > 0$ by the induction hypothesis such that we have, as long as $n^\xi \leq n' \leq n$,

$$\begin{aligned}
\hat{V}(i_0) &\geq V^*(i_0) - C'\left(\frac{1}{4}n^{\xi e(1-\alpha)}\right)^{-\gamma_{d+\frac{1}{2}}} \text{ e.s. in } n' \\
&\geq V^*(z) - (1 + C')\Delta \text{ e.s. in } n'.
\end{aligned}$$

Consider any child i_1 chosen by the algorithm at a time $n' \geq n^\xi$, i.e. the one which has the greatest score at time n'. All values being considered at time n', we have

$$\hat{V}(i_1) + \sqrt{\frac{n'^e}{n'(i_1)}} \geq \hat{V}(i_0) + \sqrt{\frac{n'^e}{n'(i_0)}},$$

$$\text{hence } \hat{V}(i_1) + \sqrt{\frac{n^e}{n'(i_1)}} \geq V^*(z) - (1 + C')\Delta \text{ e.s. in } n'. \tag{4.28}$$

To conclude this part, all we have to do is to show that some property exponentially sure in n' is also exponentially sure in n. This easily follows from the fact that $n' \geq n^{\xi}$ and that ξ, is bounded below by some constant. One can easily check from the definition of ξ that $\xi \geq \frac{2}{3}$, since $e \leq \frac{1}{2}$.

Third step : lower bound on $\hat{V}(z)$.

Consider a child i_1 selected after n^{ξ}. By the previous step, exponentially surely in n, this child must either satisfy

$$\sqrt{\frac{n^e}{n(i_1)}} \geq \Delta \tag{4.29}$$

$$\text{or } \hat{V}(i_1) \geq V(z) - (2 + C')\Delta. \tag{4.30}$$

Under this hypothesis we can split the children of z in three categories:

1. children i_1 visited only before time n^{ξ} ;
2. children i_1 visited after n^{ξ} satisfying (4.29) ;
3. children i_1 visited after n^{ξ} satisfying (4.30) .

Let us use this decomposition to lower bound the sum

$$\hat{V}(z) - V^*(z) = \sum_{i=1\cdots\lfloor n^{\alpha}\rfloor} \frac{n(i)}{n}(\hat{V}(i) - V^*(z)).$$

For the children in the first category, we have

$$\left| \sum_{i_1 \text{in cat.1}} \frac{n(i_1)}{n}(\hat{V}(i_1) - V^*(z)) \right| \leq \frac{\sum_{i_1 \text{in cat.1}} n(i_1)}{n} \leq \frac{n^{\xi}}{n}.$$

For children in the second category, since there are at most n^{α} of these children, we have

$$\left| \sum_{i_1 \text{in cat.2}} \frac{n(i_1)}{n}(\hat{V}(i_1) - V^*(z)) \right| \leq \frac{\sum_{i_1 \text{in cat.2}} n(i_1)}{n} \leq \frac{n^{\alpha}}{n}\frac{n^e}{\Delta^2} = \frac{n^{\alpha+e-1}}{\Delta^2}.$$

Finally, using (4.30) for the third category of children, we see that

$$\hat{V}(z) - V^*(z) \quad \geq \quad -(2 + C')\Delta(1 - n^{\xi-1}) - n^{\xi-1} - \frac{n^{\alpha+e-1}}{\Delta^2}.$$

Now we compare the three terms

$$\Delta, n^{\xi-1} \text{ and } \frac{n^{\alpha+e-1}}{\Delta^2}. \tag{4.31}$$

By (4.25) we have $\xi - 1 = -\xi e \gamma_{d+\frac{1}{2}}(1 - \alpha)$, thus by (4.26), $n^{\xi-1} = 4^{-\gamma_{d+\frac{1}{2}}}\Delta \leq \Delta$.

This implies that the term $n^{\xi-1}$ in the three terms (Eq. 4.31) is $O(\Delta)$. We now compare the two other terms; from the definition of Δ, we see that we must compare $n^{\alpha+e-1}$ and $\Delta^3 = 4^{3\gamma_{d+\frac{1}{2}}} n^{-3\xi e(1-\alpha)\gamma_{d+\frac{1}{2}}}$. Using the definitions of ξ, e and α, one can check that:

$$1 - e - \alpha \geq \frac{3\gamma_{d+\frac{1}{2}} + \frac{1}{2}}{1 + 4\gamma_{d+\frac{1}{2}}} \geq \frac{1}{2}$$

and, using $\xi \leq 1$, $(1-\alpha) \leq 1$, $e\gamma_{d+\frac{1}{2}} = \frac{\gamma_{d+\frac{1}{2}}}{2(1+4\gamma_{d+\frac{1}{2}})} \leq \frac{1}{8}$,

$$3\xi e(1-\alpha)\gamma_{d+\frac{1}{2}} \leq \frac{3}{8}$$

$$n^{e+\alpha-1} \leq n^{-3\xi e(1-\alpha)\gamma_{d+\frac{1}{2}}} = 4^{-3\gamma_{d+\frac{1}{2}}} \Delta^3 \leq \Delta^3$$

so that $\hat{V}(z) - V(z) \geq -(5 + C')\Delta$. Finally, one can check that $\xi e(1-a)\gamma_{d+\frac{1}{2}} \geq \frac{\gamma_{d+\frac{1}{2}}}{1+7\gamma_{d+\frac{1}{2}}}$ so that $\hat{V}(z) - V^*(z) \geq -(5+C')4^{\gamma_{d+\frac{1}{2}}} n^{\frac{\gamma_{d+\frac{1}{2}}}{1+7\gamma_{d+\frac{1}{2}}}}$ which can now be written $\hat{V}(z) - V^*(z) \geq -Cn^{-\gamma}$ with $C := (5+C')4^{\gamma_{d+\frac{1}{2}}}$ and $\gamma = \frac{\gamma_{d+\frac{1}{2}}}{1+7\gamma_{d+\frac{1}{2}}}$.□ □

4.6 Base step, initialization and conclusion of the proof

Let w be a random node of depth $d_{\max} - \frac{1}{2}$. Its children are leaf nodes, and all have a fixed reward in $[0;1]$. These children form a ensemble of independent and identically distributed variables, all following the random distribution associated with w, of mean $V^*(w)$. Hoeffding's inequality gives, for $t > 0$,

$$\mathbb{P}\left(|\frac{1}{n} \sum_{z_i child of w} V^*(z_i) - V^*(w)| \geq t\right) \leq 2\exp(-2t^2 n).$$

Setting the exploration coefficient $\alpha_{d_{\max}-\frac{1}{2}}$ to 1 (since there is no point in selecting again children with a constant reward) and t to $n^{-\frac{1}{3}}$, we obtain

$$\mathbb{P}\left(|V\hat{(w)} - V^*(w)| \geq n^{-\frac{1}{3}}\right) \leq 2\exp(-2n^{\frac{1}{3}})$$

so that $|V\hat{(w)} - V^*(w)| \leq n^{-\frac{1}{3}}$ is exponentially sure in n, i.e. $Cons(\frac{1}{3}, d_{\max} - \frac{1}{2})$ holds. Of course one can consider a coefficient different from $\frac{1}{3}$ for t, as long as it is less than $\frac{1}{2}$ – we just aim so as to simplify the definition of coefficients. This gives a singular value of $\alpha^R_{d_{\max}-\frac{1}{2}} = 1$ and an initialization of the convergence rate as $\gamma^R_{d_{\max}-\frac{1}{2}} = \frac{1}{3}$. It is now elementary to check this value of $\frac{1}{3}$ for γ at depth $d_{\max} - \frac{1}{2}$, together with recursive definitions of coefficients derived in Lemmas 2 and 4, yield the values given on Table 4.1. This concludes the proof of Theorem 4.5.

Budget (s)	0.001	0.004
HOLOP	-47.45 $\pm$.	.
DPS	-838.7 $\pm$ 78.0	-511.0 $\pm$ 100.0
PUCT+DPS	-13.84 $\pm$ 0.80	-11.11 $\pm$ 0.95

Budget (s)	0.04	0.16	0.64
DPS	-8.02 $\pm$ 0.98	-7.06 $\pm$ 0.024	-6.98 $\pm$ 0.03
PUCT+DPS	-7.23 $\pm$ 0.45	-6.69$\pm$ 0.03	-6.57 $\pm$ 0.02

Table 4.2: Left: Cart Pole results; episodes are 200 time steps long. Right: Unit Commitment results, with 2 stocks, 5 plants, and 6 time steps.

4.7 Experimental validation

In this section, we show some experimental results, by implementing PUCT on two tests problems. We used fixed parameters α and e, quickly tuned by hand. We added a custom default policy, as seen in [42], that is computed offline using Direct Policy Search (DPS), once per problem instance. We also gave heavier weights to the decisions with high average value when computing the empirical value of a state, as it showed increased performances in practice. There are many ways to finely tune PUCT that we did not explore. Our goal was simply to check that our PUCT has a satisfying behaviour, to verify our theoretical results. We acknowledge that depending on implementation subtleties, results can vary. Our source code is available upon request.

Cart pole. We used the well known benchmark of cart pole, and more precisely the version presented in [122]. As our code uses time budget, and not a limit in the number of iterations, we only approximated their limit of 200 roll outs (on our machine, 0.001 second per action. We took HOLOP as a baseline, that yields an average reward of -47.45[122]. Our results are shown in Table 4.2. Though cart pole is not as challenging as real world applications, these results are encouraging and supporting our theoretical results of consistency.

Unit commitment. We used a unit commitment problem, inspired by ongoing work with an industrial partner. The agent owns 2 water reservoirs and 5 power plants. Each reservoir is a free but limited source of energy. Each power plant has a fixed capacity, has a fixed cost to be turned on, as well as quadratic running costs that change over time. The time horizon was fixed to 6 time steps. At each time step, the agent decides how to produce energy in order to satisfy a varying demand, and the water reservoirs receive a random inflow. Failure to satisfy the demand incurs a prohibitive cost. This problem is challenging for many reasons, including: the action space is non convex, the objective function is non linear and discontinuous, there are binary and continuous variables, and finally, the action space can be subject to operational constraints that make a discretization by hand very tedious. The purpose of PUCT in this application is not to solve all of it, but rather to improve existing solvers. This is an especially promising method, with the many powerful heuristics available for this problem. The results are shown in Table 4.2. PUCT manages to reliably improve the actions suggested by DPS, and its performances increase with the time budget it is given.

4.8 Conclusion

[74] have shown the consistency of the UCT approach for finite Markov Decision Processes. We have shown the consistency of our modified version, with polynomial exploration and double progressive widening, for the more general case of infinite MDP. [39] have shown that the classical UCT is not consistent in this case and already proposed double progressive widening; we here give a proof of the consistency of this approach, when we use polynomial exploration; [39] was using logarithmic exploration.

Some extensions of our work are straightforward. We considered trees, but the extension to MDP with possibly two distinct paths leading to the same node is straightforward. Also, we assumed, only for simplifying notation, that the probability that a random node leads twice to the same decision node (when drawn independently with the probability distribution of the random node) is zero, but the extension is possible. On the other hand, we point out two deeper limitations of our work: (i) The reason why we switched to polynomial exploration was to make the proof easier. We do not know if similar results can be derived without switching. This would be an interesting direction for future research, as empirically the UCB based exploration seems to lead to a consistent MCTS. (ii) The general case of a possibly cyclic MDP with unbounded horizon is not covered by our result.

We have shown consistency in the sense that Bellman values are properly estimated. This does not explain which decision should be actually made when PUCT has been performed for generating episodes and estimating V values. Our result implies that choosing the action by empirical distribution of play (i.e. randomly draw a decision with probability equal to the frequency at which it was simulated during episodes; see discussion in [21]) is asymptotically consistent. Also, choosing the most simulated child is consistent (this is a classical method in UCT), as well as selecting the child with best $\hat{V}$ among child nodes of the root of class II; our results do not show the superiority of one or another of these recommendation methodologies.

Our experimental results on the classical Cart pole problem show that PUCT outperforms HOLOP; PUCT also outperformed a specialized DPS on a unit commitment problem. This last empirical result is especially interesting because unit commitment problems are, in practice, highly non Markovian. And, even though we worked in the framework of MDP to relate to its abundant literature, our algorithm does not actually need the random process to be Markovian, as the history is naturally embedded in the tree structure. Hence, PUCT could be a way to approach difficult and more general non Markovian continuous sequential decision making problems.

Chapter 5

Hybridization of MCTS

In this section we present some of the works relying on the hybridization of MCTS with other solvers. In Section 5.1, we show a method to mix MCTS with direct policy search, in order to improve MCTS's playouts. In Section 5.2, we explain how MCTS can be used to solve Partially Observable MDP (POMDP), and show some experimental results on the game of Mines, obtained by mixing it with domain specific heuristics. Finally, in Section 5.3, we present a meta-bandit framework, to solve problems with a high level optimization done by bandits, and microlevel problems solved by MCTS.

5.1 Custom default policies

This section's content can be found, for the most part, in [42].

5.1.1 Introduction

MCTS has been greatly improved by including Progressive Widening and Double Progressive Widening[45, 33], RAVE values[54], Blind Values[38], and hand-crafted Monte-Carlo moves[120, 80]. A crucial component is the Monte-Carlo move generator, also known as the playout generator.

In this section, we focus on the addition of specialized Monte-Carlo moves, i.e. we modify default policy, to help dealing with stochastic planning problems. Finding a default policy that is optimal for all instances of a stochastic problem can be extremely difficult and time consuming. The solution we propose here is to apply a Direct Policy Search to the available default policy. This way, even an initially poor default policy can be improved to fit different instances of one stochastic planning problem.

In Section 5.1.2 we describe existing algorithms for improving Monte-Carlo move generators (subsection 5.1.2), and Direct Policy Search for improving a Monte-Carlo move generator in Section (subsection 5.1.2). In Section 5.1.3, we experiment our algorithms on three forms of a unit commitment problem (Section 5.1.3), and on an investment problems (Section 5.1.3).

5.1.2 Algorithms

In this section we will present Direct Policy Search, and Monte-Carlo move generators improvements.

Heuristics and Monte-Carlo move generators

Whereas in the 2-player case, it is known that making a Monte-Carlo generator stronger (stronger in the sense: as a stand-alone policy), does not necessarily make the MCTS built on top of it stronger (see [120]), we conjecture that in the one-player case it is usually quite efficient.

The recent improvements in the world of Computer Go basically comes from improvements of the Monte-Carlo move generator, implemented so that the Monte-Carlo simulator does not contradict life&death known results; Zen, CrazyStone, Pachi, are examples of such strong programs, around 4 to 5 Dan for short time settings and 3 to 4 Dan for long time settings (professional human players tend to benefit from longer time settings more than machines). Other tools have been proposed as generic solutions for learning Monte-Carlo move generators:

- Simulation balancing [109] has been proposed for automatically learning the Monte-Carlo move generator in 2-player games.
- PoolRave[102], in which the Monte-Carlo move is replaced, with a fixed probability $p \in (0,1)$, by a move uniformly drawn among the c moves with best RAVE score in the last node of the simulation with at least k simulations.
- Contextual Monte-Carlo[101] in which the Monte-Carlo move-generator is improved by online learning a tile-based value function.

These tools are efficient, but the main successes in Monte-Carlo Tree Search nonetheless come from handcrafted Monte-Carlo move generators. Here, we used a specialized Monte-Carlo move generator, chosen specifically on our main target problem, as well as a less specialized function. We improve them by Direct Policy Search (Section 5.1.2).

Direct Policy Search for generating Monte-Carlo Move Generators

Direct Policy Search (DPS) is an approach very different from Upper Confidence Tree; it is based on selecting a policy among a parametric family of policies by optimization of its parameters. The pseudo-code is as follows:

Procedure *Simulate*(s, MDP, p)**:**
Inputs: a state s, a Markov Decision Process MDP, and a policy p.
Output: a reward.
Method: simulate MDP from state s with policy p until a terminal state and return the obtained reward.
Procedure Direct Policy Search:
Inputs: (i) a parametric policy $\theta \mapsto \pi(\theta)$, where $\pi(\theta)$ is a mapping from states to actions. (ii) a Markov Decision Process MDP. (iii) an initial state s.
Output: a parameter $\hat{\theta}$, leading to a policy $\pi(\hat{\theta})$.
Auxiliary method: a noisy optimization algorithm.
Apply the noisy optimization algorithm to the function $\theta \mapsto Simulate(s, MDP, \pi(\theta))$; get $\hat{\theta}$ the approximate optimum.
Return $\hat{\theta}$.

Direct Policy Search is usually applied offline, i.e. a single $\hat{\theta}$ is obtained once and for all. However, optimizing Θ to maximize $\theta \mapsto Simulate(s, MDP, \pi(\theta))$ specifically for the current state s for which we look for a decision is possible. We apply DPS and use the obtained policy $\pi(\hat{\theta})$ as a Monte-Carlo move generator in our MCTS.

The paper in [18] proposes to apply DPS (the terminology in the paper is different, but it is essentially DPS) based on a heuristic function obtained by experts, by smoothing the heuristic and adding parameters in it (the smoothing is here for making the problem easier to optimize). This is our approach here, except that we do not smooth policies as the randomized nature of our problems makes the objective function smooth enough. We use self-adaptation[88] as a noisy optimization algorithm. As a summary, our algorithm is as follows for choosing a move in state s within time t:

Procedure *OptimisticHeuristics*(s, ϕ, t, MDP)
Input: a state s, a time t, a parametric family of policies ϕ_θ.
Output: an action a.
Apply DPS with time budget $t/2$ for choosing $\hat{\theta}$ (use warm start if possible)
Apply MCTS, with $\phi_{\hat{\theta}}$ as a default policy, and time budget $t/2$, for choosing action a.

5.1.3 Experiments

Here, we compare the performances of different sequential decision making algorithms. Namely, we implemented vanilla MCTS, MCTS with a fixed default policy, MCTS with a default policy improved online by DPS, and DPS alone.

We made experiments on three different forms of the unit commitment problem, and on a more general energy management problem called bilevel.

Unit commitment problem

We work on a stock management problem, from [116].

The main points in the problem are that: (1) Unit Commitment problems can not be solved efficiently by traditional methods; these problems are usually simplified so that classical methods, like Bellman's stochastic dynamic programming, can be applied. The motivation of our work on Unit Commitment by Monte-Carlo Tree Search methods is that we want to work without simplifying too much the model. (2) Unit Commitment problems exist at many time scales (from seconds, up to years for hydroelectric stocks or tenths of years if investments are included) and many dimensionalities (from a few stocks to thousands of state variables), depending on the scope under analysis. We here work on small scale problems for the sake of statistical significance (working on our full problems requires by far too much time for reproducing runs tenths of times).

In this section, we will consider three variants of the unit commitment problem. The significant difference between these three variants is the way the stocks are connected. In the first one, they are lined up on a one dimension chain (we will call it the one river problem). In the second one, they are linked so that they form a binary tree, with the root being the last stock that the water goes through (we will call it the *binary rivers problem*). Finally, the third one is simply a random arrangement of the stocks, with one single constraint: no cycles are allowed.

Two different heuristics for the Unit commitment problem. The expert parametrized heuristic that we use has been designed using knowledge about the problem, to make it particularly efficient on the one river variant of the unit commitment problem. On the other hand, the naive heuristic uses almost no knowledge about the problem. Given a state s, it requires the current time to go t, and the average demand at the current time step D_{avg}. We provide below pseudo-codes of both heuristics.

Experimental results on the Unit Commitment problem. We present here the results obtained on all three variants of the unit commitment problem. Each time, we compared the following algorithms: (i) vanilla MCTS, as presented in Section 3.2.3, (ii) MCTS-naive, a MCTS using the naive heuristic as a default policy,(iii) MCTS-expert, a MCTS using the expert heuristic as a default policy, (iv) MCTS-naive-DPS, a MCTS using the naive heuristic improved by DPS, (v) MCTS-expert-DPS, a MCTS using the expert heuristic improved by DPS and when relevant, (vi) the non tuned naive and expert heuristics.

The x axis shows the time budget allocated per decision made, in logarithmic scale, and went from 0.01 second to 2.56 second. The y axis shows the average reward. Each average reward was computed using 1000 runs. Error bars show the 95% confidence intervals. The higher the reward, the better the algorithm performed. It should be

Algorithm 19 Expert heuristic

Input: a parameter vector θ of dimension 3 (default value is $[1, 0, 1]$), a state S, of dimension N, $D(t)$ the expected electricity demand during time step t, $D_{timeToGo}(t)$ the expected total demand after time step t, $TSA(s, timeToGo)$ the total stock available (this assume a 1 river structure),$TI(timeToGo, averageInflows)$ the expected total usable water from inflows (this assume a 1 river structure).

Output: an action a.

initialize:

- total water available $= TWA = (\theta_0 + \theta_1 \times timeToGo) \times (TSA + TI)$
- $x = production\ by\ hydroelectricity = 0$
- $increaseWater \leftarrow true$
- $S_{available} = \sum_{0 \leq i \leq N-1} s_i$

while $(increaseWater$ and $x < S_{available})$ s_i being the current level of stock i **do**

define the marginal cost mc of increasing water, approximated as

$$mc = IC(x, s, t, D(t)) + \theta_2 \times LTC(x, TWA, t, D_{timeToGo}(t))$$

where:

- $IC(x, s, t, D(t))$ is negative; it is the marginal benefit associated to the reduction of thermal production.
- $LTC(\dots)$ is the sum of thermal production cost, if expected total demand $D_{timeToGo}$, decreased by the total production from the water stocks if equally distributed on the time steps to go, is produced thermally.

if marginal cost $mc > 0$ **then**

then $increaseWater \leftarrow false$

else

$x \leftarrow x + 1$.

end if

end while

Compute q, the ratio $\min(0, \frac{x}{S_{available}})$

return the action vector a defined as follows: $\forall 0 \leq i \leq N-1, a_i = q.L_i$

Algorithm 20 Naive heuristic, polynomial with degree m

Input: a parameter vector θ of dimension $m+1$ (default values are $[1, 0, \ldots, 0]$), a state S, of dimension N, t the remaining time steps, and D_{avg} the average demand after the current time step.

Output: an action a.

Compute total amount of water to use $W_{use} = \max(0, D_{avg}.(\theta_0 + \theta_1 t + \cdots + \theta_{m+1} t^m))$

Given S and the current level L_i of each stock i, $W_{available} = \sum_{0 \leq i \leq N-1} L_i$

Compute q, the ratio $\min(0, \frac{W_{use}}{W_{available}})$

return Return the action vector a defined as follows: $\forall 0 \leq i \leq N-1, a_i = q.L_i$.

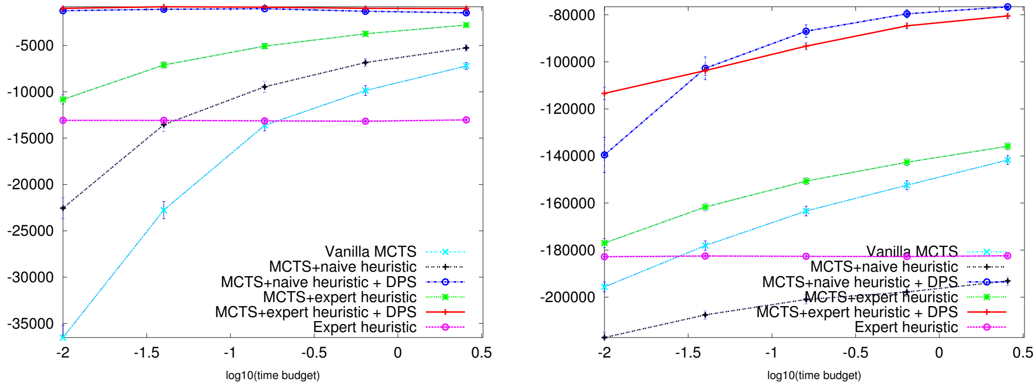


Figure 5-1: Performances of different variants of MCTS on the 1 river unit commitment problem (left) and the binary rivers (right), with 7 stocks, 24 time steps. Y axis shows the reward (the higher the better).

noted that the rewards cannot be compared between different variants of the unit commitment problem. Indeed, only the connections between the stocks change, and changing this changes the amount of water effectively available. Our results for the 1-river problem and for the binary rivers problem are shown on the left side and the right side of Fig. 5-1, respectively. We did not plot the results of the naive heuristic, that scored -150000 and -380000 respectively (far below other methods), for the sake of readability. In both experiment, MCTS-naive-DPS and MCTS-expert-DPS outperform by a factor of at least 100 the third placed algorithm, MCTS-expert. MCTS-naive and MCTS vanilla share the fourth and fifth places.

Our results on the random rivers problem are shown in Fig. 5-2. In this experiment, the most significant difference in the results is that MCTS-naive-DPS is about 10 times faster than MCTS-expert-DPS.

Over all three versions of the unit commitment problem, the most efficient and robust version has been MCTS-naive-DPS. Even on the one river problem, that the expert heuristic was particularly well tuned for, we could not see huge benefits from using it as a parametric function for DPS.

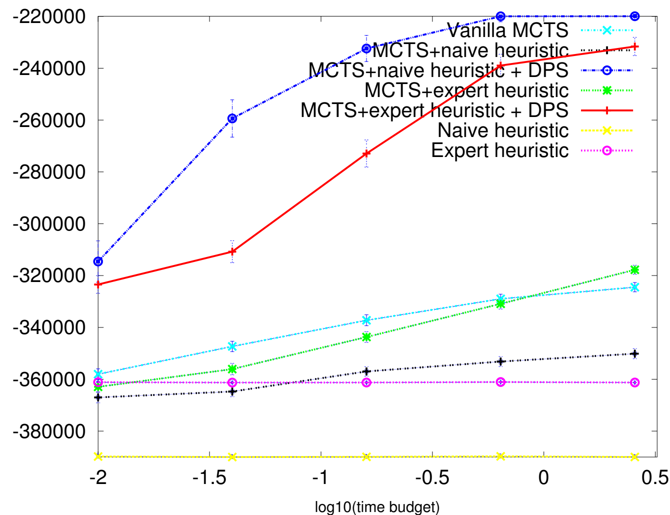


Figure 5-2: Performances of different variants of MCTS on the randomly connected unit commitment problem (7 stocks, 24 time steps).Y axis shows the reward (the higher the better).

Experiments on the investment problem

In this section we experiment our algorithm on a problem (simplified from [116]) as follows:

– At each time step, we decide investments; there is a limited amount of money to invest, and investments must be distributed over 7 different possible infrastructures. There are therefore 7 decision variables for each time step.

– At each time step, a lower level problem (the management of the energy production system) is built and solved, and its cost is the cost of the current transition of the investment problems.

– There are 10 time steps, the last one has a strong influence because it reflects the long-term.

Our results compare the following strategies:

– a heuristic which gives a constant ratio of the investment on each possible infrastructure (the parameters of the heuristic are this proportions); the default parametrization is the same ratio for all infrastructures;

– DPS on a "sum of Gaussians" policy (parameters: positions of the Gaussians, widths, associated decisions; see [116];

– DPS on a "neural network" policy (parameters: weights, theresholds; see [116]);

– DPS on a "sum of Gaussians" policy, added to the heuristic with default parametrization;

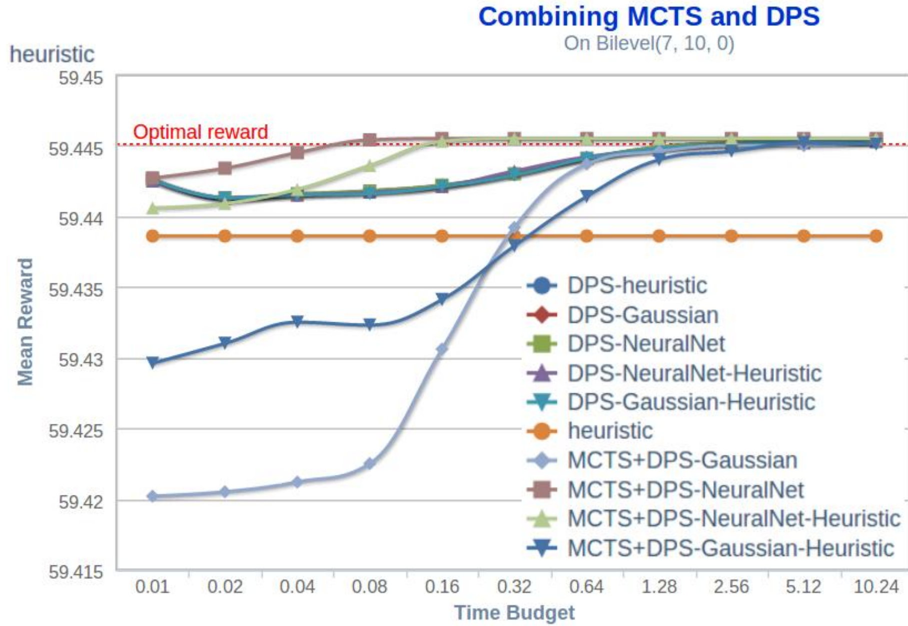


Figure 5-3: Results on the energy investment problem. The five DPS curves (curves 1 to 5) are very close to each other; results are better than for the heuristic alone, and versions without the heuristic are almost the same as versions with the heuristic. The MCTS+DPS+neural network was the most efficient strategy, outperforming MCTS+DPS+neural network+heuristic. The sums of Gaussians require more time for learning, hence the poor results for moderate budgets.

– DPS on a "neural network" policy, added to the heuristic with default parametrization;

– MCTS, on top of each of the above.

Results are presented in Fig. 5-3.

5.1.4 Conclusion

We combine DPS and MCTS. The DPS provides the Monte-Carlo simulator of the MCTS. The resulting algorithm, has no free parameter and outperforms by far the vanilla MCTS. We use human expertise at two levels: (i) For partial observation handling, i.e. the belief state estimation was hand-crafted, so that the problem is essentially a MDP rather than a partially observable MDP. The details of this are beyond the scope of work; (ii) In the Monte-Carlo move generator, because in spite of nice and interesting efforts in the literature, no generic algorithm, in the current state of the art, can define a Monte-Carlo move generator as efficiently as a human expert (in the case of Go, but also in the case of unit commitment problems). Nonetheless our DPS could strongly improve the heuristic by optimizing its parameters. We agree with the traditional statement that MCTS is surprisingly efficient when no human expertise is available, but we clearly see that human expertise was an easy key for a

speed-up 100, as well as human expertise is the key of recent progress in MCTS for the classical challenge of the game of Go.

Importantly, the need for human expertise is considerably reduced by the use of DPS for optimizing the heuristics, so that our results are a step towards generic MCTS tools.

5.2 Mixing MCTS with heuristics to solve POMPD

A large part of this section's contents can be found in [41], and the experimental results in [108].

5.2.1 Belief state estimation, from Mines to mathematics

In Section 5.2.1 we present the Mines games, which is convenient as an experimental testbed. In Section 5.2.1 we present the formalism of Markov Decision Processes and Partially Observable Markov Decision Processes. In Section 5.2.1 we formalize the problem of belief state estimation.

The Mines game

Consider the simple Mines game, which is well known for his free clones on several platforms (e.g. on Linux or Windows). Mines look like a simple game, sometimes equipped with a "hint" system, which helps you for finding a good move. However, this hint system is usually unable to solve more than simple cases, or cheats by using hidden information. Indeed, an exact choice of optimal move is far from being simple.

The rules of the game are as follows. This is a one player game. There are N mines, located randomly on a $p \times q$ grid. Mines are not visible. At each move, the player chooses one location in the grid. If there is no mine at this location, then the number of mines in the 8-neighborhood is displayed to the player; otherwise, the game is over. Usually, the score is the time before complete solving (i.e. all non-mines locations are played at least once) - the smaller, the better. No score in case of game over by playing on a mine.

We will here consider as score the number of (unique) moves before the game is over, divided by $pq - N$ (so that 1 means complete solving). Figure 5-4 shows that the game is not trivial: which move would you play here ?

All players know many cases in which a move can be played for free, because it is sure that there is no mine here - and many players know that sometimes, you can not be sure and must play with a non-zero probability of losing the game. But not many players know that in such cases, sometimes, some locations are more likely than others to be mines. Specific examples, as preliminaries for the discussion on Belief State estimation below, are given in Fig. 5-5.

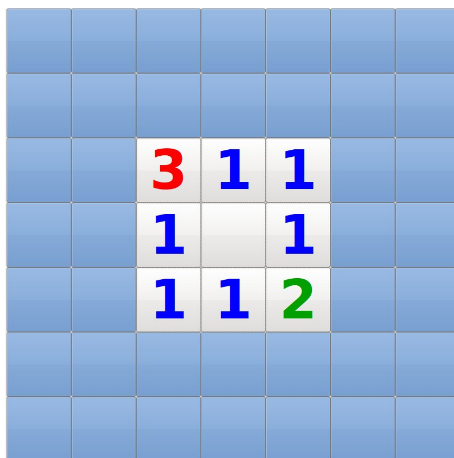


Figure 5-4: A case in which choosing the next move is non-trivial.

Some details on the rules. The Windows version of the game ensures that the first move is not on a mine (the mines are randomly distributed after the first move). Some versions on the game (Gnomine, default version on many Linux distributions) moreover ensure that the number of mines in the 8-neighbourhood is 0. In our tests we will only consider the Windows version, i.e. when one is guaranteed that the first move will not be on a mine, as this version is the most widely used.

The game is partially observable - the "complete" state (i.e. including the position of the mines) is not equal to the observed state (which is only a view of the game with covered locations and uncovered locations with a number of neighbouring mines). When you have observations, you can estimate the probability distribution on the internal state - this is the belief state. Therefore, it is a partially observable game. The trouble is that many solvers, like Monte-Carlo Tree Search, are aimed at working on fully observable Markov Decision Processes; we will see below (Section 5.2.1) the related definitions, and we will see in Section 5.2.1 how to transform a Partially Observable problem into a fully observable one in the 1-player case.

Markov Decision Processes and Partial Observation

Let's now consider Markov Decision Processes (MDP) and Partially Observable MDP (POMDP) from a more abstract point of view.

A MDP or POMDP is a directed graph, each edge being equipped with a label (action) and a number (probability); the edge between vertex x and vertex y has label m and number r if the probability of switching from state x to state y, when choosing action m, is r. The sum of probabilities in outgoing edges from a state x and equipped with a label m should be 1. By definitions, legal actions in x are labels which can be found on at least one outgoing edge from x. Nodes with no outgoing edge are termed terminal nodes and are equipped with a reward (in some definitions, edges are equipped with a reward, or non-terminal nodes; this will make no difference for this work).

Additionally, in a POMDP, each node x is equipped with an observation o_x. A

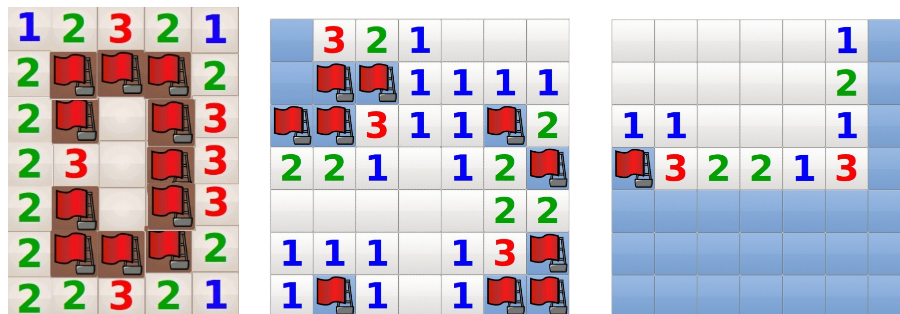


Figure 5-5: Left: here, you can deduce that one of the remaining locations is a mine. If you play in the middle location, you have an expected number of (unique) moves before losing which is, if you play perfectly, 1 (the three outcomes, losing immediately, or losing after 1 move, or completely solving the game, are equally likely) - whereas playing the top or bottom unknown location gives an expected number of (unique) moves 4/3 (with probability $\frac{1}{3}$, it's an immediate loss - otherwise, it's a complete solving). Middle: a situation with 50% probability of winning. Right: this situation is difficult to analyze mathematically; our program immediately sees that the top-right location (0,6) is a mine. However, this could easily (and faster) be found by a branch-and-bound optimization. More importantly, it also sees that the location just below (1,6) is good; but the real good point is that it can say which locations are more likely to lead to a long-term win than others.

MDP can be seen as a POMDP in which the observation is equal to the state or to a unique state identifier. A (possibly random) sequence of actions $a_1, \ldots, a_t, \ldots$ defines naturally a random sequence of states $x_1, \ldots, x_t, \ldots$ and observations $o_1, \ldots, o_t$; each state and observation is obtained from the previous one using actions and probabilities of transition.

A strategy is a (possibly random) mapping from a sequence of pairs (actions,observations) to actions. In a MDP, observations are uniquely determined as a function of states; so, equivalently, the strategy in a MDP can be defined as a mapping from sequences of pairs (actions,states) to actions (one can note that it is known that the mapping can depend only on the last state without loss of performance in optimal strategies). Basically and informally, a MDP is therefore a POMDP in which the player always knows in which state she is. Given a MDP or POMDP, and a strategy, a random sequence of states and observations is defined, as well as a random reward. The Mines game is a priori a POMDP; mines are not visible. Yet, it can be rephrased as a MDP; we'll see this in the next section.

POMDP transformed into MDP: the problem of belief state estimation

A POMDP can be rephrased as a MDP. This can be done as follows. Consider a POMDP P. The MDP M rephrasing P is built as follows:

– The state space of M is the set of sequences of pairs (actions,observations) in P;

– In M, action a in state $((a_1, o_1), \ldots, (a_t, o_t))$ leads to $((a_1, o_1), \ldots, (a, o_{t+1}))$ where o_{t+1} is the random observation obtained when applying action a in state s where s is the tth state, randomly drawn according to observations $(o_1, \ldots, o_t)$ and actions $(a_1, \ldots, a_t)$.

This transformation (originating in [2]) has the advantage of being consistent; a good strategy for M is a good strategy for P - the distribution of rewards is exactly the same. It has the drawback that it leads to a much bigger state space; possibly, M is infinite whenever P is quite small. Moreover, the conditional law above can be very hard to sample - we have to sample the next observation, conditionally to all past observations. The key part of this sampling is the sampling of s_t, the state after $t-1$ actions and observations, conditionally to past actions and observations. This is known as the problem of *belief state estimation*.

The likelihood of an observation, conditionally to the hidden state, is the probability of this observation conditionally to the hidden state; the likelihood of the hidden state, conditionally to the observation, is the probability of this hidden state, conditionally to the observation. As well known (Bayes formula), when the observation is given, the likelihood of the hidden state is proportional to the probability of observation, given the hidden state.

There are several methods for this:

– Simple rejection methods: randomly sample s and reject s unless it is accepted, which happens with probability proportional to its likelihood conditionally to observations. This is very slow, but mathematically consistent. This is summarized in Alg. 21.

Algorithm 21 The rejection method for estimating the belief state.

```
while True do
  Randomly draw x
  Let r ← likelihood of observation (given x)
  Randomly draw u in [0, 1].
  if u ≤ r then
    Break.
  end if
end while
Return x
```

– Markov-Chain Monte-Carlo (MCMC) [60] can be used as well; e.g. Metropolis-Hastings, the simplest version of MCMC.

– Randomly extend the observations in order to get a “complete” state (observed part + unobserved part); this is possible in some games, and widely used in e.g. phantom-games or dark chess which are a quite difficult challenge in terms of partial observability[32, 30]. This is certainly not consistent in general (i.e. this approximation of M is not equivalent to P), unless the random choice of the completion in a very specific manner - this is certainly not usual.

The purpose of this contribution is to propose new consistent methods for belief state estimation. The main claim of this section is that, even on an a priori simple game like Mines, a rigorous belief state estimation can provide significant improvements on a simple constraint satisfaction problem. By "consistent" methods, we mean methods which, at least at the limit of infinite computational power, lead to perfect estimates; POMDP algorithms based on this algorithm should, as a consequence, be consistent in the sense that they find, at the limit of an infinite computational power, an optimal strategy; this will be formalized in the rest of this section.

5.2.2 MCTS with belief state estimation

In this section, we explain how MCTS, as presented in 3.1.1, can be adapted to POMDP, by using belief state estimation (BSE) techniques. We then present some of the possible BSE techniques that have been tried in combination with MCTS.

Combining MCTS with BSE.

MCTS is particularly well suited to be applied to POMDP, because the tree structure it relies on naturally contains the history of each state, i.e. all explored trajectories, without needing to increase the size of the state variables. That way, if each node of the tree contains either an observation o or an observation-action pair (o, a), then each node having a unique history contained in the tree, has enough information to sample possible hidden states. What is essential at this point, and the only thing needed in addition to the traditional MCTS algorithm, is a BSE algorithm, to provide a generative model. Without it, MCTS cannot operate. The most important property that the BSE algorithm needs to verify in order for the complete algorithm to be consistent is that, given a history of observations and actions, it should generate a possible hidden state following the true random distribution of that state.

Given a consistent BSE algorithm, it is then straightforward to include it in the generative model, and to run MCTS on a tree whose nodes only contain observations. A transition from a node containing only an observation can then refer to its past actions and observations, and refer to the BSE algorithm to generate a possible hidden state, and the associated next observation.

The first idea of BSE algorithm that we combined with MCTS was the rejection algorithm21. But even though it is consistent, it is too slow for even medium scale applications. We present now a few alternatives.

Faster methods

Constraint Satisfaction Problem (CSP). CSP has been used in combination with MCTS in [108], where they give its formal description as shown in Alg. 22. Given some observations, CSP can be used to give the exact probability of each possible hidden state. Acting greedily with respect to CSP alone is a myopic strategy, i.e. it does not consider the long term consequences of the immediate move. Even though

it can be used to detect the locations with the lowest probability of having a mine, it is suboptimal.

Algorithm 22 The CSP algorithm for playing MineSweeper. The algorithm is intrinsically myopic: it only optimizes the 1-step result (minimum probability of immediate death).

CSP-function for choosing a move on a partially visible board with M mines
for each location l of the board **do**
  $Nb(l) \leftarrow 0$
end for
for each positioning p of the M mines on the board consistent with observations **do**
  for each location l with a mine for p **do**
    $Nb(l) \leftarrow Nb(l) + 1$
  end for
end for
Play move l uniformly chosen among the set of uncovered locations l such that $Nb(l)$ is minimum.

CSP with a heuristic. One of the weaknesses of CSP is that it is not equipped to deal with ties, i.e. with situations where there are multiple locations sharing the same lowest probability of hiding a mine. In [28], the authors introduce a heuristic, specific to the Mines game, to help solve these ties. This is especially important for the first move, where CSP cannot give any information. The heuristic added to CSP in [28] can be summarized as follows:

– the first move needs to be in one of the corners;

– when there is a tie between multiple locations according to CSP, pick one of the locations that is the closest to the frontier, i.e. the line between covered and uncovered locations.

5.2.3 Experimental results

Results of these methods, applied to Mines, are compiled in Table 5.1, from [28]. CSP-PGMS is the PGMS implementation of CSP[1]. HCSP is the implementation of CSP with heuristic breaking ties by playing as close as possible to the frontier between covered locations and uncovered locations[81]. BSSUCT is the implementation of UCT from [107]. OH is our Optimistic Heuristics program.

[1]http://www.ccs.neu.edu/home/ramsdell/pgms/

Format	CSP-PGMS	HCSP	BSSUCT	OH
4 mines on 4x4	64.7 %	67.0%	70.0% ± 0.9%	67.0% ± 0.5%
1 mine on 1x3	100 %		100%	100%
3 mines on 2x5	22.6%	21.0%	25.4%± 1%	23.4% ± 0.5 %
10 mines on 5x5	8.20%	8.51%	9% (p-value: 0.14)	11.4% ± 0.4 %
5 mines on 1x10	12.93%	12.7%	18.9% ± 0.2%	17.0% ± 0.4 %
10 mines on 3x7	4.50%	4.76%	5.96% ± 0.16%	6.1% ± 0.2 %
15 mines on 5x5	0.63%	0.63%	0.9% ± 0.1%	1.15% ± 0.1 %
10 mines on 8x8		79.9 %		80.2 ± 0.48
10 mines on 9x9	80%	90.5%		89.9% ± 0.3%
40 mines on 16x16	45%	76.4% ± 0.4%		74.4% ± 0.5% (100 sims per move)
99 mines on 16x30	34%	38.1% ± 0.5%		38.7 ± 1.8 % (100 sims per move)

Table 5.1: Results of various implementations on the MineSweeper games. Results are averaged over 10^5 games except 16x30 which is averaged over 10^4 games. For OH, results are obtained with 10000 simulations per move, except expert mode and intermediate mode (99 mines on 16x30 and 40 mines on 16x16) which use 100 simulations per move. BSSUCT is not documented for cases in which it was too slow for being operational in [107].

5.2.4 Conclusions

Rigorously estimating the belief state is a big challenge in partial observation problems. This turns out to be important even in simple games like Mines.

There are several tools for estimating belief states. From the simple and slow but consistent rejection method, to the faster and still consistent CSP algorithm. As these tools are used to compute the transitions, they are critical to the success of any algorithm built on top of them. Results show that even the addition of a simple tie breaking heuristic can improve the performances.

We also want to emphasize that the addition of UCT on top of tools like CSP is essential to the optimality of the overall method. That might be surprising for humans, but the best strategy does *not* only consists in playing a move with minimum probability of immediate death (see Fig. 5-5), and our algorithm, in spite of drawbacks (it can be slow for easy cases), has the strength of, asymptotically, playing optimally these subtle cases. Our example (first move in 4x4 board) might look like an artificial counter-example, but Fig. 5-5 (left) shows that such non-trivial phenomena also occur late in the game. This second example can be solved manually, but it is certainly hard for humans to solve cases like Fig. 5-5 (right) without a tool like our algorithm, and constraint programming does not provide a solution to this.

We guess that such a rigorous approach might be important in two-player games as well, at least with moderate size; this is a first natural further work. This is far more complicated as in two-player games we have no direct estimate of the likelihood. Moreover, in many games, the state space is huge, making it hard to get rid of heuristics[125, 32, 30]. As a second further work, a direct application of this work is also the test on industrial POMDPs; we guess that neglecting the PO part or handling it in a computationally fast but statistically irrelevant way might lead to

serious troubles in real world cases.

5.3 A Meta-bandit framework

5.3.1 Review of multi-armed bandit formulas

So far, we have focused on many areas of SDM problems, on MCTS, its extension to continuous domains, its consistency, and even its combination with other optimization tools, but we have not really covered an essential part of its success: the balance between exploration and exploitation. We did say that this balance is traditionally achieved in MCTS by using scoring formulas from the multi-armed bandits community[74, 75], giving its name to the UCT algorithm. Here, we want to present a short review of the works around multi-armed bandit problems. This will be one of the two bricks necessary to understand the meta-bandit framework, and its potential applications.

Stochastic bandits

The stochastic bandit setting can be described as follows: there are K arms, and an agent needs to play one of these arms at each time step $t \geq 1$. The arm played at time t is noted $a_t \in \{1, \ldots, K\}$. After playing arm a_t, the agent receives a reward r_t, that follows some unknown distribution $L_t(a_t)$. Each of these distribution has a hidden expectation, and the goal of the agent is to figure out what the best arm is.

From this setting, there are two common objective functions that can be used: the *Simple Regret*, and the *Cumulative Regret*. The simple regret is the difference between the mean of the best arm and the mean of the arm played at the last time step, i.e.:

$$R_S = \max_a \mathbb{E}(L_T(a)) - \mathbb{E}(L_T(a_T)) \tag{5.1}$$

The cumulative regret is the same thing, but cumulated over each time step, i.e.:

$$R_C = \sum_{t=1}^{T} \left(\max_a \mathbb{E}(L_t(a)) - \mathbb{E}(L_t(a_t)) \right) \tag{5.2}$$

Many contributions on this topic, starting with[78], focused on the stationary case, i.e. the case where for some arm a, $L_t(a) = L(a)$ for all $t \geq 1$. The author of [10] worked on this case, using upper confidence bounds (UCB) to minimize the cumulative regret over a finite time horizon T. The authors of [4] introduced the use of the empirical variance to improve the performance of the state of the art at the time.

In the case of non stationary bandits, one can read [65], that is specially designed to handle multi-armed bandits with large variation of the hidden distributions over time.

Finally, these ideas where put together with MCTS in[74], giving birth to the Upper Confidence Tree algorithm (UCT), that is essentially the algorithm shown in

Section 3.1.1.

An extension to the case where there are infinitely many arms (continuous bandits) has been done by [119] and [27].

In this section, we will focus on the simple regret, as it is more relevant to our application, and to the context of MCTS. In that context, it is important to distinguish two phases. The first is the *exploration phase*, i.e. time steps from 0 to $T-1$, when the aim is to gather information, even if it means playing bad arms. The second is the *final recommendation phase*, i.e. time step T, when the goal is to select the best arm possible. We will soon present several algorithms for both phases.

We will sometimes need the following notations: $\hat{L}_t(i)$ is the estimate, at time t, of the regret associated to the ith arm. Similarly, $N_t(i)$ is the number of times the ith arm has been played up to time t.

Adversarial bandits

Adversarial bandits happen when, instead of having the rewards follow (possibly time varying) random distributions, these rewards are decided by some adversary. It can then be seen as a two player game. At each time step, player 1 chooses an arm selection strategy (that could, and should be stochastic), and player 2 then chooses rewards for each arm. In such a setting, a deterministic player 1 would do very poorly, as player 2 can then always assign the lowest possible reward to the chosen arm, and high rewards to all others.

In [62], authors study this setting under a game theoretical perspective, computing the Nash equilibrium of zero sum games. [9] extended this work to an adversarial bandit setting, leading to the EXP3 technique. Finally, in [3], authors improved the previous algorithm and introduced Implicitely Normalized Forecaster (INF).

Algorithms for exploration

We present below several known algorithms for choosing a_t during the exploration phase.

- The **UCB (Upper Confidence Bound)** formula is well known since [78, 10]. It is optimal in the one player case up to some constants, for the criterion of cumulative regret. The formula is as follows, for some parameter α: $a_t = mod(t, K) + 1$ if $t \leq K$; $a_t = \arg\max_i \hat{L}_{t-1}(i) + \alpha\sqrt{\log(t)/N_{t-1}(i)}$ otherwise.

- [25] has discussed the efficiency of the very simple **uniform exploration strategy** in the one-player case, i.e. $a_t = \arg\min_{i\in\{1,\ldots,K\}} N_{t-1}(i)$, with EBA as a final recommendation method (see next section). In particular, it reaches the provably optimal expected simple regret $O(\exp(-cT))$ for c depending on the problem. [25] also shows that it reaches the optimal regret, within logarithmic terms, for the non-asymptotic distribution independent framework, with $O(\sqrt{K\log(K)/T})$.

- **Successive Reject (SR)** is a simple algorithm, quite efficient in the simple regret setting; it explores uniformly non-discarded arms and at some given time steps discards the weakest arm; see [5, 37] for more details.

Algorithms for final recommendation

Choosing the final arm, used for the real case, and not just for exploration, might be very different from choosing exploratory arms. Typical formulas are:

- **Empirically best arm (EBA)**: picks up the arm with best average reward. Makes sense if all arms have been tested at least once. Then the formula is $\hat{a} = \arg\max_i \hat{L}_T(i)$.
- **Most played arm (MPA)**: the arm which was simulated most often is chosen. This methodology has the drawback that it can not make sense if uniformity is applied in the exploratory steps, but as known in the UCT literature (Upper Confidence Tree[74]) it is more stable than EBA when some arms are tested a very small number of times (e.g. just once with a very good score - with EBA this arm can be chosen). With MPA, $\hat{a} = \text{argmax}_i N_T(i)$.
- **Upper Confidence Bound (UCB):** $\hat{a} = \text{argmax}_i \hat{L}_T(i) + \alpha\sqrt{\log(T)/N_T(i)}$ This makes sense only if $T \geq K$. UCB was used as a recommendation policy in old variants of UCT but it is now widely understood that it does not make sense to have "optimism in front of uncertainty" (i.e. the positive coefficient for $\sqrt{t/N_t(i)}$ in the UB formula) for the recommendation step.
- As Upper Confidence Bound, with their optimistic nature on the reward (they are increased for loosely known arms, through the upper bound), are designed for exploration more than for final recommendation, the **LCB (Lower Confidence Bound)** makes sense as well: $\hat{a} = \text{argmax}_i \hat{L}_T(i) - \alpha\sqrt{\log(T)/N_T(i)}$
- For SR (successive reject), there are epochs, and one arm is discarded at each epoch; therefore, at the end there is only one arm, so there is no problem for recommendation.

5.3.2 Energy management: comparing different master plans

Consider a problem in which the tactical level is hard. Then, the strategic level has to decide, for each simulation at the tactical level, the computational power devoted to this simulation. Formally, $L(\theta)$ is replaced by $L(\theta, c)$ where c is a time budget for the simulation. c large implies more precision (less bias) on the estimation; however, it also reduces the budget T for a given runtime, because the runtime will be nearly $T \times c$. Classical simple regret bounds (see e.g. [25] for a survey) show that the optimal expected precision is $\leq \sqrt{C \times K \log K/T}$ (reached e.g. by uniform exploration and EBA), where C is an absolute (i.e. distribution-independent) constant; therefore, if we want a precision ϵ, we know that we need T such $\sqrt{C \times K \log K/T} \leq \epsilon$. Deciding the

runtime c necessary for a bias upper bounded by ϵ is more difficult as existing bounds are far from being practical on non-trivial sequential decision making problems.

In our work, we consider the following problem: the agent needs to decide how to distribute a limited number $K+1$ of sites to energy production facilities. A site can be either a thermal power plant (TPP) or a hydro-electric plant (HEP), with the constraint that we must have at least one of each facility. This means that there are exactly K different possible configurations (the order does not matter): 1 TPP, 2 TPP, ..., K TPPs. The resulting problem is a non linear stochastic sequential decision making problem (inflows and energy demand are random).

Here, each possible configuration of the problem is an arm. The real value of such an arm is the expected cost of such a configuration under optimal management of the production facilities.

In our experiment, we used a version of continuous MCTS (Monte-Carlo Tree Search[44]), as described in [39], to evaluate each arm. The longer MCTS runs, the less noisy and the closer it gets to the optimal value of a given configuration. There is an obvious dilemma, between spending more time on each arm evaluation, and playing more arms. This is not the focus of this work, but we believe this should be addressed in a future work.

We considered a problem with 8 locations, and at least one HEP and one TPP ($K = 7$). The time horizon is of 5 time steps. An approximation of the real value of each arm is shown in table 5.2.

This problem is particularly difficult for two reasons. First, all the K configurations of the problem are not equally easy to solve for MCTS. Hence, for a small tactical budget c, it is very likely that $\theta^\star_c = \arg\max_\theta \mathbb{E}L(\theta, c)$ will be different from $\theta^\star = \arg\max_\theta \mathbb{E}L(\theta)$. This means that if c is not big enough, the bandit algorithm is going to converge towards a suboptimal arm. Secondly, the distribution followed by each arm is very far from satisfying common assumptions of bounded rewards and variances. Our problem has no informative bounds on the rewards. And, because for small tactical budgets, MCTS will occasionally make mistakes, the distribution of some arms is heavy tailed. This is particularly harmful to the stability of the mean estimator, that is crucial in most bandit algorithms studied here.

We compared five different bandit strategies: UCB, Successive Reject, and uniform exploration, UCB and SR using the median of the rewards instead of the mean reward. When it made sense, we used up to three different algorithms for final recommendation: LCB, EBA, and MPA. Our results are shown in table 5.2.

5.3.3 Discussion

From our results, it seems like the most determinant factor in reducing the simple regret is the budget given to MCTS. At a MCTS budget of 3.2 or 12.8 seconds, an increase in T actually decreases the performances of most algorithms. The only bandit algorithm that works in this setting is SR using the median. Looking closely at our data, it appears that the main reason for the poor performance of all the other algorithms is due to the heavy tail distribution of the arms. In such a setting, the mean estimator does not work. One needs to use a truncated version of the mean.

k	Cost
1	3.54 ± 0.025
2	3.20 ± 0.015
3	2.94 ± 0.039
4	2.36 ± 0.11
5	1.94 ± 0.11
6	**1.04** ± 0.14
7	$4.00{\cdot}10^2$ ± 0.059

c (s) T cT (s)	3.2 16 51.2	3.2 64 204.8	3.2 256 819.2	12.8 16 204.8	12.8 64 819.2
UCB+EBA	1.78 ± 0.11	2.34 ± 0.056	.	0.62 ± 0.12	1.32 ± 0.12
UCB+MPA	1.86 ± 0.12	2.36 ± 0.046	.	0.50 ± 0.091	1.50 ± 0.11
UCB+LCB	1.79 ± 0.11	2.34 ± 0.055	.	0.50 ± 0.091	1.50 ± 0.11
UCB+median+MPA	1.68 ± 0.11	2.27 ± 0.075	2.40 ± 0.059	0.39 ± 0.095	1.41 ± 0.13
UE+EBA	**1.03 ± 0.13**	2.18 ± 0.078	.	**0.21 ± 0.067**	0.93 ± 0.14
UE+LCB	1.67 ± 0.16	2.19 ± 0.077	.	0.26 ± 0.093	0.93 ± 0.14
SR	1.47 ± 0.12	2.28 ± 0.068	2.40 ± 0.036	0.32 ± 0.076	1.15 ± 0.15
SR+median	**1.11 ± 0.13**	**1.04 ± 0.15**	**0.70 ± 0.095**	0.28 ± 0.080	**0.075 ± 0.071**

Table 5.2: Left: “real” value of each arm; computed by giving MCTS a large budget (100s). Right: expected simple regret of different bandit algorithms, as a function of the budget allowed to arm plays and to T.

Taking the median is an extreme way to truncate the mean; more subtle approaches could be used, and would be an interesting basis for future work.

Most likely, increasing the MCTS budget to a very high value until the arms are no longer heavy tailed would make the mean based algorithms more efficient. But this could prove impractical. We think that an important future work would be to find an adaptive method to progressively increase the bandit budget and the MCTS budget, in order to avoid wasteful increases in bandit budget, as seen in our results (see table 5.2). As the number of arms played increases, it should be beneficial to increase c, the budget given to the tactical solver, to improve its accuracy. In addition, like MCTS itself, this framework could easily allow parallelization of the computation for each of the sub-problems.

Chapter 6

Backup operators for SDM

In this chapter, we take a closer look at the way information is backed up through the tree, after each iteration of MCTS. We start by reviewing the state of the art, mostly focused on adversarial games. We explain our motivations to move away from the traditional UCB formula, and show some alternatives, with some empirical results.

6.1 State of the art

In this section, we give a quick review of the backup operators in MCTS, both in two players and single player games.

6.1.1 Related work and how it relates with MCTS

Related work for adversarial games

An interesting work on backup operators can be found in [46], where the author reviews the common practice for backing up information in two players game trees, and introduces the Crazy Stone algorithm for computer Go. We borrow several of the observations made in that review.

In [22], the author uses central limit theorem, considering that the rewards of all arms are independent and identically distributed (iid) random variables, in order to prune branches of the tree, i.e. to focus the search in certain promising actions. Because a radical pruning approach seemed too risky (one may turn his back on the optimal action just because of a few unlucky initial simulations), researchers started to use stochastic bandit tools, leading UCT, as seen in Section 3.1.1.

However, as seen in Section 5.3.1, the strong and attractive results from the multi-armed bandit community often rely on one or more of the following assumptions:

- arms have stationary underlying distributions
- the objective is to minimize the cumulative regret

We will quickly address each of these assumptions, in the context of MCTS.

On the stationary distributions in MCTS Let a be some arm, i.e. one feasible action from state x. The very first time it is simulated, it is added to the tree, and then the default policy is applied successively until a final state is reached. At that point, the reward is computed, and backed up through the branch that was just used, that includes arm a from state x. Unless the default policy is already optimal to begin with, this first reward probably follows a distribution that has a low expectation. However, after 1000 simulations in arm a, the branch starting from it has grown significantly, and some more information about the possible futures from (x, a) have been explored. It means that the 1001st simulation of (x, a) will benefit of that extra information, and will return a reward that follows a different distribution, probably with a higher expectation and lower variance. This trend goes on as the number of simulations in (x, a) increases. It is hard to imagine a situation in which MCTS has its arms following stationary distribution.

On the objective of the SELECT function in MCTS The common criterion used to measure how good bandit algorithms are is CR. Being good means choosing bad actions a little as possible, including during the phase that could be called an exploration phase. It is then natural to think of other objectives for the SELECT function of MCTS. When in the root of the tree (i.e. the initial state), one probably wants to minimize the simple regret (SR).

However, when travelling inside the tree, the objective becomes more subtle. Then, one could aim at getting the most accurate approximation possible for the value functions ($V(.)$ or $Q(.,.)$). But one also wants to focus on promising branches. Indeed, there is no use in knowing with high certainty the value of all the states following the worst action doable at the first time step.

These remarks do not mean that one should never use UCB for MCTS. However, they do show that the theoretical grounds on which much of the commonly used bandit formula are based make little sense in our context. This means that there is probably some room for improvement in that area, as some recent research has indicated. We show some of the recent advances made in the next subsection.

A more comprehensive investigation on how computations could, and should be distributed in a setting like MCTS, under the name of *metareasoning for MCTS*, is done in[67].

Related work from single player games

In a recent publication[82], the authors developed an automatic tool to discover new backup operator, from a fixed set of variables (mean estimator, variance estimator, etc) and operators (plus, minus, etc). We believe this is an interesting and powerful method, that would be a promising future research direction.

6.1.2 When asymptotic results are not enough

Here, we quickly show some empirical results, on the toy problem called *Trap*, introduced in Section 3.2.4. We slightly modified the problem, to allow for more than two

time steps. The reward function is also modified, and only depends on the last state, so that the maximum reward is 10, the lowest -1. To obtain 10, one needs to choose the biggest action possible at each time step, which we refer to as the *risky strategy*, because doing so implies jumping over the gap. Ending in the gap returns a reward of -1. The *safe strategy*, that consists in staying more or less where we start, returns a reward of 5. This problem is a specific instance of the problem detailed in Section 6.2.2 (i.e. it has a probability of crash of 0).

In this experiment, we compared three versions of MCTS: (i) UCT (i.e. MCTS with the classical UCB formula, as described in Section 3.2.3); (ii) MCTS+expectimax and (iii) MCTS+mostSimulated, MCTS with two different variants of the SELECT function that will be described in details in the next section.

Having shown the consistency of MCTS with polynomial exploration, and UCT being similar to it (the only difference is in the exploration term, that is sensitive to online tuning anyway), one could expect UCT to work well on such a simple problem. Our results are shown in Fig. 6.1.2.

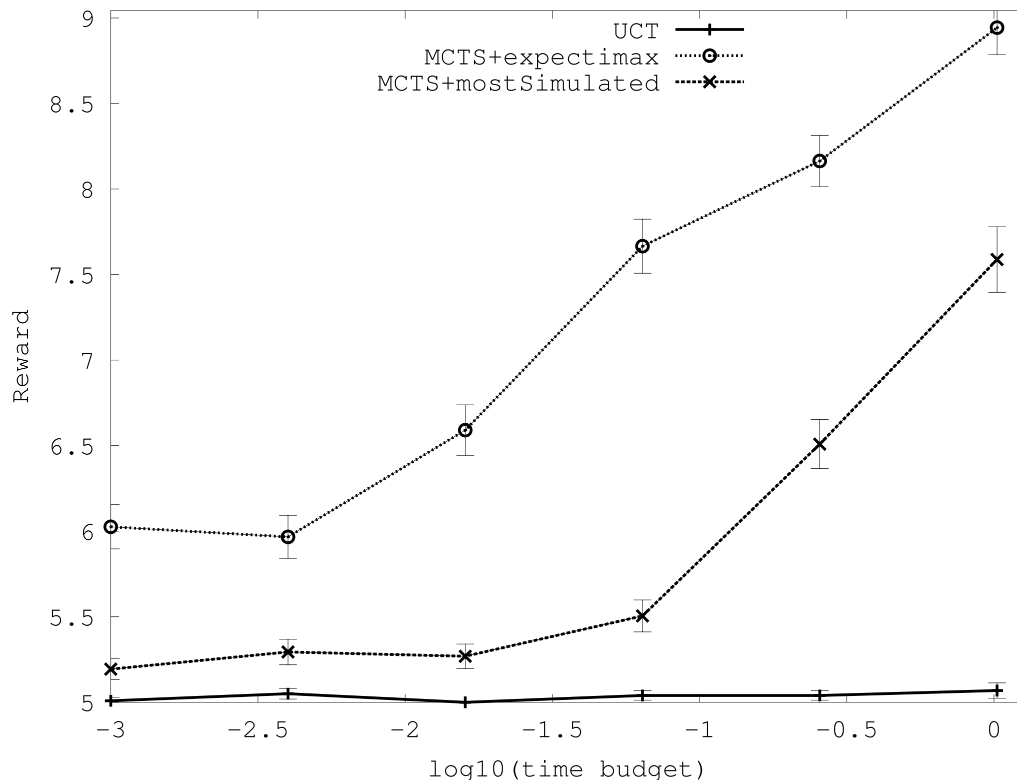


Figure 6-1: Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7.

What our experiment seems to indicate is that even for a simple problem of small size, UCT is unable to converge in a reasonable time to the optimum. Most likely, given enough time, it would converge. But the theoretical guarantees that we proved in Chapter 4 for polynomial MCTS depend on unknown constants, so it is hard to

know for sure if the convergence will happen sooner rather than later when increasing the time budget. Instead of launching longer experiments, we decided to look for alternative methods for the selection phase of MCTS, to improve, at least empirically, the convergence speed of the algorithm.

6.2 Finding the right balance between optimism and conservatism

In this section, we present two alternatives to the UCB formula to balance exploration and exploitation, namely Expectimax-MCTS (in Section 6.2.1), and MSP-MCTS (in Section 6.2.1). We show our experimental results in Section 6.2.2.

6.2.1 Alternatives to UCB

Expectimax-MCTS

First introduced in [89], *expectimax* refers to the algorithm that searches through a game tree, backing up information as follows: when in a random node, backup the average value of the children nodes; when in a decision node, backup the maximum of the children's values. This was intended for finite spaces, and was improved in [13] to reduce the complexity of the algorithm, and later in [66]. This was more recently successfully applied to some finite two-player games by [79].

As far as we know, there is no formal description of expectimax for MCTS in continuous domains. Just like the application of UCB to MCTS, applying expectimax to MCTS can only make the tree *asymptotically* converge to the typical expectimax tree for finite games.

Our implementation is a straight forward attempt to approximate expectimax in MCTS. When backing up the information from a simulation through a path of the tree, we do the following:

- in a leaf node, put the final reward, or the reward given by the Monte Carlo simulation that was just done (i.e. if this leaf is not a final state)

- in a random node, put the weighted average of the values stored in its explored children

- in a decision node, put the maximum value of its explored children.

We insisted on purpose on the word *explored*, to remind the reader that all these values are only as accurate as the number of explored children allows them. A random node with only two children might give a very inaccurate vision of the probability distribution it represents. Similarly, a decision node with only two children might be missing important good actions, thus severely underestimating its value.

We introduce a new notation to replace CR, the cumulative reward in a node, by EM, the expectimax value in a node. Alg. 23 shows how we modify the GROWTREE function of MCTS, as described in Section 3.1.1.

Algorithm 23 GROWTREE with Expectimax

Input: current tree $\mathcal{T}$, generative model $M(x,a) = [x', r]$, action sampler $s : \mathcal{X} \mapsto A$
Output: none, just updates the tree and possibly makes it grow

Initialize $x \leftarrow r(\mathcal{T})$, and $CR \leftarrow 0$
repeat
  $a \leftarrow$ SELECT(x, K)
  Children$(x) \leftarrow$Children$(x) \cup (x, a)$
  $[x', r] \leftarrow M(x, a)$
  Children$(x, a) \leftarrow$Children$(x, a) \cup x'$
  $r(x, a) \leftarrow r$
  $x \leftarrow x'$
until $n(x) == 0$ or x is a final state
$EM \leftarrow$ EVAL(x, M, ϕ)
while x not root **do**
  $n(x) \leftarrow n(x) + 1$
  $EM(x) \leftarrow \max_{a \in \mathsf{Children}(x)} EM(x, a)$
  $(x, a) \leftarrow$ Father(x)
  $n(x, a) \leftarrow n(x, a) + 1$
  $EM(x, a) \leftarrow \frac{1}{n(x,a)} \sum_{x' \in \mathsf{Children}(x,a)} n(x')EM(x')$
end while

Modifying the GROWTREE function, and more precisely the while-loop that back-propagates the information in the tree, allows us to store the expectimax values in the nodes, both in decision and in random nodes. It is important to notice that the EM values are back-propagated in the tree after each iteration of MCTS, from leaf to root. At each random node, the estimation of the expectation of the EM values over the children is re-computed. At each decision node, one needs to update the ranking of the children, according to their EM values, and to update the EM value of the decision node accordingly.

Just like UCB formula, our selection formula with expectimax has an exploration term to guarantee that all arms are, asymptotically, visited infinitely often. The resulting modified SELECT function is showed in Alg. 24.

Algorithm 24 SELECT with expectimax

Input: state-node x, $K > 0$ action sampler s
Output: an action a

$a_x = \text{argmax}_{a \in \mathsf{Children}(x)} \widehat{Q}(x, a)$
with $\widehat{Q}(x, a) = EM(x, a) + K \left(\frac{ln(n(x))}{n(x,a)}\right)^{\frac{1}{2}}$ if $n(x, a) > 0$, $+\infty$ otherwise
return a_x

The advantage of changing UCB to this version of expectimax is that as soon as one good action is found, even deep in the tree, the information is instantly backed up at the decision node levels (while still being averaged at random node levels). Informally speaking, the information travels faster in depth. The corresponding drawback

is that the estimations might end up overly optimistic, and more simulations might get invested in the luckiest branches. Even though as the number of simulation grows, this overestimation should disappear, it could still be a problem during the initial phase. In addition, as we work in continuous domains, even as the number of simulations grows to infinity, there are always new nodes to add. This is why, even though we do not have a proof, we suspect this method to be unstable, and to have no convergence guarantees, unlike the algorithm shown in Chapter 4.

Most simulated paths

One way to find a compromise between the overly optimistic estimations of expectimax in MCTS, without being as slow as UCB, is to use the *most simulated paths* (MSP) technique.

The idea is as follows: instead of backing up the value of the best action according to its EM score, like in expectimax, one backs up the value of the most simulated action. Meanwhile, when in a random node, proceed as in the case of expectimax, i.e. compute a weighted average of the values of the children. We hope that, while virtually pruning 'bad' actions from the backup operation like expectimax, this method will be more stable with respect to lucky nodes.

We introduce the notation of $MSP(.)$, the MSP of a node in the tree. The formal description of the new GROWTREE function is given in Alg. 25.

Algorithm 25 GROWTREE with Most Simulated Paths (MSP)

Input: current tree $\mathcal{T}$, generative model $M(x,a) = [x', r]$, action sampler $s : \mathcal{X} \mapsto A$
Output: none, just updates the tree and possibly makes it grow

  Initialize $x \leftarrow r(\mathcal{T})$, and $CR \leftarrow 0$
  repeat
    $a \leftarrow \text{SELECT}(x, K)$
    $\text{Children}(x) \leftarrow \text{Children}(x) \cup (x, a)$
    $[x', r] \leftarrow M(x, a)$
    $\text{Children}(x, a) \leftarrow \text{Children}(x, a) \cup x'$
    $r(x, a) \leftarrow r$
    $x \leftarrow x'$
  until $n(x) == 0$ or x is a final state
  $MSP \leftarrow \text{EVAL}(x, M, \phi)$
  while x not root **do**
    $n(x) \leftarrow n(x) + 1$
    $MSP(x) \leftarrow MSP(x, \text{argmax}_{a \in \mathsf{Children}(x)} n(x, a))$
    $(x, a) \leftarrow \text{Father}(x)$
    $n(x, a) \leftarrow n(x, a) + 1$
    $MSP(x, a) \leftarrow \frac{1}{n(x,a)} \sum_{x' \in \mathsf{Children}(x,a)} n(x') MSP(x')$
  end while

Like for expectimax, we want to ensure that all arms are visited infinitely often, and add an exploration term. This leads to a now familiar SELECT function,

described in Alg. 26.

Algorithm 26 SELECT with Most Simulated Paths

Input: state-node x, $K > 0$ action sampler s

Output: an action a

$a_x = \text{argmax}_{a \in \mathsf{Children}(x)} \widehat{Q}(x, a)$

with $\widehat{Q}(x, a) = MSP(x, a) + K \left(\frac{ln(n(x))}{n(x,a)} \right)^{\frac{1}{2}}$ if $n(x, a) > 0$, $+\infty$ otherwise

return a_x

Again, we do not have a proof of consistency for this version of MCTS, but we are more confident in its stability than in the one of expectimax-MCTS. Indeed, it should be less subject to sudden new lucky nodes. It should also be faster than UCT, especially as the number of time step grows.

6.2.2 Empirical analysis

In this section, we present some early empirical results to compare UCT, expectimax-MCTS (as seen in Section 6.2.1), and MSP-MCTS (as seen in Section 6.2.1). In order to do so, we introduce a slightly modified version of the Trap problem seen in Section 3.2.4, that we call *Crash Trap*, and present in Section 6.2.2. The experimental results are shown in Section 6.2.2.

Trap problem with probability of crash

The trap problem is interesting for several reasons. First, it is easy enough to have experiments running quickly. Second, it can be made arbitrarily difficult by changing its parameters; it was actually more difficult for UCT than many instances of our Stocks problem. Third, it is quite easy, for simple instances of Trap, to compute by hand the optimal reward.

However, the version described in Section 3.2.4 lacks an important feature for what we are interested in here: a probability distribution for the random process that can be shaped in tricky ways. Such a feature, once added, will allow us to design a Trap problem that can lead to underestimation *and* to overestimation.

By *overestimation*, we mean that some algorithm, when only limited information is available (i.e. after only a few simulations), will tend to overestimate the value of certain actions. This is actually hard to find in practice, as MCTS tends to have increasing rewards when the number of simulations increases, because its tree grows bigger. And a bigger tree usually means a better 'tree policy' (the policy followed when travelling down the tree), but still not as good as the optimal policy, hence the underestimation.

Overestimation will happen, however, if the good sequence of actions is easy to find, but the probability distribution underlying the random process is hard to estimate.

This is why we added a rare event to the Trap problem, namely the *crash event*. Of course, in order for it to matter, it needs to weigh heavily in the final reward when it actually happens. From now on, we will refer to this problem as the *Trap-Crash problem*. The modified reward function shape is shown in Figure 6.2.2.

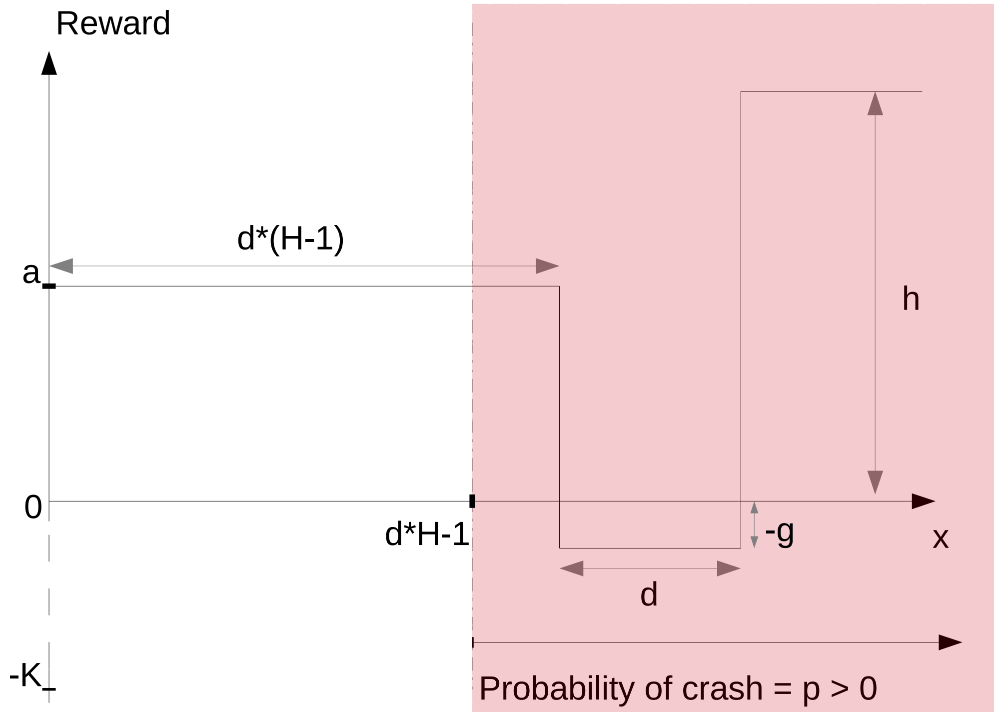


Figure 6-2: Reward function shape for the trap problem with a crash probability. x is the state variable; $0 < d < 1$ is the difficulty parameter; H is the number of time steps; $d(H-1)$ is the length between the initial state 0 and the gap; d is the width of the gap; the reward is obtained at the final time step, can be equal to a, $-g$, h, or $-K$; if the final state x_f is larger than $dH-1$, there is a probability p of 'crashing' and getting a reward of $-K$.

This problem is similar to the previous version, with three main differences:

– the reward is only computed at the final time step

– there can be multiple time steps, and the problem scales accordingly, following the core *difficulty parameter* d

– if the final state is larger than $dH-1$, then with probability p, one gets a large negative reward of $-K$.

This variant is thus aiming at two things: scalability in the number of time steps, and allowing to control the difficulty on two levels: d measures how hard it is to

find the sequence of actions that can lead to the high plateau on the right, while p measures how difficult it is to accurately estimate the underlying random process.

The resulting reward function is formally described in Eq. 6.1.

$$r(x) = \begin{cases} r_{nc}(x) & \text{with probability} \quad 1-p \\ r_c(x) & \text{with probability} \quad p \end{cases} \tag{6.1}$$

with

$$r_{nc}(x) = \begin{cases} a & \text{if} \quad x < l \\ 0 & \text{if} \quad l < x < l+w \\ h & \text{if} \quad x > l+w \end{cases}$$

and

$$r_c(x) = \begin{cases} a & \text{if} \quad x < c \\ -K & \text{if} \quad c < x \end{cases}$$

Experimental results

The results shown in Fig. 6.1.2 seem to indicate that one should always use expectimax-MCTS, rather than UCT. However, these results were obtained on a special instance of Trap-Crash, where the probability of crash was set to 0. In doing so, we made the random process almost negligible. In this section, we set the probability of crash to $p = 0.1$. We also set the crash reward to $-K = -60$, so that the expected reward for a state in the crash zone one action before the final time step would be equal to 3 (when acting optimally from there, i.e. $V^*(x,t) = 3$ for $x > dH - 1$ and $t = H - 1$).

Each time, we compare UCT, expectimax-MCTS, and MSP-MCTS. They all had progressive widening coefficients set to $\alpha^D = 0.25$ for decision nodes, and, depending on the experiment, $\alpha^R = 0.25$, 0.5, or 0.75 for random nodes. Finally, because the reward function does not have a lot of variance, we added an ϵ-greedy exploration, with $\epsilon = \frac{0.5}{(1+n(x))^{\frac{1}{4}}}$ (this value was chosen after a couple of runs, without really trying to tune it). Because of the high variance in the given rewards (the crash has a low probability but very high weight), it is hard to obtain small confidence intervals. Each point on the following figures is obtained after 10000 runs for each algorithm, but more runs would be needed for more accurate results.

Our first experiment, with $\alpha^R = 0.25$, showed how poorly expectimax-MCTS could behave when not properly estimating an important random process. It was easily deceived by the problem, choosing risky actions, overestimating their value. MSP-MCTS showed better performances, closer to UCT. Notice how even UCT struggles to reach the optimal reward, 5. The results are shown in Fig. 6.2.2.

For our second experiment, we increased the progressive widening coefficient for random nodes to $\alpha^R = 0.5$, hoping to improve all three versions of MCTS in terms of how well they estimate a probability distribution. Our results are shown in Fig. 6.2.2. With this new setting of DPW, both UCT and MSP-MCTS converge to the optimum reasonably quickly. However, expectimax-MCTS fails to converge; it even

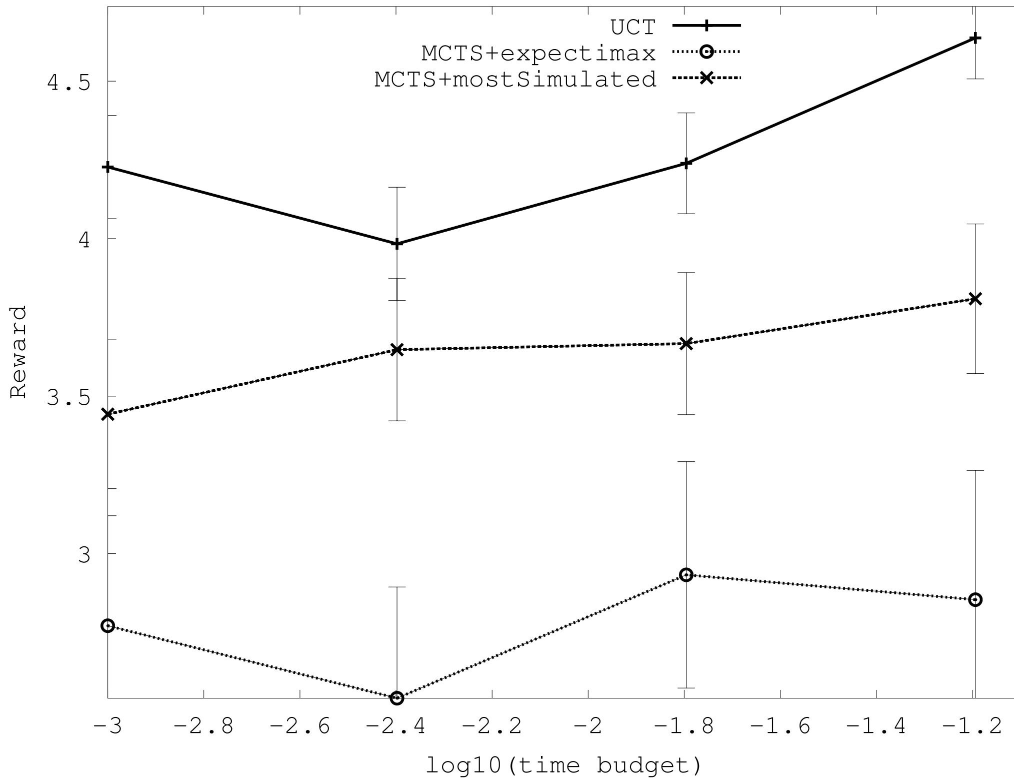


Figure 6-3: MCTS-DPW set to 0.25. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1.

diverges away from the optimum as the time budget increases. Notice however, that all obtained average rewards here are better than the ones obtained with $\alpha^R = 0.25$.

Finally, we increased α^R to 0.75. The results are shown in Fig. 6.2.2. With this setting, expectimax-MCTS seems to converge to the optimum, though still slower that MSP-MCTS and UCT. MSP-MCTS is still slightly slower than UCT, but this looks barely significant.

6.3 Conclusions

As discussed in Section 6.1, there has already been some work done to use other formulas than UCB in MCTS. Actually, most successful implementations of MCTS require a certain degree of tuning for the E/E balance to work well. Having this in mind, we think that the research direction taken by [82], where the authors use an offline meta algorithm to automatically discover E/E formulas, is promising.

However, in that work, like in most implementations of MCTS that we have found, the focus is on using the average reward and the variance of the rewards to score actions. To the best of our knowledge, the use of other tools, like expectimax, has only been done in two-player games [79], or in finite spaces[13].

Our goal here was to introduce other measures, namely expectimax and what

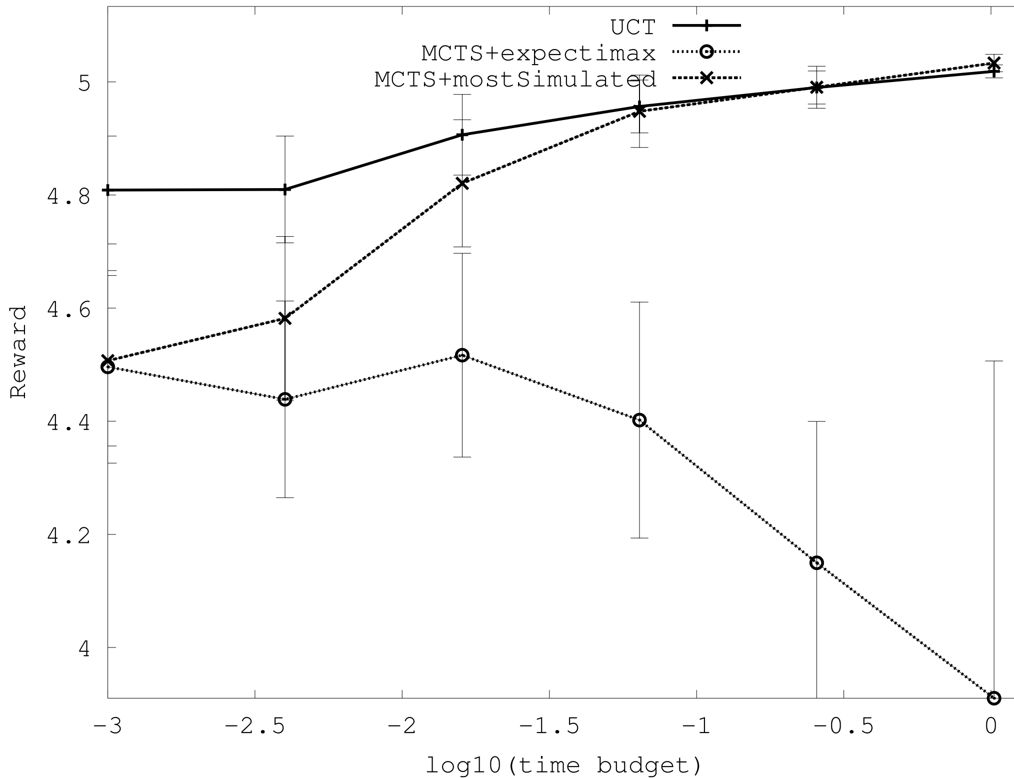


Figure 6-4: MCTS-DPW set to 0.50. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1.

we called *most simulated paths* (MSP). These measures cannot be obtained just by tuning UCB. As we observed in Figure 6.1.2, these two variants of MCTS can be dramatically faster than UCT on some problems (in that case, UCT showed no sign of convergence at all, being stuck in what looks like a local optimum). But, as we saw when we modified our test problem, both expectimax-MCTS and MSP-MCTS can perform worse than UCT (see Figure 6.2.2). As it turns out, one can increase the performances of both of our variants by changing the Double Progressive Widening coefficient (DPW), that was presented in Section 3.2.3.

It is hard to give a clear indication on what version of MCTS is better. We believe that, as it is often the case, the choice between these various implementations is up to the person behind the keyboard.

That being said, there is an important difference to notice, between the respective failures of UCT on one side, and expectimax-MCTS and MSP-MCTS on the other. When UCT failed (see Figure 6.1.2), the reason behind it was that UCB was simply far too slow to back up a good action when it found one only 3 time steps ahead of the root state. When the optimal action was found, somewhere down the tree, its good reward was usually already drowned in low rewards (the ones that are obtained when falling in the 'gap', see Section 6.2.2). This means that, in our opinion, playing with the meta-parameters of UCT as it is cannot save it from failing on this specific

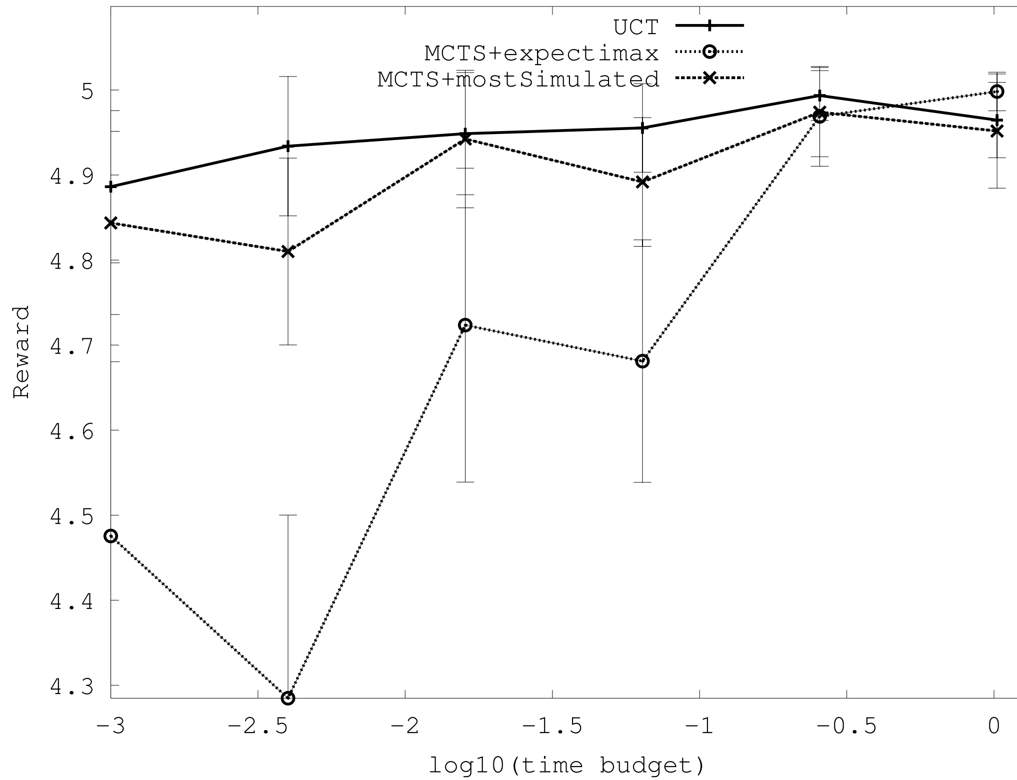


Figure 6-5: MCTS-DPW set to 0.75. Trap problem with 3 time steps, uniform additive noise of amplitude 0.03, gap of 0.7. Crash probability of 0.1.

instance. Only a more drastic change, like switching to expectimax-MCTS, can make it converge in a reasonable time.

This means that, in our opinion, when using MCTS for one player stochastic games, one should try either expectimax-MCTS or MSP-MCTS first, and then quickly tune its meta-parameters, i.e. the DPW coefficients, or the constants in the exploration term of the E/E formula. From our early empirical results, this seemed to be enough to provide good enough performance in both domains, especially in the case of MSP-MCTS.

Chapter 7

Conclusion

Even though our initial motivation was to improve the state of the art for solving energy unit commitment problems, and more specifically the ones involving hydroelectricity and hard to predict renewable sources, the results are more general and widely applicable.

In our opinion, some of the biggest problems with the existing methods to solve hydroelectric scheduling (HS) was that they almost always relied on *model simplifications*. Convexity assumptions are the most common (see Benders cuts in Section 2.3.5), but simplifications of the random processes (Markovian assumption, finite number of scenarios, etc) are also often made (see MPC in Section 2.3.5).

We do not deny that mathematical programming (MP) methods are essential to deal with large action spaces. Indeed, relying purely on RL methods cannot work when the action space is of a large dimension, eg. more than 10000 action variables, with equally many constraints. In such a setting (which occurs naturally in energy unit commitment problems), using a method like Stochastic Dual Dynamic Programming is more adequate. But, as we explained in Section 2.3.5, this implies simplifying the model. Our goal was then to explore a different approach, inspired by the RL community, and to try to understand how MCTS could be applied to HS. This means that we are mostly interested in two questions. How can MCTS work on continuous domains? And, how can we efficiently mix it with existing suboptimal but fast solvers, to obtain the best out of two worlds (namely, the MP and RL worlds)?

Our contributions to making MCTS viable and consistent for continuous SDM problems can be summarized as follows.

In Section 3.2, we presented continuous MCTS, by using the Double Progressive Widening trick. As we proved in Chapter 4, it lead to the first theoretically consistent MCTS algorithm for continuous SDM problems, with few assumptions (we assumed the availability of a generative model, and of an action sampler that has a non-zero probability of sampling actions arbitrarily close to the optimum). Although one could argue that the convergence rate guarantees are not strong (some constants in the probability of convergence are not constrained), the importance of this result lies in the convergence itself. To the best of our knowledge, there is no such result for continuous SDM problems that do not require more assumptions.

In Section 3.3.2, we presented Blind Value, an online method to improve explo-

ration in a continuous action space, without any assumption on the knowledge of that space. It can be seen as an attempt to discretize the action space first, and then focus on the more rewarding areas. The method requires some meta-parameters, that could be subject to refined tuning method.

In Section 3.4, we showed how to extend RAVE values to continuous spaces. This is one of the ways to transport information from one branch of the tree to another. To work properly, it requires the tuning of its meta-parameter, to adjust to the scale of both the state and the action spaces. We believe that attempts at improving the generalization of the information gathered during simulations (eg. transporting information from one branch to the other) are a key part to the success of MCTS in continuous spaces, and deserves more work.

One of the most interesting features of MCTS is that it can easily be combined with other existing methods. This is what we showed in Section 5.1, by using Direct Policy Search to build a better default policy to perform the rollouts of MCTS. We obtained promising experimental results. This showed one way of integrating powerful but suboptimal algorithms with MCTS, making it possible to improve an existing solution, thanks to the fact that MCTS makes less simplifications on the model of the problem.

In Section 5.2, we explained how MCTS can be used to solve POMPD. Again, the interesting part is that we can use existing heuristic to provide suboptimal solutions, and improve them with MCTS. This method has improved significantly the state of the art performance of solvers for the game of Minesweeper.

Section 5.3 showed how MCTS could be used in a meta-bandit framework, acting as a solver for the tactical part of the problem. These types of problem occur, for example, when one wants to decide how to allocate resources for an investment in energy facilities. The investment decision is done at a macro level, and each possible investment is then evaluated by a tactical solver, eg. MCTS. This framework poses the crucial question of how to divide the computing budget between the macro level and the tactical level; it would probably be a fruitful direction for future research.

Finally, in Section 6, we present our most recent work, i.e. possible alternatives to the common usage of the mean estimator in the selection function of MCTS. This function drives the exploration/exploitation balance in MCTS, and is crucial to the speed of convergence. Most implementations of MCTS need to first go through some tuning of this function. Our suggestion, supported by early experiments, is that one should consider using other backup operators, such as expectimax, and what we called *most simulated paths*(MSP). Coupled with basic tuning of the *double progressive widening coefficients*, these alternatives look promising, and possibly better than UCB. This would be our next future work: making more thorough experiments to understand how our alternatives fare for larger problems, and most importantly for problems with many time steps, where we think UCB becomes even less efficient.

Directions for future work. Many of our contributions to MCTS introduce meta-parameters (see Chapter 3). A good direction would be to design adaptive ways to tune these parameters online, as more simulations are done, to adapt them to any

specific problem. Another way would be to use meta-algorithms techniques, similarly to what has been done for finding custom selection formulas for MCTS in [82].

The proof of consistency presented in chapter 4 holds under certain assumptions, and for a MCTS with polynomial exploration. It would be interesting to prove the same result for a MCTS with UCB based exploration, since it already shows good results empirically. Also, one could try to prove the consistency of a MSP based MCTS (see chapter 6 for a description of MSP).

We think the framework presented in section 5.1 is very promising, in the sense that it allows to insert expert knowledge in MCTS easily. A way to improve it would be to design a method to improve the default policy online, during MCTS simulations, instead of improving it separately.

The results on back up operators shown in chapter 6 are also promising, but should undergo more thorough benchmarking. One option would be to test the UCB alternatives presented in our work to the game of Mine Sweeper, where it is easy to compare results to current state of the art.

The meta-bandit framework presented in section 5.3 raises a larger question: how should one distribute computing power between the meta-level (bandit algorithm) and the micro-level (in our case, MCTS). In our work, we used a constant ratio to make this distribution, but it would make sense to design an adaptive method, to avoid overspending computing power on the micro-level for example.

Finally, on the side of energy management, an important question still remains unanswered. Namely, we do not know how much the various approximations made on the model (e.g. making the assumption that the cost function is convex) cost. One could start by doing meticulous experiments as follows: establish an *accurate model* M of the system. From it, deduce an *approximate model* $\bar{M}$, that verifies the assumptions necessary to run state of the art algorithms like SDDP. Run SDDP on $\bar{M}$, obtain a suggested action a_{SDDP}, apply it to M, and repeat. Use this process to evaluate the performance of SDDP, and compare the results to the performance of MCTS on M. Ideally, this work should be done with an industrial partner, to have realistic models. We hope that the ongoing work of our team, with the company Artelys, will reach this objective in the near future.

Bibliography

[1] Broderick Arneson, Ryan Hayward, and Philip Henderson. Mohex wins hex tournament. *ICGA journal*, pages 114–116, 2009.

[2] K.J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.

[3] J.-Y. Audibert and S. Bubeck. Minimax policies for adversarial and stochastic bandits. In *proceedings of the Annual Conference on Learning Theory (COLT)*, 2009.

[4] J.-Y. Audibert, R. Munos, and C. Szepesvari. Use of variance estimation in the multi-armed bandit problem. In *NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation*, 2006.

[5] Jean-Yves Audibert and Sébastien Bubeck. Best Arm Identification in Multi-Armed Bandits. In *COLT 2010 - Proceedings*, page 13 p., Haifa, Israël, 2010.

[6] P. Audouard, G. Chaslot, J.-B. Hoock, J. Perez, A. Rimmel, and O. Teytaud. Grid coevolution for adaptive simulations; application to the building of opening books in the game of Go. In *Proceedings of EvoGames*, pages 323–332. Springer, 2009.

[7] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *The Journal of Machine Learning Research*, 3:397–422, 2003.

[8] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.

[9] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[10] Peter Auer and M. Long. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:2002, 2002.

[11] Peter Auer, Ronald Ortner, and Csaba Szepesvári. Improved rates for the stochastic continuum-armed bandit problem. In Nader H. Bshouty and Claudio Gentile, editors, *COLT*, volume 4539 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 2007.

[12] David Auger, Adrien Couetoux, and Olivier Teytaud. Continuous Upper Confidence Trees with Polynomial Exploration - Consistency. In *ECML/PKKD 2013*, Prague, Tchèque, République, September 2013.

[13] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artif. Intell.*, 21(3):327–350, September 1983.

[14] Danny Barash. A genetic search in policy space for solving markov decision processes. In *In AAAI Spring Symposium on Search Techniques for Problem Solvingunder Uncertainty and Incomplete Information*. AAAI Press, 1999.

[15] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(5):834–846, 1983.

[16] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[17] J.F. BENDERS. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[18] Yoshua Bengio. Using a financial training criterion rather than a prediction criterion. CIRANO Working Papers 98s-21, CIRANO, 1998.

[19] D.P. Bertsekas. *Dynamic Programming and Optimal Control, vols I and II*. Athena Scientific, 1995.

[20] Dimitris Bertsimas, Eugene Litvinov, Xu Andy Sun, Jinye Zhao, and Tongxin Zheng. Adaptive robust optimization for the security constrained unit commitment problem. 28(1):52 – 63, 2013.

[21] Amine Bourki, Matthieu Coulm, Philippe Rolet, Olivier Teytaud, and Paul Vayssière. Parameter Tuning by Simple Regret Algorithms and Multiple Simultaneous Hypothesis Testing. In *ICINCO2010*, page 10, funchal madeira, Portugal, 2010.

[22] Bruno Bouzy. Move pruning techniques for monte-carlo go. In *In Advances in Computer Games 11*, 2005.

[23] JustinA. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2-3):233–246, 2002.

[24] StevenJ. Bradtke and AndrewG. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1-3):33–57, 1996.

[25] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. *Theor. Comput. Sci.*, 412(19):1832–1852, 2011.

[26] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. Online optimization in x-armed bandits. In Daphne Koller, Dale Schuurmans, Yoshua Bengio, and Léon Bottou, editors, *NIPS*, pages 201–208. Curran Associates, Inc., 2008.

[27] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvari. X-Armed Bandits. *Journal of Machine Learning Research*, 12:1655–1695, April 2011.

[28] Olivier Buffet, Chang-Shing Lee, Woanting Lin, and Olivier Teytaud. Optimistic Heuristics for MineSweeper. In Ruay-Shiung Chang, Lakhmi C. Jain, and Sheng-Lung Peng, editors, *ICS - International Computer Symposium - 2012*, volume 20 of *Smart Innovation, Systems and Technologies*, pages 199–207, Hualien, Taïwan, Province De Chine, 2012. Springer.

[29] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC Pr I Llc, 2010.

[30] T. Cazenave and J. Borsboom. Golois wins phantom go tournament. *ICGA Journal*, 30(3):165–166, 2007.

[31] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101, 2007.

[32] Tristan Cazenave. A phantom-go program. In *ACG*, pages 120–125, 2006.

[33] G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.

[34] G.M.J.B. Chaslot, M.H.M. Winands, and H.J. van den Herik. Parallel Monte-Carlo Tree Search. In *Proceedings of the Conference on Computers and Games 2008 (CG 2008)*, 2008.

[35] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In Christian Darken and Michael Mateas, editors, *AIIDE*. The AAAI Press, 2008.

[36] Benjamin E. Childs, James H. Brodeur, and L. Kocsis. Transpositions and move groups in Monte Carlo tree search. 2008.

[37] Cheng-Wei Chou, Ping-Chiang Chou, Chang-Shing Lee, David Lupien Saint-Pierre, Olivier Teytaud, Mei-Hui Wang, Li-Wen Wu, and Shi-Jim Yen. Strategic Choices: Small Budgets and Simple Regret. In *TAAI*, Hualien, Taïwan, Province De Chine, 2012.

[38] Adrien Couetoux, Hassen Doghmen, and Olivier Teytaud. Improving the exploration in Upper Confidence Trees. In *Learning and Intelligent OptimizatioN Conference LION 6*, Paris, France, January 2012.

[39] Adrien Couetoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous Upper Confidence Trees. In *LION'11: Proceedings of the 5th International Conference on Learning and Intelligent OptimizatioN*, page TBA, Italie, January 2011.

[40] Adrien Couetoux, Mario Milone, Matyas Brendel, Hassen Doghmen, Michèle Sebag, and Olivier Teytaud. Continuous Rapid Action Value Estimates. In Chun-Nan Hsu and Wee Sun Lee, editors, *The 3rd Asian Conference on Machine Learning (ACML2011)*, volume 20 of *Workshop and Conference Proceedings*, pages 19–31, Taoyuan, Taïwan, Province De Chine, 2011. JMLR.

[41] Adrien Couetoux, Mario Milone, and Olivier Teytaud. Consistent belief state estimation, with application to mines. In *Proceedings of the TAAI 2011 conference*, page in press, 2011.

[42] Adrien Couetoux, Olivier Teytaud, and Hassen Doghmen. Learning a move-generator for upper confidence trees. In Ruay-Shiung Chang, Lakhmi C. Jain, and Sheng-Lung Peng, editors, *Advances in Intelligent Systems and Applications - Volume 1*, volume 20 of *Smart Innovation, Systems and Technologies*, pages 209–218. Springer Berlin Heidelberg, 2013.

[43] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.

[44] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.

[45] Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.

[46] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] Christos Dimitrakakis and Michail G. Lagoudakis. Rollout sampling approximate policy iteration. *Machine Learning*, 72(3):157–171, 2008.

[48] Peter D. Drake and Steve Uurtamo. Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go. In *Proc. 3rd North Amer. Game-On Conf.*, pages 35–42, Gainesville, Florida, 2007.

[49] Y. Ermoliev, editor. *Numerical techniques for stochastic optimization*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[50] Damien Ernst, Guy bart Stan, Jorge Gonçalves, and Louis Wehenkel. Clinical data based optimal sti strategies for hiv: A reinforcement learning approach. In *in Proc. BENELEARN, 2006*, pages 65–72.

[51] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 259–264. AAAI Press, 2008.

[52] E. Gallestey, A. Stothert, M. Antoine, and S. Morton. Model predictive control and the optimization of power plant load while considering lifetime consumption. *Power Systems, IEEE Transactions on*, 17(1):186–191, 2002.

[53] S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. The parallelization of Monte-Carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203, 2008.

[54] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.

[55] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875, July 2011.

[56] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS-2006, Online trading between exploration and exploitation Workshop*, 2006.

[57] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Rapport de recherche INRIA RR-6062, 2006.

[58] Abraham P. George and Warren B. Powell. Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming. *Mach. Learn.*, 65(1):167–198, October 2006.

[59] Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors. *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009.* AAAI, 2009.

[60] W. R. Gilks. *Markov Chain Monte Carlo in Practice.* Chapman & Hall/CRC, December 1995.

[61] Faustino Gomez, Juergen Schmidhuber, and Risto Miikkulainen. Efficient nonlinear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning*, pages 654–662, Berlin, 2006. Springer.

[62] Michael D. Grigoriadis and Leonid G. Khachiyan. A sublinear-time randomized approximation algorithm for matrix games. *Operations Research Letters*, 18(2):53–58, Sep 1995.

[63] I. Grondman, L. Busoniu, G. A D Lopes, and R. Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1291–1307, 2012.

[64] Lars Grüne. Error estimation and adaptive discretization for the discrete stochastic hamilton–jacobi–bellman equation. *Numerische Mathematik*, 99(1):85–112, 2004.

[65] C. Hartland, S. Gelly, N. Baskiotis, O. Teytaud, and M. Sebag. Multi-armed bandits, dynamic environments and meta-bandits. In *NIPS Workshop "online trading of exploration and exploitation*, 2006.

[66] Thomas Gordon Hauk. Search in trees with chance nodes, 2004.

[67] Nicholas Hay and Stuart J. Russell. Metareasoning for monte carlo tree search. Technical Report UCB/EECS-2011-119, EECS Department, University of California, Berkeley, Nov 2011.

[68] International Energy Agency (IEA). Medium-term renewable energy market report 2013 – market trends and projections to 2018. 2013.

[69] V.I. Istratescu. *Fixed Point Theory: An Introduction*. Springer, 2002.

[70] Guillaume M. J-B, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud, and Mark H. M. Winands. Meta Monte-Carlo Tree Search for Automatic Opening Book Generation. pages 7–12.

[71] Hideki Kato and Ikuo Takeuchi. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, November 2008.

[72] Robert D. Kleinberg. Nearly tight bounds for the continuum-armed bandit problem. In *NIPS*, 2004.

[73] Julien Kloetzer. Monte-carlo opening books for amazons. In H.Jaap Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, volume 6515 of *Lecture Notes in Computer Science*, pages 124–135. Springer Berlin Heidelberg, 2011.

[74] L Kocsis and Cs Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[75] L. Kocsis, Cs. Szepesvári, and J. Willemson. Improved monte-carlo search. working paper, 2006.

[76] J. Zico Kolter and Andrew Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 521–528, New York, NY, USA, 2009. ACM.

[77] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6:4–22, 1985.

[78] T.L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[79] Marc Lanctot, Abdallah Saffidine, Joel Veness, Christopher Archibald, and Mark H. M. Winands. Monte carlo *-minimax search. *CoRR*, abs/1304.6057, 2013.

[80] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[81] M. Legendre, K. Hollard, O. Buffet, and A. Dutech. Minesweeper: Where to probe? Technical Report RR-8041, INRIA, 2012.

[82] Francis Maes, Damien Ernst, and Louis Wehenkel. Meta-learning of exploration/exploitation strategies: The multi-armed bandit case. *CoRR*, abs/1207.5208, 2012.

[83] Shie Mannor, Reuven Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *In International Conference on Machine Learning*, pages 512–519. Morgan Kaufmann, 2003.

[84] Christopher R. Mansley, Ari Weinstein, and Michael L. Littman. Sample-based planning for continuous action markov decision processes. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *ICAPS*. AAAI, 2011.

[85] Peter Marbach and John N. Tsitsiklis. Approximate gradient methods in policy-space optimization of markov reward processes. *Discrete Event Dynamic Systems*, 13(1-2):111–148, January 2003.

[86] Eric Martinot, Carmen Dienst, and Liu Weiliang. Renewable energy futures: Targets, scenarios, and pathways. *ANNUAL REVIEW OF ENVIRONMENT AND RESOURCES Book Series: Annual Review of Environment and Resources*, 32:205–239, 2007.

[87] P. Massé. *Les Réserves et la Régulation de l'Avenir dans la vie Economique*. Herman, 1946.

[88] Silja Meyer-Nieberg and Hans-Georg Beyer. Self-adaptation in evolutionary algorithms. In Fernando G. Lobo, Claudio F. Lima, and Zbigniew Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, 2007.

[89] D. Michie. Game-playing and game-learning automata. *Advances in Programming and Non-numerical Computation*, pages 183—-196, 1966.

[90] Rémi Munos. Policy gradient in continuous time. *Journal of Machine Learning Research*, 7:771–791, 2006.

[91] Rémi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Mach. Learn.*, 49(2-3):291–323, November 2002.

[92] A. Nedic and D. P. Bertsekas. Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems*, 13(1-2):79–110, 2003.

[93] M. V. F. Pereira and L. M. V. G. Pinto. Stochastic optimization of a multireservoir hydroelectric system: A decomposition approach. *Water Resources Research*, 21(6):779–792, 1985.

[94] M. V. F. Pereira and L. M. V. G. Pinto. Multi-stage stochastic optimization applied to energy planning. *Math. Program.*, 52(2):359–375, October 1991.

[95] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7–9):1180 – 1190, 2008.

[96] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[97] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.

[98] S. Joe Qin and Thomas A. Badgwell. A survey of industrial model predictive control technology, 2003.

[99] Agrawal R. Sample mean based index policies with o(logn) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 1995.

[100] Martin Riedmiller and et al. Evaluation of policy gradient methods and variants on the cart-pole benchmark. *IN: ADPRL*, 2007.

[101] Arpad Rimmel and Fabien Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In *Evostar*, Istanbul Turquie.

[102] Arpad Rimmel, Fabien Teytaud, and Olivier Teytaud. Biasing Monte-Carlo Simulations through RAVE Values. In *The International Conference on Computers and Games 2010*, Kanazawa Japon, 05 2010.

[103] P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, 2009.

[104] Philippe Rolet, Michele Sebag, and Olivier Teytaud. Optimal robust expensive optimization is tractable. In *Gecco 2009*, page 8 pages, Montréal Canada, 2009. ACM.

[105] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[106] Frederik Christiaan Schadd. Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis. 2010.

[107] Michèle Sebag and Olivier Teytaud. Combining Myopic Optimization and Tree Search: Application to MineSweeper. In *LION6, Learning and Intelligent Optimization*, pages in press (14 pages, long paper), Paris, France, 2012.

[108] Michèle Sebag and Olivier Teytaud. Combining Myopic Optimization and Tree Search: Application to MineSweeper. In Youssef Hamadi and Marc Schoenauer, editors, *LION6, Learning and Intelligent Optimization*, volume 7219 of *LNCS*, pages 222–236, Paris, France, 2012. Proc. LION 6, Sringer Verlag.

[109] David Silver and Gerald Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 945–952, New York, NY, USA, 2009. ACM.

[110] Satinder Singh, Tommi Jaakkola, MichaelL. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[111] T. G. Siqueira, M. Zambelli, M. Cicogna, M. Andrade, and S. Soares. Stochastic dynamic programming for long term hydrothermal scheduling considering different streamflow models. In *Probabilistic Methods Applied to Power Systems, 2006. PMAPS 2006. International Conference on*, pages 1–6, 2006.

[112] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction.* MIT Press., Cambridge, MA, 1998.

[113] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

[114] Fabien Teytaud and Olivier Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Spain, 2009.

[115] Fabien Teytaud and Olivier Teytaud. On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. In *IEEE Conference on Computational Intelligence and Games*, Copenhagen, Denmark, August 2010.

[116] Olivier Teytaud. Including ontologies in monte-carlo tree search and applications - an open source platform, 2008.

[117] B. Tuffin. On the use of low discrepancy sequences in monte carlo methods. Technical Report Technical Report 1060, I.R.I.S.A., 1996.

[118] A. Waldock and B. Carse. Fuzzy q-learning with an adaptive representation. In *Fuzzy Systems, 2008. FUZZ-IEEE 2008. (IEEE World Congress on Computational Intelligence). IEEE International Conference on*, pages 720–725, 2008.

[119] Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21, 2008.

[120] Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182, 2007.

[121] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[122] Ari Weinstein and Michael L. Littman. Bandit-based planning and learning in continuous-action markov decision processes. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *ICAPS*. AAAI, 2012.

[123] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, May 2006.

[124] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.

[125] Shi-Jim Yen, Shih-Yuan Chiu, and I-Chen Wu. Modark wins chinese dark chess tournament. *ICGA Journal*, 33(4):230–231, 2010.

[126] Huizhen Yu and Dimitri P. Bertsekas. Basis function adaptation methods for cost approximation in mdp," to appear. In *in the proceedings of 2009 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL)*, 2009.

Index

Action variables, 18

Benders cuts, 39

Certainty Equivalent Control, 41

Decision epochs, 17
Decision Tree structure, 43

Exploration/Exploitation, 47

Hydrothermal Scheduling, 11

multi-armed bandits, 47

Random action sampler, 42

State variables, 18