# Constructing Optimal Policies for Agents with Constrained Architectures

**Dmitri A. Dolgov** and **Edmund H. Durfee**
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109
{ddolgov, durfee}@umich.edu

30 April 2003

**Abstract**

Optimal behavior is a very desirable property of autonomous agents and, as such, has received much attention over the years. However, making optimal decisions and executing optimal actions typically requires a substantial effort on the part of an agent, and in some situations the agent might lack the necessary sensory, computational, or actuating resources to carry out the optimal policy. In such cases, the agent will have to do the best it can, given its architectural constraints. We distinguish between three ways in which an agent's architecture can affect policy optimality. An agent might have limitations that impact its ability to *formulate*, *operationalize* (convert to internal representation), or *execute* an optimal policy. In this paper, we focus on agents facing the latter two types of limitations. We adopt the Markov decision problem framework in our search for optimal policies and show how gradations of increasingly constrained agent architectures create correspondingly more complex optimization problems ranging from polynomial to NP-complete problems. We also present algorithms based on linear and integer programming that work across a range of such constrained optimization problems.

## 1 Introduction

A widely-studied problem in the AI literature is the construction of optimal policies for autonomous agents [23]. Traditionally, the problem of constructing an optimal policy has been viewed separately from the problem of actually executing that policy. However, a policy is no good to an agent if it cannot carry it out, which might happen if the agent lacks the necessary sensory, computational, or actuating resources to observe the world, make the decisions, or take the actions as prescribed by the policy. As a simple example, an agent operating in a real-time environment might have a policy that optimizes expected utility but includes a requirement that the agent reacts within 1 second of a particular exogenous event whenever the event occurs. This policy might not be *realizable* within the limitations of the agent architecture. It could be unable to *operationalize* the policy because the agent might be dividing its limited perceptual resources to monitor for various important events such that it cannot assuredly notice this particular event within one second. Or the agent might be unable to dependably *execute* the policy because it draws down its battery power each time it needs to react to the particular event, and while it is capable of reacting properly at £rst, it cannot guarantee that it will have the power to react to all future event occurrences.

The concept of *bounded optimality*, as described by Russell and Subramanian [24], suggests that the best that an agent designer can expect to do in implementing an optimal agent is to formulate the best *program* that can be run on the agent's architecture. In their work, Russell and Subramanian emphasize bounded-optimal programs for the *formulation* of decisions for a resource (time) constrained agent. Our emphasis in this paper is instead on algorithms for devising bounded-optimal programs (represented as policies) for agents whose architectures constrain the policies that can be operationalized or executed. A policy is bounded-optimal with respect to an agent's architecture if, according to some evaluation criterion, that policy has the highest value in the class of policies that are realizable, given the agent's architectural constraints. A policy is said to be realizable by an agent if it satis£es all constraints imposed on the policy by the agent's architecture.

One approach to solving the constrained optimization problem of £nding optimal realizable policies is to reduce the problem to an unconstrained one. This can be accomplished in a number of ways. One might try to restrict the language for representing policies relative to the agent architecture, such that any representable policy is realizable. In this case, if the optimization algorithm uses the same language, the search can be performed in the unconstrained space, and the resulting policy will be guaranteed to be realizable. As a simple example of this approach, consider a memoryless agent that only knows the current state of the world but does not remember how it got there. In that case, even if the world is non-Markov, one can (and probably should) conduct the search for the optimal policy in the language of memoryless policies rather than to search in the space of history-dependent policies with some additional constraints. In general, however, restricting the language in just the right way may be very hard or impossible to do, as a policy composed of actions that are individually realizable (and thus should each be represented in the language) might not be realizable due to the action combinations.

Another way to reduce the constrained problem to an unconstrained one is to modify the problem representation to include the information about the agent's limitations. For example, if an agent has limited consumable resources (e.g. power), the resource usage (e.g. current power level) can be made a part of the state description. This approach, while still able to make use of ef£cient methods of solving unconstrained optimization problems, has several drawbacks. Firstly, it suffers from combinatorial (in case of multiple discrete-value resources) state-space explosion. Secondly, this approach naturally handles only the types of constraints that can be folded into the state description. The problem is that sometimes there are global constraints imposed on the policy as a whole, rather than local constraints on the execution of individual actions in particular states. One can certainly argue that any information (at the extreme, a full description of the whole policy) can be added to the state description, but we do not consider this to be a viable approach.

The alternative, which we adopt in this work, is to conduct the policy search using the unrestricted language and without modifying the state description, but to place some constraints on the space of feasible solutions. Let us say that a language $\mathcal{L}$ is used to represent a set of policies $\Pi^{\mathcal{L}}$, and an agent's architecture $\mathcal{M}$ limits the set of feasible solutions to a set of policies $\Pi^{\mathcal{L}}(\mathcal{M}) \subseteq \Pi^{\mathcal{L}}$. Moreover, there is a policy evaluation criterion $\mathcal{V}(\boldsymbol{\pi})$, which maps a policy $\boldsymbol{\pi} \in \Pi^{\mathcal{L}}$ to a number ($\mathcal{V} : \Pi \rightarrow \mathbb{R}$).

In this work, we have the following goal. Given a policy representation language $\mathcal{L}$, a restricted policy space $\Pi^{\mathcal{L}}(\mathcal{M})$, and a policy evaluation $\mathcal{V}$, £nd a policy from the constrained space ($\boldsymbol{\pi} \in \Pi^{\mathcal{L}}(\mathcal{M})$) that has the maximum value according to $\mathcal{V}$. An example problem would be to £nd a policy that has the highest total expected reward ($\mathcal{V}$) among the class of deterministic Markov policies ($\Pi^{\mathcal{L}}$) that are executable by a particular agent, given that it only has a limited amount of power, and that it has to £nish executing the policy by a certain deadline ($\mathcal{M}$). Although, we certainly do not claim to solve the stated problem in the most general case and provide solution algorithms for all possible languages, architectures, and evaluations, we do follow one relevant thread of increasingly constrained agent types and their corresponding constrained optimization problems.

The contribution of this work is a characterization of a portion of the landscape of constrained agent architectures in terms of the complexity of £nding optimal policies for those architectures, and algorithms for doing so. Not all algorithms and complexity results discussed in this paper are new. The novel results that are of the most interest include the complexity proof and the algorithm for £nding deterministic policies under linear execution constraints (section 5.1), the analysis of operationalization constraints on action utilization costs (section 5.2) and an algorithm for approximating optimal policies that bound the probability of exceeding upper bounds on the total costs of the policy (section 4.3).

We begin with an introduction of our model in section 2 and follow with a review of methods for solving unconstrained Markov decision problems in section 3. Section 4 that discusses various types of execution constraints is followed by section 5, which focuses on operationalization constraints. For each of the types of constraints, we analyze the complexity and discuss solution algorithms for the corresponding optimization problems, as well as provide some examples of agent architectures that embody these types of constraints.

# 2 The Model

## 2.1 Markov Decision Problem

In accordance with the literature, we formulate our optimization problem as a standard stationary, discrete-time, Markov decision problem with £nite state and action spaces [20]. A classical MDP can be de£ned as a tuple

$\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R} \rangle$, where:

- $\mathcal{S} = \{i\}$ is a £nite set of states.
- $\mathcal{A} = \{a\}$ is a £nite set of actions.
- $\mathbf{P} = [p_{ij}^a] : \mathcal{A} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ de£nes the transition function. The probability that the agent goes to state $j$ if it executes action $a$ in state $i$ is $p_{ij}^a$.
- $\mathbf{R} = [r_{ia}] : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ de£nes the rewards. The agent gets a reward of $r_{ia}$ for executing action $a$ in state $i$.

A policy is de£ned as a procedure for selecting an action in each state.[1] A policy is said to be *stationary* if it does not depend on time, but only on the current state, i.e. the same procedure for selecting an action is performed every time the agent encounters a particular state. A *pure* policy always chooses the same action for a state, as opposed to a *randomized* policy, which chooses actions according to some probability distribution over the set of actions. The term *deterministic* is used to refer to stationary pure policies.

A randomized Markov policy[2] $\boldsymbol{\pi} \in \Pi^{\text{MR}}$, can be described as a mapping of states to probability distributions over actions, or equivalently, as a mapping of state-action pairs to probability values: $\boldsymbol{\pi} = [\pi_{ia}] : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$; $\pi_{ia} \in [0, 1]$ de£nes the probability of executing action $a$, given that the agent is in state $i$. We assume that an agent must execute an action (a noop is considered a trivial action) in every state, thus $\sum_a \pi_{ia} = 1$.

A deterministic policy can be viewed as a degenerate case of a randomized policy, for which there is only one action for each state that has a nonzero probability of being executed. That probability must obviously equal 1. This class of deterministic policies is traditionally labeled $\Pi^{\text{MD}}$. Since there is only one action that can be executed in any state, a deterministic policy $\boldsymbol{\pi} \in \Pi^{\text{MD}}$ can be represented simply as a mapping of states to actions: $\mathbf{d} = [d_i] : \mathcal{S} \rightarrow \mathcal{A}$. The conversion between the two representations ($\boldsymbol{\pi}$ and $\mathbf{d}$) of deterministic policies is obvious:

$$d_i = a \iff \pi_{ia} = 1 \tag{1}$$

Clearly, the total probability of transitioning out of a state, given a particular action, cannot be greater than 1, i.e. $\sum_j p_{ij}^a \leq 1$. As we discuss below, we are actually interested in domains where there exist states for which $\sum_j p_{ij}^a < 1$.

If, at time 0, the agent has an initial probability distribution $\boldsymbol{\alpha} = [\alpha_i]$ over the state space, and the system obeys the Markov assumption (namely that the transition probabilities depend only on the current state and the chosen action), the system's trajectory (de£ned as a sequence of probability distributions on states) will be as follows:

$$\begin{aligned} \boldsymbol{\rho}^{t+1} &= \widetilde{\mathbf{P}} \boldsymbol{\rho}^t \\ \boldsymbol{\rho}^0 &= \boldsymbol{\alpha}, \end{aligned} \tag{2}$$

where $\boldsymbol{\rho}^t = [\rho_i^t]$ is the probability distribution of the system at time $t$ ($\rho_i^t$ is the probability of being in state $i$ at time $t$), and $\widetilde{\mathbf{P}} = [\widetilde{p}_{ij}]$ is the transition probability matrix implied by the policy $\boldsymbol{\pi}$ ($\widetilde{p}_{ij} = \sum_a p_{ij}^a \pi_{ia}$).

## 2.2 Assumptions

Typically, Markov decision problems are divided into two categories: *£nite-horizon* problems, where the total number of steps that the agent spends in the system is £nite and is known a priori, and *in£nite-horizon* problems, where the agent is assumed to stay in the system forever (see [20] for a detailed discussion of both types of models).

In this work (as will be clearer from the next sections) we concentrate on dynamic real-time domains, where agents have tasks to accomplish (typically before a given deadline). Moreover, besides succeeding or failing to carry out these tasks, agents might obtain some rewards during the execution of their policies. For example, consider an agent ¤ying a plane, whose goal is to safely get to its destination and land there. This example does not naturally correspond to a £nite-horizon problem, because the duration of executing various policies is not predetermined (unless we arti£cially impose such a £nite duration, which is not easily justi£able). On the other hand, the problem does not naturally £t the

[1] We only consider stationary Markovian problems and, therefore, do not make a distinction between decision rules and policies for ease of exposition.

[2] MR (Markov Randomized) and MD (Markov Deterministic) are the standard abbreviations used to refer to the classes ofrandomized and deterministic policies, respectively [20].

de£nition of the in£nite-horizon model, because the plane obviously cannot keep on ¤ying forever (assuming that it does not have the option of in-¤ight refueling and is not powered by a *perpetuum mobile*).

This leads us to adopt a slightly different assumption about how much time the agent spends executing its policy. We assume that there is no prede£ned number of steps that the agent spends in the system, but that optimal policies always yield *transient* Markov processes (decision problems of this type were extensively studied by Kallenberg [11]). A policy is said to yield a transient Markov process if the agent executing that policy will eventually leave the corresponding *Markov chain*, after spending a £nite number of time steps in it. The term Markov chain is used to refer to the stochastic process that results once the agent's policy has been £xed. Given a £nite state space, this assumption implies that the Markov chain corresponding to the optimal policy has no recurrent states (states that are visited in£nitely often) or, in other words:

$$\forall i \quad \lim_{t\to\infty} \rho_i^t = 0 \tag{3}$$

This means that there has to be some "leakage" of probability out of the system, i.e. there have to exist some states $\{i\}$ for which $\sum_j \sum_a p_{ij}^a < 1$. One particular case where the above assumption holds is in a system in which all trajectories lead to *absorbing* states. Once an agent enters an absorbing state, it has £nished (or failed to £nish) some task and has nowhere else to go, i.e. the probability of transitioning out of an absorbing state $i$ is zero: $\sum_j p_{ij}^a = 0$.[3] In the plane-¤ying example, all trajectories lead to either a safe landing or a crash, and once the agent enters one of these states, the probability of transitioning to other states is zero.

We also assume that the rewards that an agent receives while executing a policy are bounded, as discussed in more detail in the next sub-section.

## 2.3 Policy Value and Optimality

There are multiple ways in which one can assign numeric values to policies. In general, the value of a policy $\boldsymbol{\pi}$, given the initial probability distribution $\boldsymbol{\alpha}$, can be expressed as follows:

$$V(\boldsymbol{\pi}, \boldsymbol{\alpha}) = \sum_{h\in\mathcal{H}} V(h) P(h|\boldsymbol{\pi}, \boldsymbol{\alpha}), \tag{4}$$

where $h$ is a state history, which is just a sequence of state-action pairs $h = \{\langle s^t, a^t\rangle\}$, $\mathcal{H}$ is a set of all possible state-action histories, and $P(h|\boldsymbol{\pi}, \boldsymbol{\alpha})$ is the probability of sequence $h$ actually occurring, conditioned on $\boldsymbol{\pi}$ and $\boldsymbol{\alpha}$.

One of the more popular choices for evaluating policies, which is the most natural one for transient problems, is the expected total reward criterion.[4] If the agent receives a reward whenever it executes an action, the total expected utility of a policy can be expressed as follows:[5]

$$V(\boldsymbol{\pi}, \boldsymbol{\alpha}) = \sum_{t=0}^{T} \sum_i \rho_i^t \sum_a \pi_{ia} r_{ia}, \tag{5}$$

where $T$ is the number of steps during which the agent accumulates utility. If the rewards $\mathbf{r}$ are bounded, and the process is transient, the above sum converges for any $T$, and the value of the policy is, thus, also bounded.

Moreover, if the system obeys the Markov assumption and, as a result, follows the trajectory in (eq. 2), the value of a policy can be expressed in terms of the initial probability distribution, transition probabilities, and rewards as follows:

$$V(\boldsymbol{\pi}, \boldsymbol{\alpha}) = \sum_t^{\infty} \mathbf{r}\boldsymbol{\rho}^t = \sum_t^{\infty} \mathbf{r}\widetilde{\mathbf{P}}^t\boldsymbol{\alpha} \tag{6}$$

It is clear that the value of a policy depends on the initial state probability distribution $\boldsymbol{\alpha}$. Moreover, in general, the relative order of two policies can change depending on the initial probability distributions, i.e. $\exists \boldsymbol{\pi}, \boldsymbol{\pi}', \boldsymbol{\alpha}, \boldsymbol{\alpha}'$ : $V(\boldsymbol{\pi}, \boldsymbol{\alpha}) > V(\boldsymbol{\pi}', \boldsymbol{\alpha})$, and $V(\boldsymbol{\pi}, \boldsymbol{\alpha}') < V(\boldsymbol{\pi}', \boldsymbol{\alpha}')$. However, in some cases (such as for an unconstrained MDP), there exist policies that are optimal for *any* initial probability distribution, i.e. $V(\boldsymbol{\pi}^*, \boldsymbol{\alpha}) \geq V(\boldsymbol{\pi}, \boldsymbol{\alpha})\ \forall \boldsymbol{\pi}, \boldsymbol{\alpha}$. These policies are the ones that are commonly called "optimal" in the unconstrained literature and are typically computed,

[3] An alternative way of handling these states is to treat them as in£nitely-recurrent, i.e. once the agent gets there, it stays there forever. We do not adopt this model, because it is less natural for our domains and also leads to unnecessary complications in the optimization problems.

[4] Other widely-used criteria include, for example, the expected average reward and the discounted total reward.

[5] This assumes a linear additive utility, which is characteristic to risk-neutral, time-indifferent agents.

in a very ef£cient manner, by using some form of dynamic programming, such as value or policy iteration, which is based on Bellman optimality equations [4]. We will refer to these policies as *uniformly optimal* (using the terminology from [3]). These uniformly optimal policies $\boldsymbol{\pi}^*$ always produce a history of states that is at least as good as a history produced by any other policy, regardless of the initial conditions. Therefore, for an unconstrained case, it is suf£cient to compute a single uniformly optimal policy and use it for all instances of the problem with arbitrary initial probability distributions.

The trouble is, of course, that in the constrained case these dominant optimal policies are not always operationalizable and executable. Indeed, imagine a problem where executing the same action ($a^*$) in all states yields the best results. Therefore, the policy $\boldsymbol{\pi}^*$ that prescribes the execution of that action $a^*$ in all states is clearly uniformly optimal. However, executing $a^*$ might not be operationalizable for an agent with limited resources, and it might have to settle for sub-optimal actions for some states. Furthermore, imagine that if the agent starts in state $0$ and executes $\boldsymbol{\pi}^*$, all other states are reachable, and the agent has to spend a long time in the resulting Markov chain, thus exceeding its executable (consumable) resource limitations. However, if the agent starts in a different state $1$ and executes the same policy $\boldsymbol{\pi}^*$, only a small subset of all states is reachable and the agent can £nish executing $\boldsymbol{\pi}^*$ much faster, and has enough resources to do so. Therefore, in a constrained case, a policy is only optimal with respect to the initial probability distribution.

Moreover, once constraints are introduced, we can no longer assume that an optimal policy can be composed from optimal sub-policies. This thus violates the principle of optimality, which is the basis for most unconstrained algorithms such as value and policy iteration. Hence, the problem of constructing an optimal realizable policy becomes much more dif£cult, as shown in the next sections, where we analyze the complexity of various constrained optimization problems after looking £rst at the unconstrained case.

# 3 Unconstrained Agents

This section is devoted to a discussion of omnipotent agents that do not have any operational or execution constraints. In particular, we describe a linear programing formulation of Markov decision problems that is due to D'Epenoux [6] and cite some known results about the complexity of unconstrained MDPs. The purpose of this section is to provide a baseline and the necessary background for subsequent sections that deal with constrained agents.

## 3.1 Linear Programming Formulation

The most commonly used algorithms for solving unconstrained MDPs are value and policy iteration (see, for example, [20]). As we pointed out earlier, these algorithms and their modi£cations are very ef£cient and are very well-suited to unconstrained problems. However, it is not easy to adapt value or policy iteration to constrained problems, since these algorithms are based on the principle of optimality, which (as just mentioned) does not always hold for constrained problems. Another method for solving unconstrained MDPs that allows for an easier addition of constraints is based on a linear programing formulation of the problem.

In this section, we review how to construct a linear program, the solution to which yields a policy that maximizes the total expected reward, as de£ned in (eq. 5), over an in£nite horizon (again, assuming all states are transient):

$$\begin{aligned} V(\boldsymbol{\pi}, \boldsymbol{\alpha}) &= \sum_{t=0}^{T} \sum_{i} \rho_i^t \sum_{a} \pi_{ia} r_{ia} \\ &= \sum_{i} \sum_{a} \Big( \sum_{t=0}^{T} \rho_i^t \pi_{ia} \Big) r_{ia} = \sum_{i} \sum_{a} x_{ia} r_{ia}, \end{aligned} \tag{7}$$

where $x_{ia} = \sum_{t=0}^{T} \rho_i^t \pi_{ia}$ is the expected number of times action $a$ is excuted in state $i$. Then, $x_i = \sum_a x_{ia}$ is the total expected number of times state $i$ is visited.

Clearly, $x_{ia}$ constitutes an alternative way of representing a randomized policy. It is often referred to as the *occupancy measure* of a policy [3]. The mapping from $x_{ia}$ to $\pi_{ia}$ is trivial:

$$\pi_{ia} = \frac{x_{ia}}{\sum_a x_{ia}} = \frac{x_{ia}}{x_i} \tag{8}$$

Notice, however, that the $x_{ia}$ representation contains more information about the system than just a policy $\boldsymbol{\pi}$, because the former, besides giving us the mapping (eq. 8), also speci£es the expected number of times each state will be visited if $\boldsymbol{\pi}$ is executed.

If a policy is deterministic, then in its occupancy-measure representation only one $x_{ia}$ for a given $i$ will have a nonzero value. In that case, the occupancy-measure representation can be converted to a deterministic policy $\mathbf{d}$ as follows:

$$d_i = \begin{cases} a & \text{if } x_{ia} > 0 \\ \text{arbitrary} & \text{if } x_i = 0 \end{cases} \tag{9}$$

Notice that in the second case an arbitrary action can be assigned to state $i$, because $x_i = \sum_a x_{ia} = 0$ implies that the state $i$ will never be visited.

We can now formulate a linear program, whose solution is a set of occupancy variables that yields a policy with the highest expected value, as de£ned by (eq. 7). The linear program is:

$$\max \sum_i \sum_a x_{ia} r_{ia}$$

subject to the constraints:

$$\sum_a x_{ja} - \sum_i \sum_a x_{ia} p_{ij}^a = \alpha_j,$$
$$x_{ia} \geq 0$$

or, equivalently:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \alpha_j, \\ x_{ia} \geq 0, \end{array} \right. \tag{10}$$

where $\delta_{ij}$ is the Kronecker delta, de£ned as $\delta_{ij} = 1 \iff i = j$.

The constraints in (eq. 10) just represent the conservation of probability and have nothing to do with external constraints (such as resource limitations) imposed on the problem. That is, the expected number of times that state $j$ is visited minus the expected number of times that $j$ is entered across all state-action pairs better equal the initial probability of being in $j$.

Recall that, for unconstrained problems, there always exist deterministic policies that are uniformly optimal, i.e. they are optimal for any initial probability distribution. An important question is whether the policies constructed using the above LP belong to the class of deterministic uniformly optimal policies. A detailed discussion of this topic is beyond the scope of this paper, and we refer the reader to [20] or [11] for a comprehensive exploration of this matter. Here we just cite the main results and provide some intuitive rationale for them. As it turns out, the solution to the above LP yields a uniformly optimal solution, if the initial probability distribution $\boldsymbol{\alpha}$ used in the LP is strictly positive ($\forall i \ \alpha_i > 0$). Intuitively, the justi£cation for this condition is that it "forces" the solution to assign an optimal action for every state, whereas some states could be unreachable given a different initial probability distribution with some zero components. In the latter case, any actions could be chosen for the unreachable states, which, in general, would not be optimal for all initial distributions.

We have provided some justi£cation for why the resulting policies are uniformly optimal. Another question is whether a solution to the above LP represents a deterministic or a randomized policy. The answer is not de£nitive and depends on the algorithm that is used to solve the LP. However, a key property of linear programs such as (eq. 10) is that there always exists a set of solutions that map to deterministic policies. Recall from the theory of linear programming that $\mathbf{x}$ is a *basic feasible solution* of an LP if it cannot be expressed as a nontrivial convex combination of any other feasible solutions to the LP. An important property of basic feasible solutions of an LP with $n$ rows is that they have at most $n$ positive components. Therefore, a basic feasible solution to (eq. 10) contains at most $n = |S|$ positive components. Now, if we assume that the initial probability distribution over the states is strictly positive ($\forall i \ \alpha_i > 0$), it is clear that $x_i \geq \alpha_i > 0$, which means that there have to be *at least* $n$ positive elements in $\mathbf{x}$. Hence, there are exactly $n$ positive components in $\mathbf{x}$, one for each state $i$, which means that the corresponding policy is deterministic. Therefore, for problems with strictly positive initial probability distributions, deterministic policies are uniformly optimal and can be obtained by using an exterior point method (e.g. simplex) for solving linear programs.

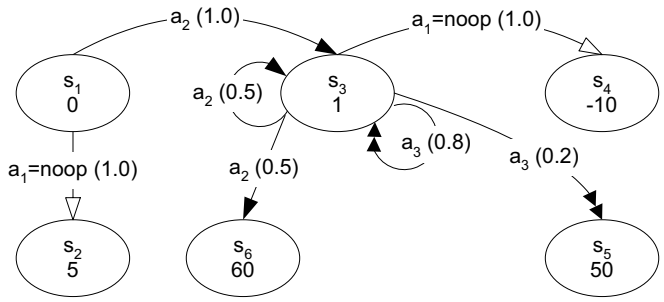


Figure 1: A simple MDP: $S = \{s_1, \ldots, s_6\}, A = \{a_1 = noop, a_2, a_3\}$. Ovals represent states (rewards for visiting them are shown inside the ovals), and arcs represent possible state transitions (probabilities are shown in parentheses). States $\{s_2, s_4, s_5, s_6\}$ are absorbing.

Therefore, in order to £nd a deterministic uniformly optimal policy for an unconstrained MDP, one just needs to solve the LP in (eq. 10) with $\boldsymbol{\alpha} > 0$. Since the resulting policy will be deterministic and uniformly optimal, the action to be executed in every state, as prescribed by (eq. 1), will be optimal for any probability distribution over the initial states. However, it must be noted that the occupancy measure representation of a policy ($x_{ia}$) that is obtained by solving (eq. 10) retains its additional interpretation as the expected number of times state $i$ is visited and action $a$ is executed *only* for the initial probability distribution $\boldsymbol{\alpha}$ that was used in the LP.

## 3.2 Example

We now introduce an extremely simple example that serves to illustrate the method described in the previous subsection. Despite the overly simplistic nature of this problem, it can be used as a skeleton for examples of constrained problems that are analyzed in the following sections.

Consider the stochastic environment depicted in Figure 1. There are six states in the problem $\{s_1, \ldots s_6\}$ and three actions $\{a_1, a_2, a_3\}$, where $a_1$ is a noop that represents the fact that the agent has a choice of not executing any action. The agent starts in $s_1$. If it does not do anything in $s_1$ (executes a noop), it will go to state $s_2$ with probability of $1.0$ and get a reward of 5 there. It can also execute action $a_2$, which will with certainty take it to state $s_3$. Similarly, in state $s_3$, the agent has three choices of actions: action $a_1$ takes it to state $s_4$ for a negative reward of $-10$, $a_2$ has a 50/50 chance of leading to $s_3$ or $s_6$ and, in the latter case, yielding a reward of 60. If, however, the agent chooses to execute $a_3$, it has a $0.8$ probability of staying in $s_3$ and a $0.2$ probability of going to $s_5$ and getting a reward of 50.

This problem yields the following linear program (as in (eq. 10)):

$$\max(\mathbf{rx}) \mid \mathbf{Ax} = \boldsymbol{\alpha},\ \mathbf{x} \geq 0, \tag{11}$$

where

$$\begin{aligned}
\mathbf{x} =& [(x_{11}, x_{12}), x_{21}, (x_{31}, x_{32}, x_{33}), x_{41}, x_{51}, x_{61}]^T, \\
\mathbf{r} =& [(0,0), 5, (1,1,1), -10, 50, 60], \quad \boldsymbol{\alpha} = [0.1, 0.1, 0.1, 0.1, 0.1, 0.5]^T, \\
\mathbf{A} =& \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0.5 & 0.2 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -0.2 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & -0.5 & 0 & 0 & 0 & 1
\end{pmatrix}
\end{aligned} \tag{12}$$

In these equations, we used round parentheses to group the components of $\mathbf{x}$ and $\mathbf{r}$ that refer to the same state. These parentheses have no other meaning and are only used to improve readability, We will also use the same notation for other vectors (such as $\boldsymbol{\pi}$) that are indexed by $ia$.

Notice that we chose an $\boldsymbol{\alpha}$ with all positive components to get a uniformly optimal solution. The solution to the above LP is $\mathbf{x} = [(0, 0.1), 0.1, (0, 0.4, 0), 0.1, 0.1, 0.7]$, which maps (per (eq. 1)) to the following deterministic uniformly-optimal policy: $\mathbf{d} = [a_2, a_1, a_2, a_1, a_1, a_1]$ (execute $a_2$ in $s_1$ and in $s_3$, and execute $a_1$ in all other states).

If, however, we set $\boldsymbol{\alpha}$ to the actual initial probability distribution as speci£ed in the example ($\boldsymbol{\alpha} = [1, 0, 0, 0, 0, 0]^T$), the LP would yield the following solution: $\mathbf{x} = [(0, 1), 0, (0, 2, 0), 0, 0, 1]$, which maps to the following deterministic policy: $d_1 = a_2, d_3 = a_2, d_6 = a_1$ and arbitrary actions for $s_2$, $s_4$, and $s_5$, since these states are not reachable ($x_2 = x_4 = x_5 = 0$). Notice that this solution prescribes the execution of the same actions for reachable states ($s_1$,$s_3$, and $s_6$) as the uniformly optimal one. This solution, besides giving us an optimal policy, also indicates that the expected number of times of visiting states $s_1$, $s_3$, and $s_6$ is 1, 2, and 1, respectively, and that the agent can expect a total reward of $\mathbf{rx} = 62$.

### 3.3 Complexity of Unconstrained MDPs

The complexity of unconstrained Markov decision problems has been a topic of extensive discussion (see [15] for a summary). Papadimitriou and Tsitsiklis [19] analyzed the computational complexity of various classes of MDPs and, in particular, showed that the problem of £nding optimal solutions for in£nite-horizon MDPs under the total expected cost criterion is P-complete. [6] The proof that these problems can be solved in polynomial time is based on the fact that an in£nite-horizon MDP can be formulated as a linear program (as described above in subsection 3.1), and since LPs are known to be in P, the reduction shows that the MDPs of this type are also in P. Actually, the polynomial-time algorithms for solving linear programs [12, 13] are extremely slow in practice. For this reason, the most popular methods for solving linear programs are based on Dantzig's simplex method [5]. These methods are usually very ef£cient, but most have been shown to have an exponential worst-case running time (e.g., [14]), and no algorithm based on the simplex method has been shown to have polynomial running time.

Recall from the previous subsection that, for unconstrained problems, there always exists a uniformly optimal deterministic policy, meaning that for any initial probability distribution, there exists a deterministic policy that is at least as good as any randomized policy. Therefore, the above complexity result holds for both deterministic ($\Pi^{\text{MD}}$) and randomized ($\Pi^{\text{MR}}$) classes of policies.

These results on the complexity of unconstrained MDPs can serve as a comparison baseline for the complexity of constrained problems, which are analyzed in the following sections.

## 4 Execution Constraints

We begin our discussion of constrained agent architectures by analyzing execution constraints. Recall that constraints of this type arise due to the inability of an agent to carry out the actions as prescribed by the policy. We model this situation by assigning resource costs to action execution and putting bounds on the total costs incurred during the process of executing a policy.

### 4.1 Constant Action Costs

In this section, we assume that every time an agent executes a given action in a particular state it incurs the same costs. In that case, the total resource costs incurred by the agent are proportional to the number of times the actions using the corresponding resources are performed. Therefore, the total costs can be expressed as linear functions of $\mathbf{x}$. The treatment presented in this section is the standard constrained MDP (CMDP) model [3, 11, 20] that is well-known and is widely used primarily in the operations research community.

As an example, recall the toy problem from section 3.2 (Figure 1) and suppose that each action $a$ takes a certain amount of time $t_a$ to execute. Moreover, imagine that the agent is placed in a real-time situation, where it must £nish executing a policy before time $T$ (this requirement could be due to a power limitation of the agent's architecture, or it could be imposed by the system designer). This constraint can be expressed as $\sum_a t_a \sum_i x_{ia} \leq T$, which assumes that $x_{ia}$ is exactly the expected number of times action $a$ is executed in state $i$. In order for the constraint to be valid, $x_{ia}$ will have to retain this meaning in whatever method we use for solving this constrained problem. We postpone

[6] Assuming that all numbers that are needed to de£ne a particular optimization problem (such as transition probabilities $p_{ij}^a$, rewards $r_{ia}$, etc.) can be written using a constant number of bits. Then, the size of the input is polynomial in $|\mathcal{S}|$ and $|\mathcal{A}|$. To be more precise, it is in $O(|\mathcal{A}||p_{ij}^a|) = O(|\mathcal{A}||\mathcal{S}|^2)$.

the solution to our running example for the moment and now turn to the general formulation of problems with linear constraints.

A CMDP is commonly described by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q} \rangle$, where $\mathcal{S}$, $\mathcal{A}$, $\mathbf{P}$, and $\mathbf{R}$ have the same meaning as in the standard MDP model (described in section 2). Additionally, in a CMDP associated with each action there is a vector of costs $\mathbf{C} = [c_{ia}^k]$ (action $a$ incurs $c_{ia}^k$ units of resource $k$ when it is executed in state $i$). Furthermore, there is a vector of upper bounds on the total costs $\mathbf{Q} = [q^k]$, and a solution to a CMDP should satisfy the following inequalities on the total expected cost:

$$\sum_i \sum_a x_{ia} c_{ia}^k \leq q^k \tag{13}$$

In order to £nd an optimal policy that satis£es these constraints, it seems that all we have to do is add the constraints in (eq. 13) to the unconstrained LP (eq. 10) and solve the resulting linear program:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \alpha_j, \\ \sum_i \sum_a x_{ia} c_{ia}^k \leq q^k, \\ x_{ia} \geq 0 \end{array} \right. \tag{14}$$

However, although the linear program for the constrained case looks very similar to the one for the unconstrained case (eq. 10), there are a few signi£cant differences between solutions to these two problems.

Recall that, in the unconstrained case, there exist policies that are optimal for any initial probability distribution $\boldsymbol{\alpha}$. As we mentioned earlier, this is generally not true in the constrained case, and optimal policies depend on $\boldsymbol{\alpha}$. Therefore, we cannot just solve a single LP and use the resulting policy for all initial probability distributions, as we did in the unconstrained case. Now, if we want to obtain an optimal policy for an initial probability distribution $\boldsymbol{\alpha}$, we need to solve a linear program that uses the same $\boldsymbol{\alpha}$. The good news is that, since we always solve linear programs that contain exactly the right initial probability distributions, the interpretation of $x_{ia}$ as the expected number of times action $a$ is executed in state $i$ will be valid, and thus our constraints mean precisely what we expect.

Another difference between the unconstrained and the linearly constrained problems is that, in the latter case, there does not necessarily exist a deterministic policy that is at least as good as any randomized policy (as was the case in the unconstrained situation). The reason for this is that the LP in (eq. 14) has additional constraints, and its basic feasible solutions can have more than $n = |\mathcal{S}|$ positive components.

### 4.1.1 Example

We now return to our running example from section 3.2 and show how the above method can be used to solve the constrained problem. Recall that we would like to £nd a solution to the example from the previous subsection, such that the solution satis£es the constraint $\sum_a t_a \sum_i x_{ia} \leq T$. In terms of the formulation given above, we have only one resource, action costs are $c_{ia}^1 = t_a$, and the single resource bound is $q^1 = T$. Let us say that the action costs are $t_a = (0, 5, 1)$ and $T = 11$. Given these numbers, the cost of the policy obtained for the unconstrained case is $\sum_a t_a \sum_i x_{ia} = 15 > T$, and thus that policy does not satisfy our constraint. Therefore, we have to add the constraint to (eq. 11), and solve the new linear program. The solution thus obtained is $\mathbf{x} = [(0, 1), 0, (0, 0.4, 4), 0, 0.8, 0.2]$, which maps to a randomized policy $\boldsymbol{\pi} = [(0, 1), 0, (0, 0.1, 0.9), 0, 0.8, 0.2]$ for the expected total reward of 56.4. Notice that the new policy prescribes the execution of the "cheaper" action $a_3$ in place of the more "expensive" $a_2$ 90% of the time for the state $s_3$. Notice that the optimal policy is randomized. The problem of £nding optimal solutions in the class of deterministic policies for linearly constrained problems is discussed in the next section dealing with operationalization constraints.

### 4.1.2 Complexity

It is easy to see that linear constraints do not add to the complexity of the corresponding optimization problem. Indeed, since linearly constrained problems can be formulated as linear programs, they are also P-complete, and, thus, belong to the same complexity class as unconstrained MDPs.

## 4.2 Variable Action Costs

In this section we relax the assumption made in the previous section and consider action costs that are not constant, but depend on the expected number of times a given action is executed in a particular state. We, therefore, no longer

limit ourselves to constraints that can be expressed as linear functions of $\mathbf{x}$. However, we do make the assumption that the constraints can be represented by continuous functions of $\mathbf{x}$. This section does not really provide any great insight into solving these kinds of problems and is included here for the sake of completeness.

For example, consider an agent that learns to perform a particular action more ef£ciently by adjusting some internal parameters every time it executes the action. In that case, the average cost of performing this action will decrease with the expected number of times the agent executes the action. Another example of a domain, where variable action costs occur is the example of ¤ying a plane: the amount of fuel needed to ¤y a certain distance decreases with the distance already ¤own, since the plane uses up fuel and loses weight.

In general, to handle nonlinear constraints we have to solve the following nonlinear program:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p^a_{ij}) x_{ia} = \alpha_j, \\ c^k(\mathbf{x}) \leq q^k, \\ x_{ia} \geq 0, \end{array} \right. \tag{15}$$

where $c^k(\mathbf{x})$ is a continuous function that speci£es the usage of resource $k$, given the occupancy measure $\mathbf{x}$.

The above program contains a linear objective function and nonlinear constraints. Unfortunately, there is no ef£cient method for solving general nonlinear programs. However, there are stochastic methods and approximations for solving these problems. For example, a linearization of the program might be possible using a method such as the Wolfe's grid method [1] or the Grif£th and Steward method based on Taylor series expansion [9].

#### 4.2.1 Example

Let us once again revisit our running example from section 3.2. Suppose that there are time-costs associated with action execution, but they are not constant (as in section 4.1.1), but decrease with the expected frequency of execution. Let us say, just an example, that it takes the agent $5$ time units (as in section 4.1.1) to execute action $a_2$ the £rst time, and that the agent learns to perform the action more ef£ciently, so that the cost of executing $a_2$ decreases as:

$$c^1(\mathbf{x}) = 2 + \frac{9}{1 + 2\sum_i x_{i1}}$$

If we use the numbers from the linear case, the constraint becomes

$$\Big(2 + \frac{9}{1 + 2\sum_i x_{i1}}\Big) \sum_i x_{i1} \leq 11$$

In order to £nd an optimal policy, we have to add this constraint to (eq. 11) and solve the resulting nonlinear program. It is easy to verify that we obtain the same solution as in the unconstrained case, which makes sense because the cost of executing $a_2$ three times is approximately $3.3 \times 3 = 9.9$ rather than $5 \times 3 = 15$ (as with constant execution costs in section 4.1.1), which no longer breaks the constraint.

#### 4.2.2 Complexity

In general, there is not much one can say about the complexity of nonlinear programs such as (eq. 15), except that they are no easier than linear programs from the previous section (the latter are obviously a special case of the former). In particular, it is not even clear how one would formulate a general nonlinear program (consisting, in part, of a number of arbitrary functions) as an input to a Turing machine, without restricting oneself to a speci£c class of functions. In the case of nonlinear optimization, it makes more sense to talk about the rate of convergence of an optimization algorithm, and the results would greatly depend on the instance of the nonlinear program and the algorithm used.

### 4.3 Risk-sensitive Execution Constraints

In the previous sections, we discussed constraints that were imposed on the total *expected* cost. However, in some domains such *risk-neutral* constraints and optimization criteria are not expressive enough (discussed in more detail in [21, 8]). In this section we survey some work in the area of risk-sensitive constraints and optimization criteria.

To handle deviations from the expected case, Sobel [25] proposed to maximize the *mean to variance* ratio with constraints on the expected. Huang and Kallenberg [10] describe a general approach to handling risk-sensitive constraints and optimization criteria.

Ross and Varadarajan [22] have considered constraints that are imposed on the *sample-path* costs and thus allow one to £nd policies that are guaranteed, in the limit, to have a zero probability of exceeding the cost bounds. Altman and Shwartz [2] also analyzed the sample path costs in both constraints and optimization criteria.

Dolgov and Durfee [8] propose an approximation that allows one to *explicitly* bound the probability of exceeding the bounds on expected total cost for transient problems with constant action execution costs. Namely, approximations to the following optimization problem are computed:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_a x_{ja} - \sum_i \sum_a x_{ia} p^a_{ij} = \alpha_j \\ P(c_k \geq q^k) \leq p^k_0, \quad x_{ia} \geq 0, \end{array} \right. \tag{16}$$

where $P(c_k) > q_k$ is the probability that the total amount $c_k$ of resource $k$ that is used by the policy exceeds an upper bound $q_k$ on that resource. The proposed linear approximation makes use of the Markov inequality to bound the probability of resource overutilization and yields the following liner program:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_a x_{ja} - \sum_i \sum_a x_{ia} p^a_{ij} = \alpha_j \\ \frac{1}{q^k} \sum_i \sum_a x_{ia} c^k_{ia} \leq p_0 \\ x_{ia} \geq 0 \end{array} \right. \tag{17}$$

Even thought the above LP yields suboptimal policies, it allows one to make explicit guarantees about the probability of resource overutilization, which can be useful if such overutilization has dire consequences. Furthermore, if penalties for resource overutilization are known at design-time and can be expressed in the same units as the rewards, it is possible to include the penalties in the policy evaluation criterion, as opposed to modeling the constraints on the overutilization probability. This yields an LP with the following objective function:

$$\max \Big( \sum_i \sum_a x_{ia} r_{ia} - \sum_k \frac{W^k}{q^k} \sum_i \sum_a x_{ia} c^k_{ia} \Big), \tag{18}$$

where $W_k$ is the penalty (in units of $r_{ia}$) incurred for overutilizing resource $k$. The maximization is subject to just the standard "conservation of probability" constraints as in (eq. 10). A bene£t of this formulation (as compared to 17) is that, for certain initial probability distributions, deterministic policies are optimal. A downside is that the formulation does not allow one to explicitly control the acceptable overutilization probabilities.

A more detailed description of this method, including an experimental evaluation can be found in [8].

#### 4.3.1 Example

Let us turn to our running example and add the following probabilistic constraint to it (using the same numbers as in section 4.1.1):

$$P(c \geq 11) \leq p_0, \tag{19}$$

where $p_0$ is some upper bound on the probability that the total cost $c$ exceeds the upper bound $q = 11$. Let us £rst calculate the probability that the optimal policy for the unconstrained agent (3.2) exceeds this bound. Recall that the optimal unconstrained policy prescribes the execution of action $a_2$ in $s_1$ and $s_3$. If the agent starts in state $s_1$ with probability 1, the total cost of the policy is a geometric random variable with the following distribution:

$$P(c = 5i) = 0.5^{i-1} \quad \forall i \in [2, \infty]$$

Thus, the total probability of exceeding the upper bound of $q = 11$ is

$$P(c \geq 11) = \sum_{i=3}^{\infty} P(c = 5i) = \sum_{i=2}^{\infty} 0.5^i = 0.5$$

However, if we set $p_0 = 0.5$ and plug the numbers into (eq. 17), we get $\mathbf{x} = [(0.45, 0.55), 0.45, (0, 0, 2.75), 0, 0.55, 0]$, which maps to the following policy $\boldsymbol{\pi} = [(0.45, 0.55), 1, (0, 0, 1), 0, 1, 0]$ that has a total reward of 32.5. Clearly, this is suboptimal, because the optimal policy that satis£es the constraint $P(c \geq 11) \leq 0.5$ is the policy for the unconstrained agent, whose probability of exceeding the bound on total cost is exactly 0.5. As mentioned above, the suboptimality of our approximation is due to the fact that the Markov inequality gives a very rough bound on the probability.

#### 4.3.2 Complexity

One of the bene£ts of using the linear approximation described above is that it is no harder to solve than the unconstrained problem. However, this is only an approximation, and we have not yet analyzed the true complexity of £nding optimal policies that satisfy probabilistic constraints. This is one of the directions of our future work.

# 5 Operationalization Constraints

As we mentioned in the introduction, some agent architectures, besides having limitations on what they can execute, also have limitations on what policies that they can represent internally. In this section, we discuss constraints of this type.

## 5.1 Deterministic Policies for Constrained Agents

As discussed earlier, optimal policies for MDPs with execution constraints are randomized, in general, and deterministic policies are suboptimal. However, a lot of work in AI has focused on deterministic actions (probably due to the existence of uniformly optimal policies). Therefore, in this section, we focus on agents with linear execution constraints that can only represent (or operationalize) deterministic policies. We analyze the complexity of this problem and present an algorithm for solving it.

This section slightly deviates from the structure of the previous sections in that we £rst formally de£ne a problem and analyze its complexity and then present a solution algorithm.

#### 5.1.1 Complexity

In order to determine how hard it is to £nd an optimal deterministic policy for a CMDP with linear constraints, let us formulate a corresponding decision problem, which we call DET-CMDP:

> *Given an instance of a transient CMDP* $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q}, \boldsymbol{\alpha} \rangle$ *and a rational number* $U$*, does there exist a deterministic policy* $\mathbf{d}$*, whose expected total reward equals or exceeds* $U$*?*

The following result characterizes the complexity of this decision problem.

**Theorem 1** DET-CMDP *is NP-complete.*

**Proof:** The presence of DET-CMDP in NP is obvious. Clearly, one can always guess a deterministic policy and calculate its expected total reward in polynomial time.

In order to show NP-completeness of DET-CMDP, we will reduce Hamiltonian Circuit (HC) to it. Recall that HC asks whether, for a given directed graph $G(V, E)$, there exists a path of length $|V|$ that begins in a given starting state $v_0$, visits every state exactly once, and returns back to the starting state. HC is known to be NP-complete. Therefore, if we could show that any instance of HC could be reduced to DET-CMDP, we would show that DET-CMDP is also NP-complete. The reduction is illustrated in Figure 2 and proceeds as follows:

1. Create a CMDP with states that correspond to vertices of the HC, and with unique deterministic actions that correspond to edges of the HC.

2. Add action cost constraints that ensure that each action is executed no more than once, which implies that every edge in the HC is also traversed no more than once. This constraint is only needed to correctly implement the next ones.

3. Assign rewards to actions and choose a $U$ in such a way that ensures that the total cost can be greater than or equal to $U$ if and only if all states are visited *at least once*.

4. Create additional constraints on action costs that, in conjunction with the above ones, are satis£ed if and only if all states are visited *exactly once*.

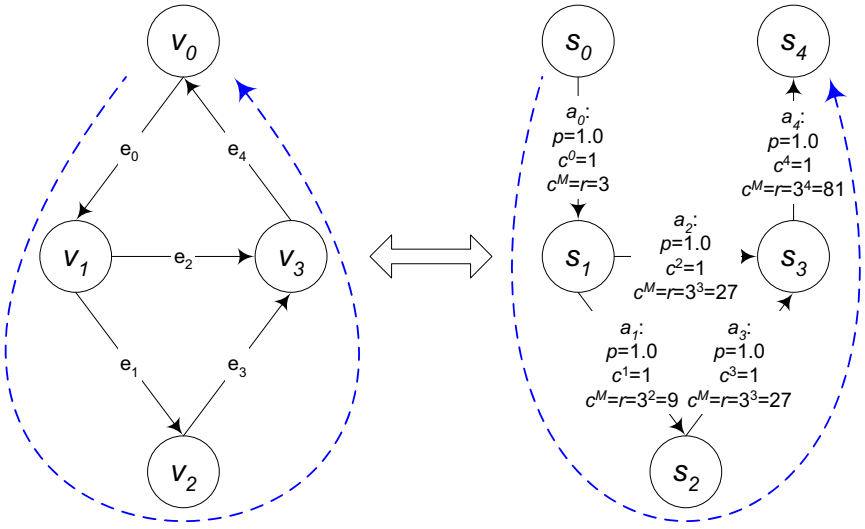


Figure 2: Reduction of HC to DET-CMDP.

Our construction will have the property that a policy that satis£es these constraints will only exist if and only if the original graph has a Hamiltonian circuit. We now describe the reduction in detail.

For any graph $G(V, E)$, let us construct a DET-CMDP that has a state $s_i$ for every vertex $v_i \in V$ and one additional state $s_n$. Therefore, for a graph with $n$ vertices, we create a state space with $n + 1$ states. Furthermore, let us add a unique action to every state $s_i \in \{s_0 \dots s_{n-1}\}$ for every outgoing edge of $v_i$ and set the transition probabilities for these actions as follows:

$$\forall a, \forall i \in [0, n-1],\ \forall j \in [1, n-1]:$$
$$p_{ij}^a = \begin{cases} 1 & \text{if the edge corresponding to } a \text{ leads from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$
$$\forall a, \forall i \in [0, n-1]:$$
$$p_{in}^a = \begin{cases} 1 & \text{if the edge corresponding to } a \text{ leads from } v_i \text{ to } v_0 \\ 0 & \text{otherwise} \end{cases}$$

Essentially, this creates a unique deterministic action $a$ for every edge $e = (v_i, v_j)$ in the original graph, such that the action transitions from state $s_i$ to state $s_j$, except the edges that lead into the starting vertex $v_0$, which are mapped to actions that lead to $v_n$ instead. In a way, we have split the starting vertex into two states: $s_0$ and $s_n$, where $s_0$ inherited all outgoing edges from $v_0$, and $s_n$ got all the edges incoming into $v_0$.

Let us also assign to every action a unit cost of a unique type. Hence, there will be $M = |\mathcal{A}|$ different resources in the CMDP – one for each action.

$$\forall k \in [0, M-1]: \quad c_{ia}^k = \begin{cases} 1 & k = a \\ 0 & k \neq a \end{cases} \tag{20}$$

Using these costs, we can add a constraint that ensures that every action is executed no more than once:

$$\forall k \in [0, M-1]: \quad \sum_i \sum_a c_{ia}^k x_{ia} \leq 1 \tag{21}$$

Let us also assign rewards to actions in the following manner. Let $m$ be some integer that is greater than the maximum number of incoming edges for any vertex in the HC. Then, for every action $a$ that leads from state $s_i$ to state $s_j$ let us assign the following reward:

$$r_{ia} = m^j, \tag{22}$$

where, $j$ is an exponent, not an index. For example, in Figure 2, the maximum number of edges leading into any vertex is 2, so we choose $m = 3$, and assign the rewards according to (eq. 22).

If we now choose

$$U = \sum_{i=0}^{M-1} m^i = \frac{m^M - 1}{m - 1}, \tag{23}$$

the policy value will be greater than or equal to $U$ if and only if every state is visited at least once. Indeed, we can view the total value of the policy as a number in base $m$ of length $M$, where the digit in position $s_i$ exactly equals the number of visits to state $s_i$. Since we have ensured (via (eq. 21)) that no action can be executed more than once, the digit in position $s_i$ can only be in the interval $[0, m - 1]$. Therefore, recalling that state $s_n$ is absorbing (the most signi£cant digit is 1), and viewing $U$ as a number in base $m$ that has 1's in all positions, it is clear that the total value of a policy can only be greater than or equal to $U$ if and only if every state is visited at least once.

Let us now assign an additional cost of a new type to every action, which will allow us to make sure that every state is visited exactly once. For every action $a$ that leads from state $s_i$ to state $s_j$, let us assign the following cost of type $M$ that exactly equals the reward for executing $a$ in $s_i$:

$$c_{ia}^M = r_{ia} = m^j \tag{24}$$

Now, let us add the following constraint on the total cost of type $M$:

$$\sum x_{ia} c_{ia}^M \leq \sum_{i=0}^{M-1} m^i = \frac{m^M - 1}{m - 1} \tag{25}$$

Clearly, since the total cost of type $M$ for a given policy must equal its total reward, the constraint (eq. 25) implies that the total reward exactly equals $U$. Obviously, the latter can happen if and only if every state of the CMDP is visited exactly once, which in turn can happen if and only if the original graph $G(V, E)$ has a Hamiltonian cycle. This concludes the reduction of HC to DET-CMDP and the proof of the fact that the latter is NP-complete. ■

### 5.1.2 Algorithm

As discussed earlier, it is not clear how one can adapt value or policy iteration to handle constraints. Also, now that we have shown that DET-CMDP is NP-complete, it is clear that we cannot solve the problem using linear programming (as the latter is P-complete).[7]

However, the problem of £nding deterministic policies slightly resembles that of Integer Linear Programing (or Integer Programming (IP), for short). Recall that integer programming is the discrete version of linear programming, with an additional constraint that all variables are limited to integer values; if only some of the variables are limited to integer values, then the problem is called a Mixed Integer Program (MIP). The problem of £nding the optimal deterministic policy for a constrained MDP is somewhat similar to IP and MIP in that in a deterministic solution, in the occupancy measure representation, only one $x_{ia}$ for a given $i$ can have a nonzero value. Integer Programming and Mixed Integer Programming are known to NP-complete, and therefore there might be a way of formulating DET-CMDP as an IP or an MIP. This section does exactly that – it describes a reduction of DET-CMDP to a Mixed Integer Program. The ability to formulate the constrained policy optimization problem as an MIP allows one to make use of a wide variety of highly optimized algorithms and tools for solving MIPs.

The £rst step in reducing DET-CMDP to MIP is to notice that it is not easy to express the "determinism" constraint as a linear function of $x_{ia}$. We, therefore, augment the representation with a set of binary occupancy variables $\Delta_{ia}$, where

$$\Delta_{ia} = \begin{cases} 1 & \text{if } d_i = a \text{ (action } a \text{ is planned for state } i) \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

Then, the condition that a certain policy is deterministic can be expressed as a linear constraint on $\Delta_{ia}$:

$$\forall i \ \sum_a \Delta_{ia} \leq 1 \tag{27}$$

[7] Unless P=NP, or we use an exponential number of optimization variables.

Therefore, we can now express our objective function and the original constraints as linear functions of $x_{ia}$, and we can also express the condition that the policy is deterministic via linear functions of $\Delta_{ia}$. Therefore, if we could "synchronize" all $x_{ia}$'s with the corresponding $\Delta_{ia}$'s with a linear function, we would have a MIP solution to which would yield an optimal deterministic policy.

The problem is, of course, that the relationship between $\Delta_{ia}$ and $x_{ia}$ is not quite linear:

$$\Delta_{ia} = \begin{cases} 1 \text{ if } x_{ia} > 0 \\ 0 \text{ if } x_{ia} = 0 \end{cases} \tag{28}$$

However, we can capture the essence of this relationship with a linear function with a couple of "tricks".

First, we need to normalize our occupancy measure representation to be in the interval $[0, 1]$:

$$y_{ia} = \frac{x_{ia}}{X}, \tag{29}$$

where $0 < X = \sup x_{ia} < \infty$ is some constant upper bound on $x_{ia}$, which exists for any transient MDP and can be computed in polynomial time. Indeed, we can just replace the expected reward in the objective function with $\sum_i \sum_a x_{ia}$ in the usual LP formulation of the constrained MDP, solve this LP in polynomial time and let $X$ equal the resulting value of the objective function.

Given the normalized occupancy measure, we can then capture the essence of the relationship between the occupancy measure variables and $\Delta_{ia}$ as a linear constraint:

$$\forall i, a: \; y_{ia} \leq \Delta_{ia} \tag{30}$$

Clearly, if $y_{ia} > 0$, the above constraint forces the corresponding $\Delta_{ia}$ to be 1, which is in accordance with (eq. 28). On the other hand, if $y_{ia} = 0$, the above constraint will hold for both $\Delta_{ia} = 0$ and $\Delta_{ia} = 1$, which does not quite satisfy (eq. 28). However, it turns out that this is not a problem for the following reason. If some $\Delta_{ia} = 1$, even if the corresponding $y_{ia} = 0$ (which is allowed by constraint (eq. 28)), the worst that can happen is that the "determinism" constraint (eq. 27) becomes unnecessarily broken. This might seem problematic, but in fact it is not, since the important thing is that another (feasible) solution with the offending deltas corrected always exists and has the same value of the objective function. Basically, the above condition has no false positives, but can have false negatives, which are not lethal. This means that if the algorithm for solving MIPs is complete (most of them are), it will always £nd the optimal deterministic policy that satis£es all the constraints (assuming one exists).

To summarize, the problem of £nding optimal deterministic policies for constrained MDPs can be formulated as the following MIP:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \frac{\alpha_j}{X}, \\ \sum_i \sum_a x_{ia} c_{ia}^k \leq \frac{q^k}{X}, \\ \sum_a \Delta_{ia} \leq 1, \\ x_{ia} \leq \Delta_{ia}, \\ X = \sup x_{ia}, \\ 0 \leq x_{ia} \leq 1, \quad \Delta_{ia} \in \{0, 1\} \end{array} \right. \tag{31}$$

As mentioned earlier, even though solving such programs is, in general, an NP-complete problem, there is a wide variety of very ef£cient tools for doing so (see [26, 18, 27] and references therein). Therefore, one of the bene£ts of reducing the optimization problem to MIP is that it allows one to make use of the available highly optimized tools, instead of having to implement a solution algorithm by hand.

### 5.1.3 Example

Let us now £nd the optimal deterministic policy for our example with the same linear constraints as in section 4.1.1. Recall that the action costs for the three actions are $[0, 5, 1]$, and the upper bound on the cost is $q^1 = 11$. Plugging the numbers into (eq. 31), we get the solution $\mathbf{x} = [(0, 1), 0, (0, 0, 5), 0, 1, 0]$, which maps to the following deterministic policy: $d_1 = a_2, d_3 = a_3, d_5 = a_1$, yielding a total expected reward of 55 and the total expected cost of 10. Notice that unlike the randomized solution to the same problem (section 4.1.1) which uses action $a_2$ in state $s_3$ 10% of the time, this deterministic policy has to always execute action $a_3$, giving a slightly lower total reward.

## 5.2 Action Utilization Costs

We continue our discussion of operationalization constraints by analyzing problems where costs are not incurred for executing actions (as in section 4), but rather actions have utilization that is incurred for including the actions in the policy. We begin this section by introducing a motivating problem.

### 5.2.1 Motivating Problem

Let us consider a problem from the domain of real-time intelligent control. This example is of interest in that the constraints imposed on the policy are not due to costs incurred during the *execution* of a policy, but there are limitations on the agent's ability to *schedule* a control cycle that is used by the architecture to execute the policy.

To illustrate the kind of problem that we are attempting to address here, consider an intuitive example of driving a car in a dynamic city environment. In order to ensure safety, the driver needs to continuously cycle through a list of activities such as watching the road ahead, checking the surrounding traf£c, reading the speedometer, etc. Depending on the result of these observations the driver takes an action (such as accelerating, slowing down, turning, etc.) that is most appropriate in the current situation. Therefore, the driver needs to formulate a *control policy* that prescribes what observation acts should be performed and what actions should be executed, based on the outcomes of the observations. However, an agent might have limited resources, and the time spent on one activity (looking at the speedometer) might take away from the time available to execute other actions (watching other cars) and to react to sudden changes in the environment (the car in front braking). The challenge is then to come up with a control policy that is executable, given the agent's resource limitations.

A solution to this real-time control problem was proposed by Musliner, Durfee, and Shin for their Cooperative Intelligent Real-Time Control Architecture (CIRCA) [16, 17]. However, CIRCA uses a *satis£cing* approach to £nding executable policies, meaning that it does not continue searching for optimal policies once it £nds one that is good enough (for a discussion of the details of this satis£cing search see [7]).

One of the main parts of CIRCA's architecture is its Real-Time System (RTS), whose purpose is to execute a *control cycle*, which consists of a scheduled sequence of *test-action pairs* (TAPs). The test part of a TAP constitutes a boolean test on the current state of the environment, and the action part speci£es the action to be executed if the environment matches the test. The control cycles for the RTS are produced by the scheduler (another component of CIRCA), which takes a standard (state to action mapping) representation of a policy and converts it to a sequence of TAP's. One of the underlying assumptions in CIRCA is that sensing and acting takes time, which, of course, means that not all policies yield schedulable TAP policies. This problem is addressed in CIRCA by a myopic generate-and-test approach, which works as follows. CIRCA £rst constructs a policy without taking into account the limitations of the RTS and then tries to schedule it. If the scheduler fails, a greedy heuristic is used to remove some TAP's (effectively replacing the corresponding actions with noops), until a set of schedulable TAP's remains. In some cases, this approach can produce suboptimal control cycles.

The approach that we suggest below is different in that it attempts to reason about the limitations of the architecture of the RTS *while* searching for an optimal policy, so that any produced policy will be guaranteed to be executable on the RTS. In making this leap, however, we need to also acknowledge that our treatment here does not yet deal with some complicated aspects of CIRCA's policy-generation problem, and in particular with some of its non-Markovian aspects.

As mentioned earlier, a characteristic feature of the constraints imposed on policies by the CIRCA RTS is that actions (other than the noop) incur costs in terms of the *utilization* of the RTS. If some action is used in a policy, it has to be scheduled and thus periodically utilizes resources of the RTS. However, this utilization cost is *constant*, regardless of how many times the action might actually be executed. Let us now de£ne the problem more formally.

### 5.2.2 Complexity

In accordance with the example described above, let us de£ne the total cost of including action $a$ in the policy to be:

$$c_a(\mathbf{x}) = u_a \theta\big(\sum_i x_{ia}\big), \tag{32}$$

where $u_a$ is a constant utilization cost incurred for including action $a$ in the policy[8], and $\theta(z)$ is a step function de£ned as follows:[9]

$$\theta(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases}$$

Therefore, we would like to solve the following program:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \alpha_j, \\ \sum_i \sum_a c_{ia}^k x_{ia} \leq q^k \\ \sum_a u_a^k \theta\big(\sum_i x_{ia}\big) \leq w^k \\ x_{ia} \geq 0, \end{array} \right. \tag{33}$$

where $\mathbf{U} = [u_a^k]$ is a vector of utilization costs, and $\mathbf{W} = [w^k]$ is a vector of utilization upper bounds. We have allowed multiple utilization costs (to resemble CMDPs with execution costs) for generality reasons.

Let us now formulate a corresponding decision problem and analyze its complexity. Consider the following question, which we call UTIL-CMDP:

> *Given an instance of a transient CMDP with constraints on action utilization $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q}, \mathbf{U}, \mathbf{W}, \boldsymbol{\alpha} \rangle$ and a rational number $V$, does there exist a policy $\boldsymbol{\pi}$, whose expected total reward equals or exceeds $V$?*

**Theorem 2** UTIL-CMDP *is NP-complete.*

**Proof:** The presence of UTIL-CMDP in NP is trivial. We can prove that UTIL-CMDP is NP-complete, by noticing that DET-CMDP is just a special case of UTIL-CMDP, and reducing the former to the latter.

Indeed, given any instance of DET-CMDP $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q}, \boldsymbol{\alpha} \rangle$, we can construct an equivalent UTIL-CMDP $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q}, \mathbf{U}, \mathbf{W}, \boldsymbol{\alpha} \rangle$ with the same state and action spaces, transition probabilities, rewards, and execution costs and bounds. All we have to do is to augment the DET-CMDP with some utilization constraints $\mathbf{U}$, $\mathbf{W}$.

For every state in the problem, we will create a unique type of utilization cost and will assign unit costs to all actions

$$\forall k \in [1, n]: \quad u_a^k = 1 \tag{34}$$

Now, we can add a constraint that ensures that only one action can be used for any given state:

$$\forall k \in [1, n]: \quad \sum_a u_a^k \theta\big(\sum_i x_{ia}\big) \leq 1 \tag{35}$$

Clearly, there exists a solution to the UTIL-CMDP constructed above if and only if there exists one for the original DET-CMDP. We have, therefore, reduced DET-CMDP to UTIL-CMDP and have, thus, shown that the latter is NP-complete. ■

### 5.2.3 Algorithm

In this section, we present an algorithm for solving CMDPs with constraints on action utilization costs via a reduction to a mixed integer program. This reduction is very similar to the one used in section 5.1 for £nding optimal deterministic policies for problems with linear execution constraints (DET-CMDP). Therefore, in this section we only brie¤y survey the differences before presenting the resulting MIP.

The £rst observation is that the optimization program for UTIL-CMDP (eq. 33) can be re-written as:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \alpha_j, \\ \sum_a u_a^k \Delta_i \leq w^k, \quad x_{ia} \geq 0, \end{array} \right. \tag{36}$$

where $\Delta_i = \theta\big(\sum_i x_{ia}\big)$ is a binary variable that shows whether action $a$ is used anywhere in the policy.

Now, we can proceed exactly as in section 5.1.2 to construct the MIP that is equivalent to the program given above. Namely, we normalize our optimization variables $\mathbf{x}$ such that $\sum_i x_{ia} \in [0, 1]$. This can be easily accomplished via the

[8] In our CIRCA example, this would be the utilization cost incurred by the RTS for scheduling action $a$.

[9] Notice that $\theta(0) = 0$, unlike the more commonly occurring step (or Heaviside) function whose value at 0 is de£ned as 1. The two are otherwise identical.

transformation $y_{ia} = x_{ia}/X$, where $X = \sup_a \sum_i x_{ia}$. Instead of computing the exact maximum, we can again take $X = \max \sum_i \sum_a x_{ia}$, which can be easily computed.

We can now use the same trick as in section 5.1.2 to synchronize $\Delta_i$ and $\sum_i x_{ia}$ to yield the following MIP:

$$\max \sum_i \sum_a x_{ia} r_{ia} \left| \begin{array}{l} \sum_i \sum_a (\delta_{ij} - p_{ij}^a) x_{ia} = \frac{\alpha_j}{X}, \\ \sum_a u_a^k \Delta_a \leq w^k, \\ \sum_i x_{ia} \leq \Delta_a, \\ X = \sup \sum_i x_{ia}, \\ 0 \leq x_{ia} \leq 1, \quad \Delta_a \in \{0,1\} \end{array} \right. \tag{37}$$

### 5.2.4 Example

As a simple example of this type of constraint, let us again turn to our running example with six states and three actions. Imagine that the system that executes policies for this domain has a (very) limited amount of memory and can represent a state-to-action mapping for only a single state-action pair (all other states default to noop actions). This is a highly simpli£ed analog of CIRCA's RTS. The optimization program for this example is:

$$\max(\mathbf{rx}) \left| \begin{array}{l} \mathbf{Ax} = \boldsymbol{\alpha}/X \\ \forall a \in \{2,3\}: \quad \sum_i \Delta_a \leq 1, \\ \forall a \in \{2,3\}: \quad \sum_i x_{ia} \leq \Delta_a, \\ X = 10, \quad 0 \leq \mathbf{x} \leq 1, \quad \Delta_a \in \{0,1\} \end{array} \right. \tag{38}$$

The solution to the above program is $\mathbf{x} = [(1,0), 1, (0,0,0), 0, 0, 0]$, which corresponds to a deterministic policy of executing action $a_1$ (noop) in state $s_1$, yielding a total reward of 5. Given the constraint that the agent cannot plan to execute more than one action (besides noops), this solution is the obvious optimum, because if the agent wastes its only available action to go to $s_3$, the best it will be able to do from there is -10.

It is worth noting that it is possible to satisfy the constraints by using a heuristic to repair the solution to the unconstrained problem (section 3.2). However, the solutions thus obtained would be suboptimal. For example, a greedy heuristic that incrementally made minimal changes to the original unconstrained policy would arrive at the suboptimal solution $\mathbf{x} = [(0,1), 0, (1,0,0), 1, 0, 0]$, which yields a reward of -10.

## 5.3 Action Combination Constraints

In the previous sections, we have analyzed some constraints that can can be expressed as linear (or at least continuous) functions of actions' execution or utilization costs. However, sometimes, coming up with such functional representations for the constraints is not an easy task, as the interactions between various actions of a policy can be quite complex. In that case, one might have to explicitly specify which combinations of actions are realizable, and which are not. In this section, we brie¤y discuss the problem of £nding policies that are optimal in the class of deterministic policies ($\mathbf{d} \in \Pi^{\text{MD}}$), where constraints are given as propositional formulas on boolean variables $\Delta_{ia}$ that have the same meaning as in the DET-CMDP problem (eq. 26). Namely:

$$\Delta_{ia} = \begin{cases} \text{true} & \text{if } d_i = a \\ \text{false} & \text{otherwise} \end{cases} \tag{39}$$

Clearly, $\boldsymbol{\Delta}$ is an alternative representation of a deterministic policy that is analogous to the occupancy measure ($\mathbf{x}$) representation of randomized policies.

### 5.3.1 Example

Let us make use of our running example one more time. Suppose that the agent can execute $a_2$ either in $s_1$ or $s_3$ but not in both (as prescribed by the optimal policy in the unconstrained case of section 3.2). This constraint could be expressed as $\varphi(\boldsymbol{\Delta}) = \neg(\Delta_{12} \wedge \Delta_{32})$. Then, the agent's best course of action is to execute $a_2$ in $s_1$ and $a_3$ in $s_3$ for a total reward of 50.

### 5.3.2 Complexity

The problem of £nding optimal deterministic policies that satisfy a given boolean formula $\varphi(\mathbf{\Delta})$ corresponds to the following decision question, which we call SAT-CMDP:

> *Given an instance of a transient CMDP $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{Q}, \boldsymbol{\alpha} \rangle$, a boolean formula $\varphi(\mathbf{\Delta})$, where $\mathbf{\Delta}$ is as de£ned above, and a rational number $V$, does there exist a deterministic policy $\mathbf{\Delta}$ that satis£es $\varphi(\mathbf{\Delta})$ and whose expected total reward equals or exceeds $V$?*

The following (quite obvious) proposition characterizes the complexity of this problem.

**Proposition 1** SAT-CMDP *is NP-complete.*

**Proof:** Clearly, SAT-CMDP is in NP. It is also NP-complete, because it contains SAT as a subproblem, which makes a reduction of SAT to SAT-CMDP a trivial matter.■

At the point of writing, we have not extensively investigated these types of problems and, therefore, do not have a good solution algorithm. However, we are interested in further pursuing this line of research and investigating hybrid approaches that combine optimization and SAT techniques.

# 6 Opaque Constraints

We conclude our discussion of constrained architectures with the most general case – a "black-box" scenario, where the agent's architectural constraints are completely opaque to whomever is trying to construct an optimal policy. Although this class of constraints is not very interesting or practical, we include it here for completeness, as being the polar opposite of the unconstrained case. We assume that there exists a "black box" that can classify policies as being realizable or not. In other words, the optimization algorithm is completely clueless as to the set of realizable policies $\Pi^{\mathcal{L}}(\mathcal{M})$, but it has access to an oracle which, given a policy $\boldsymbol{\pi}$, tells it whether $\boldsymbol{\pi} \in \Pi^{\mathcal{L}}(\mathcal{M})$. Clearly, given the situation, the time required to £nd an optimal realizable policy in the worst case is $\sim |\Pi^{\mathcal{L}}|$. Indeed, if there are no realizable policies ($\Pi^{\mathcal{L}}(\mathcal{M}) = \oslash$), one will have to ask the oracle about *all* policies from $\Pi^{\mathcal{L}}$ before a decisive answer can be obtained. If the search is conducted in the class of deterministic policies $\Pi^{\text{MD}}$, the worst case running time is $\sim |\Pi^{\text{MD}}| = |\mathcal{A}|^{|\mathcal{S}|}$, and the problem is in EXP, with the best possible worst-case execution time in $O(|\mathcal{A}|^{|\mathcal{S}|})$. In the case of randomized policies $|\Pi^{\text{MR}}| = \infty$, and the problem is, in general, undecidable.

# 7 Summary

In this paper, we have analyzed increasingly constrained agent architectures, and discussed methods for constructing optimal policies for these architectures. Through a combination of algorithmic descriptions and complexity analyses, we have provided a systematic and nearly comprehensive characterization of how operational and execution constraints associated with different agent architectures can impact the dif£culty of designing optimal agents to be embodied in those architectures. We have also presented new algorithms and complexity results for the following three problems: 1) £nding optimal deterministic policies for agents with linear execution constraints (section 5.1), 2) £nding optimal policies for agents with action utilization 5.2 costs, and 3) approximating optimal policies that bound the probability of exceeding upper bounds on the total costs of the policy (section 4.3).

# 8 Acknowledgements

This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under contract F30602-00-C-0017 as a subcontractor through Honeywell Laboratories. The authors would like to thank Kang Shin, Haksun Li, and David Musliner for their valuable contributions to this work.

# References


[1] J. Abadie, editor. *Nonlinear Programming*. North-Holland, 1967.

[2] E. Altman. Adaptive control of constrained markov chains: Criteria and policies. *Annals of Operations Research, special issue on Markov Decision Processes*, 28:101–134, 1991.

[3] E. Altman. *Constrained Markov Decision Processes*. Chapman and HALL/CRC, 1999.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[6] D'Epenoux. A probabilistic production and inventory problem. *Management Science*, 10:98–108, 1963.

[7] D. A. Dolgov and E. H. Durfee. Satis£cing strategies for resource-limited policy search in dynamic environments. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS-02)*, 3, pages 1325–1332, 2002.

[8] D. A. Dolgov and E. H. Durfee. Approximating optimal policies for agents with limited execution resources. In *Proceedings of IJCAI-2003*, 2003.

[9] R. Grif£th and R. Steward. A nonlinear programming technique for the optimization of continuous processing systems. *Management science*, 7:379–392, 1961.

[10] Y. Huang and L. Kallenberg. On £nding optimal policies for Markov decision chains. *Math. of Operations Research*, 19:434–448, 1994.

[11] L. Kallenberg. *Linear Programming and Finite Markovian Control Problems*. Mathematisch Centrum, Amsterdam, 1983.

[12] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–396, 1984.

[13] L. Khachian. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[14] V. Klee and G. Minty. How good is the simplex algorithm? *O. Sisha, editor, Inequalities III*, 20:191–194, 1972.

[15] M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the complexity of solving Markov decision problems. In *UAI–95*, pages 394–402, Montreal, 1995.

[16] D. Musliner, E. Durfee, and K. Shin. CIRCA: A cooperative intelligent real time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1561–1574, - 1993.

[17] D. J. Musliner, E. H. Durfee, and K. G. Shin. World modeling for the dynamic construction of real-time control plans. *Arti£cial Intelligence*, 74(1):83–127, 1995.

[18] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience, 1988.

[19] C. Papadimitriou and J. Tsitsiklis. The complexity of markov chain decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[20] M. L. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, 1995.

[21] K. Ross and B. Chen. Optimal scheduling of interactive and non-interactive traf£c in telecommunication systems. *IEEE Transactions on Auto Control*, 33:261–267, 1988.

[22] K. Ross and R. Varadarajan. Markov decision processes with sample path constraints: the communicating case. *OR*, 37:780–790, 1989.

[23] S. Russell and P. Norvig. *Arti£cial Intelligence A Modern Approach*. Prentice-Hall, New Jersey, 1995.

[24] S. J. Russell and D. Subramanian. Provably bounded optimal agents. *JAIR*, 2:575–609, 1995.

[25] M. Sobel. Maximal mean/standard deviation ratio in undiscounted mdp. *OR Letters*, 4:157–188, 1985.

[26] W. P. W.J. Cook, W.H. Cunningham and A. Schrijver. *Combinatorial Optimization*. Wiley Interscience, 1997.

[27] L. Wolsey. *Integer Programming*. John Wiley & Sons, 1998.