A Decentralized Approach to Multi-agent Planning in the
Presence of Constraints and Uncertainty

Aditya Undurti and Jonathan P. How

Abstract—We address the problem of planning in the presence of uncertainty and constraints for teams of unmanned vehicles. The problem is formulated as a Constrained Markov Decision Process (C-MDP). We allow for plans with a non-zero but bounded probability of violating constraints, a quantity that we define as risk and provide a solution technique that keeps the risk below a specified threshold while optimizing reward. We also use the decoupling between the dynamics of individual agents to assume transition independence and use this assumption to reduce the complexity of the problem. We provide representative simulation results to show that our technique achieves high reward while keeping risk bounded.

I. MOTIVATION

The use of teams of UAVs and UGVs in the aftermath of a Chemical, Biological, Radiological and Nuclear, Explosive (CBRNE) incident has several advantages [1]. However to operate successfully, these agents require the ability to plan in an uncertain, dynamic environment. The agents must also ensure their own safety at all times. This can be challenging since performing tasks in such an environment might entail taking risk, which is defined as the probability of violating a constraint. Constraints include danger zones or obstacles – situations that an agent can expect to encounter. Planning in the presence of uncertainty while balancing objectives with risks is therefore key to mission success.

II. LITERATURE REVIEW

Ono and Williams investigated the problem of constraints in the presence of process noise [2], [3]. The planning framework is a Mixed-Integer Linear Program (MILP) and obstacle avoidance constraints are written as chance constraints, i.e. the probability of violating a constraint is required to be below a certain threshold. The probabilistic constraints are then converted into tightened deterministic constraints where the tightening factor is proportional to the bound on the constraint violation probability. A similar formulation was used by Luders and How [4] in the context of Rapidly-Exploring Random Trees (RRTs). Improvements by Ono and Williams [2] to the methods presented in [3] account for the fact that there may be several obstacles and therefore risk needs to be “allocated” among them. However, in contrast to the work presented in this paper, [2] does not specifically look at multiple agents, in particular, the scaling problems that arise when there are many agents and joint actions.

Also note, that Dolgov [5] investigates resource constraint problems and includes the constrained resources in the state space. In this work, the resource we consider is risk which is a continuous variable whereas Dolgov et al. do not consider continuous state spaces. Solving MDPs with continuous state spaces is difficult and is best accomplished using function approximation techniques. Solving constrained MDPS in continuous state spaces is an area of ongoing work. However in this paper, we define risk in a manner that avoids having to add it to the state space. Furthermore, Dolgov’s work mostly looked at off-line methods [6], whereas in this paper we seek fast online algorithms to deal with a dynamic environment. Other work has also been done on solving constrained POMDPs by Isom et al. [7], but the proposed methods are again off-line methods that are computationally too expensive for fast planning.

Transition independence plays a key role in reducing the complexity of the multi-agent planning problem in this paper. Becker, Zilberstein and Goldman [8] investigate using transition independence to handle reward coupling in the unconstrained case – it is extended to the constrained case in this work.

III. APPROACH

Due to the presence of uncertainty, the problem lends itself to being formulated as a Markov Decision Process (MDP). An MDP is defined by the tuple \( < S, A, R, T > \), where \( s \in S \) is the state space, \( a \in A \) are the actions, \( R(s, a) : S \times A \rightarrow \mathbb{R} \) is a reward model that maps states and actions to rewards, and \( T(s', a, s) : S \times A \times S \rightarrow \mathbb{R} \) is the transition model that gives the probability that state \( s' \) is reached when action \( a \) is taken in state \( s \). The stochastic dynamics of the system are captured in the transition model \( T \), and the mission objectives in the reward model \( R \). A solution to an MDP is a policy \( \pi : S \rightarrow A \) that maps states to actions. The optimal policy \( \pi^* \) is defined as one that satisfies the following optimality condition:

\[
\pi^* = \arg \max_\pi E \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) \right]
\] (1)

where \( \gamma \in [0, 1] \) is a discount factor that time-discounts rewards. The standard MDP shown here does not provide a mechanism by which to capture risk. In this work, we define risk as the probability that a hard constraint on the system will be violated. MDPS do not provide a means by which to capture risk due to the fact that there is no natural means by which hard constraints can be encoded. Typically, hard constraints are treated as large penalties in the reward...
model. While this may be suitable for problems in which hard constraints must be satisfied at all times, the current work considers problems in which we allow some probability for constraints to be violated, and may be required in order to achieve high reward. Modeling constraints as penalties leads to problems in such situations - the choice of penalty value that achieves the desired balance between risk and reward is not clear, and tuning that penalty might require more a priori knowledge of the optimal policy than can be expected. Furthermore, there might not even be a value for which this balance is achieved - it has been shown [9] that in some situations a planner that models both rewards and risks in the reward model can switch abruptly between being too conservative and too risky.

Previous work by the authors extended the standard MDP framework by defining a constrained MDP as the tuple $< S, A, R, T, C >$ [9]. $S$, $A$, $R$, and $T$ are as defined previously, and $C(s, a): S \times A \rightarrow \mathbb{R}$ is a constraint model that gives the probability of violating a constraint when action $a$ is taken in state $s$. The optimal policy that solves a constrained MDP is defined as a policy $\pi^*$ that satisfies the following:

$$\pi^* = \arg \max_\pi \mathbb{E} \left[ \sum_{t=0}^T \gamma^t R(s_t, \pi(s_t)) \right]$$

subject to

$$\mathbb{E} \left[ \sum_{t=0}^T C(s_t, \pi(s_t)) \right] \leq \alpha$$

The constrained states (states where $C(s, a) = 1 \ \forall a$) are required to be absorbing states to ensure that $\mathbb{E} \left[ \sum_{t=0}^T C(s_t, \pi(s_t)) \right] \leq 1.0$ for all $s$ and $t$.

A. Two Ways to Measure Risk

Previously we defined risk as the probability that a constraint will be violated. However, there are two different ways to measure risk - defined here as accumulated and memoryless. Accumulated risk is the total risk that has been accumulated since $t = 0$, i.e. we set $t_0 = 0$ in Equations 2 and 3. Under this definition, we require that the total risk - the risk taken in the past, plus the risk that is expected in the future - be less than $\alpha$. On the other hand, memoryless risk is the risk that is expected in the future. Since the agent cannot perform anymore actions after violating a constraint (constrained states being absorbing states), it is reasonable to assume that no constraints have been violated so far. Risk that was not realized is ignored, and only the risk that is expected in the future is constrained.

Both methods of measuring risk are equally valid, and choice of accumulated versus memoryless risk is problem-dependent. Solving a problem with accumulated risk requires risk to be included in the state vector $s$. However, since risk is a continuous variable (a probability), the problem becomes a Continuous MDP which are in general computationally challenging. Such problems are best solved using function approximation. Using function approximation for constrained MDPs is an area of ongoing work. In this paper, we focus our attention on problems with memoryless risk.

In our previous work [9], we proposed an on-line algorithm for solving constrained MDPs with memoryless risk. However that work has some limitations, in particular for large teams with dynamic constraint maps. We propose a new decentralized approach that helps address some of these limitations. We specifically consider the multi-agent case and take advantage of the fact that the agents’ dynamics are independent and the only coupling arises only when overall team risk is accounted. But first, we briefly discuss the on-line constrained MDP algorithm presented in [9].

IV. PREVIOUS WORK

The solution that was previously proposed was to use an on-line forward search to optimize the reward and check for constraint feasibility up to a horizon of length $D$. Constraint feasibility for times beyond $D$ is ensured by using a constraint-penalty-to-go estimate that is computed off-line. The solution requires two quantities to be tracked - the reward $R$ and the constraint value $C$. For this reason, we maintain two separate value functions - one associated with the reward $R(s, a)$ which we call $V_R(s)$ and another associated with the constraint value $C(s, a)$ which we call $V_C(s)$. Thus the algorithm has two components - an off-line component where an approximate solution for $V_C$ is obtained, and an on-line component where the off-line estimate for $V_C$ is refined and $V_R$ is computed over the finite horizon to select the best action.

A. Off-line Constraint Penalty Estimate

In the off-line component of the algorithm, we solve for a policy $\pi^*_C$ that will minimize the constraint penalty. If the minimum constraint penalty from a certain state is below $\alpha$, we can guarantee that there exists at least one action in that state (the action associated with the policy $\pi^*_C$) that is constraint-feasible. We obtain the minimum constraint penalty, $U_C(s)$, by solving the following unconstrained optimization problem:

$$U_C(s) = \min_\pi \mathbb{E} \left[ \sum_{t=0}^T C(s_t, \pi(s_t)) \right]$$

If, for any state $s$, $U_C(s) \leq \alpha$, then there exists at least one policy (the optimal policy $\pi^*_C$ that solves problem 4) that guarantees that starting from $s$, the constraints will never be violated. In solving the optimization problem shown in Equation 4, we can use any approximate technique that overestimates $U_C$. We know that if the overestimated value for $U_C(s)$ remains less than $\alpha$, the exact value for $U_C(s)$ is also less than $\alpha$. During the on-line portion of the algorithm (described in detail the next section) we use this guarantee to ensure constraint feasibility for times beyond the on-line planning horizon.

B. on-line Reward Optimization

During the on-line part of the algorithm, we compute the future expected reward $V_R(s)$ and the future expected
Algorithm 1 Pseudocode for an on-line algorithm to solve constrained POMDPs

1: Function Constrainedon-lineMDPSolver()
2:   Static:
3:     \( s_c \): The current state of the agent
4:     \( T \): The current search tree
5:     \( D \): Expansion Depth
6:     \( U_C \): An upper bound on \( V_C \)
7:     \( \Phi(s, a) \): System dynamics
8: 2: s_c \leftarrow s_0
9: 3: Initialize \( T \) to contain only \( s_c \) at the root
10: while ExecutionTerminated() do
11:    \( s_c \leftarrow \Phi(s_c, a) \)
12:    Update tree \( T \) so that \( s_c \) is the new root
13: end while

Algorithm 2 The Constrained MDP Algorithm

1: Function Expand(s, \( U_C, D \))
2: if \( D = 0 \) then
3:   \( V_C(s) = U_C(s); V_R(s) = 0; \pi(s) = \pi_C(s) \)
4: return \( \pi(s), V_R(s), V_C(s) \)
5: else
6: for \( a \in A \) do
7:   \( [V_R(s'), V_C(s'), \pi(s')] = \text{Expand}(s', U_C, D - 1) \)
8:   \( Q_R(s, a) = \sum_{s'} V_R(s') P(s', s, a) + R(s, a) \)
9:   \( Q_C(s, a) = \sum_{s'} V_C(s') P(s', s, a) + C(s, a) \)
10: \( \pi'(t) = \pi(s) \forall t \neq s \)
11: \( \pi'(s) = a \)
12: \( p(a) = \text{ComputeP}(\pi') \)
13: end for
14: \( a_C = [a : Q_C(s, a) < \alpha] \)
15: \( \pi'(s) = \arg\max_{a \in a_C} Q_R(s, a) \)
16: \( V_R(s) = Q_R(s, \pi'(s)) \)
17: \( V_C(s) = Q_C(s, \pi'(s)) \)
18: return \( \pi'(s), V_R(s), V_C(s) \)
19: end if

constraint penalty \( V_C(s) \). The previously-obtained minimum constraint-penalty-to-go is then evaluated at the leaves of the tree to ensure constraint feasibility beyond the planning horizon.

Algorithm 1 is the main on-line planning algorithm. The inputs to this algorithm are the current state \( s \), the planning horizon length \( D \), and the minimum constraint penalty (or its overestimate) \( U_C \). The planning algorithm then begins the cycle of planning and executing. First, the function Expand is called on the current state. This function, shown in Algorithm 2, builds a tree of depth \( D \) and computes the best action to execute. It is discussed in more detail in the next paragraph. Once the action is executed the state \( s \) is updated and the Expand routine is again called on the most current belief. This cycle is repeated until execution is terminated.

When the Expand subroutine is called with a depth of 0, i.e. if the belief node on which the function has been called is a leaf node, the function simply returns the minimum constraint penalty \( U_C \). For nodes that are not leaf nodes, the algorithm looks through all possible successor states \( s' \) by looping through all possible actions. Any successor state that does not satisfy the constraints \( (V_C(s') < \alpha) \), is not considered. For those states that do satisfy \( V_C(s') < \alpha \), the Expand routine is called recursively to get the expected reward and the expected constraint penalty for that state. The action that provides the best reward \( V_R \) is returned as the best action to execute in the current state.

There are several complications that arise in the above solution technique when applied to multiple agents and dynamic environments. First, the size of the state and action spaces grow exponentially in the number of agents. An on-line search becomes computationally expensive as the branching factor of the search tree becomes large, in this case \( |A|^N \) where \( N \) is the number of agents. Second, the above approach assumes that the constraint map of the environment is fixed. But this may not be the case in a dynamic environment. The ability to recompute \( U_C(s) \) quickly becomes important. For both these reasons, it is desirable to have a solution scheme that scales reasonably as the number of agents grows.

V. Proposed Solution

We approach the problem by first noting that the agents are transition independent. If the state vector can be decomposed into \( N \) subvectors \( s = [s_1^T, s_2^T, \ldots, s_N^T] \) and the action vector into subvectors \( a = [a_1^T, a_2^T, \ldots, a_N^T] \) such that \( p(s_i'|s_i, a_i, a_k) = p(s_i'|s_i, a_i) \) for all \( k \neq i \), the MDP \( < S, A, T, R > \) is said to be transition independent. In this case, \( s_i \) is the state of agent \( i \), and \( a_i \) is the action of agent \( i \). Therefore the multi-agent system under consideration is transition independent due to the fact that the agent dynamics are decoupled. Each agent’s individual transition model will henceforth be denoted \( T_i(s_i', a_i, s_i) \). The only coupling between the agents occurs in rewards and constraints. Becker, Zilberstein and Goldman [8] investigate reward coupling in the unconstrained case, and in this work we account for coupled constraints as well.

The constraint coupling between the agents is represented as follows. We define an event \( e_i \) as the tuple \( < S_i, A_i, S_i > \). Agent \( i \) is said to have experienced an event \( e_i \) when there is a transition from \( s_i \in S_i \) to \( s_i' \in S_i \) under the action \( a_i \in A_i \). Then, we define a joint event as the tuple \( < e_1, e_2, \ldots, e_N, JC > \) where \( e_i \) is an event associated with agent \( i \) and \( JC \) is the joint constraint penalty that is awarded to every agent when events \( e_1, \ldots, e_N \) all occur. Note that this is in addition to the penalty awarded by the constraint model \( C(s_i, a_i) \). We further define \( p_{ij}(\pi_i) \) as the probability that event \( e_i \) in the joint event \( j \) will occur when agent \( i \) follows policy \( \pi_i \). With these definitions, we can write the
constrained optimization problem from Eq. 2 and Eq. 3 as
\[ \pi^* = \arg \max_{\pi} E \left[ \sum_{t=0}^{T} \gamma^t R(s_t, \pi(s_t)) \right] \]  
(5)

s.t. \[ E \left[ \sum_{t=0}^{T} C(s_t, \pi(s_t)) \right] + \sum_{j=0}^{E} JC_j \prod_{i=0}^{N} p_{ij}(\pi_i) \leq \alpha \]  
(6)

Knowledge of the other agents’ policies is summarized in \( p_{ij}(\pi_i) \). The additional term added to the left side of the constraint (Equation 6) is the expected joint constraint penalty. Since one MDP must be solved for each agent, we have to solve \( N \) MDPs of size \( S \times A \), rather than one single MDP of size \( S^N \times A^N \). However, the policy for any agent depends on \( p_i(\pi_i) \) of all the other agents, which in turn depend on their policies. In the next section, we present an iterative scheme for finding the solutions for all agents given the mutual dependency described above.

The technique to be described below has two components - first, we compute a “library” of policies that give the minimum risk for chosen values of \( p_{ij}(\pi_i) \). This library is computed off-line. The agents then communicate with each other and decide which one of the policies from their libraries they will use as the minimum risk-to-go policy. This policy, and its corresponding value function, provide the libraries they will use as the minimum risk-to-go policy.

\( A. \) Off-line Library Computation and Joint Policy Iteration

First we need to compute the minimum constraint penalty \( U_C(s) \). Since we have multiple agents all potentially subject to different constraints, we let each agent \( i \) maintain its own individual \( U_C(s_i) \). By definition, \( U_C(s) \) satisfies

\[ U_C(s_i) = \min_{\pi_i} E \left[ \sum_{t=0}^{T} C(s_{it}, \pi_i(s_{it})) \right] + \sum_{j=0}^{E} JC_j \prod_{k=0}^{N} p_{kj}(\pi_k) \]  
(7)

In order to compute \( U_C(s_i) \) as shown above, it is essential to know the \( p_{kj}(\pi_k) \), i.e. the probability that the other agents \( k \) will perform their respective actions in event \( j \). Since these are not known, the above optimization problem is solved off-line by assuming some values for \( p_{kj}(\pi_k) \). The \( U_C(s_i) \) corresponding to those values of \( p_{kj}(\pi_k) \) are stored in a library \( U_C \). Once this library is created, the agent simply needs to look up the appropriate \( U_C(s_i) \in U_C \) when the other agents communicate their actual \( p_{kj}(\pi_k) \) values.

It is important to note here that the size of the library is crucial - for example, if a very dense set of sample \( p_{kj}(\pi_k) \) are used in computing the library, the library will be large but will also be more complete than for a smaller sample. If the \( p_{kj}(\pi_k) \) received do not correspond to any of the sample points, interpolation is used to find the appropriate \( U_C(s_i) \). However it was found that the number of policies that need to be maintained to span the entire space of \( p_{kj}(\pi_k) \) is finite and reasonably small. With two agents, a library size of not more than three policies per event was needed to span the space of \( p_{ij}(\pi_i) \).

Then, each agent picks one policy out of this library and computes the \( p_{ij}(\pi_i) \) for itself and communicates this value to all other agents. Since all the other agents perform the same procedure, agent \( i \) also receive new values for \( p_{kj}(\pi_k) \) for all other agents \( k \neq i \). Agent \( i \) then picks the policy in the library that is optimal for the new set of \( p_{kj}(\pi_k) \). This results in a new \( p_{ij}(\pi_i) \), and the process is repeated until all the agents converge on a policy. This is called the joint policy iteration and is shown in Algorithm 3. Once this process is completed (and it is known to converge, see [8]), the agents each have a policy \( \pi_i^* \) that minimizes the risk-to-go. The value function corresponding to this policy, \( U_C(s_i) \), is then used as the risk-to-go estimate during the on-line search. This process is described in the following.

\( B. \) on-line Search

During the on-line search (Algorithm 4), every agent maintains two quantities - the expected future reward \( V_R(s_i) \) and the expected future risk \( V_C(s_i) \). Both values are initialized to be zero. The \( Q \) values for every state \( s \) and action \( a \) are computed using the definition of expected reward:

\[ Q_R(s_i, a_i) = R(s_i, a_i) + \sum_{s' \in S} P_{ij}(s_i) V_R(s_i') \]  
(8)

\[ Q_C(s_i, a_i) = C(s_i, a_i) + \sum_{s' \in S} P_{ij}(s_i) V_C(s_i') \]  
(9)

\[ \pi_i(s) = \arg \max_{a_i \in A_i} Q_R(s_i, a_i) \]  
(10)

\[ a_{C_i} = \{ a_i : Q_C(s_i, a_i) < \alpha \} \]  
(11)

\[ V_R(s_i) = Q_R(s_i, \pi_i(s_i)) \]  
(12)

\[ V_C(s_i) = Q_C(s_i, \pi_i(s_i)) \]  
(13)

\( V_R(s_i') \) and \( V_C(s_i') \) are obtained by calling Equations 8–12 recursively. At the end of the search horizon (the end of the recursion) we use \( V_R(s_i') = 0 \) and \( V_C(s_i') = U_C(s_i') \).

In the process of optimizing the reward, the policy \( \pi_i(s_i) \) (Equation 10) is modified. In general, this will lead to a change in the value of \( p_{ij} \), which will then require all the agents to repeat the joint policy iteration. Since doing so at each node in the forward expansion can be computationally expensive, we use a heuristic instead. Each time the policy is modified (Equation 10), we compute the new \( p'_{ij} \). If \( p'_{ij} > p_{ij} \), we discard that action – we do not allow the agent to take more risk than was announced to the rest of the team. Since the joint constraint penalty \( \sum_{t=0}^{T} JC_j \prod_{k=0}^{N} p_{kj}(\pi_k) \) is proportional to \( p_{kj}(\pi_k) \), the constraint-penalty-to-go estimate \( U_C(s_i) \) that was computed previously becomes an overestimate and remains valid without repeating the joint policy iteration. This ensures that the joint constraint is not violated despite a unilateral change in policy. The disadvantage is
Algorithm 3 Joint policy iteration for computing the minimum constraint-penalty-to-go

1: Function JointPolicyIteration(U_{C1})
2: Initialize p_{k} = 1/|k|, p'_{k} = 0
3: while p'_{k} \neq p_{k} do
4: \pi_{C1} = U_{C1}(p_{k})
5: p'_{ij} = ComputeP(\pi_{C1})
6: if p'_{ij} \neq p_{ij} then
7: Broadcast p'_{ij}
8: p_{ij} = p'_{ij}
9: end if
10: Receive p'_{kj}
11: end while
12: return \pi_{C1}

Algorithm 4 Decentralized Constrained MDP Algorithm

1: Function Expand(s_{i}, U_{C}, D)
2: if D = 0 then
3: V_C(s_{i}) = U_{C_{i}}(s_{i}); V_R(s_{i}) = 0; \pi_{i}(s_{i}) = \pi_{C_{i}}(s_{i})
4: return \pi_{i}(s_{i}), V_R(s_{i}), V_C(s_{i})
5: else
6: for a_{i} \in A_{i} do
7: [V_R(s_{i}', V_C(s_{i}'), \pi_{i}(s_{i}'))] = Expand(s_{i}', U_{C_{i}}, D - 1)
8: Q_{R}(s_{i}, a_{i}) = \sum_{s_{i}'} V_{R}(s_{i}') P(s_{i}', s_{i}, a_{i}) + R(s_{i}, a_{i})
9: Q_{C}(s_{i}, a_{i}) = \sum_{s_{i}'} V_{C}(s_{i}') P(s_{i}', s_{i}, a_{i}) + C(s_{i}, a_{i})
10: \pi_{i}^{*}(t_{i}) = \pi_{i}(s_{i}) \forall t_{i} \neq s_{i}
11: \pi_{i}^{*}(s_{i}) = a_{i}
12: p_{ij}(a_{i}) = ComputeP(\pi_{i}^{*}) \forall j
13: end for
14: a_{C_{i}} = \{a_{i} : Q_{C}(s_{i}, a_{i}) < \alpha \text{ AND } p_{ij}(a_{i}) < p_{ij}\}
15: \pi_{i}^{*}(s_{i}) = \arg\max_{a_{i} \in a_{C_{i}}} Q_{R}(s_{i}, a_{i})
16: V_{R}(s_{i}) = Q_{R}(s_{i}, \pi_{i}^{*}(s_{i}))
17: V_{C}(s_{i}) = Q_{C}(s_{i}, \pi_{i}^{*}(s_{i}))
18: return \pi_{i}^{*}(s_{i}), V_{R}(s_{i}), V_{C}(s_{i})
19: end if

the additional conservatism introduced - if an agent finds an opportunity to increase reward by taking more risk that opportunity cannot be exploited. Despite this conservatism, the main computational advantage of the proposed method are significant - we only need to solve N MDPs of size S \times A, instead of one MDP of size S^{N} \times A^{N}. In practice, the proposed method achieves good performance. These results are presented next.

VI. RESULTS

This section presents simulation results that compare the performance of the on-line Constrained MDP algorithm presented here with that of a standard MDP and an off-line centralized Constrained MDP solver. The problem layout is characterized by three quantities - the constraint density, the reward density and the obstacle density. The constraint density is the fraction of states that are constrained states, the reward density is the fraction of states that are reward states, and the obstacle density is the fraction of states that are blocked by obstacles. The goal is to maximize the rewards gathered while keeping the risk at \alpha < 0.15. The transition model is shown in Figure 1 – the probability of executing the intended action is 0.9. To enable a comparison with centralized MDP solutions, the problem size is restricted to two agents.

First, we use a standard MDP with constraints treated as penalties in the reward. The shortcoming of this method is that as the penalty is increased, the MDP solution becomes more conservative, taking less risk at the cost of a lower reward, whereas when the penalty is lowered, the solution becomes risky. Furthermore, this trade-off is determined by the constraint density - a low constraint density would have very little effect on the MDP solution, whereas a high constraint density would have a significant impact. The second technique that explicitly accounts for risk is a centralized, constrained MDP value iteration technique. This method solves the following optimization problem:

\[
\pi^{*}(s) = \arg\max_{a_{i} \in a_{C_{i}}} [R(s, a) + \sum_{s'} T(s', s, a)V_{R}(s')] \\
V_{R}(s) = \max_{a_{i} \in a_{C_{i}}} [R(s, a) + \sum_{s'} T(s', s, a)V_{R}(s')] \\
a_{C_{i}} = \{a_{i} : Q_{C}(s_{i}, a_{i}) < \alpha\} \\
Q_{C}(s, a) = C(s, a) + \sum_{s'} T(s', s, a)V_{C}(s') \\
V_{C}(s) = Q_{C}(s, \pi^{*}(s))
\]

The main shortcoming of this method is that the plan is made off-line. It does not account for the fact that in most cases, risk is not realized. This method effectively uses accumulated risk, making the planner more conservative than the proposed method.

In order to compare the three methods, we generated problem layouts randomly for varying constraint densities while keeping the reward and obstacle densities fixed at \rho_{R} = \rho_{O} = 0.05. The placement of the rewards, constraints and obstacles was random. For each value of the constraint density, 40 such random problems were generated. The plans computed by each of the three methods were executed and average reward for each method computed. The fraction of runs in which a constraint was violated was also computed, and this value was compared against the specified risk of \alpha < 0.15. The results for the average reward and the risk are shown in Figures 2 and 3 respectively. First, we note that the reward for all three techniques decreases as the constraint density increases. This is because rewards become less accessible as the number of constrained states increases. Unless risk is explicitly accounted for and kept bounded, the risk correspondingly increases. This is seen in the results for Value Iteration. Figures 2 and 3 show that the reward is highest for MDP Value Iteration but the risk associated is extremely high. The Decentralized MDP technique proposed here achieves approximately 25% lower...
reward than MDP value iteration for any given value of the constraint density, but the risk remains bounded below the threshold of $\alpha \leq 0.15$. C-MDP Value Iteration also achieves the risk levels specified, but the reward is much lower than the Decentralized MDP due to the fact that unrealized risk from the past is still accounted for in planning. The results show that the Decentralized MDP obtains good reward while keeping the risk bounded. Finally, in Figure 4, we show that similar trends hold for larger teams, in this case five agents. The MDP solution is not shown due to the computational infeasibility of solving an MDP for that size team.

VII. CONCLUSIONS

We proposed a technique for planning in the presence of uncertainty and constraints. The problem is formulated as an MDP with constraints that must be satisfied with a specified probability, defined as risk. The bound on risk is achieved by maintaining two separate value functions, one for reward and another for risk. We begin by computing the minimum risk policy off-line, and use this minimum risk to estimate the risk-to-go during execution. The risk-to-go estimate is used to restrict the set of policies over which we search for the maximum reward. We address the issue of scaling by noticing that the dynamics of individual agents are decoupled. The resulting simplification is used to speed up the algorithm. Simulation results show that algorithm performs well when compared to other methods such as value iteration and constrained value iteration.

VIII. ACKNOWLEDGMENTS

This research was supported by the Office of Naval Research under grant number N00014-07-1-0749.

REFERENCES