NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

THE DYNAMIC DEFENSE OF NETWORK AS POMDP AND THE DESPOT POMDP SOLVER

Nikolaos P. Karaiskakis

Supervisor: **Nicholas Kalouptsidis,** Professor

ATHENS

OCTOBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Η ΑΥΤΟΜΑΤΟΠΟΙΗΜΕΝΗ ΑΜΥΝΑ ΔΙΚΤΥΟΥ ΩΣ POMDP ΚΑΙ Ο ΑΛΓΟΡΙΘΜΟΣ DESPOT ΓΙΑ ΤΗΝ ΕΠΙΛΥΣΗ POMDP

Νικόλαος Π. Καραϊσκάκης

Επιβλέπων: **Νικόλαος Καλουπτσίδης,** Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2020

BSc THESIS

The Dynamic Defense of Network as POMDP and the DESPOT POMDP Solver

Nikolaos P. Karaiskakis
S.N.: 1115201400062

Supervisor: **Nicholas Kalouptsidis,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Η Αυτοματοποιημένη Άμυνα Δικτύου ως POMDP και ο Αλγόριθμος DESPOT για την επίλυση POMDP

Νικόλαος Π. Καραϊσκάκης

Α.Μ.: 1115 2014 00 062

Επιβλέπων: **Νικόλαος Καλουπτσίδης,** Καθηγητής

ABSTRACT

In recent years, artificial intelligence becomes all the more significant for our lives with many applications most of us would not even imagine. Representing the real world demands sophisticated models, which we “feed” to agents to see how they will respond. This is where Markov Decision Processes (MDPs) and Partially Observed Markov Decision Processes (POMDPs) shine. POMDPs provide us with a general framework to depict many different kinds of problems. The capabilities seem endless; from agents that play games optimally to driverless cars. One of these problems that is becoming more and more relevant today is the dynamic defense of a cyber network, which basically means a network that protects itself from intruders in real time by trying to predict their moves and stop them from progressing further into the network and reaching vital points. The development of such a defense system is complicated, since the attackers do not use simplistic methods, but instead rely on a complex sequence of exploits, combining many vulnerabilities. The POMDP model can provide a quite realistic representation of this problem. However, as with most demanding problems modeled as such, it is difficult to solve them efficiently due to the complicated structure of the POMDP model itself. Researchers focus on creating sufficient algorithms that can tackle these problems in realistic situations.

We will begin with introducing the basic information needed to understand the MDP model and then we continue with the POMDP model which extends the idea to more realistic applications. Then, we can present the formulation of the dynamic defense problem as POMDP and after that we take a look into the DESPOT POMDP solver, which is one of the best algorithms to scale up and cope with such complicated problems.

SUBJECT AREA: Artificial Intelligence, Decision making under uncertainty

KEYWORDS: reinforcement learning, MDP, POMDP, dependency graph, dynamic defense of network, POMDP solvers

ΠΕΡΙΛΗΨΗ

Όλοι ακούμε για την Τεχνητή Νοημοσύνη που τα τελευταία χρόνια αποτελεί όλο και μεγαλύτερο κομμάτι της ζωής μας με εφαρμογές που οι περισσότεροι δε θα φανταζόμασταν ποτέ. Η αναπαράσταση του πραγματικού κόσμου απαιτεί πολύπλοκα μοντέλα που να μπορούμε να δώσουμε σε πράκτορες και να δούμε πώς θα ενεργήσουν. Οι Μαρκοβιανές Διαδικασίες Αποφάσεων (MDP) και κυρίως οι Μερικώς Παρατηρούμενες Μαρκοβιανές Διαδικασίες Αποφάσεων (POMDP) αφορούν τη λήψη αποφάσεων υπό αβεβαιότητα και βοηθούν ιδιαίτερα στην πιστή αναπαράσταση ενός περιβάλλοντος. Οι δυνατότητες φαίνονται ατελείωτες, καθώς οι εφαρμογές κυμαίνονται από «έξυπνους» παίκτες παιγνίων μέχρι αυτοματοποιημένα συστήματα οδήγησης. Ένα τέτοιο πρόβλημα που κεντρίζει συνεχώς το ενδιαφέρον είναι η αυτοματοποιημένη άμυνα ενός δικτύου, δηλαδή ένα δίκτυο που προστατεύεται μόνο του από επίδοξους εισβολείς, προβλέποντας τις κινήσεις τους και παίρνοντας τα κατάλληλα μέτρα ώστε να τους αποτρέψει από το να φτάσουν σε ζωτικά σημεία του δικτύου. Οι επιτηθέμενοι δεν κάνουν απλές ενέργειες, αλλά χρησιμοποιούν πολύπλοκες τακτικές συνδυάζοντας πολλά τρωτά σημεία του δικτύου κι έτσι η ανάπτυξη ενός τέτοιου συστήματος άμυνας καθίσταται αρκετά δύσκολη. Αν και μπορούμε να αναπαραστίσουμε το πρόβλημα αρκετά πιστά σαν POMDP, υπάρχει το ζήτημα της γρήγορης επίλυσης, καθώς το POMDP μοντέλο είναι ήδη περιπλεγμένο αυτό καθ'αυτό. Οι ερευνητές, λοιπόν, εστιάζουν την προσοχή τους στην ανάπτυξη γρήγορων αλγορίθμων που να μπορούν να λύνουν αυτά τα προβλήματα σε ρεαλιστικές καταστάσεις.
Αρχικά, θα εισάγουμε τις βασικές έννοιες και πληροφορίες προκειμένου να γίνει κατανοητό το MDP μοντέλο και θα συνεχίσουμε με το POMDP που επεκτείνει το προηγούμενο, κάνοντάς το ρεαλιστικά εφαρμόσιμο. Έπειτα, γίνεται η παρουσίαση του προβλήματος της αυτοματοποιημένης άμυνας σαν POMDP και καταλήγουμε στον αλγόριθμο DESPOT, που είναι από τους καλύτερους που μπορούν να ανταπεξέλθουν σε POMDP προβλήματα τέτοιας κλίμακας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη, Λήψη αποφάσεων υπό αβεβαιότητα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ενισχυτική μάθηση, MDP, POMDP, γραφήματα εξάρτησης, αυτοματοποιημένη άμυνα δικτύου, αλγόριθμοι που επιλύουν POMDP

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to especially thank my supervisor, Mr. Nicholas Kalouptsidis, for his guidance and patience. I would also like to thank his group for the cooperation and assistance. Finally, I would like to thank my family and friends who support me in all my endeavours.

CONTENTS

PREFACE....11

1. INTRODUCTION....13

1.1 Markov Decision Processes....13

1.2 Infinite Horizon Discounted Cost MDP....15

1.2.1 Objective and Dynamic Programming Equation....15

1.2.2 Numerical Methods....15

1.3 Infinite Horizon Average Cost MDP....17

1.3.1 Numerical Methods....18

2. PARTIALLY OBSERVED MARKOV DECISIN PROCESSES....20

2.1 Finite horizon POMDP and Objective....20

2.2 Belief State Formulation of POMDP....21

2.3 Stochastic Dynamic Programming for POMDP....22

2.4 Discounted Infinite Horizon POMDP....23

2.5 Classes of POMDP Algorithms....24

2.5.1 Exact Algorithms: Incremental Pruning....24

2.5.2 Point-Based Value Iteration Algorithms....25

2.5.3 Online POMDP solvers....26

3. DYNAMIC DEFENSE OF CYBER NETWORKS....29

3.1 POMDP Formulation of the Dynamic Defense Problem....29

3.1.1 Dependency Graphs....29

3.1.2 Belief Formulation based on History....30

3.2 Defender Problem....31

4. DESPOT....33

4.1 Determinized Sparse Partially Observable Tree Structure....33

4.2 Dynamic Programming....34

4.3 Anytime Heuristic Search....36
4.3.1 Forward Exploration....36
4.3.2 Termination of Exploration....37
4.3.3 Backup....37
4.3.4 Complexity....38

4.4 DESPOT-alpha....38

4.5 Hyp-DESPOT....40

TABLE OF TERMINOLOGY....42

ABBREVIATIONS - ACRONYMS....43

REFERENCES....44

LIST OF FIGURES

Figure 2.1: Comparison between offline and online approaches…….....27

Figure 3.1: Example of a dependency graph with 12 security conditions (SCs) and 13 exploits……30

Figure 1.2: Standard Belief Tree and DESPOT…….33

Figure 1.3: DESPOT search tree for small and large observation spaces……....39

Figure 1.4: DESPOT-α search tree……..…….39

PREFACE

This thesis gave me the opportunity to work with great scientists and learn a lot. I occupied myself with some very interesting topics in the field of artificial intelligence and mathematics and got a glimpse of the future of IT technologies. I am really grateful I had the chance to do so as an undergraduate student.

1. INTRODUCTION

Markov Models are widely used in artificial intelligence applications. We will analyze particularly the Markov Decision Process (MDP) and Partially Observed Markov Decision Process (POMDP) models. Both are used to tackle problems concerning an agent who tries to achieve a goal in an environment under uncertainty. The POMDP model extends the idea of the MDP by adding the element of partial observability. In other words, this is the case where the agent cannot "see" the exact state of the environment. This feature allows us to represent more complicated and realistic environments and thus formulate the equivalent problems.

However, there is an issue with these problems concerning scalability. Most realistic environments are characterized by large state spaces as well as observation spaces. This means that the algorithms used to examine the agent's possible actions and take decisions can be non-efficient. Researchers have created some complicated algorithms that can scale up for large size POMDPs and this appears to be an area with a lot more to discover.

We will begin with analyzing some basic information on the MDP and POMDP models. Then, we present the POMDP formulation for the problem of the dynamic (or automated) defense of a cyber network. Finally, we present the DESPOT algorithm, which is a sophisticated POMDP-solver that belongs to the state-of-the-art family.

This chapter continues with some basic definitions and information on Markov Decision Processes, which are the basis for the rest.

1.1 Markov Decision Processes

The finite state MDP model consists of the following ingredients:

1. $\mathcal{X} = \{1, 2, ..., \mathrm{X}\}$ denotes the state space and $x_k \in \mathcal{X}$ denotes the state of the controlled Markov chain at time $\mathrm{k} = 0, 1, ..., \mathrm{N}$.
2. $\mathcal{U} = \{1, 2, ..., \mathrm{U}\}$ denotes the action space. The elements $u \in \mathcal{U}$ are called actions. In particular, $u_k \in \mathcal{U}$ denotes the action chosen at time k.
3. For each action $\mathrm{u} \in \mathcal{U}$ and time $\mathrm{k} \in \{0, ..., \mathrm{N}-1\}$, $\mathrm{P(u, k)}$ denotes an $\mathrm{X} \times \mathrm{X}$ transition probability matrix with elements

$$\mathrm{P_{ij}(u, k) = P(x_{k+1} = j | x_k = i, u_k = u), i, j \in \mathcal{X}}.$$

4. For each state $i \in X$, action $u \in U$ and time $\mathrm{k} \in \{0, ..., \mathrm{N}-1\}$, $\mathrm{c(i, u, k)}$ denotes the one-stage cost incurred by the decision-maker (controller).
5. At time N, for each state $\mathrm{i} \in \mathcal{X}$, $c_N(i)$ denotes the terminal cost.

This definition concerns the environment of the agent. We now need to specify an objective function or a way to enable the decision-maker to do the work of taking the best actions. Assuming a problem is modeled with a finite horizon, then the objective of the decision-maker is:

$$J_\pi(x) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{N-1} c(x_k, \pi_k(h_k), k) + c_N(x_N) | x_0 = x \right\} \qquad (1.1)$$

which is the expected cumulative cost incurred by using policy π up to time k. Here, $\mathbb{E}_\pi$ denotes expectation with respect to the probability distribution induced by $h_k = \{x_0, u_0, x_1, u_1, \ldots, x_{k-1}, u_{k-1}, x_k\}$.

The decision-maker determines the optimal policy in the following way:

$$\pi^* = \arg\min_{\pi} J_\pi(x). \qquad (1.2)$$

The meaning of these equations is that the decision-maker searches for the policy sequence π that minimizes the expected cumulative cost (1.1) for every initial state x. If $\mathcal{X}$ and $\mathcal{U}$ are finite, a policy with minimum cumulative cost always exists. The policy with the smallest expected cumulative cost amongst all policies for every initial state is called the optimal policy and is denoted as π^*. Obviously, the optimal policy sequence π^* may not be unique. (For example, if the costs $c(x, u)$ are identical for all x and u, then all policies are optimal.) [1].

Classes of policies

To solve the MDP (1.2) for an optimal policy π^*, we need to examine the space of policies more carefully. We distinguish three types of policies:

1. *General policies*: The most general class of policies $\pi = (\pi_0, \pi_1, \ldots, \pi_{N-1})$ are randomized history dependent. That is, at each time *k*, action u_k is chosen according to probability distribution $\pi_k(h_k)$ (More on h_k in [1] section 6.4). So u_k is a probabilistic function of h_k).
2. *Randomized Markovian policies*: Action u_k is chosen according to probability distribution π_k (x_k). That is, u_k is a probabilistic function of state x_k only.
3. *Deterministic Markovian policies*: Action u_k is chosen based on a deterministic mapping from the state space $\mathcal{X}$ to action space $\mathcal{U}$.

Bellman's stochastic dynamic programming algorithm

Consider the MDP problem (1.1) with objective (1.2). The optimal policy π^* is obtained via Bellman's stochastic dynamic programming algorithm [5].

Theorem 1.1 (Bellman's dynamic programming algorithm) *The optimal policy* $\pi^* = (\pi_0, \pi_1, \ldots, \pi_{N-1})$ *for the finite horizon MDP can be obtained as the solution of the following backward recursion:*

Initialize $J_N(i) = c(i,N)$. *Then for* $k = N-1, \ldots, 0$ *evaluate*

$$J_k(i) = \min_{u \in U} \left\{ c(i,u,k) + \sum_j P_{ij}(u,k)\, J_{k+1}(j) \right\}$$

$$\pi_k^*(i) = \arg\min_{u \in U} \left\{ c(i,u,k) + \sum_j P_{ij}(u,k)\, J_{k+1}(j) \right\} \qquad (1.3)$$

For any initial state $i \in \{1, \ldots, X\}$, the expected cumulative cost of the optimal policy π^*, namely $J_{\pi^*}(i)$ in (1.2) is obtained as $J_0(i)$ from (1.3) .

1.2 Infinite Horizon Discounted Cost MDP

In this section we consider the case where the horizon length $\mathrm{N} \to \infty$. Also, the transition probabilities and costs are assumed not to be explicit functions of time, and there is no terminal cost. The infinite horizon discounted MDP model considered here is the 5-tuple:

$$(\mathcal{X}, \mathcal{U}, \mathrm{P_{ij}(u)}, \mathrm{c(i,u)}, \rho), \mathrm{i,j} \in \mathcal{X}, \mathrm{u} \in \mathcal{U}.$$

where $\rho \in [0, 1)$ is a discount factor. The discount factor ρ weights the costs in the following manner: the cost incurred by the decision-maker at time k is $\rho^k c(x_k, u_k)$. Therefore, the first few decisions are much more important than subsequent decisions.

1.2.1 Objective and Dynamic Programming Equation

The aim is to determine the optimal policy $\pi^* = \arg\min_{\pi} J_\pi(i)$ where $J_\pi(i)$ denotes the infinite horizon discounted cumulative cost

$$J_\pi(i) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \rho^k c(x_k, u_k) | x_0 = x \right\} \qquad (1.4)$$

Here $\pi = (\pi_0, \pi_1, \ldots)$ is a sequence of policies where π_k at time k maps $h_k = \{x_0, u_0, x_1, u_1, \ldots, x_{k-1}, u_{k-1}, x_k\}$ to action u_k.

Theorem 1.2 *Consider an infinite horizon discounted cost Markov decision process with discount factor $\rho \in [0, 1)$. Then*

1. For any initial state i, the optimal cumulative cost $\mathrm{J_{\pi^*}(i)}$ is attained by the value function $V(i)$ which satisfies Bellman's equation (1.5).
2. For any initial state i, the optimal cumulative cost $\mathrm{J_{\pi^*}(i)}$ achieved by the stationary deterministic Markovian policy π^* which satisfies Bellman's equation (1.5).
3. The value function V is the unique solution to Bellman's equation (1.5). *(The optimal policy may not be unique.)*

$$V_k(i) = \min_{u \in \mathrm{U}} Q_k(i,u), \quad \pi^* = arg \min_{u \in \mathrm{U}} Q_k(i,u),$$

$$Q_k(i,u) = c(i,u) + \rho \sum_j P_{ij}(u) V(j) \qquad (1.5)$$

1.2.2 Numerical methods

Now we can take a look at three classical methods for solving infinite horizon discounted cost Markov decision processes.

Value iteration algorithm

The value iteration algorithm is a successive approximation algorithm to compute the value function V of Bellman's equation. It proceeds as follows. Choose the number of iterations N (typically large). Initialize $V_0(i) = 0$. Then for iterations $\mathrm{k} = 1, 2, \ldots, \mathrm{N}$, compute

$$V_k(i) = \min_{u \in \mathrm{U}} Q_k(i,u), \;\; \pi_k^* = arg \min_{u \in \mathrm{U}} Q_k(i,u),$$

$$Q_k(i,u) = c(i,u) + \rho \sum_j P_{ij}(u) V_{k-1}(j).$$

Then use the stationary policy π_N^* at each time instant in the real-time controller. The value iteration algorithm is identical to the finite horizon MDP dynamic programming algorithm, except that in the controller implementation the stationary policy π_N^* is used at each time.

Policy iteration algorithm

This is an iterative algorithm that computes an improved policy at each iteration compared to that of the previous iteration. Since for a U-action, X-state Markov decision process, there are a finite number X^U possible stationary policies, the policy iteration algorithm converges to the optimal policy in a finite number of iterations.

The policy iteration algorithm proceeds as follows.
Assume a stationary policy π_{n-1} and its associated cumulative cost $J_{\pi_{n-1}}$ from iteration $\mathrm{n}-1$ are given. At iteration n:

1. *Policy improvement*: Compute stationary policy π_n as

$$\pi_n(i) = \; arg \min_{u \in \mathrm{U}} \left[c(i,u) + \rho \sum_j P_{ij}(u) J_{\pi_{n-1}}(j) \right],$$

where *i* ∈ $\mathcal{X}$.

2. *Policy evaluation*: Given policy π_n, compute the discounted cumulative cost associated with this policy as

$$J_{\pi_n}(i) = c\big(i, \pi_n(i)\big) + \; \rho \sum_j P_{ij}(\pi_n(i)) J_{\pi_n}(j),$$

where *i* ∈ $\mathcal{X}$.

This is a linear system of equations and can be solved for J_{π_n}. If $J_{\pi_n}(i) < J_{\pi_{n-1}}(i)$ for all i, then set $\mathrm{n} = \mathrm{n} + 1$ and continue. Else stop.

Linear programming

Bellman's equation can be seen as a linear programming optimization problem.
Define $\bar{V}(i)$ such that

$$\bar{V}(i) \le \min_u \left\{ c(i,u) + \rho \sum_j P_{ij}(u) \, \bar{V}(j) \right\}$$

From the value iteration algorithm we have that $\bar{V}(i) \le V(i)$ where V is the value function from Bellman's equation. This means that the value function V is the largest $\bar{V}$ that satisfies the above inequality. Then V is the solution of the optimization problem

$$\max_{\bar{V}} \sum_{\iota} \alpha_{\iota} \bar{V}(i) \quad subject\ to\ \bar{V}(i) \leq \min_{u} \left\{ c(i,u) + \rho \sum_{j} P_{ij}(u)\, \bar{V}(j) \right\},$$

where $i \in \mathcal{X}$ and α_{ι} are arbitrary nonnegative scalars.

V is the solution of the linear programming problem:

$$\max_{\bar{V}} \sum_{\iota} \alpha_{\iota} \bar{V}(i)$$

$$subject\ to\ \bar{V}(i) \leq\ c(i,u) + \rho \sum_{j} P_{ij}(u)\, \bar{V}(j),$$

where $i \in \mathcal{X}$ and $u \in \mathcal{U}$.

1.3 Infinite Horizon Average Cost MDP

As in the discounted cost case, by a stationary policy we mean the sequence of policies $\boldsymbol{\pi} = (\pi,\pi,\pi, ...)$ and for notational convenience we denote $\boldsymbol{\pi}$ by π. For any stationary policy π, let $\mathbb{E}_{\pi}$ denote the corresponding expectation and define the infinite horizon average cost as

$$J_{\pi}(x_0) = \lim_{N\to\infty} \frac{1}{N+1} \mathbb{E}_{\pi} \left\{ \sum_{k=0}^{N} c(x_k, u_k) | x_0 \right\}. \qquad (1.6)$$

We say that a stationary policy π^* is optimal if $J_{\pi}^*(x_0) \leq J_{\pi}(x_0)$ for all states $x_0 \in \mathcal{X}$.

Unlike discounted infinite horizon problems, average cost problems have more issues. The existence of an optimal stationary policy depends on the structure of the transition matrices $\mathrm{P(u)}$. We will not go into depth with that. However, we introduce the term "unichain" to present the following condition, which is sufficient for our needs, since our description considers finite state Markov chains.

Definition of Unichain: An MDP is said to be unichain if every deterministic stationary policy yields a Markov chain with a single recurrent class.

It turns out that if an average cost MDP is unichain, then an optimal stationary policy π^* always exists. Checking the unichain condition can be computationally intractable though since there are X^U possible stationary policies; for each such policy one needs to check that the resulting transition matrix $(\mathrm{P_{ij}}\ (\pi(\mathrm{i}))\ , \mathrm{i}, \mathrm{j} \in \mathcal{X})$ forms a recurrent class.
If all the transition probabilities $P_{ij}\ (u)$ are strictly positive for each π, then the unichain condition trivially holds [1]. The next theorem is equivalent to Bellman's dynamic programming.

Theorem 1.3 *Consider a finite-state finite-action unichain average cost Markov decision process. Then the optimal policy π^* satisfies*

$$g + V(i) = \min_u \left\{ c(i,u) + \sum_j P_{ij}(u) V(j) \right\} \qquad (1.7)$$

$$\pi^*(\mathrm{i}) = \arg\min_u \left\{ c(i,u) + \sum_j P_{ij}(u) V(j) \right\}. \qquad (1.8)$$

Here π^* is a stationary nonrandomized policy. Also, $\mathrm{g} = J_{\pi^*}(x_0)$ denotes the average cost achieved by the optimal policy π^* and is independent of the initial state x_0. Furthermore, g is unique while if $V(i)$, $\mathrm{i} \in \mathcal{X}$ satisfies (1.7) then so does $\mathrm{V(i)} + \mathrm{K}$ for any constant K.

1.3.1 Numerical methods

Relative value iteration algorithm

Since by Theorem 1.3 any constant added to the value function also satisfies (1.7) , it is is convenient to rewrite (1.7) in the following form that is anchored at state 1. Define $\tilde{V}(i) = V(i) - V(1)$. So obviously $V(1) = 0$. Then (1.7) can be rewritten relative to state 1 as:

$$g + \tilde{V}(i) = \min_u \left\{ c(i,u) + \sum_{j>1} P_{ij}(u) \tilde{V}(j) \right\}, i > 1,$$

$$g + V(1) = \min_u \left\{ c(1,u) + \sum_j P_{1j}(u) \tilde{V}(j) \right\} + V(1)$$

or

$$g = \min_u \left\{ c(1,u) + \sum_j P_{1j}(u) \tilde{V}(j) \right\}. \qquad (1.9)$$

In analogy to the value iteration algorithm described for the discounted cost MDP (section 1.2.2), (1.9) yields the following *relative value iteration algorithm* that operates for $\mathrm{k} = 1, 2, \ldots$

$$\tilde{V}_k(i) = \min_u \left\{ c(i,u) + \sum_{j>1} P_{ij}(u) \tilde{V}_{k-1}(j) \right\} - g_k, i > 1,$$

$$g_k = \min_u \left\{ c(1,u) + \sum_j P_{1j}(u) \tilde{V}_{k-1}(j) \right\}. \qquad (1.10)$$

Linear programming

An average cost MDP can be formulated as a linear programming problem and solved using interior point methods or simplex algorithms [1]. From Theorem 1.3, solving Bellman's equation is equivalent to the following linear program

Maximize g
subject to $g + V(1) \leq c(i,u) + \sum_{j \in X} P_{ij}(u)V(j)$. (1.11)

The dual of the above problem can be formulated as the following linear programming problem. Let $\mathrm{p(x, u)} = \mathbb{P}\ (x_k = \mathrm{x},\ u_k = \mathrm{u})$, $\mathrm{x} \in \mathcal{X}$, $\mathrm{u} \in \mathcal{U}$, denote the joint action state probabilities.

Theorem 1.4 *Consider a finite-state finite-action unichain average cost Markov decision process. Then the optimal policy* π^* *is*

$$\pi^*(x) = u \; with \; probability \; \theta_{x,u} = \frac{p^*(x,u)}{\sum_{i \in \mathrm{X}} p^*(i,u)}$$

The $X \times U$ *elements of* p^* *are the solution of the linear programming problem:*

$$minimize \sum_{i \in \mathrm{X}} \sum_{u \in \mathrm{U}} c(i,u)p(i,u) \quad with \; respect \; to \; p$$

$$subject \; to \; p(i,u) \geq 0, \; i \in \mathcal{X}, \; u \in \mathcal{U}$$

$$\sum_i \sum_u p(i,u) = 1$$

$$\sum_u p(i,u) = \sum_i \sum_u p(i,u)P_{ij}(u), \;\; j \in \{1, \ldots, X).$$

2. PARTIALLY OBSERVED MARKOV DECISION PROCESSES

The POMDP model extends the MDP by adding the feature of *partial observability.* A model where the agent does not fully observe the environment provides a framework to formulate more realistic problems.

2.1 Finite Horizon POMDP and Objective

A POMDP model with finite horizon N is a 7-tuple $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \mathrm{P(u)}, \mathrm{O(u)}, \mathrm{c(u)}, c_N)$.
1. $\mathcal{X} = \{1, 2, ..., \mathrm{X}\}$ denotes the state space and $x_k \in \mathcal{X}$ denotes the state of a controlled Markov chain at time $\mathrm{k} = 0, 1, ..., \mathrm{N}$.
2. $\mathcal{U} = \{1, 2, ..., \mathrm{U}\}$ denotes the action space with $u_k \in \mathcal{U}$ denoting the action chosen at time k by the controller.
3. $\mathcal{Y}$ denotes the observation space which can either be finite or a subset of R. $y_k \in \mathcal{Y}$ denotes the observation recorded at each time $\mathrm{k} \in \{1, 2, ..., \mathrm{N}\}$.
4. For each action $u \in \mathcal{U}$, $P(u)$ denotes a $\mathrm{X} \times \mathrm{X}$ transition probability matrix with elements $P_{ij}(u) = \mathbb{P}(x_{k+1} = j | x_k = i, u_k = u)$, $\mathrm{i, j} \in \mathcal{X}$. (2.1)
5. For each action $u \in \mathcal{U}$, $\mathrm{O(u)}$ denotes the observation distribution with
$O_{iy}(u) = \mathbb{P}(y_{k+1} = y | x_{k+1} = i, u_k = u)$, $i \in \mathcal{X}$, $\mathrm{y} \in \mathcal{Y}$. (2.2)
6. For state x_k and action u_k, the decision-maker incurs a cost $\mathrm{c}(\mathrm{x_k}, \mathrm{u_k})$.
7. Finally, at terminal time N, a terminal cost $c_N(x_N)$ is incurred.

The decision-maker does not observe the state x_k, but only observes noisy observations y_k that depend on the action and the state specified by the probabilities in (2.2).

Objective

We have to specify a performance criterion or objective function for the decision-maker. For the finite horizon, the objective is:

$$J_\pi(b_0) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{N-1} c(x_k, u_k) + c_N(x_N) | b_0 \right\} \qquad (2.3)$$

which is the expected cumulative cost incurred by the decision-maker when using policy π up to time N given the initial distribution b_0 of the Markov chain.
Here, $\mathbb{E}_\pi$ denotes expectation with respect to the joint probability distribution of $(x_0, y_0, x_1, y_1, ..., x_{N-1}, y_{N-1}, x_N, y_N)$. The goal of the decision-maker is to determine the optimal policy sequence

$$\pi^* = \mathrm{arg}\min_\pi J_\pi(b_0), \qquad (2.4)$$

for any initial prior b_0 that minimizes the expected cumulative cost. Of course, the optimal policy sequence π^* may not be unique.

Remarks

1. The decision-maker does not observe the state x_k. It only observes noisy observations y_k that depend on the action and the state. Also, the decision-maker knows the cost matrix $c(x, u)$ for all possible states and actions in $\mathcal{X}$, $\mathcal{U}$. But since the decision-maker does not know the state x_k at time k, it does not know the cost accrued at time k or terminal cost. Of course, the decision-maker can estimate the cost by using the noisy observations of the state.
2. The term POMDP is usually used in the case where the observation space $\mathcal{Y}$ is finite. However, we consider both finite and continuous valued observations.
3. The action u_k affects the evolution of the state and observation distribution. In controlled sensing applications such as radars and sensor networks, the action only affects the observation distribution and not the evolution of the target.

2.2 Belief State Formulation of POMDP

This section details a crucial step in the formulation and solution of a POMDP, namely, the belief state formulation. In this formulation, a POMDP is equivalent to a continuous-state MDP with states being the belief states. These belief states are simply the posterior state distributions computed via the HMM filter described in Part 1 of [1]. We then formulate the optimal policy as the solution to Bellman's dynamic programming recursion written in terms of the belief state. The main outcome of this section is the formulation of the POMDP dynamics in terms of the belief state.
Recall from that for a fully observed MDP, the optimal policy is Markovian and the optimal action $u_k = \pi_k^*(x_k)$. In comparison, for a POMDP the optimal action chosen by the decision-maker is in general

$$u_k = \pi_k^*(h_k), \text{ where } h_k = (b_0, u_0, y_1, \ldots, u_{k-1}, y_k). \tag{2.5}$$

Since h_k is increasing in dimension with k, to implement a controller, it is useful to obtain a sufficient statistic that does not grow in dimension. In [1] section 3.5, a HMM filter is used to compute the posterior distribution b_k, which is a sufficient statistic for h_k.
We define the posterior distribution of the Markov chain given h_k as:

$$b_k(i) = \mathbb{P}(x_k = i | h_k),\ i \in \mathcal{X} \quad \text{where } h_k = (b_0, u_0, y_1, \ldots, u_{k-1}, y_k) \tag{2.6}$$

We will call the X-dimensional probability vector $b_k = [b_k(1), \ldots, b_k(X)]'$ as the *belief state* or *information state* at time k.

The main point established below in Theorem 2.1 is that (2.5) is equivalent to

$$u_k = \pi_k^*(b_k), \tag{2.7}$$

In other words, the optimal controller operates on the belief state b_k (HMM filter posterior) to determine the action u_k.
In light of (2.7), let us first define the space where b_k lives in.
The beliefs b_k, $k = 0, 1, \ldots$ defined in (2.6) are X-dimensional probability vectors. Therefore, they lie in the $X-1$ dimensional unit simplex denoted as

$$B(X) = \{b \in \mathrm{R}^X : 1'b = 1, 0 \le b(i) \le 1 \ \text{ for all } i \in \mathcal{X} = \{1, 2, \ldots, X\}\}.$$

$\mathrm{B}(\mathrm{X})$ is called the *belief space*. $\mathcal{B}(2)$ is a one-dimensional simplex (unit line segment), $\mathcal{B}(3)$ is a two-dimensional simplex (equilateral triangle); $\mathcal{B}(4)$ is a tetrahedron, etc. Note that the unit vector states e_1, e_2, …, e_X of the underlying Markov chain $\mathcal{X}$ are the vertices of $\mathrm{B}(\mathrm{X})$.
We now formulate the POMDP objective in terms of the belief state. Consider the objective (2.3). Then

$$\begin{aligned} J_\pi(b_0) &= \mathbb{E}_\pi\left\{\sum_{k=0}^{N-1} c(x_k,u_k) + c_N(x_N)|b_0\right\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=0}^{N-1} \mathbb{E}\{c(x_k,u_k)|h_k\} + \mathbb{E}\{c_N(x_N)|h_N\}|b_0\right\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=0}^{N-1}\sum_{i=1}^{X} c(i,u_k)b_k(i) + \sum_{i=1}^{X} c_N(i)b_N(i)|b_0\right\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=0}^{N-1} c'_{u_k} b_k + c'_N b_N|b_0\right\} \end{aligned} \tag{2.8}$$

In (2.8), the X-dimensional cost vectors $c_u(k)$ and terminal cost vector c_N are defined as:

$$c_u = [c(1,u)\ \ldots c(X,u)]',\ c_N = [c_N(1)\ldots\ c_N(X)]. \tag{2.9}$$

In this way the POMDP has been expressed as a continuous-state (fully observed) MDP. This continuous-state MDP has belief state b_k which lies in unit simplex belief space $\mathrm{B}(\mathrm{X})$. Determining the optimal policy for a POMDP is equivalent to partitioning $\mathrm{B}(\mathrm{X})$ into regions where a particular action $u \in \{1, 2, \ldots, \mathrm{U}\}$ is optimal.

2.3 Stochastic Dynamic Programming for POMDP

In this section, we present how we can use Bellman's dynamic programming for POMDPs according to the formulation we previously saw.

Theorem 2.1 *For a finite horizon POMDP with model* $(\mathcal{X}, \mathcal{U}, \mathcal{Y}, \mathrm{P}(\mathrm{u}), \mathrm{O}(\mathrm{u}), \mathrm{c}(\mathrm{u}), c_N)$:

1. The minimum expected cumulative cost $J_{\pi^*}(b)$ *is achieved by deterministic policies*

$$\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{N-1}^*), \quad where\ u_k = \pi_k^*(b_k).$$

2. The optimal policy $\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{N-1}^*)$ *for a POMDP is the solution of the following Bellman's dynamic programming backward recursion: Initialize* $J_N(b) = c'_N b$ *and then for* $\mathrm{k} = \mathrm{N} - 1, \ldots, 0$

$$J_k(b) = \min_{u \in \mathrm{U}} \left\{ c'_u b + \sum_{y \in \mathrm{Y}} J_{k+1}\big(T(\pi, y, u)\big)\sigma(b, y, u)) \right\}$$

$$\pi_k^*(b) = arg \min_{u \in \mathrm{U}} \left\{ c'_u b + \sum_{y \in Y} J_{k+1}\big(T(\pi, y, u)\big)\sigma(b, y, u)) \right\}$$

(2.10)

The expected cumulative cost $J_{\pi*}(b)$ (2.10) *of the optimal policy* π^* *is given by the value function* $J_0(b)$ *for any initial belief* $\mathrm{b} \in \mathrm{B(X)}$.

Since the belief space $\mathrm{B(X)}$ is uncountable, the above dynamic programming recursion does not translate into practical solution methodologies. $J_k(b)$ needs to be evaluated at each $\mathrm{b} \in \mathrm{B(X)}$, an uncountable set.

2.4 Discounted Infinite Horizon POMDP

So far we have considered finite horizon POMDPs. This section considers infinite horizon discounted cost POMDPs. The discounted POMDP model is a 7-tuple ($\mathcal{X}$, $\mathcal{U}$, $\mathcal{Y}$, $\mathrm{P(u)}$, $\mathrm{O(u)}$, $\mathrm{c(u)}$, ρ) where $\mathrm{P(u)}$, $\mathrm{O(u)}$ and c are no longer explicit functions of time and $\rho \in [0, 1)$ is an economic discount factor. Also, there is no terminal cost c_N.

We define a stationary policy sequence as $\boldsymbol{\pi}$=(π,π,π,...) where π is not an explicit function of time k. For stationary policy $\pi : \mathrm{B(X)} \rightarrow \mathcal{U}$, initial belief $\mathrm{b}_0 \in \mathrm{B(X)}$, discount factor $\rho \in [0, 1)$, define the objective function as the discounted expected cost:

$$J_\pi(b_0) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \rho^k c(x_k, u_k) \right\}, where\ u_k = \pi(b_k)$$

or

$$J_\pi(b_0) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \rho^k c'_\pi(b_k) \right\},$$

(2.11)

where $c_u = [c(1,u), \dots, c(X,u)]', u \in \mathcal{U}$ is the cost vector for each action.

The aim is to compute the optimal stationary policy $\pi^* : \mathrm{B(X)} \rightarrow \mathcal{U}$ such that $\mathrm{J}_{\pi^*}(\mathrm{b}_0) \leq \mathrm{J}_\pi(\mathrm{b}_0)\ \forall\ \mathrm{b}_0 \in \mathrm{B(X)}$.

Theorem 2.2 *Consider an infinite horizon discounted cost POMDP with discount factor* $\rho \in [0, 1)$*. Then with* $\mathrm{b} \in \mathrm{B(X)}$ *denoting the belief state,*

1. The optimal expected cumulative cost is achieved by a stationary deterministic Markovian policy π^**.*

2. The optimal policy $\pi^*(b)$ *and value function* $V(b)$ *satisfy Bellman's dynamic programming equation*

$$\pi^*(b) = arg\min_{u\in \mathrm{U}} Q(b,u), \quad J_{\pi^*}(b_0) = V(b_0) \tag{2.12}$$

$$V(b) = \min_{u\in \mathrm{U}} Q(b,u), \quad Q(b,u) = c'_{\pi}(b_k) + \rho \sum_{y\in Y} V\big(T(b,y,u)\big)\sigma(b,y,u),$$

3. The value function $V(\pi)$ *is continuous and concave in* $\mathrm{b} \in \mathrm{B(X)}$.

For more general theoretical background on MDPs and POMDPs, except for [1], also [2] and [3] provide the reader with much information.

2.5 Classes of POMDP Algorithms

In this section, we present the basic categories of POMDP solvers. First, we will talk about exact algorithms, which are used to solve optimally finite horizon POMDPs. However, these algorithms are not efficient, so we need to take a look into approximating algorithms that search for near-optimal solutions and can scale pretty well even for large scale POMDPs. The most important categories of such solvers are the point-based (PB) algorithms and the online algorithms.

2.5.1 Exact Algorithms: Incremental Pruning

These algorithms solve optimally finite horizon POMDPs. They are based on Sodnik's idea [4], which was the first exact algorithm for POMDPs of the finite horizon. Exact here means that there is no approximation involved in the dynamic programming algorithm.

According to [1] (section 7.5.1), Bellman's dynamic programming recursion (2.10) can be expressed as the following three steps:

$$Q_k(b,u,y) = \frac{c'_u b}{Y} + J_{k+1}\big(T(b,y,u)\big)\sigma(b\,,y\,,u)$$

$$Q_k(b,u) = \sum_{y\in \mathrm{Y}} Q_k(b,u,y)$$

$$J_k(b) = \min_u Q_k(b,u).$$

(2.13)

Based on the above three steps, the set of vectors Γ_k that form the piecewise linear value function [1], can be constructed as:

$$\Gamma_k(u,y) = \left\{\frac{c_u}{Y} + P(u)O_y(u)\gamma | \gamma \in \Gamma_{k+1}\right\}$$

$$\Gamma_k(u) = \bigoplus \Gamma_k(u,y)$$

$$\Gamma_k = \cup_{u\in \mathrm{U}}\, \Gamma_k(u).$$

(2.14)

Here $\oplus$ denotes the cross-sum operator: given two sets of vectors *A* and *B*, $A \oplus B$ consists of all pairwise additions of vectors from these two sets.

In general, the set Γ_k constructed according to (2.14) may contain superfluous vectors (we call them "inactive vectors" below) that never arise in the value function J_k (*b*).
The incremental pruning algorithm seeks to eliminate useless vectors by pruning Γ_k to maintain a parsimonious set of vectors. Below is the Incremental pruning algorithm:

Algorithm 2.1: Incremental pruning

1: Given set Γ_{k+1} generate Γ_k as follows:
2: Initialize $\Gamma_k\ (u, y), \Gamma_k\ (u), \Gamma_k$ as empty sets
3: for each u $\in\ \mathcal{U}$
4: **for each** y $\in\ \mathcal{Y}$
5: $\Gamma_k(u, y)\ \leftarrow prune\left(\left\{\frac{c_u}{Y} + P(u)O_y(u)\gamma | \gamma \in \Gamma_{k+1}\right\}\right)$
6: $\Gamma_k\ (u) \leftarrow$ prune $(\Gamma_k\ (u) \oplus \Gamma_k\ (u, y))$
7: **end for**
7: $\Gamma_k \leftarrow$ prune $(\Gamma_k \cup\ \Gamma_k\ (u))$
8: **end for**

The value function $J_k(b) = \min_{\gamma \in \Gamma_k} \gamma b'$ with set of vectors Γ_k is piecewise linear and concave.

Suppose there is a vector $\gamma\ \in \Gamma_k$ such that for all $b \in B(X)$, it holds that $\gamma' b \geq \bar{\gamma}' b$ for all vectors $\bar{\gamma} \in \Gamma_k\ - \{\gamma\}$. Then γ dominates every other vector in Γ_k and is never active.

The following linear programming dominance test can be used to identify and prune inactive vectors:

$$\begin{aligned} &\min x \\ &\text{subject to: } (\gamma - \bar{\gamma})' b \geq x, \forall\ \bar{\gamma} \in \Gamma - \{\gamma\} \end{aligned} \qquad (2.15)$$

If the above linear program yields a solution $x \geq 0$ for a vector γ, then γ dominates all other vectors in $\Gamma\ - \{\gamma\}$, which means it is inactive and can be excluded from Γ. This pruning method makes the Incremental Pruning Algorithm more efficient compared to other exact POMDP-solvers.

Other known exact algorithms are Monahan's algorithm and Witness algorithm.

2.5.2 Point-Based Value Iteration Algorithms

Point-based value iteration methods seek to compute an approximation of the value function at special points in the belief space. The main idea is to compute solutions only for those belief states that have been visited by running the POMDP. This urges researchers to develop approximate solution techniques that use a sampled set of belief states on which the POMDP is solved.
Such a method can be inefficient given the exponential growth in value function representation size. When performing value iteration over the belief space, it is crucial to limit the size of the set of vectors representing the value function. The issue here lies in

the decision on which vectors should be removed. Of course, there is a trade-off between avoiding the growth of the set of vectors of the value function, at the cost of compromising the accuracy of the value function.
PB algorithms belong to the offline family, since they construct a policy by dividing the belief space into areas based on the domination relations among the vectors representing the value function. Also, they need an initial estimation of the value function. More details on PB algorithms can be found at [7].

We will now take a look into the Value Iteration algorithm for discounted cost POMDPs. Let $\mathrm{n} = 1, 2, \ldots, \mathrm{N}$ denote iteration number. The value iteration algorithm for a discounted cost POMDP is a successive approximation algorithm for computing the value function $\mathrm{V}(\pi)$ of Bellman's equation (2.12) and proceeds as follows: initialize $\mathrm{V}_0(\mathrm{b}) = 0$. For iterations $\mathrm{n} = 1, 2, \ldots, \mathrm{N}$, evaluate

$$\mathrm{V}_\mathrm{n}(\mathrm{b}) = \min_{u \in \mathrm{U}} Q_n(b,u), \quad \pi_n^*(b) = arg \min_{u \in \mathrm{U}} Q_n(b,u),$$

$$Q_n(b,u) = c'_u\, b + \rho \sum_{\mathrm{y} \in Y} V_{n-1}\big(T(b,y,u)\big)\sigma(b,y,u) \qquad (2.16)$$

Finally, the stationary policy π_n^* is used at each time instant k in the real-time controller. The POMDP value iteration algorithm (2.16) is identical to the finite horizon dynamic programming recursion (2.10). So at each iteration n, $\mathrm{V}_\mathrm{n}(\mathrm{b})$ is piecewise linear and concave in b (by Theorem 2.1). The value iteration algorithm (2.16) generates a sequence of value functions $\{\mathrm{V}_\mathrm{n}\}$ that will converge uniformly (sup-norm metric) as $\mathrm{N} \to \infty$ to $\mathrm{V}(\mathrm{b})$, the optimal value function of Bellman's equation.
As mentioned already, the number of piecewise linear segments that characterize $\mathrm{V}_\mathrm{n}(\mathrm{b})$ can grow exponentially with iteration n. Therefore, we cannot expect great results except for POMDP problems with small state, action and observation spaces.

2.5.3 Online POMDP solvers

An offline POMDP solver returns a policy defining which action to execute in every possible belief state. Given the complexity of the POMDP model, it is obvious that this practice is not sufficient. Exact algorithms can only be useful when it comes to small to mid-size domains, since the policy construction step takes significant time. Online algorithms tackle this issue by computing good local policies at each time step.
Approximating offline algorithms may be used in order to compute upper and lower bounds of the value function. Then, an online algorithm takes into account these bounds so as to search for policies based on the more promising areas of the belief space and save time.
Online approaches try to find a good local policy for the current belief. These approaches tend to be more appropriate for large POMDPs, because they only need to consider belief states that are reachable from the current belief state. This focuses computation on a small set of beliefs and thus reduces running time drastically. In addition, since online planning is done at every step, it is sufficient to calculate only the maximal value for the current belief state. In this setting, the policy construction steps and the execution steps are interleaved with one another as shown in Figure 2.1.
In some cases, online approaches may require a few extra execution steps (and online planning), since the policy is locally constructed and therefore not always optimal.
However, the policy construction time is often substantially shorter.

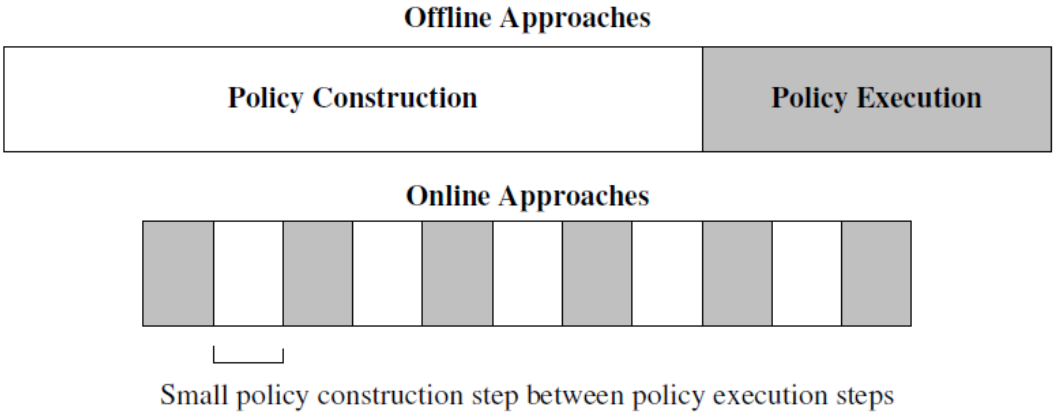

Figure 2.1: Comparison between offline and online approaches.

Consequently, the overall time for the policy construction and execution is normally less for online approaches. In practice, a potential limitation of online planning is when we need to meet short real-time constraints. In such case, the time available to construct the plan is very small compared to offline algorithms.
Online algorithms comprise two basic steps; the planning phase and the execution phase. In the planning phase, the algorithm is given the current belief state and computes the best action to execute in the current belief. This is usually achieved in two steps.
First a tree of reachable belief states from the current belief state is built by looking at several possible sequences of actions and observations that can be taken from the current belief. In this tree, the current belief is the root node and subsequent reachable beliefs are added to the tree as child nodes of their immediate previous belief. Belief nodes are represented using OR-nodes (at which we must choose an action) and actions are included in between each layer of belief nodes using AND-nodes (at which we must consider all possible observations that lead to subsequent beliefs) [8]. Then the value of the current belief is estimated by propagating value estimates up from the fringe nodes, to their ancestors, all the way to the root, according to Bellman's equation.
After the planning phase is done, the execution phase proceeds by executing the best action found for the current belief in the environment, and updating the current belief and tree according to the observation obtained.

Algorithm 2.2 provides an outline on a generic online algorithm's implementation of the planning phase (lines 5-9) and the execution phase (lines 10-13). The algorithm first initializes the tree to contain only the initial belief state (line 2). Then given the current tree, the planning phase of the algorithm proceeds by first selecting the next node to expand (line 6). The Expand function (line 7) constructs the next reachable beliefs under the selected leaf for some pre-determined expansion depth D and evaluates the approximate value function for all newly created nodes. The new approximate value of the expanded node is propagated to its ancestors via the UpdateAncestors function (line 8). The planning phase goes on until some terminating condition is met (either an optimal action is found or planning time has run out).
Then, the algorithm proceeds to the execution phase, where it executes the best action u^* found during planning (line 10) and gets a new observation o from the environment (line 11). Next, the algorithm updates the current belief state and the search tree T according to the most recent action u^* and observation o (lines 12-13).

Algorithm 2.2: Online POMDP-solver

Static: b_c: The current belief state of the agent.
T: An AND-OR tree representing the current search tree.
D: Expansion depth.
L: A lower bound on V^*.
U: An upper bound on V^*.
$b_c \leftarrow b_0$
Initialize T to contain only b_c at the root
while not ExecutionTerminated() **do**
    while not PlanningTerminated() **do**
        $b^* \leftarrow$ ChooseNextNodeToExpand()
        Expand(b^*, D)
        UpdateAncestors(b^*)
    end while
    Execute best action u^* for b_c
    Perceive a new observation o
    $b_c \leftarrow \tau(b_c, u^*, o)$
    Update tree T so that b_c is the new root
end while

3. DYNAMIC DEFENSE OF CYBER NETWORKS

After seeing the basic elements of POMDPs, we can introduce a problem modeled as such. The problem is the protection of a cyber network from intruders in real time. Specifically, given a network, the defender attempts to prevent the attacker from reaching important points, by blocking possible future actions that further the intrusion, while maintaining an adequate level of availability for trusted users.
In order to do so, the defender has to rely on the representation of the network he has during each time step. The defender does not know the true strategy of the attacker and is unable to perfectly observe the attacker's actions, resulting in a lack of certainty of the security status of the network at any given time. The defender only has access to a stream of noisy security information generated in real-time (for example, security alerts generated via intrusion detection systems). Oftentimes, this information suffers from a high-rate of false alarms, that is, alarms being triggered when nothing of concern has actually occurred. This element constitutes the partial observability aspect of the problem and thus urges us to adopt the POMDP formulation.
The defender aims to decide on his actions based on a near-optimal policy. So the problem is modeled as a POMDP, where the decision-maker is the defender, and based on the information from the environment, choses the best action for the given situation each time.

3.1 POMDP Formulation of the Dynamic Defense Problem

3.1.1 Dependency Graphs

First of all we need a way to represent the network and especially the relation between exploits the attacker can attempt and network states, as well as the states we cannot allow which are the cases the intruder has succeeded in reaching critical points.
One way to model such interactions are Attack trees/graphs. Attack trees/graphs (first introduced in [11]) model the dependencies between exploits and system states in a cyber network, allowing one to construct the specific attack paths that intruders can take to enter a network. However, these graphs tend to be enormously large even for modestly-sized systems, rendering them restricting for any realistic applications. One way to improve scalability, is the assumption of monotonicity on the attacker's behavior, which means that the success of a previous exploit will not interfere with the success of a future exploit. Monotonicity enables one to restrict attention to dependencies between exploits and security conditions, in what is termed a dependency graph, avoiding the need to enumerate over all system states. In this way the amount of information required to describe network attacks is drastically reduced.
A dependency graph is used to model how the attacker progresses through the cyber network over time. The dependency graph is represented as a hypergraph, where nodes represent possible security conditions and directed hyperedges (edges that connect a pair of sets of nodes) represent exploits, thus presenting the relations between preconditions, the security conditions that must be enabled in order for the exploit to be attempted, and postconditions, the security conditions that become enabled if the attacker is successful with that exploit.
A security state is the set of currently enabled security conditions. In this sense, the security state at any given time represents the current capabilities of the attacker. For a given security state, the attacker uses its current capabilities (the set of enabled security conditions) to attempt exploits, with the goal of enabling more security conditions until one (or more) goal condition is achieved. The specific strategy that the attacker employs

is its own private information and is assumed to dynamically adjust according to the defender's actions.

Below is an example of a dependency graph, where we see how each exploit leads to a new SC. The cycled *SCs* are goal states for the attacker and need to be protected.

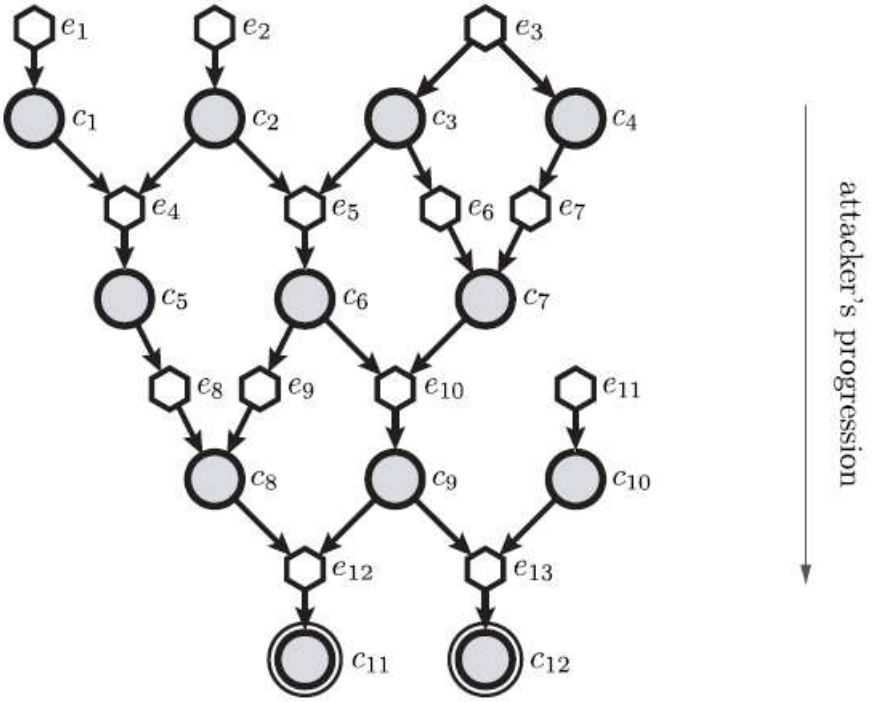

Figure 3.1: Example of a dependency graph with 12 security conditions (SCs) and 13 exploits

Formally, we represent a condition dependency graph as a directed acyclic hypergraph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{c_1, ..., c_{n_c}\}$ is the set of security conditions (nodes) and $\mathcal{E} = \{e_1, ..., e_{n_e}\}$ is the set of exploits (hyperedges). The acyclic nature of the graph follows from the monotonicity assumption. Each security condition $c_i \in \mathcal{N}$ in the hypergraph can either be true or false. The truth value of each condition is interpreted as follows: a true (enabled) condition means that the attacker possesses condition c_i, and a false (disabled) condition means that the attacker does not possess c_i. An enabled condition is interpreted as the attacker having a particular capability.
Some of the conditions in the hypergraph, when enabled, designate that an attacker has reached a goal. Such nodes are termed *goal conditions* and are denoted by the subset $N_g \subseteq \mathcal{N}$. Goal conditions are defined by the defender and correspond to something that it wants to protect.

3.1.2 Belief Formulation based on History

As mentioned above the defender, does not know exactly which SCs are enabled by the attacker. The decision-maker can only receive noisy observations based on the attacker's previous actions and so keeps a belief over the current state each moment.

The defender has to take into consideration the possible attacker type as well, meaning that, for a given network state, different attackers will attempt different actions in order to deepen their progression.

Using the environment, in this case the received security alerts, the defender constructs a belief over the current situation, denoted by b_k , that summarizes its uncertainty over both the security state and the attacker type. This belief is constructed using all of the defender's available information at time k: the (distribution over the) initial security state and attacker type, the history of all defense actions from time 0 to time $\mathrm{k}-1$, and all observations from time 0 to k, denoted by $h_k = (b_0, u_0, y_0, \ldots, u_{k-1}, y_k)$. The belief represents the joint probability distribution over security states and attacker types, and is defined as:

$$b_k = \begin{bmatrix} b_k^{1,1} & b_k^{1,2} & \ldots & b_k^{1,|\Phi|} \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ b_k^{|X|,1} & b_k^{|X|,2} & \ldots & b_k^{|X|,|\Phi|} \end{bmatrix} \in B_{X\times\Phi}$$

where $\mathrm{b}_{\mathrm{k}}^{\mathrm{ij}} = \mathrm{P}(\mathrm{x}_{\mathrm{k}} = \mathrm{i}, \varphi_{\mathrm{k}} = \mathrm{j} | \mathrm{h}_{\mathrm{k}} = \mathrm{h})$ is the likelihood that x_{k} is the true security state and φ_{k} is the true type given the realized information h_k . The space $B_{X\times\Phi}$ is the probability simplex over the state-type space $X \times \Phi$. Notice that b_k is a doubly-stochastic matrix for each k; each row represents a probability mass function over the type space for a given state and each column represents a probability mass function over the space of security states for a given type.

Based on the above, the problem of the dynamic defense of a network can be modeled as a POMDP, where the defender is the decision-maker and the attacker is part of the environment of the agent [6].

3.2 Defender problem

The defender wishes to determine an optimal defense action to deploy for any belief that it may encounter. This means trying to stop the attacker from progressing through the network, while keeping a sufficient availability level for trusted users in the network. The decision rule determining this action is termed a defense policy and is represented by the function $\pi : B_{X\times\Phi} \rightarrow \mathcal{U}$, mapping a belief matrix $b \in B_{X\times\Phi}$ to a defense action $\mathrm{u} \in \mathcal{U}$.

The problem of determining π can be cast as a POMDP:

For a given initial belief $\mathrm{b}=b_0$, the objective of the decision maker is:

$$J_\pi(b_0) = \min_\pi \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \rho^k c(b_k, u_k) | b_0 \right\} \quad (3.1)$$

And the optimal policy is:

$$\pi^* = arg\min_\pi \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \rho^k c(b_k, u_k) | b_0 \right\}, \quad (3.2)$$

where ρ with $0 < \rho < 1$ is the discount factor. The function $c(b_k, u_k)$ represents the expected cost for being in belief state b_k when defense action u_k is selected and is defined as $c(b_k, u_k) = \sum_{x_i \in X, \varphi_j \in \Phi} b_k^{ij}\ c(x_i, \varphi_j, u_k)$.

$c(s, \phi, u)$ is the weighted cost [6] taking into account security and availability and is defined as $c(x, \varphi, u) = \omega c_s(x, \varphi) + (1 - \omega) c_u(u)$, where $0 \leq \omega \leq 1$.

4. DESPOT

The DESPOT algorithm is an online POMDP solver considered state-of-the-art. It approximates the belief tree so as to search for optimal policies in a much smaller belief space.

4.1 Determinized Sparse Partially Observable Tree Structure

A DESPOT is a sparse approximation of a standard belief tree. While a standard belief tree captures the execution of all policies under all possible scenarios, a DESPOT captures the execution of all policies under a set of randomly sampled scenarios (Figure 4.1).

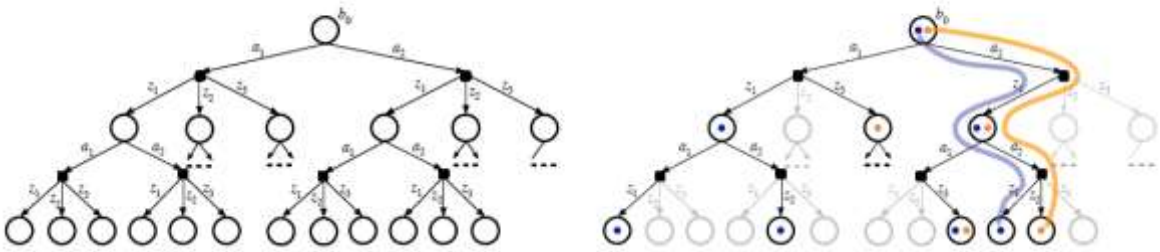

Figure 4.1: Standard Belief Tree and DESPOT

To overcome the computational challenge of online planning under uncertainty, DESPOT samples a small finite set of K scenarios as representatives of the future. Each scenario contains a sampled initial state and random numbers which determinize the uncertain outcomes of future actions and observations.
This approximation of the belief tree contains all the action branches, but not all the observation branches. Instead, it only contains those observation branches encountered under the sampled scenarios.
DESPOT is constructed by applying a deterministic simulative model to all possible action sequences under K sampled scenarios.
A scenario is an abstract simulation trajectory starting from a state x_0. Formally, a scenario for a belief b is a random sequence $\varphi = (x_0, \varphi_1, \varphi_2, \ldots)$, in which the start state x_0 is sampled according to b and each φ_i, is a real number sampled independently and uniformly from the range $[0, 1]$. The deterministic simulative model is a function $g : \mathcal{X} \times \mathcal{U} \times R \rightarrow \mathcal{X} \times \mathcal{Y}$, such that if a random number φ_i is distributed uniformly over $[0, 1]$, then $(x', y') = g(x, u, \varphi)$ is distributed according to $p(x', y'|x, u) = T(x,u,x')O(x', u, y')$.

Applying this simulative model for an action sequence $(u_1, u_2, \ldots)$ under a scenario $(s_0, \varphi_1, \varphi_2, \ldots)$ generates a simulation trajectory $(x_0, u_1, x_1, y_1, u_2, x_2, y_2 \ldots)$,where $(x_k, y_k) = g(x_{t-1}, u_k, \varphi_k)$ for k = *1, 2,….*
The simulation trajectory traces out a path $(u_1, y_1, u_2, y_2, \ldots)$ from the root of the standard belief tree. Now all the nodes and edges on this path are added to the DESPOT $\mathcal{D}$ being constructed. Each node b of $\mathcal{D}$ contains a set Φ_b of all scenarios that it encounters. We insert the scenario $(x_0, \varphi_0, \varphi_2, \ldots)$ into the set Φ_{b_0} at the root b_0 and insert the scenario $(x_k, \varphi_{k+1}, \varphi_{k+2}, \ldots)$ into the set Φ_{b_k} at the belief node b_k reached at the end of the subpath $(u_1, y_1, u_2, y_2, \ldots, u_k, y_k)$, for k = *1, 2,….*

Repeating this process for every action sequence under every sampled scenario completes the construction of $\mathcal{D}$.

A DESPOT is completely determined by the set of K random sequences sampled a priori. Hence the name Determinized Sparse Partially Observable Tree. In a DESPOT tree every node b represents a belief (just like a standard belief tree) and contains a set Φ_b of scenarios starting from this belief node. The start states of the scenarios in Φ_b form a particle set that represents b approximately.
It is possible to search for near-optimal policies using a DESPOT instead of a standard belief tree. The empirical value $\hat{V}_\pi(\mathrm{b})$of a policy π under the sampled scenarios encoded in a DESPOT is the average total discounted reward obtained by simulating the policy under each scenario for a belief node. Formally, let $V_{\pi,\Phi}$ be the total discounted reward of the trajectory obtained by simulating π under a scenario $\varphi \in \Phi_b$ for some node b in a DESPOT, then

$$\hat{V}_\pi(\mathrm{b}) = \sum_{\varphi \in \Phi_b} \frac{V_{\pi,\Phi}}{|\Phi_b|}$$

where $|\Phi_b|$ is the number of scenarios in Φ_b. Since $\hat{V}_\pi(\mathrm{b})$ converges to $V_\pi(\mathrm{b})$ almost surely as $\mathrm{K} \to \infty$, the problem of finding an optimal policy at b can be approximated as that of doing so under only the sampled scenarios. However, overfitting can be an issue, since a policy optimized for finitely many sampled scenarios may not be optimal in general, as many scenarios are excluded from set Φ_b. To overcome overfitting, a regularization of the empirical value of a policy is introduced by adding a term that penalizes large policy size. There is a chance that the agent encounters an observation not present in π, as π contains only the observation branches resulting from the simulation of the scenarios. In this case, the agent follows a default policy from then on. Similarly, it follows the default policy when reaching a leaf node of π.

To simplify the presentation, we assume without loss of generality that all rewards are non-negative and are bounded by R_{max}.

DESPOT iterates over two main steps: action selection and belief update. A standard particle filtering method, sequential importance resampling (SIR) (Gordon, Salmond, & Smith, 1993) is used for belief update.

There are two action selection methods. The first approach consists of a simple dynamic programming method that constructs a DESPOT fully before finding the optimal action. However, constructing the DESPOT fully in advance is not practical for large POMDPs. The second approach, which is more useful, is an anytime DESPOT algorithm that performs anytime heuristic search. The anytime algorithm uses a heuristic to construct the DESPOT incrementally. This allows this method to scale particularly well even for large scale POMDPs. The algorithm converges to an optimal policy when the heuristic is admissible and that the performance of the algorithm degrades gracefully even when the heuristic is not admissible [9].

4.2 Dynamic Programming

This approach wants to construct a fixed DESPOT $\mathcal{D}$ with K randomly sampled scenarios and derive from $\mathcal{D}$ a policy that maximizes the regularized empirical value under the sampled scenarios:

$$\max_{\pi}\{\hat{V}_{\pi}(b_0) - \lambda|\pi|\}$$

where b_0 is the current belief, at the root of $\mathcal{D}$. A DESPOT policy is represented as a policy tree. For each node b of π, we define the regularized weighted discounted utility (RWDU):

$$v_{\pi}(\mathrm{b}) = \frac{|\Phi_b|}{K}\gamma^{\Delta(b)}\hat{V}_{\pi_b}(b) - \lambda|\pi_b|, \qquad (4.1)$$

where $|\Phi_b|$ is the number of scenarios passing through node b, γ is the discount factor, Δ(b) is the depth of b in the policy tree π, π_b is the subtree rooted at b, and $|\pi_b|$ is the size of π_b. The ratio $|\Phi_b|/\mathrm{K}$ is an empirical estimate of the probability of reaching b. For root node b_0 we have $v_{\pi}(b_0) = \hat{V}_{\pi_b}(b) - \lambda|\pi|$, which we want to optimize.

For every node b of $\mathcal{D}$, define $v^*(\mathrm{b})$ as the maximum RWDU of b over all policies in Π_D. Assuming that $\mathcal{D}$ has finite depth and that $|\pi_0| = 0$, the following dynamic programming procedure computes $v^*(b_0)$ recursively from bottom up. At a leaf node b of $\mathcal{D}$, the agent follows default policy π_0 under the sampled scenarios:

$$v^*(\mathrm{b}) = \frac{|\Phi_b|}{K}\gamma^{\Delta(b)}\hat{V}_{\pi_b}(b).$$

For each node b, τ(b, u, y) represents the child of b following the action branch u and the observation branch y at b. Then

$$v^*(\mathrm{b}) = \max\left\{\frac{|\Phi_b|}{K}\gamma^{\Delta(b)}\hat{V}_{\pi_0}(b), \max_{u\in U}\left\{\rho(b,u) + \sum_{y\in Y_{b,u}} v^*\big(\tau(b,u,y)\big)\right\}\right\} \qquad (4.2)$$

where

$$\rho(b,u) = \frac{1}{K}\sum_{\varphi\in\Phi_b}\gamma^{\Delta(b)}\,R\big(x_{\varphi},u\big) - \lambda$$

the state x_{φ} is the start state of the scenario φ, and $Y_{b,u}$ is the set of observations following the action branch u at the node b. The outer maximization in (4.2) decides either to execute the default policy or expand the subtree at b. The inner maximization chooses among the different actions available. When the algorithm terminates, the maximizer at the root b_0 of $\mathcal{D}$ gives the best action at b_0.

In cases where $\mathcal{D}$ has unbounded depth, there is the option of truncating $\mathcal{D}$ to a depth of $\lceil R_{max}/\lambda(1-\gamma)\rceil + 1$ and run the above algorithm, provided that λ > 0. This approach is sufficient because an optimal regularized policy $\hat{\pi}$ cannot include the truncated nodes of $\mathcal{D}$. Otherwise, $\hat{\pi}$ has size at least $\lceil R_{max}/\lambda(1-\gamma)\rceil + 1$ and thus RWDU $v_{\hat{\pi}}(b_0) < 0$. Since the default policy π_0 has RWDU $v_{\pi_0}(b_0) \geq 0$ is then better than π, a contradiction.

We first simulate the deterministic model to construct the tree, then do a bottom-up dynamic programming to initialize $\hat{V}_{\pi_0}(b)$, and finally compute $v^*(\mathrm{b})$ using Equation (4.2).

Based on these, the complexity of the standard dynamic programming approach algorithm for a DESPOT is $\mathcal{O}(|U|^D \mathrm{K} D)$.

4.3 Anytime Heuristic Search

The bottom-up dynamic programming algorithm presented in section 4.2 constructs the full DESPOT $\mathcal{D}$ in advance. This is generally not practical, when it comes to large scale POMDPs. Instead, we use the DESPOT approach based on an anytime forward search algorithm to scale up. In that way, we do not construct the DESPOT fully in advance, but incrementally. The algorithm selects the action by incrementally constructing a DESPOT $\mathcal{D}$ rooted at the current belief b, using heuristic search, and approximating the optimal RWDU $v^*(\mathrm{b})$. The main components of the algorithm are described below.
To guide the heuristic search, we maintain a lower bound $\mathrm{l}(\mathrm{b})$ and an upper bound $\mu(\mathrm{b})$ on the optimal RWDU at each node b of $\mathcal{D}$, so that $\mathrm{l}(\mathrm{b}) \leq v^*(\mathrm{b}) \leq \mu(\mathrm{b})$.To prune the search tree, we additionally maintain an upper bound $\mathrm{U}(\mathrm{b})$ on the empirical value $\hat{V}^*(b)$ of the optimal regularized policy so that $\mathrm{U}(\mathrm{b}) \geq \hat{V}^*(b)$ and compute an initial lower bound $L_0(\mathrm{b})$ with $L_0(\mathrm{b}) \leq \hat{V}^*(b)$. In particular, we use $L_0(\mathrm{b}) \leq \hat{V}^*_{\pi_0}(b)$ for the default policy π_0 at b [9].
The aim is to construct and search a DESPOT $\mathcal{D}$ incrementally, using K sampled scenarios. At first, $\mathcal{D}$ contains only a single root node with belief b_0 and the associated initial upper and lower bounds. The algorithm makes a series of explorations to expand $\mathcal{D}$ and reduce the gap between the bounds $\mu(b_0)$ and $\mathrm{l}(b_0)$ at the root node b_0 of $\mathcal{D}$. Each exploration follows a heuristic and traverses a promising path from the root of $\mathcal{D}$ to add new nodes to $\mathcal{D}$. Specifically, it keeps on choosing and expanding a promising leaf node and adds its child nodes into $\mathcal{D}$ until current leaf node is not heuristically promising. Once this happens, the algorithm traces the path back to the root and performs backup on the upper and lower bounds at each node in the path, using Bellman's principle. The explorations continue, until the gap between the bounds $\mu(b_0)$ and $\mathrm{l}(b_0)$ reaches a target level $e_0 \geq 0$ or the online planning time runs out. More details in section 4.2 of [9].

4.3.1 Forward Exploration

Let $\mathrm{e}(\mathrm{b}) = \mu(\mathrm{b}) - \mathrm{l}(\mathrm{b})$ denote the gap between the upper and lower RWDU bounds at a node b. Each exploration aims to reduce the current gap $\mathrm{e}(b_0)$ at the root b_0 to $\xi \mathrm{e}(b_0)$ for some given constant $0 < \xi < 1$. An exploration starts at the root b_0. At each node b along the trial path, we choose the action branch optimistically according to the upper bound $\mu(\mathrm{b})$:

$$u^* = arg \max_{u \in U} \mu(b,u) = arg \max_{u \in U} \left\{ \rho(b,u) + \sum_{y \in Y_{b,u}} \mu(b') \right\}, \qquad (4.3)$$

where $\mathrm{b}' = \tau(\mathrm{b}, \mathrm{u}, \mathrm{y})$ is the child of b following the action branch u and the observation branch y at b. We then choose the observation branch y that leads to a child node $\mathrm{b}' = \tau(\mathrm{b}, \mathrm{u}^*, \mathrm{y})$ maximizing the excess uncertainty $\mathrm{E}(\mathrm{b}')$ at b':

$$y^* = arg \max_{y \in Y_{b,u*}} E(b') = arg \max_{y \in Y_{b,u*}} \left\{ e(b') - \frac{|\Phi_{b'}|}{K} \xi e(b_0) \right\}. \qquad (4.4)$$

Intuitively, the excess uncertainty $E(b')$ measures the difference between the current gap at b' and the "expected" gap at b' if the target gap $\xi e(b_0)$ at b_0 is satisfied. The exploration strategy seeks to reduce the excess uncertainty in a greedy manner.

If the exploration encounters a leaf node b, we expand b by creating a child b′ of b for each action $u \in \mathcal{U}$ and each observation encountered under a scenario $\varphi \in \Phi_b$. For each new child b′, we need to compute the initial bounds $\mu_0(b')$, $l_0(b')$, $U_0(b')$ and $L_0(b')$. The RWDU bounds $\mu_0(b')$, and $l_0(b')$, can be expressed in terms of the empirical value bounds $U_0(b')$ and $L_0(b')$, respectively.

Applying the default policy π_0 at b′ and using the definition of RWDU in (4.1), we have

$$l_0(b') = v_{\pi 0}(b') = \frac{|\Phi_{b'}|}{K}\, \gamma^{\Delta(b)} L_0(b')$$

as $|\pi_0| = 0$. For the initial upper bound $\mu_0(b')$, there are two cases. If the policy for maximizing the RWDU at b' is the default policy, then we can set $\mu_0(b') = l_0(b')$. Otherwise, the optimal policy has size at least 1, and it follows from (4.1) that $\mu_0(b') = \frac{|\Phi_b|}{K}\, \gamma^{\Delta(b)} U_0(b)$-λ is an upper bound. So we have

$$\mu_0(b) = \max\left\{ l_0(b), \frac{|\Phi_b|}{K}\, \gamma^{\Delta(b)} U_0(b) - \lambda \right\}.$$

There are various ways to construct the initial empirical value bounds $U_0$and L_0. More about this at [9] sections 4.3 and 4.4.

4.3.2 Termination of Exploration

We terminate the exploration at a node b under three conditions. First, $\Delta(b) > D$, i.e., the maximum tree height is exceeded. Second, $E(b) < 0$, indicating that the expected gap at b is reached and further exploration from b onwards may be unprofitable. Finally, b is blocked by an ancestor node b′:

$$\frac{|\Phi_b|}{K}\, \gamma^{\Delta(b')} \big(U(b') - L_0(b') \big) \le \lambda\, l(b', b)$$

where $l(b', b)$is the number of nodes on the path from b′ to b. The intuition behind this condition is that there is insufficient number of sampled scenarios at the ancestor node b′ . Further expanding b and thus enlarging the policy subtree at b' . may cause overfitting and reduce the regularized utility at b′ . We thus prune the search by applying the default policy at b and setting the bounds accordingly.

4.3.3 Backup

When the exploration terminates, the anytime DESPOT algorithm traces the path back to the root to perform backup on the bounds at each node b along the way, using Bellman's principle:

$$\mu(b) = \max\left\{ l_0(b), \quad \max_{u \in U}\left\{ \rho(b,u) + \sum_{y \in Y_{b,u}} \mu(b') \right\} \right\},$$

$$l(b) = \max\left\{ l_0(b), \quad \max_{u \in U}\left\{ \rho(b,u) + \sum_{y \in Y_{b,u}} l(b') \right\} \right\},$$

$$U(b) = \max_{u \in U}\left\{ \frac{1}{|\Phi_b|} \sum_{\varphi \in \Phi_b} R(s_\varphi, u) + \gamma \sum_{y \in Y_{b,u}} \frac{|\Phi_{b'}|}{|\Phi_b|} U(b') \right\},$$

where b' is a child of b with $b' = \tau(b, u, y)$.

4.3.4 Complexity

The EXPLORE method [9] of the algorithm traverses a path from the root to a leaf node of a DESPOT $\mathcal{D}$, visiting at most $D + \mathrm{K} - 1$ nodes along the way because a path has at most D nodes, and at most $\mathrm{K} - 1$ nodes not on the path can be added. At each node, the following steps dominating the running time. Checking the condition for pruning takes time $\mathcal{O}(D^2)$ in total and $\mathcal{O}(\mathrm{D})$ per node. Adding a new node to $\mathcal{D}$ and initializing the bounds take time $\mathcal{O}(I)$. Choosing the action branch takes time $\mathcal{O}(|\mathcal{U}|)$. Choosing the observation branch takes time $\min\{|\mathcal{Y}|, \mathrm{K}\} \in \mathcal{O}(\mathrm{K})$, which is loose because only the sampled observation branches are involved. Thus, the running time at each node is $\mathcal{O}(D + I + |\mathcal{U}| + \mathrm{K})$. Assuming that the anytime search algorithm invokes EXPLORE method N times, time complexity is $\mathcal{O}(\mathrm{N}(D + I + |\mathcal{U}| + \mathrm{K}))$.

As far as space complexity is concerned, the anytime search algorithm constructs a partial DESPOT with at most $\mathrm{N}(D + \mathrm{K})$ nodes, while the dynamic programming algorithm (section 4.2) constructs a DESPOT fully with $\mathcal{O}(|U|^D \mathrm{K} D)$ nodes. While the bounds are not directly comparable, $\mathrm{N}(D + \mathrm{K})$ is typically much smaller than $|U|^D \mathrm{K} D$ in many practical settings. This is the main differentiator between the two algorithms.

4.4 DESPOT-alpha

DESPOT-alpha (DESPOT-α) (analyzed in detail in [10]) is a variation of the standard DESPOT algorithm. It aims to surpass the state-of-the-art POMDP solvers by overcoming the difficulties the latter face due to particle divergence, when it comes to problems with large observation spaces. DESPOT-α improves the practical performance of online planning for POMDPs with large observation as well as state spaces. Like DESPOT, DESPOT-α uses the particle belief approximation and searches a determinized sparse belief tree. To tackle large observation spaces, DESPOT-α shares sub-policies among many observations during online policy computation.

DESPOT-α makes use of both a sparse sampling method and DESPOT. A sparse sampling method by Kearns et al. [12] is an online algorithm which can potentially deal with large observation spaces because it samples a fixed number of C observations for each action branch resulting in tree size of $\mathcal{O}(C^D |U|^D)$.

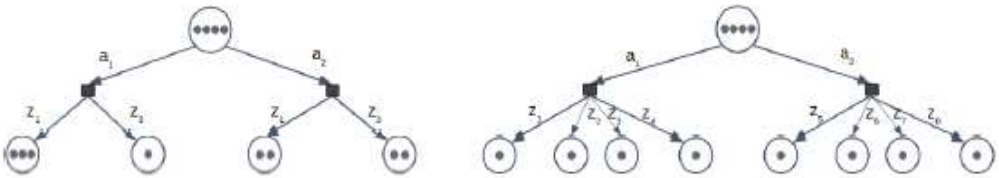

Figure 4.2: DESPOT search tree for small and large observation spaces

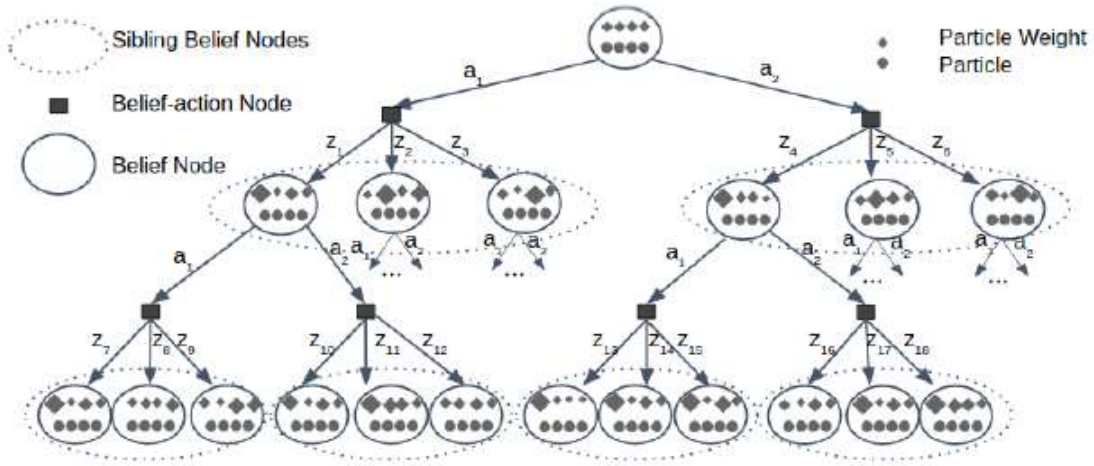

Figure 4.3: DESPOT-α search tree

K is much smaller than C^D for many problems but DESPOT suffers from the particle divergence problem when the observation space is large. The DESPOT-α variation constructs a tree of size $\mathcal{O}(C^D|U|^D)$ like sparse sampling but use determinized scenarios like DESPOT.

DESPOT-alpha (DESPOT-α) does a similar anytime forward search as standard DESPOT through trials consisting of exploration and backup on sampled scenarios. However instead of propagating only the particles producing the same observation to the child of a belief-action node, this variation suggests that we propagate all the particles to the child nodes (Figure 4.3) and update the weights of particles according to relative likelihood of observation $\mathrm{p(y \mid x, u)}$. This is similar to a particle filter.

$\mathrm{p(y \mid x, u)}$ values are also generally available for particle filtering. For a belief b, represented by the particle set Φ_b, with each particle having weight $w_b(\mathrm{x})$, the weight of particles in child belief node $\tau(\mathrm{b, u, y})$ is:

$$w_{\tau(b,u,y)}(x') = \frac{p(y \mid x', u) \sum_{x \in \Phi_b} p(x' \mid x, u) w_b(\mathrm{x})}{p(y \mid b, u)} \qquad (4.5)$$

where

$$p(y \mid b, u) = \sum_{x' \in \Phi_{\tau(b,u,y)}} p(y \mid x', u) \sum_{x \in \Phi_b} p(x' \mid x, u) w_b(\mathrm{x}) \qquad (4.6)$$

In our determinized tree, a particle x transitions to only one particle x' i.e. Φ_b has one to one correspondence with $\Phi_{\tau(b,u,y)}$). Let x'_- be the particle in Φ_b that transitions to particle x' and let x'_+ be the particle in $\Phi_{\tau(b,u,y)}$ to which particle x transitions. Then:

$$p(y \mid b,u) = \sum_{x' \in \Phi_{\tau(b,u,y)}} w_b(x'\_)p(y \mid x',u) = \sum_{x \in \Phi_b} w_b(\mathrm{x})p(y \mid x_+,u) \qquad (4.7)$$

and

$$w_{\tau(b,u,y)}(x') = \frac{p(y \mid x',u)w_b(x'\_)}{p(y \mid b,u)}$$

The resulting tree is an approximation of the belief tree based on the DESPOT definition, meaning a determinized sparse belief tree as it still contains only the observation branches reachable by the K sampled scenarios. However, every belief-action node can have up to C child belief nodes: as we do not use observations to decide which particles will go into each child node, we can sample only $\mathrm{C}(\leq \mathrm{K})$ instead of K observations from K scenarios by using only C out of K scenarios to generate observations. Always having C child belief nodes prevents over optimistic evaluation of value of belief but also makes the tree size $(\mathrm{C}|\mathrm{U}|)^D$.
Note that eventually after few information gathering actions, most of the weight would be concentrated around a few particles in the search tree. Particle filters do re-sampling when this happens. However, in the search tree, re-sampling is not required as we only need to estimate the reward which gets discounted as depth increases.

4.5 HyP-DESPOT

The Hybrid Parallel DESPOT (HyP-DESPOT) is a variation of the DESPOT algorithm that seeks to surpass the state-of-the-art algorithms for POMDPs by leveraging both CPU and GPU parallelization in order to achieve near real-time online planning performance for complex tasks with large state, action, and observation spaces.

HyP-DESPOT is a massively parallel online planning algorithm that integrates CPU and GPU parallelism in a multi-level scheme. It performs parallel DESPOT tree search by simultaneously traversing multiple independent paths using multi-core CPUs and performs parallel Monte-Carlo simulations at the leaf nodes of the search tree using GPUs. The research presented in [13] shows that HyP-DESPOT can speed up online planning by up to several hundred times, compared with the original DESPOT algorithm, in several challenging robotic tasks in simulation.

The aim of HyP-DESPOT is to parallelize all key steps of the standard DESPOT algorithm. The fact that these key steps exhibit different structural properties for parallelization needs to be taken into consideration. The two tree search steps, forward search and back-up, are irregular; leaf node initialization, which consists of many identical Monte Carlo simulations with different initial states, is regular and embarrassingly parallel. HyP-DESPOT builds a CPU-GPU hybrid parallel model to treat them separately. It uses the more flexible CPU threads to handle the two irregular tree search steps. It uses massively parallel GPU threads to handle the embarrassingly parallel Monte Carlo simulations for leaf node initialization.

HyP-DESPOT integrates CPU-based parallel tree search and GPU-based parallel Monte Carlo simulations in a multilevel scheme. Specifically, HyP-DESPOT launches multiple CPU threads to simultaneously search different paths and discover leaf nodes. At the same time, It relies on the GPU threads to takes over these leaf nodes, expand them, and initialize their children through massively parallel Monte Carlo simulations. Further, HyP-DESPOT factors the dynamics model and the observation model within a single simulation step and simulates the factored elements in parallel, in order to maximally

exploit GPU parallelization. The reader can find an extensive analysis of the HyP-DESPOT algorithm in [13].

TABLE OF TERMINLOGY

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Markov Decision Process	Μαρκοβιανή Διαδικασία Αποφάσεων
Partially Observed Markov Decision Process	Μερικώς Παρατηρούμενη Μαρκοβιανή Διαδικασία Αποφάσεων
Dynamic Defense of Network	Αυτοματοποιημένη Ασφάλεια Δικτύου

ABBREVIATIONS – ACRONYMS

MDP	Markov Decision Process
POMDP	Partially Observed Markov Decision Process
PB	Point-Based
DESPOT	Determinized Sparse Partially Observable Trees
SC	Security Condition

REFERENCES

[1] V. Krishnamurthy. *Partially Observed Markov Decision Processes*. Cambridge University Press 2016.

[2] D. Bertsekas. *Dynamic Programming and Optimal Control (Vol I)*. Nashua, NH: Athena Scientific, 3rd edition edition, 2007.

[3] D. Bertsekas. *Dynamic Programming and Optimal Control (Vol II)*. Nashua, NH: Athena Scientific, 3rd edition edition, 2007.

[4] E. J. Sondik, *The Optimal Control of Partially Observed Markov Processes*. PhD thesis, Electrical Engineering, Stanford University, 1971.

[5] R. Bellman. *Dynamic Programming*. Princeton University Press, 1st edition edition, 1957.

[6] E. Miehling, M. Rasouli, and D. Teneketzis. A POMDP Approach to the Dynamic Defense of Large-Scale Cyber Networks. *IEEE Trans. Information Forensics and Security*, 13(10):2490–2505, 2018.

[7] J. Pineau G. Shani and R. Kaplow. A Survey of Point-Based POMDP Solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

[8] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online Planning Algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008.

[9] N. Ye, A. Somani, D. Hsu, and W. Lee. DESPOT: Online POMDP Planning with Regularization. *Journal of Artificial Intelligence Reasearch*, 58:231–266, 2017

[10] N. Priyadarshini Garg, D. Hsu, and W. Lee. DESPOT-Alpha: Online POMDP Planning with Large State and Observation Spaces. In *Robotics: Science and Systems XV, University of Freiburg, Freiburg imBreisgau, Germany, June 22-26, 2019.*

[11] B. Schneier, “Attack trees,” *Dr. Dobb’s J.*, vol. 24, no. 12, pp. 21–29, 1999.

[12] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. Machine Learning, 49(2):193–208, Nov 2002. ISSN 1573-0565. doi: 10. 1023/A:1017932429737

[13] P. Cai, Y. Luo, D. Hsu, and W. Lee. HyP-Despot: A Hybrid Parallel Algorithm for Online Planning under uncertainty. *CoRR*, abs/1802.06215, 2018