On Opportunistic Techniques for Solving Decentralized Markov Decision Processes with Temporal Constraints

Janusz Marecki and Milind Tambe
Computer Science Department
University of Southern California
941 W 37th Place, Los Angeles, CA 90089
{marecki, tambe}@usc.edu

ABSTRACT
Decentralized Markov Decision Processes (DEC-MDPs) are a popular model of agent-coordination problems in domains with uncertainty and time constraints but very difficult to solve. In this paper, we improve a state-of-the-art heuristic solution method for DEC-MDPs, called OC-DEC-MDP, that has recently been shown to scale up to larger DEC-MDPs. Our heuristic solution method, called Value Function Propagation (VFP), combines two orthogonal improvements of OC-DEC-MDP. First, it speeds up OC-DEC-MDP by an order of magnitude by maintaining and manipulating a value function for each state (as a function of time) rather than a separate value for each pair of state and time interval. Furthermore, it achieves better solution qualities than OC-DEC-MDP because, as our experimental results show, it does not overestimate the expected total reward like OC-DEC-MDP. We test both improvements independently in a crisis-management domain as well as for other types of domains. Our experimental results demonstrate a significant speedup of VFP over OC-DEC-MDP as well as higher solution qualities in a variety of situations.

Categories and Subject Descriptors
I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence - Multi-agent Systems

General Terms
Algorithms, Theory

Keywords
Multi-agent systems, Decentralized Markov Decision Process, Temporal Constraints, Locally Optimal Solution

1. INTRODUCTION
The development of algorithms for effective coordination of multiple agents acting as a team in uncertain and time critical domains has recently become a very active research field with potential applications ranging from coordination of agents during a hostage rescue mission [11] to the coordination of Autonomous Mars Exploration Rovers [2]. Because of the uncertain and dynamic characteristics of such domains, decision-theoretic models have received a lot of attention in recent years, mainly thanks to their expressiveness and the ability to reason about the utility of actions over time.

Key decision-theoretic models that have become popular in the literature include Decentralized Markov Decision Processes (DEC-MDPs) and Decentralized, Partially Observable Markov Decision Processes (DEC-POMDPs). Unfortunately, solving these models optimally has been proven to be NEXP-complete [3], hence more tractable subclasses of these models have been the subject of intensive research. In particular, Network Distributed POMDP [13] which assume that not all the agents interact with each other, Transition Independent DEC-MDP [2] which assume that transition function is decomposable into local transition functions or DEC-MDP with Event Driven Interactions [1] which assume that interactions between agents happen at fixed time points constitute good examples of such subclasses. Although globally optimal algorithms for these subclasses have demonstrated promising results, domains on which these algorithms run are still small and time horizons are limited to only a few time ticks.

To remedy that, locally optimal algorithms have been proposed [12] [4] [5]. In particular, Opportunity Cost DEC-MDP [4] [5], referred to as OC-DEC-MDP, is particularly notable, as it has been shown to scale up to domains with hundreds of tasks and double digit time horizons. Additionally, OC-DEC-MDP is unique in its ability to address both temporal constraints and uncertain method execution durations, which is an important factor for real-world domains. OC-DEC-MDP is able to scale up to such domains mainly because instead of searching for the globally optimal solution, it carries out a series of policy iterations; in each iteration it performs a value iteration that reuses the data computed during the previous policy iteration. However, OC-DEC-MDP is still slow, especially as the time horizon and the number of methods approach large values. The reason for high runtimes of OC-DEC-MDP for such domains is a consequence of its huge state space, i.e., OC-DEC-MDP introduces a separate state for each possible pair of method and method execution interval. Furthermore, OC-DEC-MDP overestimates the reward that a method expects to receive for enabling the execution of future methods. This reward, also referred to as the opportunity cost, plays a crucial role in agent decision making, and as we show later, its overestimation leads to highly suboptimal policies.

In this context, we present VFP (= Value Function Propagation), an efficient solution technique for the DEC-MDP model with temporal constraints and uncertain method execution durations, that builds on the success of OC-DEC-MDP. VFP introduces our two orthogonal ideas: First, similarly to [7] [9] and [10], we maintain
2. MOTIVATING EXAMPLE

We are interested in domains where multiple agents must coordinate their plans over time, despite uncertainty in plan execution duration and outcome. One example domain is large-scale disaster, like a fire in a skyscraper. Because there can be hundreds of civilians scattered across numerous floors, multiple rescue teams have to be dispatched, and radio communication channels can quickly get saturated and useless. In particular, small teams of fire-brigades must be sent on separate missions to rescue the civilians trapped in dozens of different locations.

Picture a small mission plan from Figure 1, where three fire-brigades have been assigned a task to rescue the civilians trapped at site B, accessed from site A (e.g. an office accessed from the fire lift). General fire fighting procedures involve both: (i) putting out the flames, and (ii) ventilating the site to let the toxic, high temperature gases escape, with the restriction that ventilation should not be performed too fast in order to prevent the fire from spreading. The team estimates that the civilians have 20 minutes before the fire at site B becomes unbearable, and that the fire at site A has to be put out in order to open the access to site B. As has happened in the past in large scale disasters, communication often breaks down; and hence we assume in this domain that there is no communication between the fire-brigades 1,2 and 3 (denoted as FB1, FB2 and FB3). Consequently, FB2 does not know if it is already safe to ventilate site A, FB1 does not know if it is already safe to enter site A and start fighting fire at site B, etc. We assign the reward 50 for evacuating the civilians from site B, and a smaller reward 20 for the successful ventilation of site A, since the civilians themselves might succeed in breaking out from site B.

One can clearly see the dilemma, that FB2 faces: It can only estimate the durations of the “Fight fire at site A” methods to be executed by FB1 and FB3, and at the same time FB2 knows that time is running out for civilians. If FB2 ventilates site A too early, the fire will spread out of control, whereas if FB2 waits too long, the ventilation strategy for too long, fire at site B will become unbearable for the civilians. In general, agents have to perform a sequence of such difficult decisions; in particular, decision process of FB2 involves first choosing when to start ventilating site A, and then (depending on the time it took to ventilate site A) choosing when to start evacuating the civilians from site B. Such sequence of decisions constitutes the policy of an agent, and it must be found fast because time is running out.

3. MODEL DESCRIPTION

We encode our decision problems in a model which we refer to as Decentralized MDP with Temporal Constraints (DC-MDP). Each instance of our decision problems can be described as a tuple \(\langle M, A, C, P, R \rangle\) where \(M = \{m_i\}_{i=1}^{|M|}\) is the set of methods, and \(A = \{A_k\}_{k=1}^{|A|}\) is the set of agents. Agents cannot communicate during mission execution. Each agent \(A_k\) is assigned to a set \(M_k\) of methods, such that \(\bigcup_{k=1}^{|A|} M_k = M\) and \(\forall i,j \in |M|, M_i \cap M_j = \emptyset\). Also, each method of agent \(A_k\) can be executed only once, and agent \(A_k\) can execute only one method at a time. Method execution times are uncertain and \(P = \{p_i\}_{i=1}^{|M|}\) is the set of distributions of method execution durations. In particular, \(p_i(t)\) is the probability that the execution of method \(m_i\) consumes time \(t\). \(C\) is a set of temporal constraints in the system. Methods are partially ordered and each method has fixed time windows inside which it can be executed, i.e., \(C = C_{\prec} \cup C_{\not{\prec}}\) where \(C_{\prec}\) is the set of predecessor constraints and \(C_{\not{\prec}}\) is the set of time window constraints. For \(c \in C_{\prec}\), \(c = (m_i, m_j)\) means that method \(m_i\) precedes method \(m_j\), i.e., execution of \(m_j\) cannot start before \(m_i\) terminates. In particular, for an agent \(A_k\), all its methods form a chain linked by predecessor constraints. We assume, that the graph \(G = (M, C)\) is acyclic, does not have disconnected nodes (the problem cannot be decomposed into independent subproblems), and its source and sink vertices identify the source and sink methods of the system.

For \(c \in C_{\not{\prec}}\), \(c = (m_i, EST, LET)\) means that execution of \(m_i\) can only start after the Earliest Starting Time \(EST\) and must finish before the Latest End Time \(LET\); we allow methods to have multiple disjoint time window constraints. Although distributions \(p_i\) can extend to infinite time horizons, given the time window constraints, the planning horizon \(\Delta = \max(m_i, \tau, \tau_C)\) is considered as the mission deadline. Finally, \(R = \{r_i\}_{i=1}^{|M|}\) is the set of non-negative rewards, i.e., \(r_i\) is obtained upon successful execution of \(m_i\).

Since there is no communication allowed, an agent can only estimate the probabilities that its methods have already been enabled

---

1 We explain the EST and LET notation in section 3

2 One could also use the OC-DEC-MDP framework, which models both time and resource constraints
by other agents. Consequently, if \( m_j \in M_k \) is the next method to be executed by the agent \( A_k \) and the current time is \( t \in [0, \Delta] \), the agent has to make a decision whether to Execute the method \( m_j \) (denoted as \( E \)), or to Wait (denoted as \( W \)). In case agent \( A_k \) decides to wait, it remains idle for an arbitrary small time \( \epsilon \), and resumes operation at the same place (= about to execute method \( m_j \)) at time \( t + \epsilon \). In case agent \( A_k \) decides to Execute the next method, two outcomes are possible:

**Success:** The agent \( A_k \) receives reward \( r_j \) and moves on to its next method (if such method exists) so long as the following conditions hold: (i) All the methods \( \{m_i, \langle m_i, m_j \rangle \in C_{-\epsilon} \} \) that directly enable method \( m_j \) have already been completed, (ii) Execution of method \( m_j \) started in some time window of method \( m_i \), i.e., \( \exists \langle m_i, \tau, \tau' \rangle \in C_{-\epsilon} \) such that \( t \in [\tau, \tau'] \), and (iii) Execution of method \( m_j \) finished inside the same time window, i.e., agent \( A_k \) completed method \( m_j \) in time less than or equal to \( \tau' - \epsilon \).

**Failure:** If any of the above-mentioned conditions does not hold, agent \( A_k \) stops its execution. Other agents may continue their execution, but methods \( m_k \in \{m_i, \langle m_i, m_j \rangle \in C_{\epsilon} \} \) will never become enabled.

The policy \( \pi_k \) of an agent \( A_k \) is a function \( \pi_k : M_k \times [0, \Delta] \rightarrow \{W, E\} \) and \( \pi_k(m, t) = a \) means, that if \( A_k \) is at method \( m \) at time \( t \), it will choose to perform the action \( a \). A joint policy \( \pi = [\pi_k]_{k=1}^{K} \) is considered to be optimal (denoted as \( \pi^* \)), if it maximizes the sum of expected rewards for all the agents.

### 4. SOLUTION TECHNIQUES

#### 4.1 Optimal Algorithms

Optimal joint policy \( \pi^* \) is usually found by using the Bellman update principle, i.e., in order to determine the optimal policy for method \( m_j \), optimal policies for methods \( m_i \in \{m, \langle m, m_j \rangle \in C_{\Delta}\} \) are used. Unfortunately, for our model, the optimal policy for method \( m_j \) also depends on policies for methods \( m_i \in \{m, \langle m, m_j \rangle \in C_{\Delta}\} \). This double dependency results from the fact, that the expected reward for starting the execution of method \( m_j \) at time \( t \) also depends on the probability that method \( m_j \) will be enabled by time \( t \). Consequently, if time is discretized, one needs to consider \( \Delta \) candidate policies in order to find \( \pi^* \). Thus, globally optimal algorithms used for solving real-world problems are unlikely to terminate in reasonable time [11]. The complexity of our model could be reduced if we considered its more restricted version; in particular, if each method \( m_i \) was allowed to be enabled at time points \( t \in T_j \subset [0, \Delta] \), the Coverage Set Algorithm (CSA) [1] could be used. However, CSA complexity is double exponential in the size of \( T_j \), and for our domains \( T_j \) can store all values ranging from 0 to \( \Delta \).

#### 4.2 Locally Optimal Algorithms

Following the limited applicability of globally optimal algorithms for DEC-MDPs with Temporal Constraints, locally optimal algorithms appear more promising. Specially, the OC-DEC-MDP algorithm [4] is particularly significant, as it has shown to easily scale up to domains with hundreds of methods. The idea of the OC-DEC-MDP algorithm is to start with the earliest starting time policy \( \pi^0 \) (according to which an agent will start executing the method as soon as \( m \) has a non-zero chance of being already enabled), and then improve it iteratively, until no further improvement is possible. At each iteration, the algorithm starts with some policy \( \pi \), which uniquely determines the probabilities \( P_{[\tau, \tau']} \) that method \( m_i \) will be performed in the time interval \( [\tau, \tau'] \). It then performs two steps:

**Step 1:** It propagates from sink methods to source methods the values \( V_{[\tau, \tau']} \), that represent the expected utility for executing method \( m_i \) in the time interval \( [\tau, \tau'] \). This propagation uses the probabilities \( P_{[\tau, \tau']} \) from previous algorithm iteration. We call this step a value propagation phase.

**Step 2:** Given the values \( V_{[\tau, \tau']} \) from Step 1, the algorithm chooses the most profitable method execution intervals which are stored in a new policy \( \pi' \). It then propagates the new probabilities \( P_{[\tau, \tau']} \) from source methods to sink methods. We call this step a probability propagation phase. If policy \( \pi' \) does not improve \( \pi \), the algorithm terminates.

There are two shortcomings of the OC-DEC-MDP algorithm that we address in this paper. First, each of OC-DEC-MDP states is a pair \( \{m_j, [\tau, \tau']\} \), where \( [\tau, \tau'] \) is a time interval in which method \( m_j \) can be executed. While such state representation is beneficial, in that the problem can be solved with a standard value iteration algorithm, it blurs the intuitive mapping from time \( t \) to the expected total reward for starting the execution of \( m_j \) at time \( t \). Consequently, if some method \( m_j \) enables method \( m_j \), and the values \( V_{[\tau, \tau']} \) are known, the operation that calculates the values \( V_{[\tau, \tau']} \) runs in time \( O(I^2) \), where \( I \) is the number of time intervals [11]. Since the runtime of the whole algorithm is proportional to the runtime of this operation, especially for big time horizons \( \Delta \), the OC-DEC-MDP algorithm runs slow.

Second, while OC-DEC-MDP emphasizes on precise calculation of values \( V_{[\tau, \tau']} \), it fails to address a critical issue that determines how the values \( V_{[\tau, \tau']} \) are split given that the method \( m_j \) has multiple enabling methods. As we show later, OC-DEC-MDP splits \( V_{[\tau, \tau']} \) into parts that may overestimate \( V_{[\tau, \tau']} \) when summed up again. As a result, methods that precede the method \( m_j \) overestimate the value for enabling \( m_j \), which, as we show later, can have disastrous consequences. In the next two sections, we address both of these shortcomings.

### 5. VALUE FUNCTION PROPAGATION (VFP)

The general scheme of the VFP algorithm is identical to the OC-DEC-MDP algorithm, in that it performs a series of policy improvement iterations, each one involving a Value and Probability Propagation Phase. However, instead of propagating separate values, VFP maintains and propagates the whole functions, we therefore refer to these phases as the value function propagation phase and the probability function propagation phase. To this end, for each method \( m_i \in M \), we define three new functions:

**Value Function**, denoted as \( v_i(t) \), that maps time \( t \in [0, \Delta] \) to the expected total reward for starting the execution of method \( m_i \) at time \( t \).

**Opportunity Cost Function**, denoted as \( V(t) \), that maps time \( t \in [0, \Delta] \) to the expected total reward for starting the execution of method \( m_i \) at time \( t \) assuming that \( m_i \) is enabled.

**Probability Function**, denoted as \( P(t) \), that maps time \( t \in [0, \Delta] \) to the probability that method \( m_i \) will be completed before time \( t \).

Such functional representation allows us to easily read the current policy, i.e., if an agent \( A_k \) is at method \( m_i \) at time \( t \), then it will wait as long as value function \( v_i(t) \) will be greater in the future.

Formally:

\[
\pi_k(\langle m_i, t \rangle) = \begin{cases} W & \text{if } 3\tau_{m_{i+1}} < t \text{ and } v_i(t) < v_i(t') \\ E & \text{otherwise} \end{cases}
\]

We now develop an analytical technique for performing the value function and probability function propagation phases.\[1\]Similarly for the probability propagation phase
5.1 Value Function Propagation Phase

Suppose, that we are performing a value function propagation phase during which the value functions are propagated from the sink methods to the source methods. At any time during this phase we encounter a situation shown in Figure 2, where opportunity cost functions \( \{V_{n,0}\}_{n=0}^{N} \) of methods \( \{m_{n}\}_{n=0}^{N} \) are known, and the opportunity cost \( V_{n,0} \) of method \( m_{n} \) is to be derived. Let \( p_{n} \) be the probability distribution function of method \( m_{n} \), execution duration, and \( r_{n} \) be the immediate reward for starting and completing the execution of method \( m_{n} \) inside a time interval \([\tau, \tau']\) such that \( \{m_{n}, \tau, \tau'\} \in C_{1} \). The function \( V_{n} \) is then derived from \( r_{n} \) and opportunity costs \( V_{n,0} \) of \( n = 1, \ldots, N \) from future methods. Formally:

\[
V_{n} (t) = \begin{cases} 
\tau' - t \cdot p_{n} (t) \left( r_{n} + \sum_{m=0}^{N} V_{m,0} (t + \tau') \right) & \text{if } \exists (m_{n}, \tau, \tau') \in C_{1} \text{ such that } t \in [\tau, \tau'] \setminus \{0\} \\
0 & \text{otherwise} 
\end{cases}
\]

Note, that for \( t \in [\tau, \tau'] \), if \( h(t) := r_{n} + \sum_{m=0}^{N} V_{m,0} (t + \tau') \) then \( V_{n} \) is a convolution of \( p \) and \( h \): \( V_{n} (t) = (p * h) (\tau' - t) \) then \( V_{n} \) is a convolution of \( p \) and \( h \): \( V_{n} (t) = (p * h) (\tau' - t) \).

Assume now, that \( V_{n,0} \) represents a full opportunity cost, postponing the discussion on different techniques for splitting the opportunity cost \( V_{n} \) into \( \{V_{n}, v_{n}\}_{n=0}^{N} \) until section 6. We now show how to derive \( V_{n,0} \) (derivation of \( V_{n,0} \) for \( n \neq 0 \) follows the same scheme).

Knowing the opportunity cost \( V_{n} \), we can then easily derive the value function \( v_{n} \). Let \( A_{n} \) be an agent assigned to the method \( m_{n} \). If \( A_{n} \) is about to start the execution of \( m_{n} \), it means, that \( A_{n} \) must have completed its part of the mission plan up to the method \( m_{n} \). Since \( A_{n} \) does not know if other agents have completed methods \( \{m_{k}\}_{k=0}^{K} \), in order to derive \( v_{n} \), it has to multiply \( V_{n} \) by the probability functions of all methods of other agents that enable \( m_{n} \). Formally:

\[
v_{n} (t) = V_{n} (t) \prod_{k=1}^{K} P_{k} (t)
\]

Where the dependency of \( \{P_{k}\}_{k=1}^{K} \) is also ignored.

We have consequently shown a general scheme how to propagate the value functions: Knowing \( \{v_{n}\}_{n=0}^{N} \) and \( \{V_{n,0}\}_{n=0}^{N} \) of methods \( \{m_{n}\}_{n=0}^{N} \) we can derive \( v_{n} \) and \( V_{n} \) of method \( m_{n} \). In general, the value function propagation scheme starts with sink nodes. It then visits at each time a method \( m_{n} \) such that all the methods that \( m_{n} \) enables have already been marked as visited. The value function propagation phase terminates when all the source methods have been marked as visited.

5.2 Reading the Policy

In order to determine the policy of agent \( A_{k} \) for the method \( m_{j} \), we must identify the set \( Z_{j} \) of intervals \([z, z'] \subseteq [0, \ldots, \Delta]\), such that:

\[
\forall (k, z, z') \exists (m_{j}, t) \in W.
\]

One can easily identify the intervals of \( Z_{j} \) by looking at the time intervals in which the value function \( v_{j} \) does not decrease monotonically.

5.3 Probability Function Propagation Phase

Assume now, that value functions and opportunity cost values have all been propagated from sink methods to source nodes and the sets \( Z \), for all methods \( m_{j} \in M \) have been identified. Since value function propagation phase was using probabilities \( P_{j} (t) \) for methods \( m_{j} \in M \) and times \( t \in [0, \Delta] \) found at previous algorithm iteration, we now have to find new values \( P_{j} (t) \), in order to prepare the algorithm for its next iteration. We now show how in the general case (Figure 2) propagate the probability functions forward through one method, i.e., we assume that the probability functions \( \{P_{k}\}_{k=0}^{K} \) of methods \( \{m_{k}\}_{k=0}^{K} \) are known, and the probability function \( P_{j} \) of method \( m_{j} \) must be derived. Let \( p_{j} \) be the probability distribution function of method \( m_{j} \), execution duration, and \( Z_{j} \) be the set of intervals of inactivity for method \( m_{j} \), found during the last value function propagation phase. If we ignore the dependency of \( \{P_{k}\}_{k=0}^{K} \) then the probability \( P_{j} (t) \) that the execution of method \( m_{j} \) starts before time \( t \) is given by:

\[
P_{j} (t) = \prod_{k=0}^{K} P_{k} (\tau) \text{ if } \exists (\tau, \tau') \in Z_{j} \text{ s.t. } t \in (\tau, \tau') \prod_{k=0}^{K} P_{k} (t) \text{ otherwise.}
\]

Given \( P_{j} (t) \), the probability \( P_{j} (t) \) that method \( m_{j} \) will be completed by time \( t \) is derived by:

\[
P_{j} (t) = \int_{0}^{t} \int_{t'}^{t} \left( \frac{\partial P_{j}}{\partial t'} \right) dt'' \cdot P_{j} (t' - t'') dt'' + dt'
\]

Which can be written compactly as \( \frac{\partial P_{j}}{\partial t} = P_{j} * \sigma_{j} \).

We have consequently shown how to propagate the probability functions \( \{P_{k}\}_{k=0}^{K} \) of methods \( \{m_{k}\}_{k=0}^{K} \) to obtain the probability function \( P_{j} \) of method \( m_{j} \). The general, the probability function propagation phase starts with source methods \( m_{k} \) for which we know that \( P_{k} = 1 \) since they are enabled by default. We then visit at each time a method \( m \) such that all the methods that enable

Figure 2: Fragment of an MDP of agent \( A_{k} \). Probability functions propagate forward (left to right) whereas value functions propagate backward (right to left).

Let \( \Gamma_{j,0} (t) \) be the opportunity cost of starting the execution of method \( m_{j} \) at any time \( t \) that \( m_{j} \) has been completed. It is derived by multiplying \( V_{j,0} \) by the probability functions of all methods other than \( m_{j} \) that enable \( m_{j} \). Formally:

\[
\Gamma_{j,0} (t) = V_{j} (t) \prod_{k=1}^{K} P_{k} (t).
\]

Where similarly to [4] and [5] we ignored the dependency of \( \{P_{k}\}_{k=0}^{K} \) on \( t \). Observe that \( \Gamma_{j,0} \) does not have to be monotonically decreasing, i.e., delaying the execution of the method \( m_{j} \) can sometimes be profitable. Therefore the opportunity cost \( \Gamma_{j,0} (t) \) of enabling method \( m_{j} \) at time \( t \) must be greater than or equal to \( V_{j,0} (t) \). Furthermore, \( \Gamma_{j,0} \) should be non-increasing. Formally:

\[
\Gamma_{j,0} (t) = \min_{f} f (t) \text{ s.t. } \forall f \in F
\]

Where \( F = \{ f | f \geq \Gamma_{j,0} \text{ and } f (t) \geq f (t') \forall t' \} \).
have already been marked as visited. The probability function propagation phase terminates when all the sink methods have been marked as visited.

5.4 The Algorithm

Similarly to the OC-DEC-MDP algorithm, VFP starts the policy improvement iterations with the earliest starting time policy $π^0$. Then at each iteration it: (i) Propagates the value functions $[v_i]_{i=1}^m$ using the old probability functions $[P_i]_{i=1}^m$ from previous algorithm iteration and establishes the new sets $[Z_i]_{i=1}^m$ of method inactivity intervals, and (ii) propagates the new probability functions $[P_i]_{i=1}^m$ using the newly established sets $[Z_i]_{i=1}^m$. These new functions $[P_i]_{i=1}^m$ are then used in the next iteration of the algorithm. Similarly to OC-DEC-MDP, VFP terminates if a new policy does not improve the policy from the previous algorithm iteration.

5.5 Implementation of Function Operations

So far, we have derived the functional operations for value function and probability function propagation without choosing any function representation. In general, our functional operations can handle continuous time, and one has freedom to choose a desired function approximation technique, such as piecewise linear [7] or piecewise constant [9] approximation. However, since one of our goals is to compare VFP with the existing OC-DEC-MDP algorithm, that works only for discrete time, we also discretize time, and choose to approximate value functions and probability functions with piecewise linear (PWL) functions.

When the VFP algorithm propagates the value functions and probability functions, it constantly carries out operations represented by equations (1) and (3) and we have already shown that these operations are convolutions of some functions $p(t)$ and $b(t)$. If time is discretized, functions $p(t)$ and $b(t)$ are discrete; however, $h(t)$ can be nicely approximated with a PWL function $h(t)$, which is exactly what VFP does. As a result, instead of performing $O(Δ^2)$ multiplications to compute $f(t)$, VFP only needs to perform $O(k · Δ)$ multiplications to compute $f(t)$, where $k$ is the number of linear segments of $h(t)$ (note, that since $h(t)$ is monotonic, $h(t)$ is usually close to $h(t)$ with $k < Δ$). Since $P_i$ values are in range $[0,1]$ and $V_i$ values are in range $[0,\sum_{m\in M} r_i]$, we suggest to approximate $V_i(t)$ with $\tilde{V}_i(t)$ within error $ε_V$, and $P_i(t)$ with $\tilde{P}_i(t)$ within error $ε_P$. We now prove that the overall approximation error accumulated during the value function propagation phase can be expressed in terms of $ε_P$ and $ε_V$:

**Theorem 1.** Let $C_\prec = \{m \in M | \forall j \in K : m \not\prec j\}$ be a set of precedence constraints of a DEC-MDP with Temporal Constraints, and $ε_P$ and $ε_V$ be the probability function and value function approximation errors respectively. The overall error $ε = \max_V \sup_{t \in [0,\Delta]} [V(t) - \tilde{V}(t)]$ of value function propagation phase is then bounded by:

$$ |C_\prec| \left( ε_V + ((1 + ε_P)C_\prec - 1) \sum_{m\in M} r_i \right) .$$

**Proof.** In order to establish the bound for $ε_V$, we first prove by induction on the size of $C_\prec$, that the overall error of value function propagation phase, $ε(V) = \max_V \sup_{t \in [0,\Delta]} |P(t) - \tilde{P}(t)|$ is bounded by $(1 + ε_P)C_\prec - 1$.

**Induction base:** If $n = 1$ only two methods are present, and we will perform the operation identified by Equation (3) only once, introducing the error $|ε(V)| = ε_P = (1 + ε_P)C_\prec - 1$.

**Induction step:** Suppose, that $ε(V)_{m_1}$ for $|C_\prec| = n$ is bounded by $(1 + ε_P)^n - 1$, and we want to prove that this statement holds for $|C_\prec| = n + 1$. Let $G = (M, C_\prec)$ be a graph with at most $n + 1$ edges, and $\overline{G} = (M, \overline{C_\prec})$ be a subgraph of $G$, such that $\overline{C_\prec} = C_\prec - \{(m_j, m_i)\}$, where $m_j \in M$ is a sink node in $G$. From the induction assumption we have, that $C_\prec$ introduces the probability propagation phase error bounded by $(1 + ε_P)^{n} - 1$. We now add back the link $\{(m_i, m_j)\}$ to $\overline{C_\prec}$, which affects the error of only one probability function, namely $P_j$, by a factor of $(1 + ε_P)$. Since probability propagation phase error in $\overline{C_\prec}$ was bounded by $(1 + ε_P)^{n} - 1$, in $C_\prec = \overline{C_\prec} \cup \{(m_i, m_j)\}$ it can be at most $(1 + ε_P)^{n+1} - 1$. Thus, if opportunity cost functions are not underestimated, they are bounded by $\sum_{m\in M} P_i$, and the error of a single value function propagation operation will be at most

$$ \int_0^t p(t)(ε_V + ((1 + ε_P)C_\prec - 1) \sum_{m\in M} r_i) dt < ε_V + ((1 + ε_P)C_\prec - 1) \sum_{m\in M} r_i .$$

Since the number of value function propagation operations is $|C_\prec|$, the total error $ε_V$ of the value function propagation phase is bounded by:

$$ |C_\prec| \left( ε_V + ((1 + ε_P)C_\prec - 1) \sum_{m\in M} r_i \right) .$$

6. SPLITTING THE OPPORTUNITY COST FUNCTIONS

In section 5 we left out the discussion about how the opportunity cost function $V_{m_0}$ of method $m_{0}$ is split into opportunity cost functions $V_{jk} = V_{j \rightarrow k}$ that directly enable method $m_{0}$. So far, we have taken the same approach as in [4] and [5] in that the opportunity cost function $V_{jk}$ that the method $m_{j}$ sends back to the method $m_{k}$ is a minimal, non-increasing function that dominates function $V_{jk}(t) = (V_{m_j} - \sum_{k \in K} P_{jk}(t))$. We refer to this approach, as heuristic $H_{(1,1)}$. Before we prove that this heuristic overestimates the opportunity cost, we discuss three problems that might occur when splitting the opportunity cost functions: (i) overestimation, (ii) underestimation and (iii) starvation. Consider the situation in Figure 3.

![Figure 3: Splitting the value function of method $m_{0}$ among methods $\{m_{i}\}_{i=1}^{K}$](image-url)

(3) when value function propagation for methods $\{m_{i}\}_{i=1}^{K}$ is performed. For each $k = 0, \ldots, K$, Equation (1) derives the opportunity cost function $V_{m_k}$ from immediate reward $r_k$ and opportunity cost function $V_{m_0,k}$. If $m_0$ is the only method that precedes method $m_k$, then $V_{m_0,k} = V_{m_0,0}$ is propagated to method $m_0$, and consequently the opportunity cost for completing the method $m_0$ at time $t$ is equal to $\sum_{k=0}^{K} V_{m_0,k}(t)$. If this cost is overestimated, then an agent $A_0$ at method $m_0$ will have too much incentive to finish the execution of $m_0$ at time $t$. Consequently, although the probability $P(t)$ that $m_0$ will be enabled by other agents at time $t$ is low, agent $A_0$ might still find the expected utility of starting the execution of $m_0$ at time $t$ higher than the expected utility of doing it later. As a result, it will choose at time $t$ to start executing method $m_0$ instead of waiting, which can have disastrous consequences. Similarly, if $\sum_{k=0}^{K} V_{m_0,k}(t)$ is underestimated, agent $A_0$ might lose interest in enabling the future methods $\{m_{i}\}_{i=1}^{K}$ and just focus on
maximizing the chance of obtaining its immediate reward, since this chance is increased when agent \( A_0 \) waits, it will consider at time \( t \) to be more profitable to wait, instead of starting the execution of \( m_{30} \), which can have similarly disastrous consequences. Finally, if \( V_{30} \) is split in a way, that for some \( k, V_{30,k} = 0 \), it is the method \( m_{30} \) that underestimates the opportunity cost of enabling method \( m_{30,k} \), and the similar reasoning applies. We call such problem a starvation of method \( m_{3k} \). That short discussion shows the importance of splitting the opportunity cost function \( V_{30} \) in such a way, that overestimation, underestimation, and starvation problem is avoided. We now prove that:

**Theorem 2:** Heuristic \( H_{(1,1)} \) can overestimate the opportunity cost.

**Proof.** We prove the theorem by showing a case where the overestimation occurs. For the mission plan from Figure (3), let \( H_{(1,1)} \) split \( V_{30} \) into \( V_{30,k} = \sum_{k' \in \{0,...,K\}} \Pi_{k'}(k) \) sent to methods \( \{m_{3k}\}_{k=0}^{K} \) respectively. Also, assume that methods \( \{m_{3k}\}_{k=0}^{K} \) provide no local reward and have the same time windows, i.e., \( r_{3k} = 0 \), \( EST_{3k} = 0 \), \( LET_{3k} = \Delta \) for \( k = 0, ..., K \). To prove the overestimation of opportunity cost, we must identify \( t_0 \in [0, \Delta] \) such that the opportunity cost \( \sum_{k=0}^{K} V_{30,k}(t) \) for methods \( \{m_{3k}\}_{k=0}^{K} \) at time \( t \in [0, \Delta] \) is greater than the opportunity cost \( V_{30}(t) \). From Equation (1) we have:

\[
V_{30}(t) = \int_0^{\Delta-t} p_{0}(t) V_{30,0}(t + t') dt'
\]

Summing over all methods \( \{m_{3k}\}_{k=0}^{K} \) we obtain:

\[
\sum_{k=0}^{K} V_{30,k}(t) = \sum_{k=0}^{K} \int_0^{\Delta-t} p_{0}(t) V_{30,0}(t + t') dt'
\]

\[
> \sum_{k=0}^{K} \int_0^{\Delta-t} p_{0}(t) V_{30,0}(t + t') dt'
\]

\[
= \sum_{k=0}^{K} \int_0^{\Delta-t} p_{0}(t) V_{30,0}(t + t') \prod_{k' \in \{0,...,K\}} \Pi_{k'}(t + t') dt'
\]

Let \( c \in (0,1) \) be a constant and \( t_0 \in [0, \Delta] \) be such that \( V_{30,t_0} > 0 \) and \( V_{30,0} \) non-increasing. Then:

\[
\sum_{k=0}^{K} V_{30,k}(t_0) > \sum_{k=0}^{K} \int_0^{\Delta-t_0} p_{0}(t') V_{30,0}(t_0 + t') \cdot c \cdot dt'
\]

But \( P_{30} \) is non-decreasing. Now, suppose there exists \( t_1 \in (t_0, \Delta) \) such that \( \sum_{k=0}^{K} \int_0^{t_1-t_0} p_{0}(t') dt' > \sum_{k=0}^{K} V_{30,k}(t_0) \). Since decreasing the upper limit of the integral over positive function also decreases the integral, we have:

\[
\sum_{k=0}^{K} V_{30,k}(t_0) > \sum_{k=0}^{K} \int_0^{t_1} p_{0}(t' - t_0) V_{30,0}(t_0 + t') dt'
\]

And since \( V_{30,0}(t') \) is non-increasing we have:

\[
\sum_{k=0}^{K} V_{30,k}(t_0) > c \cdot V_{30,0}(t_0) \sum_{k=0}^{K} \int_0^{t_1} p_{0}(t' - t_0) dt'
\]

\[
= c \cdot V_{30,0}(t_0) \sum_{k=0}^{K} \int_0^{t_1-t_0} p_{0}(t') dt'
\]

\[
> c \cdot V_{30,0}(t_0) \frac{V_{30,0}(t_0)}{V_{30,0}(t_0)} = V_{30,0}(t_0)
\]

Consequently, the opportunity cost \( \sum_{k=0}^{K} V_{30,k}(t_0) \) of starting the execution of methods \( \{m_{3k}\}_{k=0}^{K} \) at time \( t_0 \in [0, \Delta] \) is greater than the opportunity cost \( V_{30}(t_0) \) which proves the theorem. Figure 4 shows that the overestimation of opportunity cost is easily observable in practice.

To remedy the problem of opportunity cost overestimation, we propose three alternative heuristics that split the opportunity cost functions:

- **Heuristic \( H_{(1,0)} \):** Only one method, \( m_{3k} \) gets the full expected reward for enabling method \( m_{3j} \), i.e., \( V_{30,j-k}(t) = 0 \) for \( k' \in \{0, ..., K\} \setminus \{k\} \) and \( V_{30,j-k}(t) = (V_{30} \Pi_{k'}(t)) \).

- **Heuristic \( H_{(1/2,1)} \):** Each method \( m_{3k} \) gets the full opportunity cost for enabling method \( m_{3j} \), divided by the number \( K \) of methods enabling the method \( m_{3j} \), i.e., \( V_{30,j-k}(t) = \frac{1}{K} V_{30} \Pi_{k'}(t) \) for \( k' \in \{0, ..., K\} \).

- **Heuristic \( \tilde{H}_{(1,1)} \):** This is a normalized version of the \( H_{(1,1)} \) heuristic in that each method \( m_{3k} \) initially gets the full opportunity cost for enabling the method \( m_{3j} \). To avoid opportunity cost overestimation, we normalize the split functions when their sum exceeds the opportunity cost function to be split. Formally:

\[
V_{30,j-k}(t) = \begin{cases} V_{30,j-k}^{H_{(1,1)}}(t) & \text{if } \sum_{k' \in \{0, ..., K\}} V_{30,j-k}^{H_{(1,1)}}(t) < V_{30}(t) \\ V_{30,j-k}(t) & \text{otherwise} \end{cases}
\]

Where \( V_{30,j-k}^{H_{(1,1)}}(t) = (V_{30} \Pi_{k'}(t)) \).

For the new heuristics, we now prove that:

**Theorem 3:** Heuristics \( H_{(1,0)} \), \( H_{(1/2,1)} \) and \( \tilde{H}_{(1,1)} \) do not overestimate the opportunity cost.

**Proof.** When heuristic \( H_{(1,0)} \) is used to split the opportunity cost function \( V_{30} \), only one method (e.g., \( m_{3k} \)) gets the opportunity cost for enabling method \( m_{3j} \). Thus:

\[
\sum_{k'=0}^{K} V_{30,k'}(t) = \int_0^{\Delta-t} p_{0}(t') V_{30,0}(t + t') dt'
\]

And since \( V_{30} \) is non-increasing:

\[
\leq \int_0^{\Delta-t} p_{0}(t') V_{30}(t + t') \prod_{k' \in \{0,...,K\}} P_{30}(t + t') dt';
\]

\[
\leq \int_0^{\Delta-t} p_{0}(t') V_{30}(t + t') dt';
\]

The last inequality is also a consequence of the fact that \( V_{30} \) is non-increasing. For heuristic \( H_{(1/2,1)} \) we similarly have:

\[
\sum_{k'=0}^{K} V_{30,k'}(t) \leq \frac{1}{K} \int_0^{\Delta-t} p_{0}(t') V_{30}(t + t') \prod_{k' \in \{0,...,K\}} P_{30}(t + t') dt';
\]

\[
\leq \frac{1}{K} \int_0^{\Delta-t} p_{0}(t') V_{30}(t + t') dt';
\]

For heuristic \( \tilde{H}_{(1,1)} \), the opportunity cost function \( V_{30} \) is by definition split in such manner, that \( \sum_{k'=0}^{K} V_{30,k'}(t) \leq V_{30}(t) \). Consequently, we have proved, that our new heuristics \( H_{(1,0)}, H_{(1/2,1)} \) and \( \tilde{H}_{(1,1)} \) avoid the overestimation of the opportunity cost.
The reason why we have introduced all three new heuristics is the following: Since $H(1,1)$ overestimates the opportunity cost, one has to choose which method $m_{j0}$ will receive the reward for enabling the method $m_{j0}$, which is exactly what the heuristic $H(1,0)$ does. However, heuristic $H(1,0)$ leaves $K - 1$ methods that precede the method $m_{j0}$ without any reward which leads to starvation. Starvation can be avoided if opportunity cost functions are split using heuristic $H(1/2,1/2)$, that provides reward to all enabling methods. However, the sum of split opportunity cost functions for the $H(1/2,1/2)$ heuristic can be smaller than the non-zero split opportunity cost function for the $H(1,0)$ heuristic, which is clearly undesirable. Such situation (Figure 4, heuristic $H(1,0)$) occurs because the mean $\frac{f+g}{2}$ of two functions $f, g$ is not smaller than $f$ nor $g$ only if $f = g$. This is why we have proposed the $\bar{H}(1,1)$ heuristic, which by definition avoids the overestimation, underestimation and starvation problems.

7. EXPERIMENTAL EVALUATION

Since the VFP algorithm that we introduced provides two orthogonal improvements over the OC-DEC-MDP algorithm, the experimental evaluation performed consisted of two parts: In part 1, we tested empirically the quality of solutions that a locally optimal solver (either OC-DEC-MDP or VFP) finds, given it uses different opportunity cost function splitting heuristic, and in part 2, we compared the runtimes of the VFP and OC-DEC-MDP algorithms for a variety of mission plan configurations.

**Part 1:** We first ran the VFP algorithm on a generic mission plan configuration from Figure 3 where only methods $m_{j0}, m_{j1}, m_{j2}$ and $m_{j3}$ were present. Time windows of all methods were set to 400, duration $p_{j0}$ of method $m_{j0}$ was uniform, i.e., $p_{j0}(t) = \frac{\mu}{\tau}$ and durations $p_{j1}, p_{j2}$ of methods $m_{j1}, m_{j2}$ were normal distributions, i.e., $p_{j1} = N(\mu = 250, \sigma = 20)$, and $p_{j2} = N(\mu = 200, \sigma = 100)$. We assumed that only method $m_{j0}$ provided reward, i.e. $\tau_{j0} = 10$ was the reward for finishing the execution of method $m_{j0}$ before time $t = 400$. We show our results in Figure (4) where the x-axis of each of the graphs represents time whereas the y-axis represents the opportunity cost. The first graph confirms, that when the opportunity cost function $V_{j0}$ was split into opportunity cost functions $V_{j1}$ and $V_{j2}$ using the $H(1,1)$ heuristic, the function $V_{j1} + V_{j2}$ was not always below the $V_{j0}$ function. In particular, $V_{j1}(280) + V_{j2}(280)$ exceeded $V_{j0}(280)$ by 69%. Since heuristics $H(1/0), H(1/2,1/2)$ and $H(1,1)$ were used (graphs 2.3 and 4), the function $V_{j1} + V_{j2}$ was always below $V_{j0}$.

We then shifted our attention to the civilian rescue domain introduced in Figure 4 for which we sampled all action execution durations from the normal distribution $N = (\mu = 5, \sigma = 2)$. To obtain the baseline for the heuristic performance, we implemented a globally optimal solver, that found a true expected total reward for this domain (Figure 6a). We then compared this reward with a expected total reward found by a locally optimal solver guided by each of the discussed heuristics. Figure (6a), which plots which on the y-axis the expected total reward of a policy complements our previous results: $H(1,1)$ heuristic overestimated the expected total reward by 280% whereas the other heuristics were able to guide the locally optimal solver close to a true expected total reward.

**Part 2:** We then chose $H(1,1)$ to split the opportunity cost functions and conducted a series of experiments aimed at testing the scalability of VFP for various mission plan configurations, using the performance of the OC-DEC-MDP algorithm as a benchmark. We began the VFP scalability tests with a configuration from Figure (5a) associated with the civilian rescue domain, for which method execution durations were extended to normal distributions $N(\mu = 30, \sigma = 5)$, and the deadline was extended to $\Delta = 200$.

We decided to test the runtime of the VFP algorithm running with three different levels of accuracy, i.e., different approximation parameters $\epsilon_P$ and $\epsilon_V$ were chosen, such that the cumulative error of the solution found by VFP stayed within 1%, 5% and 10% of the solution found by the OC-DEC-MDP algorithm. We then ran both algorithms for a total of 100 policy improvement iterations. Figure (6b) shows the performance of the VFP algorithm in the civilian rescue domain (y-axis shows the runtime in milliseconds). As we see, for this small domain, VFP runs 15% faster than OC-DEC-MDP when computing the policy with an error of less than 1%. For comparison, the globally optimal solved did not terminate within the first three hours of its runtime which shows the strength of the opportunistic solvers, like OC-DEC-MDP.

We next decided to test how VFP performs in a more difficult domain, i.e., with methods forming a long chain (Figure 5b). We tested chains of 10, 20 and 30 methods, increasing at the same time method time windows to 350, 700 and 1050 to ensure that later methods can be reached. We show the results in Figure (7a), where we vary on the x-axis the number of methods and plot on the y-axis the algorithm runtime (notice the logarithmic scale). As we observe, scaling up the domain reveals the high performance of VFP. Within 1% error, it runs up to 6 times faster than OC-DEC-MDP.

We then tested how VFP scales up, given that the methods are arranged into a tree (Figure 5c). In particular, we considered trees with branching factor of 3 and depth of 2, 3 and 4, increasing at the same time the time horizon from 200 to 300, and then to 400. We show the results in Figure (7b). Although the speedups are smaller than in case of a chain, the VFP algorithm still runs up to 4 times faster than OC-DEC-MDP when computing the policy with an error of less than 1%.

We finally tested how VFP handles the domains with methods arranged into a $n \times n$ mesh, i.e., $C_x = \{ (m_{i,j}, m_{k,l,j}) \}$ for $i = 1, ..., n; k = 1, ..., n; j = 1, ..., n - 1$. In particular, we consider...
meshes of $3 \times 3$, $4 \times 4$, and $5 \times 5$ methods. For such configurations we have to greatly increase the time horizon since the probabilities of enabling the final methods by a particular time decrease exponentially. We therefore vary the time horizons from 3000 to 4000, and then to 5000. We show the results in Figure (7c) where, especially for larger meshes, the VFP algorithm runs up to one order of magnitude faster than OC-DEC-MDP while finding a policy that is within less than $1\%$ from the policy found by OC-DEC-MDP.

8. CONCLUSIONS

Decentralized Markov Decision Process (DEC-MDP) has been very popular for modeling of agent-coordination problems, it is very difficult to solve, especially for the real-world domains. In this paper, we improved a state-of-the-art heuristic solution method for DEC-MDPs, called OC-DEC-MDP, that has recently been shown to scale up to large DEC-MDPs. Our heuristic solution method, called Value Function Propagation (VFP), provided two orthogonal improvements of OC-DEC-MDP: (i) It speeded up OC-DEC-MDP by an order of magnitude by maintaining and manipulating a value function for each method rather than a separate value for each pair of method and time interval, and (ii) it achieved better solution qualities than OC-DEC-MDP because it corrected the overestimation of the opportunity cost of OC-DEC-MDP.

In terms of related work, we have extensively discussed the OC-DEC-MDP algorithm [4]. Furthermore, as discussed in Section 4, there are globally optimal algorithms for solving DEC-MDPs with temporal constraints [11][12]. Unfortunately, they fail to scale up to large-scale domains at present time. Beyond OC-DEC-MDP, there are other locally optimal algorithms for DEC-MDPs and DEC-POMDPs [8][12], [13], yet, they have traditionally not dealt with uncertain execution times and temporal constraints. Finally, value function techniques have been studied in context of single agent MDPs [7] [9]. However, similarly to [6], they fail to address the lack of global state knowledge, which is a fundamental issue in decentralized planning.

Acknowledgments

This material is based upon work supported by the DARPA/IPTO COORDINATORS program and the Air Force Research Laboratory under Contract No. FA875005C0030. The authors also want to thank Sven Koenig and anonymous reviewers for their valuable comments.

9. REFERENCES