Sample-Based Policy Iteration for Constrained DEC-POMDPs

Feng Wu¹ and Nicholas R. Jennings¹ and Xiaoping Chen²

Abstract. We introduce constrained DEC-POMDPs — an extension of the standard DEC-POMDPs that includes constraints on the optimality of the overall team rewards. Constrained DEC-POMDPs present a natural framework for modeling cooperative multi-agent problems with limited resources. To solve such DEC-POMDPs, we propose a novel sample-based policy iteration algorithm. The algorithm builds on multi-agent dynamic programming and benefits from several recent advances in DEC-POMDP algorithms such as MB-DP [12] and TBDP [13]. Specifically, it improves the joint policy by solving a series of standard nonlinear programs (NLPs), thereby building on recent advances in NLP solvers. Our experimental results confirm the algorithm can efficiently solve constrained DEC-POMDPs that cause general DEC-POMDP algorithms to fail.

1 Introduction

Markov decision processes (MDPs) and their partially observable counterparts (POMDPs) are widely used for planning under uncertainty. A natural extension of these models to cooperative multi-agent settings is provided by the decentralized POMDP (DEC-POMDP) framework. Unlike single-agent POMDPs, there is no centralized belief state during the execution of DEC-POMDPs. Rather, each agent, with different partial information of the environment, must reason about the decisions of the other agents and how they may affect the environment. The complexity of finite-horizon DEC-POMDPs has been proved to be NEXP [3], much harder than single-agent POMDPs. Nevertheless, many exact and approximate solutions have been developed for solving DEC-POMDPs [1, 2, 5, 11, 12, 13].

In more detail, each joint action executed in the environment has an immediate reward specified by the reward function in DEC-POMDPs. The goal is to find a joint policy that maximizes the long-term accumulated reward as measured by the expected value function. However, in many real-world settings, the resources available to the agents are limited. Typical examples are disaster-response applications where multiple battery-equipped UAVs are employed to search for survivors given a finite amount of energy. The goal of the UAVs is to maximize saved lives while making energy usage below the prescribed thresholds so that they have sufficient power to return to the charging stations. Another scenario is the rock sampling task on Mars. Since solar power is the main energy resource of the rovers, they must sample as many rocks as possible before running out of battery. In both of these cases and many other besides, the utility depends on multiple factors. There is one reward (e.g., number of saved lives and number of rocks sampled) to be maximized, but this is subject to several constraints (e.g., battery level of UAVs or the rovers) with respect to limited resource budgets.

To model the above problems using standard DEC-POMDPs, it is often required to manually balance different constraints into a single reward function until the corresponding joint policy exhibits the desired behavior. Simply adding constraints in the state space would not work since the resource consumption accumulates over time and depends on both states and actions. However, tuning a model with different constraints is generally difficult, even for domain experts, since the concept of value functions is not intuitive. To address this, we extend the standard model to consider constraints. Specifically, the consumption of resources is modeled as a set of cost functions. For each cost function, an upper bound cost is defined which is the prescribed budget for the resource such as the battery capability of UAVs or Mars rovers. The objective is then to find a solution that maximizes the long-term reward without violating the constraints at each time step. This naturally models multi-agent sequential decision-making problems involving constraints and the constrained DEC-POMDP can be viewed as a multi-agent extension of the single-agent constrained POMDP [6]. However, solving this multi-agent extension is much more challenging given that DEC-POMDPs are significantly harder than POMDPs. Additionally, our model is fundamentally different from the work on DEC-MDPs with temporal constraints [4, 7, 14] where each task is assigned a temporal window during which it should be executed.

In this paper, we propose Sample-Based Policy Iteration (SBPI) for solving constrained DEC-POMDPs. This borrows ideas from dynamic programming of standard DEC-POMDPs and constructs the policies from the last step up to the first step. The approximation is motivated especially by MB-DP [12] where a portfolio of top-down heuristics is used to sample belief-cost pairs. The belief-cost pairs contain information about reachable belief states and admissible costs [9] for the current step. Intuitively, the admissible costs are the remain in resource, e.g., battery, that can be used in the future steps without violating the constraints. At each iteration, the joint policies are improved for the corresponding belief-cost pairs. The policy improvement procedure is formulated as a standard NLP that can be solved by any off-the-shelf NLP solver such as Snopt and Ipopt. We use stochastic polices with a fixed amount of memory so the algorithm has linear time and space complexity over horizons. Given this, the main contribution of this paper lies in the general solution framework for constrained DEC-POMDPs, as well as the approximation we make for solving large problems. It is straightforward to extend our work to include other constraints such as integer and stochastic constraints or take advantage of the many NLP solvers developed in the optimization community. In short, this is the first work towards solving constrained DEC-POMDPs. Moreover, our experimental results on standard benchmark problems confirm the advantage of

¹ University of Southampton, UK, {fw611, nrj}@ecs.soton.ac.uk
² University of Science and Technology of China, xpchen@ustc.edu.cn
SBPI compared to TBDP [13] and its variant.

The remainder of the paper is organized as follows. We first introduce standard DEC-POMDPs and its constrained extensions. Then, we review the DP framework and present the SBPI algorithm. Finally, we show the empirical results and conclude the paper.

2 Background

2.1 Decentralized POMDPs

Formally, a decentralized POMDP (DEC-POMDP) is defined as a tuple \( \langle I, S, b^0, \{ A_i \}, P, \{ \Omega_i \}, O, R, T \rangle \), where:

- \( I \) is a set of agents identified by \( 1, 2, \ldots, n \in I \).
- \( S \) is a finite set of system states and \( b^0 \in \Delta(S) \) is the initial state distribution.
- \( A_i \) is a finite set of actions for agent \( i \), and \( \bar{A} = \times_{i \in I} A_i \) is the joint action set.
- \( P : S \times \bar{A} \rightarrow \Delta(S) \) is a state transition function and \( P(s'|s, \bar{a}) \) denotes the probability of the next state \( s' \) when taking joint action \( \bar{a} \) in state \( s \).
- \( \Omega_i \) is a finite set of observations for agent \( i \), and \( \bar{\Omega} = \times_{i \in I} \Omega_i \) is the joint observation set.
- \( O : S \times \bar{A} \rightarrow \Delta(\bar{\Omega}) \) is an observation function and \( O(\bar{a}|s') \) denotes the probability of observing joint observation \( \bar{\omega} \) after taking \( \bar{a} \) with outcome state \( s' \).
- \( R : S \times \bar{A} \rightarrow \mathbb{R} \) is a reward function and \( R(s, \bar{a}) \) is the immediate reward after taking joint action \( \bar{a} \) in state \( s \).
- \( T \) is the time horizon of the problem.

A local policy of agent \( i \), \( \pi_i \), is a mapping from the observation sequences \( \Omega_i^t = (\omega_{i1}, \omega_{i2}, \ldots, \omega_{it}) \) to its action set \( A_i \), and a joint policy is a set of local policies, \( \bar{\pi} = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle \), one for each agent.

The value function of a joint policy \( \bar{\pi} \) is defined as:

\[
V_{\bar{\pi}}(s, \bar{q}) = R(s, \bar{a}) + \sum_{s',\bar{a}} P(s'|s, \bar{a})O(\bar{a}|s', \bar{a})V_{\bar{\pi}}(s', \bar{q}')
\]  

where \( \bar{a} \) is the joint action specified by joint policy \( \bar{\pi} \) and \( \bar{q}' \) is the joint sub-policy of \( \bar{\pi} \) after observing the joint observation \( \bar{\omega} \). The goal of solving a DEC-POMDP is to find a joint policy \( \bar{q}^* \) that maximizes the expected value of \( b^0 \):

\[
q^* = \arg \max_{\bar{q}} \sum_{s \in S} b^0(s)V_{\bar{\pi}}(s, \bar{q})
\]

Notice that DEC-POMDPs are equivalent to POMDPs when there is only one agent. While the execution of policies is inherently decentralized with only local information for each agent, the computation of policies during the planning phase can be centralized.

2.2 Constrained DEC-POMDPs

The constrained DEC-POMDP is formally defined as a tuple \( \langle I, S, b^0, \{ A_i \}, P, \{ \Omega_i \}, O, R, T, \{ C_k \}_{k=1}^{K}, \{ c_k \}_{k=1}^{K} \rangle \) with the following additional components:

- \( C_k(s, \bar{a}) \) is the cost of type \( k \) incurred for executing action \( \bar{\omega} \) in state \( s \) and all the costs are non-negative, i.e. \( C_k(s, \bar{a}) \geq 0 \).
- \( c_k \) is the upper bound on the cumulative cost of type \( k \).

For example, in the UAV coordination problem, the cost function \( C_k(s, \bar{a}) \) is the energy usage for action \( \bar{a} \) in state \( s \) and the upper bound \( c_k \) is the total capability of the battery pack.

Solving a constrained DEC-POMDP corresponds to finding an optimal joint policy \( \bar{q}^* \) computed by Equation 2 subject to the cumulative cost constraints:

\[
\forall k \in 1..K, \frac{\sum_{t=1}^{T} C_k(s^*_t, \bar{a}^*_t)}{b^0} \leq c_k
\]

where \( \bar{a}^*_t \) is the joint action specified by the joint policy \( \bar{q}^* \). Similarly, the \( k \)-th expected cumulative cost can be recursively defined as:

\[
V_c(s, \bar{q}^*_k) = C_k(s, \bar{a}^*_k) + \sum_{s', \bar{a}'} P(s'|s, \bar{a})O(\bar{a}|s', \bar{a})V_c(s', \bar{q}^*_k)
\]

Therefore, the cost constraints for a joint policy \( \bar{q} \) in state \( s \) can be simply written as:

\[
\forall k \in 1..K, V_c(s, \bar{q}) \leq c_k
\]

The solution of a constrained DEC-POMDP is to maximize the value function in Equation 1, while making all accumulated costs below the prescribed thresholds as described in Equation 5. Generally, constrained DEC-POMDPs are harder than standard DEC-POMDPs as they have the same worst-case policy space and each agent has no information about the cost occurred by the other agents.

3 Multi-Agent Dynamic Programming

In standard DEC-POMDPs, an agent’s policy is usually represented as a decision tree and a joint policy as a collection of trees, one for each agent. When running a policy tree, the agent follows a path from the root to a leaf node depending on its received observations and the actions at the nodes of the path are executed. Offline planning algorithms usually take input of the DEC-POMDP model and output a joint policy tree that maximizes the expected value. Then the joint policy is distributed and each agent takes its own part for execution. Generally, it is intractable for large problems to directly search for the best joint policy trees since the number of all possible joint trees grows double-exponentially with the horizon.

Our approach is based on the exact dynamic programming (DP) algorithm for standard DEC-POMDPs [5]. It incrementally constructs policy trees from the last step towards the first step. At each iteration, it performs an exhaustive backup on each of the sets of trees to create new policy trees for each agent. In the backup operation, for each action and each resulting observation, a branch to any of the previous-step trees is considered. The DP iteration also recursively computes the values for every new joint policy. If all policy trees are generated for every step in the horizon, the total number of complete policy trees for each agent is of the order \( O(|\bar{A}|^{|S|}) \). This double exponential blow-up presents the key challenge for the DP solution and it will quickly run out of memory even for toy problems. Given this, a crucial step of the multi-agent DP operator is to prune dominated policy trees. A policy tree \( q_0 \) of agent \( i \) is dominated if for every possible belief point and every possible policy of the other agents there exists at least one other policy tree \( q' \) that is as good as or better than \( q_0 \). This test for dominance is performed using a linear program and removing a dominated policy tree does not reduce the value of the optimal joint policy [5].

To solve constrained DEC-POMDPs with the DP method, there are two additional steps. The first one is an update step that recursively computes the expected costs for every \( k \in 1..K \) according to Equation 4. This is analogous to the evaluate step where the value
We propose Sample-Based Policy Iteration (SBPI) to address these, which is a dynamic programming approach that iteratively improves the policy. The idea of SBPI is to sample belief states and policies, then evaluate the performance of the sampled policies and update the policy based on the performance. This process is repeated until no further improvement is possible.

### Algorithm 1: Multi-Agent Dynamic Programming

**Input:** A constrained DEC-POMDP model.

∀i ∈ I, Q_i ← exhaustively initialize all last-step policy trees

**for** t=1 to T-1 **do** // Bottom-up iterations.

∀i ∈ I, Q_i+1 ← exhaustive backup Q_{i+1}

V_i^t ← recursively evaluate all joint policies Q_i^t

∀k, V_i^t ← recursively update all expected costs

**repeat**

i ← randomly select an agent in I

q_i^t ← find a policy tree in Q_i^t where

// A constraint is violated.

∀b ∈ Δ(S), ∀q_i^t ∈ Q_i^t:

∃k ∈ 1..K, V_i^t(b, q_i^t)k > c_k

Q_i^t ← Q_i^t - {q_i^t} // Prune the policy.

**until no more pruning is possible.**

return ∀i ∈ I, Q_i^t

---

### Algorithm 2: Sample-Based Policy Iteration

**Input:** A constrained DEC-POMDP model.

∀i ∈ I, Q_i ← initialized with a random policy

**for** m=1 to M **do** // Bottom-up iterations.

∀i ∈ I, Q_i ← recursively evaluate all joint policies Q_i

∀k, V_i^m ← recursively update all expected costs

return ∀i ∈ I, Q_i

---

### Figure 1. Example of Stochastic Policy for Two Agents

4. **Sample-Based Policy Iteration**

In standard DEC-POMDPs, the MBDP algorithm [12] first generates a set of reachable belief states using top-down heuristics and then keeps only a fixed number of the best policies for these beliefs. It offers linear time and space complexity w.r.t the time horizon and can solve much larger problems with essentially arbitrarily long horizons. Intuitively, we can apply similar ideas to constrained DEC-POMDPs and test the constraints when choosing the best policies as follows. First, a set of belief states are sampled by some pre-computed heuristics. Then, we backup the policies and for each belief point prune new policies that violate the constraints using the same method as described in the previous section. However, this simple idea still suffers from the exponential growth in the number of policies since the upper bounds of the costs are still very loose.

In this work, we borrow ideas from MBSP and its successors [12, 13] for efficient policy generation, and address the looseness by reasoning about the potential cost of the current step when sampling beliefs. Specifically, at each iteration, SBPI first samples pairs of beliefs and the accumulated costs (b_i, d_i) up to the current step using heuristics. Then it searches the best joint policy for each belief-cost pair by solving a NLP. Algorithm 2 outlines the main processes of SBPI. In the following subsections, we first introduce our NLP formulation and then present our belief-cost sampling method.

4.1 **Stochastic Policy Improvement**

We use stochastic policies [13] instead of deterministic policy trees to represent the solutions for two main reasons. First, the stochastic policies are parameterized. This enables us to search over the policy space by optimization methods instead of enumerating all possible policy trees. Second, as discussed in [6], the randomization introduced by the stochastic policies is useful for avoiding sub-optimality of deterministic policies. Note that a constrained DEC-POMDP is equivalent to a constrained POMDP when there is only one agent. The stochastic policies used in this paper are similar to finite state controllers (FSC) [2] but with layered structures, which is also called periodic FSC [8]. Each policy has a total of T layers and each layer contains a fixed number (M) of nodes.

Formally, each node of the stochastic policy for agent i is defined as a tuple q_i = (ψ_i, η_i), where

- ψ_i : Q_i → Δ(A_i) is an action selection function that specifies a distribution over the actions, i.e. p(α_i | q_i).
- η_i : Q_i × Ω_i → Δ(Q_i) is a node transition function that defines the probability distribution over the next nodes q_i' when q_i is observed, i.e. p(q_i' | q_i, α_i).
The value function of a joint stochastic policy \( \bar{q} \) in state \( s \) can be computed as:

\[
V_{\bar{s}}(s, \bar{q}) = \sum_{a} \sum_{|q_i|} p(a_i|q_i)|R(s, \bar{a}) + \sum_{a} P(s'|s, \bar{a})
\]

\[
\sum_{a} O(s'|s, \bar{a}) \sum_{|q_i|} \sum_{q_i} p(q_i|q_i) V_{\bar{s}}(s', \bar{q}')
\]

(6)

For a given joint belief \( b \), the value of joint policy \( \bar{q} \) is \( V_{\bar{s}}(b, \bar{q}) = \sum_{s \in S} b(s)V_{\bar{s}}(s, \bar{q}) \). Similarly, we have the expected cost function for the k-th constraint as:

\[
V_{\bar{s}}(s, \bar{q})_{k} = \sum_{a} \sum_{|q_i|} p(a_i|q_i)C_{k}(s, \bar{a}) + \sum_{a} P(s'|s, \bar{a})
\]

\[
\sum_{a} O(s'|s, \bar{a}) \sum_{|q_i|} \sum_{q_i} p(q_i|q_i) V_{\bar{s}}(s', \bar{q}')_{k}
\]

(7)

Then, the cost function for a joint belief can be defined as \( V_{\bar{s}}(b, \bar{q})_{k} = \sum_{s \in S} b(s)V_{\bar{s}}(s, \bar{q})_{k} \) for every constraint \( k \).

Before the improvement procedure, each node \( q_i \) of every agent \( i \) is initialized with random parameters \( \psi_i, \gamma_i \). Then, for each sampled belief point \( (b, \bar{d}) \) and joint policy node \( \bar{q} \), the NLP as described in Table I is formulated with the objective of maximizing the expected value. The cost constraints ensure that the new joint policy uses only bounded resources and the probability constraints guarantee that the corresponding parameters of the new policy are probabilities. This NLP can be efficiently solved with any off-the-shelf solver, with the output containing the new parameters for the joint node.

For problems with many agents, the number of variables and constraints may grow beyond the capability of NLP solvers. They may run out of memory or take too much time to find the solution. To alleviate this, we can use an approximation as follow: (1) Select a subgroup of agents with heuristics; (2) Improve the agents’ policies in this group while keeping policies of the other agents fixed; (3) Repeat (1) and (2) several times until no improvements are possible for every constraint. Before the improvement procedure, each node \( q_i \) of every agent \( i \) is initialized with random parameters \( \psi_i, \gamma_i \). Then, for each sampled belief point \( (b, \bar{d}) \) and joint policy node \( \bar{q} \), the NLP as described in Table I is formulated with the objective of maximizing the expected value. The cost constraints ensure that the new joint policy uses only bounded resources and the probability constraints guarantee that the corresponding parameters of the new policy are probabilities. This NLP can be efficiently solved with any off-the-shelf solver, with the output containing the new parameters for the joint node.

For problems with many agents, the number of variables and constraints may grow beyond the capability of NLP solvers. They may run out of memory or take too much time to find the solution. To alleviate this, we can use an approximation as follow: (1) Select a subgroup of agents with heuristics; (2) Improve the agents’ policies in this group while keeping policies of the other agents fixed; (3) Repeat (1) and (2) several times until no improvements are possible for every constraint. Before the improvement procedure, each node \( q_i \) of every agent \( i \) is initialized with random parameters \( \psi_i, \gamma_i \). Then, for each sampled belief point \( (b, \bar{d}) \) and joint policy node \( \bar{q} \), the NLP as described in Table I is formulated with the objective of maximizing the expected value. The cost constraints ensure that the new joint policy uses only bounded resources and the probability constraints guarantee that the corresponding parameters of the new policy are probabilities. This NLP can be efficiently solved with any off-the-shelf solver, with the output containing the new parameters for the joint node.

4.2 Belief and Cost Sampling

In standard DEC-POMDPs, a joint belief state is a probability distribution over the states, i.e. \( b \in \Delta(S) \). Unlike single-agent POMDPs, the execution of DEC-POMDP policies does not require maintaining a belief state over time. Given policy node \( q_i \) at time \( t \), agent \( i \) selects an action \( a_i \sim p(A_i|q_i) \), executes it, receives a subsequent observation \( q_i^{t+1} \), then updates its policy node to \( q_i^{t+1} \sim p(Q_i^{t+1}|q_i^{t}, q_i^{t+1}) \).

<table>
<thead>
<tr>
<th>Table 1. Nonlinear Program for Policy Improvement</th>
</tr>
</thead>
</table>

Maximize \( \sum_{a} \prod_{k} x(a_i|q_i) \left[ R(b, \bar{a}) + \sum_{a'} C_k(b, \bar{a}) + \sum_{a'} P(s'|s, \bar{a}) \prod_{k} y(q_i'|q_i, a_i, \alpha_i) V_{\bar{s}}(s', \bar{q}') \right] \), s.t.

(1) The cost constraints:

\[ \forall k \sum_{a} \prod_{k} x(a_i|q_i) \left[ C_k(b, \bar{a}) + \sum_{a'} P(s'|s, \bar{a}) \prod_{k} y(q_i'|q_i, a_i, \alpha_i) V_{\bar{s}}(s', \bar{q}') \right] \leq c_k - d_k \]

(2) The probability constraints:

\[ \sum_{a} x(a_i|q_i) = 1, \forall a_i, \alpha_i \prod_{k} y(q_i'|q_i, a_i, \alpha_i) = x(a_i|q_i), \forall a_i, \alpha_i \prod_{k} x(a_i|q_i) \geq 0, y(q_i'|q_i, a_i, \alpha_i) \geq 0. \]

where \( x(a_i|q_i) \), \( y(q_i'|q_i, a_i, \alpha_i) \) are variables of each agent’s policy \( q_i \), \( \forall k d_k \) are the admissible costs, and \( R(b, \bar{a}) = \sum_{s} b(s)R(s, \bar{a}), C_k(b, \bar{a}) = \sum_{s} b(s)C_k(s, \bar{a}), P(s'|s, \bar{a}) = \sum_{s} b(s)P(s'|s, \bar{a})O(\bar{s}|s', \bar{a}) \]

Algorithm 3: Belief and Cost Sampling

**Input:** A constrained DEC-POMDP model and time \( h \).

\( \forall s \in S, b(s) \leftarrow 0 \)

\( \forall k \in 1..K, d_k \leftarrow 0 \)

for \( n=1 \) to \( N \) do // Sample \( N \) times.

\( s \leftarrow \) randomly draw a state from \( b^0 \)

for \( i=1 \) to \( h \) do

\( \forall i \in I, a_i \leftarrow \) select an action w.r.t. the policy \( q_i \)

run a simulator of the system with \( (s, \bar{a}) \)

\( \forall k \in 1..K, d_k \leftarrow d_k + C_k(s, \bar{a}) \)

\( s \leftarrow \) get the new system state

\( b(s) \leftarrow b(s) + \prod_{i} p(q_i'|q_i^{t-1}, a_i) \]

normalize \( b \) and \( \forall k \in 1..K, d_k \leftarrow d_k/N \)

return \( (b, d_1..K) \)

However, a joint belief state is useful for the DP process to compute the expected value of a joint policy and identify the best one. Although a belief state can be recursively computed by Bayesian updating \( b^{t+1} = Pr(S|b^t, \bar{a}^t, q^{t+1}) \), it is generally inefficient since each belief state is a vector of size \(|S|\). In this paper, we adopt sampling methods to generate the set of belief states.

With the stochastic policy representation, a random policy has been provided for sampling. The basic procedure we consider is the use of a particle filter. Starting from \( b^0 \), we run the the simulation \( N \) times and collect a set of weighted state particles. The \( j \)-th particle is a pair \( (s_j, w_j) \) and the total weight of the particle is \( w = \sum_j w_j \). Then, the particle set represents the state distribution as:

\( b(s) = \frac{1}{w} \sum_{j=1}^{N} \{w_j : s_j = s\} \)

(8)

where \( w_j : s_j = s \) = \( w_j \) if \( s_j = s \) and 0 otherwise. This sequential importance sampling process will converge to the true distribution if \( N \) is sufficiently large. One key issue with the filtering algorithm is to decide the weight \( w_j \) for each particle. Since we will use the joint belief to improve \( q^t \), the weight can be set as:

\( w_j = \prod_{i \in I} p(q_i'|q_i^{t-1}, o_i') \)

(9)

where \( q_i^{t-1} \) is the last sampled policy and \( o_i' \) is the observation received by agent \( i \) after we run the joint action associated with \( q_i^{t-1} \). Obviously, \( w_j \) is the joint probability transitioning from \( q_i^{t-1} \) to \( q_i' \) given the joint observation \( o_i' \). To obtain information on the cumulative cost for each sampled belief, we introduce a new variable \( d^b_k \) representing the expect-
ed cumulative cost that has been incurred up to time step \( h \), i.e.
\[
d_k^h = \sum C_k(s^h, a^h).
\]
Then the expected cumulative cost that can be additionally incurred for the remaining time steps without violating the overall constraint is the difference between \( d_k^h \) and \( d_k^h \), which is called the admissible cost \([9]\) at time step \( h \). Note that we use the expected accumulated cost instead of the actually incurred cost when improving the policies. When sampling the beliefs, we collect pairs of the state and the cost and use the average value to estimate the expected accumulate cost as shown in Algorithm 3.

5 Experiments

5.1 Experimental Settings

For each problem, we defined cost functions \( C_k(s, a) \) as well as the corresponding upper bounds \( c_k \). Specifically, we assume that agents are battery-equipped and each action takes a certain amount of energy. The total capability of the battery packs is the upper bound that can be consumed during the process. Generally speaking, the upper bound can be set to an arbitrary value. However, if the upper bound is very large, none of the policies will violate the constraint. On the other hand, if the upper bound is too small, none of the valid policies exist subject to the constraint. To illustrate the usefulness of constraints, we deliberately chose the upper bound so that only a subset of the policies are valid for the constraints.

We compared our results with TBDP [13] — currently the leading algorithm for finite-horizon DEC-POMDPs — which consistently outperforms other approximate algorithms. To date, there is no algorithm in the literature focusing on constrained DEC-POMDPs. For comparisons, therefore we solved each benchmark problem with the standard version of TBDP that ignores the constraints (TBD-P) and a variation that takes input of a new reward that linearly mixes the original reward and costs (TBDP-MIXED): \( \tilde{R}(s, a) = R(s, a) - \sum C_k(s, a) \). TBDP-MIXED illustrates an example of the technique that fits the rewards and costs into a single reward and solves the constrained DEC-POMDPs with standard solvers. Although more sophisticated methods of combining reward and costs may exist, they are domain-dependent.

In the experiments, we computed the policies of each benchmark and evaluated them by a simulator designed for the model. It checked every constraint at each step and terminated when any of the constraints were violated. Each value was produced by the simulator with 100 trials. We reported the values of accumulated rewards (Total Value) and the percentage of trials where constraints are violated (Failure Rate). All results are averaged over 20 runs of the algorithms on each of the problems. SBPI was implemented in Java 1.6 and ran on a Mac OSX machine with 2.66GHz Intel Core 2 Duo CPU and 2GB of RAM available for JVM. Nonlinear programs were solved using Snoot with the AMPL interface.

5.2 Experimental Results

The Cooperative Box Pushing problem [11] is a common benchmark for DEC-POMDPs with two agents pushing 3 boxes (1 large and 2 small boxes) in a 3×3 grid. This domain has total 100 states, 4 actions and 5 observations for each agent. We defined the cost function as: 0.5 for action turn-left and turn-right, 1.0 for action move-forward and 0.0 for action stay. The upper bound costs were set to 10 for \( T=20 \) and 50 for \( T=100 \). As can be seen from Table 2, SBPI achieved higher value than TBDP and TBDP-MIXED, and also had much lower failure rate. The policies computed by TBDP violated the constraints throughout the 100 trials and had a failure rate of 100%. This suggests that the upper bounds were quite tight. Therefore, the agents constantly ran out of battery with the policies considering no constraints. TBDP-MIXED performed much better than TBDP when the horizon was 20, but the performance dropped dramatically for horizon 100 since the policy space grows double-exponentially. In contrast, SBPI had more stable performance when the horizon was shifted from 20 to 100.

The Stochastic Mars Rover problem [1] simulates two rovers with the task of cooperative rock sampling on Mars. This domain has 256 states and each agent has 6 actions and 8 observations. The cost function was defined as: 0.5 for action up, down, left and right, and 1.0 for action drill and sample. The upper bound costs were set to 24 for \( T=20 \) and 120 for \( T=100 \). In Table 2, we can see SBPI also achieved much better performance than TBDP and TBDP-MIXED. Interestingly, TBDP-MIXED worked better for a longer rather than a shorter horizon as also shown in the Cooperative Box Pushing domain. The reason is that the actions drill and sample are critical for the rock sampling task, but also have higher cost than the moving actions. Given a longer horizon with a higher upper bound, it is possible to complete more tasks and thereby gain more reward. This set of experiments show that the performance of TBDP-MIXED depends on many factors such as the structures of reward and cost functions, the horizon, etc. Therefore, it is difficult to design a single reward function for a problem with multiple objectives.

The Meeting in a 3×3 Grid problem [2] has 81 states, 5 actions and 9 observations per agent. The cost function was defined as: 0.5 for action up, down, left and right, and 0.0 for action stay. The upper bound costs were set to 20 for \( T=20 \) and 100 for \( T=100 \). In this domain, SBPI had better performance than TBDP, as expected. However, TBDP-MIXED worked surprisingly well, especially when the horizon was long. We observed that the agent with this policy
tended to stay in a grid for a long period of time because the stay action has 0 cost. For the instance with short horizon, this policy led to lower reward since there were few chances for the agents to meet in the same grid — the task of this domain. However, when the horizon is long and the grid world is relatively small (3×3), this policy might get high value since it had less chance to violate the constraints but more chance to meet. For some domains such as Meeting in a 3×3 Grid, TBDP-MIXED can work better than SBPI because it is less likely to get stuck in a local optima.

The Broadcast Channel [2] and Multi-Agent Tiger [10] problems are classical benchmarks for DEC-POMDPs. We included them here for the sake of completeness. The cost function was defined as: $\text{cost}_t = \begin{cases} 1.0, & \text{if \text{send}:1.0, \text{not-send}:0.5} \\ 0, & \text{otherwise} \end{cases}$ for Broadcast Channel (upper bound: 28 for $T=20$ and 140 for $T=100$) and $\text{cost}_t = \begin{cases} 1.0, & \text{if \text{open-left}, \text{open-right}:1.0, \text{listen}:0.5} \\ 0, & \text{otherwise} \end{cases}$ for Multi-Agent Tiger (upper bound: 30 for $T=20$ and 150 for $T=100$). We can see from Table 2 that SBPI outperforms TBDP and TBDP-MIXED with higher value and lower failure rate in both domains. TBDP and TBDP-MIXED violated the constraints in almost all trials with a failure rate of near 100%.

Notice that TBDP-MIXED first computed a new reward function that mixed the original reward and costs and then solved the new model with TBDP-MIXED. The mixed reward can be defined as $R(s, \bar{a}) = R(s, \bar{a}) - x \cdot C(s, \bar{a})$ with a cost ratio $x$. In this set of experiments, we varied the cost ratio $x$ and solved the Cooperative Box Pushing problem ($T=20$) with TBDP-MIXED. As we can see from Figure 2, the total values and failure rates of TBDP-MIXED fluctuate with different cost ratios. For the range of $x \in (0, 10)$, the total values produced by TBDP-MIXED are always lower than SBPI’s and the failure rates are higher than SBPI’s. In the line graph, the best total value and failure rate are achieved by TBDP-MIXED when $x = 3.5$. However, the cost ratio is domain-dependent and it is generally hard to find the “right” value with good performance.

6 Conclusions

Constrained DEC-POMDPs are a natural model of cooperative multi-agent problems with limited resources where the goal is to find a joint policy that maximizes the long-term reward, while keeping the accumulated costs below the prescribed thresholds. The SBPI algorithm is proposed for solving constrained DEC-POMDPs. It has several important advantages. Like MBDP and its successors, it has linear time and space complexity over the horizons [12]. This is a crucial property for problems with very long horizons. Similarly to PBVI for constrained POMDPs [6], SBPI estimates the admissible cost by sampling with heuristics. Hence SBPI can concentrate on the policies using only a “reasonable” amount of resources given the previous steps. At each iteration, SBPI improves policies with a series of standard nonlinear programs. One benefit of so doing is that SBPI can take advantage of existing NLP solvers. Another strength is that the algorithm can be easily extended to consider other types of constraints such as integer and stochastic constraints. In the experiments, SBPI performs very well on several standard benchmark problems, outperforming the leading solver with much better social welfare and a lower failure rate.

In terms of future work, one limitation of SBPI is with regard to becoming stuck in local optima. To overcome this, strategy such as random restarts may be helpful. Also from the experiments, we note that the policy generated by SBPI may have a small chance to violate the cost constraints since the cumulative cost function is computed recursively using the sampled beliefs. This may cause serious issues for some domains. Thus, it may be useful to approximate the cost function for the whole belief space instead of a limited number of belief points and guarantee all constraints are certainly satisfied.

REFERENCES