An Iterative Algorithm for Solving Constrained Decentralized Markov Decision Processes

Aurélie Beynier, Abdel-Illah Mouaddib
GREYC-CNRS, Bd Marechal Juin, Campus II
BP 5186 , 14032 Caen cedex, France
{abeynier, mouaddib}@info.unicaen.fr

Abstract

Despite the significant progress to extend Markov Decision Processes (MDP) to cooperative multi-agent systems, developing approaches that can deal with realistic problems remains a serious challenge. Existing approaches that solve Decentralized Markov Decision Processes (DEC-MDPs) suffer from the fact that they can only solve relatively small problems without complex constraints on task execution. OC-DEC-MDP has been introduced to deal with large DEC-MDPs under resource and temporal constraints. However, the proposed algorithm to solve this class of DEC-MDPs has some limits: it suffers from overestimation of opportunity cost and restricts policy improvement to one sweep (or iteration). In this paper, we propose to overcome these limits by first introducing the notion of Expected Opportunity Cost to better assess the influence of a local decision of an agent on the others. We then describe an iterative version of the algorithm to incrementally improve the policies of agents leading to higher quality solutions in some settings. Experimental results are shown to support our claims.

Introduction

Solving optimally Decentralized Markov Decision Processes (DEC-MDPs) is known to be NEXP (Bernstein, Zilberstein, & Immerman 2002). One way to overcome this complexity barrier consists in identifying specific classes of DEC-MDPs that can be solved more easily thanks to characteristics such as transition or observation independence (Goldman & Zilberstein 2004; Becker, Lesser, & Zilberstein 2004; Becker *et al.* 2003). Other approaches develop approximate solution algorithms (Nair *et al.* 2003; Pynadath & Tambe 2002; Peshkin *et al.* 2000).

Nevertheless, solving large problems remains a serious challenge even for approximating approaches, that can solve relatively small problems. Moreover few approaches consider temporal or resource constraints on task execution. To address this shortcoming, OC-DEC-MDP has been introduced (Beynier & Mouaddib 2005). This formalism can be used to represent large decision problems with temporal, precedence and resource constraints. Furthermore, a polynomial algorithm efficiently computes an approximation of the optimal solution considering these constraints. This framework proposes to solve local MDPs in a decentralized way. Coordination is guaranteed thanks to Opportunity Cost (OC) which allows the agents to consider the consequences of their decisions on the other agents. Experiments proved that good quality policies are obtained even for large problems. Nonetheless, the algorithm for solving OC-DEC-MDP suffers from limitations. First of all, OC overestimates the influence of a decision on the other agents. Then, it may lead to fallacies or bad decisions. Moreover, this algorithm improves the initial policy of each task only once, the improvement process is not repeated. We suggest that reiterating the policy improvement should lead to better performance in some settings. In this paper, we propose to overcome these limits by first introducing the notion of Expected Opportunity Cost to better assess the influence of a decision on the other agents. Then we develop an iterative version of the algorithm. Higher quality policies could therefore be obtained in some settings.

We begin by formalizing the problem we are considering and by giving an overview of the approach described in (Beynier & Mouaddib 2005). Next, we describe two directions to improve the existing algorithm for solving OC-DEC-MDP. We start with the description of the Expected Opportunity Cost and we present its benefits. We then describe an iterative version of the algorithm that allows policy improvement to be repeated. Finally, we give some experimental results on the scalability and the performance of our approach and conclude by discussing possible future issues.

Problem Statement

We consider a set of agents, for instance a fleet of robots, that have to execute a set of tasks respecting temporal, precedence and resource constraints. The problems $\mathcal{X}$ we deal with (introduced by Beynier and Mouaddib (Beynier & Mouaddib 2005)) can be described by a tuple $< T, A, TC, Pred, P_c, P_r, R, \mathcal{R}_{ini} >$. T is a set of tasks t_i to execute. A is a set of agents A_i that have to execute T. T_i denotes the set of tasks A_i must execute. TC are temporal constraints among the tasks. Each task t_i is assigned a temporal window $TC(t_i) = [EST_i, LET_i]$ during which it should be executed. EST_i is the earliest start time of t_i and LET_i is its latest end time. $Pred$ are precedence constraints between the tasks. Each task t_i has predecessors: the tasks to be executed before t_i can start. The execution time of each

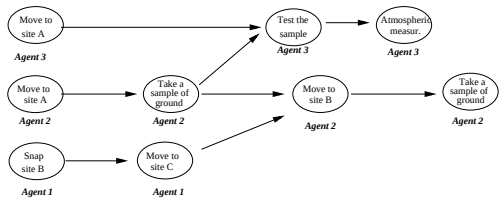

Figure 1: A mission graph for planetary rovers

task is uncertain. $P_c(\delta_c)_i$ is the probability that the activity t_i takes δ_c time units for its execution. Each task t_i has several resource consumptions. $P_r(\Delta r)_i$ is the probability that t_i consumes Δr units of resources. R is a reward function. R_i is the reward an agent obtains when it has accomplished the task t_i. $\mathcal{R}_{ini}$ is the initial resource rate available for each agent before the mission starts.

During task execution, each agent has a local view of the system and do not know the other agents' states nor actions. The agents must be able to coordinate without communicating during the execution. This class of problems can be met in NASA scenarios dealing with rovers operating on Mars. In order to increase the scientific feedback, researchers plan to send fleets of robots that could operate autonomously and cooperatively. Since the environment is unknown, robots must deal with uncertainty regarding the duration of actions and consumption of resources. Once a day, a daily mission of a hundred tasks is sent to the robots via satellite. It allows the agents to communicate during a fixed amount of time. The agents can then plan their actions and communication is possible. For the rest of the day, the rovers must complete their tasks and cannot communicate via satellite due to orbital rotation. In addition, because of bandwidth limitations, the distance between the agents and obstacles, the agents are unable to communicate directly. To guarantee scientific feedback, temporal and precedence constraints must be respected. Pictures, for example, must be taken at sunset or sunrise because of illumination constraints. The tasks of an agent are ordered by precedence, temporal and resource constraints. Precedence constraints give a partial order which can be reduced to a total order while considering temporal constraints (possible execution intervals) and space constraints (we aim to minimize distance because of resource consumptions).

As shown on Figure 1, a problem $\mathcal{X}$ can be represented using a graph of tasks. This example describes a mission involving three planetary rovers. Edges stand for precedence constraints. Each task is assigned an agent. The problem of task allocation is out of the scope of this paper.

Temporal constraints restrict the possible intervals of execution and precedence constraints partially order the tasks. If the execution of a task t_i starts before its predecessors finish, it fails. Because the agent could retry to execute t_i later, this kind of failure is called a partial failure. Partial failures consume restricted resources and can lead to insufficient resources. If an agent lacks resources it will be unable to execute its remaining tasks. Consequently, the agents tend to avoid partial failures. One way to restrict partial failures consists in delaying the execution of the tasks. As a result, the likelihood that the predecessors have finished when an agent starts to execute a task increases and less resources are "wasted" by partial failures. Nonetheless, the more the execution of a task is delayed, the more the successors are delayed and the higher the probability of violating temporal constraints. In fact, the probability the deadline is met increases. If temporal constraints are violated during task execution, the agent fails permanently executing the task. This assumption can be easily relaxed without modifications of our approach.

The problem is to find a local policy for each agent that maximizes the sum of the rewards of all the agents. The agents must trade off the probability of partially failing and consuming resources to no avail against the consequences of delaying the execution of a task. To maximize the sum of the expected rewards, each agent must consider the consequences of a delay on itself and on its successors.

Existing OC-DEC-MDP

Such problems have been formalized byg OC-DEC-MDPs (Beynier & Mouaddib 2005).

Definition 1 *An **OC-DEC-MDP** for n agents is a set of n local MDPs. A local MDP for an agent A_i is defined by a tuple $< S_i, T_i, P_i, R_i >$ where: S_i is the finite set of states of the agent A_i, T_i is the finite set of tasks of the agent A_i, P_i is the transition function and R_i is the reward function.*

Each component of a local MDP is defined by considering and representing precedence, temporal and resource constraints. The actions consist of *Execute the next task t_i at time st:* $E(t_i, st)$. Precedence constraints restrict the possible start times of each task. Consequently there is a finite set of start times for each task and a finite action set. Decisions rely on the last executed task, the end time of its execution, the current time and the available resources.

At each decision step, two kinds of states are considered: safe states and partial failure states. Safe states $[t_i, I, r]$ describe cases where the agent has just finished executing task t_i during an interval of time I and its has r remaining resources. Partial failures $[t_i, I, et(I'), r]$ are also taken into account when the agent has just tried to execute task t_{i+1} duringq interval I but failed because execution of the predecessors was not finished. t_i is the last task the agent has safely finished to execute at $et(I')$.

Theorem 1 *An OC-DEC-MDP has a complexity exponential in the number of states.*

Proof: Each agent's state is supposed to be locally fully observable. A joint policy is a mapping from world states $S = \langle S_1, \cdots, S_n \rangle$ to joint actions $A = \langle A_1, \cdots, A_n \rangle$ and the number of joint policies is exponential in the number of states $|S|$. Evaluating a joint policy can be done in polynomial time through the use of dynamic programming. The OC-DEC-MDP is therefore exponential in the number of states $|S|$. □

Constraints affect the policy space but have no effect on the worst case complexity. They reduce the state space and the action space. Thus, the policy space can be reduced. Nonetheless, the number of policies remains exponential. Consequently, dealing with constraints does not result in a lower complexity.

Expected Opportunity Cost

As explained in (Beynier & Mouaddib 2005), each agent computes its own local policy by improving an initial policy. The probabilities on the end times of the other agents can therefore be deduced from this initial policy and each agent's transition function can be computed. If an agent modifies its own local policy without taking into account the influence of such changes on the other agents, the agents will not coordinate. When an agent decides when to start the execution of a task t_i, its decision influences all the tasks that can be reached from t_i in the mission graph. In order to coordinate, the agents must consider the consequences of their decisions on the other agents. The notion of Opportunity Cost is used in order to obtain coordinated policies. It allows each agent to measure the consequences of its decision on the other agents. While deciding when to start the execution of a task t_i, each agent must trade off its expected utility against the OC it will provoke on the other agents. More specifically, the OC is the loss of expected value by the other agents resulting from delaying the execution of a task.

Previous work has demonstrated the usefulness of OC to predict the influence of a decision on the other agents. Nonetheless, the OC described previously does not compute exactly the loss in expected value due to a delay Δt. First, resources available by the other agents are not taken into account. It has been assumed that resources do not influence the OC. Next, it has been assumed that the delay provoked by a task t_i cannot be increased or decreased by other tasks. As these assumptions may lead to overestimation of the OC provoked on an agent, the influence of a decision may then be approximately computed leading the agents to fallacies.

Opportunity Cost and resources

Each agent's policy is influenced by the available resource rate. Expected utility depends on resources and the OC represents a loss in expected utility. It can be deduced that resources must be taken into account while computing the OC. For instance, if resources are very tight, the probability the agent will fail because of lack of resources is very high. Delaying a task has a low cost because whatever its start time, the likelihood its execution will fail is very high.

For each start time st_{t_j} of t_j, bounded by $[LB_{t_j}, UB_{t_j}]$, and each resource rate r, OC is given by : $OC_{t_j}(\Delta t, r) = V_{t_j}^{*0,r} - V_{t_j}^{*\Delta t,r}$ where $V_{t_j}^{*0,r}$ is the expected value if t_j can start in $[LB_j, UB_j]$ with r resources, and $V_{t_j}^{*\Delta t,r}$ is the expected value if t_j can start in the interval $[LB_j + \Delta t, UB_j]$ with r resources.

Policy Computation and EOC

Revising the policy of a task t_i consists in revising the policy for each state associated with t_i. Previous work introduced an equation to compute a new policy for each task. Nonetheless, it considers the OC provoked on the successors. Such computation may lead to overestimation of OC because the delay provoked on an agent can be considered more than once. If there are several paths from a successor k to a task t_{j+1}, then the OC on t_{j+1} is considered several times. To overcome this issue, we propose to compute the EOC provoked on the other agents. The tasks that are influenced by the end time of t_{i+1} are the nearest tasks of the other agents. The nearest task of t_{i+1} executed by A_j is the nearest task executed by A_j that can be reached from t_{i+1} in the mission graph (in Figure 1, "agent 3: test the sample" is the nearest task of "agent 2: move to site A"). Assume that an agent A_i ends a task t_{i+1} at et_{i+1} and t_j is the nearest task executed by another agent A_j. t_{i+1} induces an EOC on t_j. If t_{i+1} ends at et_{i+1} this does not imply that t_j can start at et_{i+1}. If t_j is not a successor of t_{i+1}, there may be tasks between t_{i+1} and t_j that can increase or decrease the delay. One solution to tackle this issue consists in computing the probability to provoke a delay Δt_j on t_j when t_{i+1} ends at et_{i+1}. This explains why we consider EOC instead of OC.

New equations are therefore defined. The first one is a standard Bellman equation:

$$V(s) = \overbrace{R_i(t_i)}^{\text{immediate gain}} + \overbrace{max_{E(t_{i+1},st),st \geq et(I)}(V')}^{\text{Expected value}} \quad (1)$$

where $s = [t_i, I, r]$ or $s = [t_i, [st, st+1], et(I'), r]$. If s is a permanent failure state, $V(s) = -R_{t_i} - \sum_{suiv} R_{suiv}$ where $suiv$ is the set of tasks A_i should execute after t_i.

The second equation computes the best foregone action using a modified Bellman equation in which an EOC is introduced. It allows the agent to select the best action to execute in a state s, considering its expected utility and the EOC induced on the other agents:

$$\Pi^*(s) = arg\_max_{E(t_{i+1},st),st \geq et(I)} \left(\overbrace{V'}^{\text{Expected utility}} - \overbrace{EOC(t_{i+1}, st)}^{\text{EOC}} \right) \quad (2)$$

where $EOC(t_{i+1}, st)$ is the EOC induced on the other agents when t_{i+1} starts at st. Since actions are not deterministic, we do not know exactly when the task t_{i+1} will end and the EOC it will induce. Different kinds of transitions must be taken into account so as to consider the possible end times of the task. The EOC induced on the other agents when t_{i+1} starts at st can be rewritten as follows:

$$EOC(t_{i+1}, st) = P_{suc} \cdot \sum_{ag \in \text{other\_agents}} EOC_{ag,t_{i+1}}(et_{i+1}) \quad (3)$$
$$+ P_{fail} \sum_{ag \in \text{other\_agents}} EOC_{ag,t_{i+1}}(fail) + P_{PCV} \cdot EOC(t_{i+1}, st')$$

where $other\_agents$ are the agents that do not execute t_{i+1}, et_{i+1} is a possible end time of t_{i+1}, $EOC_{ag,t_i}(et_{i+1})$ is the EOC induced on the agent ag when t_{i+1} ends at et_{i+1}. It is computed using Equation 3. $EOC(t_{i+1}, st')$ is the OC when the execution of t_{i+1} partially fails and the agents retries to execute the task at st' (the next start time given by the EOC-policy). The execution of t_{i+1} can lead to different transitions, therefore all these transitions must be considered in $EOC(t_{i+1}, st)$. P_{suc} stands for the probability to successfully execute the task, P_{PCV} is the probability to fail partially because the predecessors have not finished. P_{fail} is the probability to fail permanently.

Revision algorithm The previous work algorithm has been adapted to deal with EOC computation leading to a new revision algorithm (Algorithm 1). Both algorithms allow the agents to evaluate at the same time their own local

MDP and to derive a new local policy from the initial policy. While computing its policy, an agent first considers the last level of its graph which consists of the last task it must execute. It then passes through its graph of tasks from the leaves to the root and evaluates its policy level by level.

Algorithm 1 Revision Algorithm

1: **for** level L_n from the leaves to the root **do**
2: **for all** task t_i in level L_n **do**
3: **while** the agent doesn't have the OC values it needs **do**
4: wait
5: **end while**
6: Compute V for the failure state: $[failure(t_i), *, *]$
7: **for all** start time st from UB_{t_i} to LB_{t_i} **do**
8: **for all** resource rate r_{t_i} of a partial failure **do**
9: Compute V and Π^* for partial failure states: $[t_{i-1}, [st, st + 1], et(I'), r_{t_i}]$
10: **end for**
11: **for all** resource rate r_{t_i} of t_i's safe execution **do**
12: **for all** duration Δt_1 of t_i **do**
13: Compute V and Π^* for the safe states: $[t_i, [st, st + \Delta t_1], r_{t_i}]$
14: **end for**
15: Compute $V_k^{*\Delta t, r_{t_i}}$ where $\Delta t = st - LB_{t_i}$
16: **end for**
17: **end for**
18: **for all** $V_k^{*\Delta t, r_{t_i}}$ computed previously **do**
19: Compute $OC(\Delta t, r_{t_i}) = V_k^{*0, r_{t_i}} - V_k^{*\Delta t, r_{t_i}}$
20: Send $OC(\Delta t, r_{t_i})$ to the predecessors
21: **for all** task t_j executed by another agent and for which OC values has been received **do**
22: Update the OC values and Send to predecessors
23: **end for**
24: **end for**
25: **end for**
26: **end for**

Using equation 3 we need to know the EOC provoked on the other agents when t_{i+1} ends at et_{i+1}. The delay provoked on the nearest tasks t_k must therefore be known. This delay depends on the policies of the tasks between t_{i+1} and t_j. Nonetheless, the algorithm passes through the graph from the leaves to the node. The policies of the tasks t_{i+1} and t_j have been revised when t_{i+1} is considered. If the initial policy is used to compute these probabilities, it may lead to inaccurate results because of policy changes. Accurate probabilities can be deduced from the revised policies but A_i does not know them unless all policy modifications are communicated to all agents.

To overcome this difficulty, an update method has been developed. When an agent A_j has finished revising the policy of a task t_j, it computes the OC values of t_j for each delay Δt_{t_j} and each resource rate r_{t_j} of t_j (line 19 of algorithm 1). These OC values are sent (line 20) to t_j's predecessor agents. These agents now revise the policy of t_j's predecessor tasks. They also update the received OC of t_j following their revised policy (line 22) and send them to their predecessors. Finally, A_i is going to receive the updated OC values of t_j from its successors. Consequently, even if t_j is not a direct successor of t_i, when t_i delays the execution of t_k, the exact OC provoked on t_j is known.

Thanks to our update method, EOC values can be deduced by considering the delay provoked on the successors. The OC described in Equation 3 is given by :

$$EOC_{ag,t_i}(et_{t_i}) = \sum_{r_{t_j}} P_{ra}^{t_j}(r_{t_j}) \cdot OC_{t_j}(\Delta t_{succ}, r_{t_j})$$

where t_j is the nearest task that will be executed by ag and $OC_{t_j}(\Delta t_{succ}, r_{t_j})$ is the OC provoked on t_j when the successors of t_i are delayed by Δt_{succ}. This OC value is known since the agent received it from its successors. $P_{ra}^{t_j}(r_{t_j})$ is the probability that ag has r_{t_j} resources when it starts to execute t_j. If t_i has several successors and several delays are provoked, Δt_{succ} represents the maximum of these delays.

This method allows each agent to know the exact EOC provoked on the other agents without communicating revised policies. The agents only need to communicate OC values. When an agent evaluates the states associated with a task t_i, it needs to know the EOC it will provoke on the other agents. Therefore, it must have received OC values from its successors. If these values have not been received yet, the agent waits until delivery. We assume that there is no loss of messages. As soon as OC values are known, the agent can compute its expected value and the policy of each state related to t_i. Notice that even if decentralized execution of the algorithm requires off-line communication, the agents never communicate during the execution of the mission.

Theorem 2 *The complexity of the revision algorithm is polynomial time in* $|S|$.

Proof: In the worst case, none of the states can be evaluated at the same time and the algorithm has the same time complexity as a centralized algorithm which evaluates all the agents' states one by one. Therefore only one state is evaluated at a time. The time needed to pass trough the state space of each local MDP and to value each state is $|S|$. States have a form of [task, (end_time), interval, resources], so $|S|$ includes the number of tasks. We will assume that communication requires one time unit. There is $\#n_{OC}$ values of OC to communicate. Then, the overall complexity of the algorithm is $O(\#n_{OC} + |S|)$. As $\#n_{OC} < |S|$, the complexity can be re-written as $O(|S|)$.□

Experiments

Experiments have been developed to test the influence of OC computation on performance. We have compared three kinds of OC : Opportunity Cost without resources (OC), OC with resources (OCr) and Expected Opportunity Cost (EOC). Our experiments have been achieved with a benchmark composed of different mission sizes.These experiments show the benefits of considering resources and EOC. We present results obtained on a scenario involving 2 agents and about 20 tasks. Figure 2 plots the reward the agents obtain over 1000 executions for different resource rates. When the agents have very few resources, OC computation has no influence on the performance. In fact, whatever their policy, the agents do not have enough resources to execute their tasks, the gain is near zero and delaying a task does not decrease the expected value. With large or unlimited resources, resources have no influence on the expected utility nor the OC. OCr and OC lead to the same performance. Unless,

there are tight temporal constraints, delaying a task does not reduce the expected value, EOC and OCr are quite the same when resources are large.

Improvements are much larger when resources are tight. In this case, the agents have just enough resources to execute their tasks and partial failures may lead to permanent failures through lack of resources. Taking into account resources while computing the OC increases performance. Moreover, delaying a task can lead the other agents to fail partially and waste resources to no avail. Consequently, the agents accurately measure the consequences of their decisions on the other agents. EOC therefore achieves best improvement under tight resources ($\mathcal{R}_{ini} \in [20, 30]$).

Iterative Algorithm

The algorithm presented so far consists in improving an initial policy. When the revision algorithm stops, each task has been considered once and a new local policy is available for each agent. In order to obtain better solutions, we re-execute the revision algorithm considering that the initial policy is the policy we have just computed.

Iterative Process

At each iteration step, the agents improve their initial local policy at the same time. The outcome policies of iteration $N-1$ are the initial policies of iteration N. Obviously, the transition function depends upon the initial policy of the current iteration and must be updated for each iteration step. Once the new transition function is known, each agent can execute the revision algorithm to obtain new local policies. This process is repeated until no changes are made. A local optimum is then reached. Since each policy is considered once and there is a finite set of policies, we assure convergence.

Algorithm 2 Iterative Algorithm

```
1: nbChanges = 1
2: while nbChanges > 0 do
3:    Compute the new transition function
4:    nbChanges = 0
5:    for all Agents A_i do
6:       Revise local policies π_i
7:       nbChanges += number of local policy changes
8:    end for
9: end while
```

By iterating the algorithm each task is considered several times. We will consider a task t_i and its predecessor task t_j. At first iteration step t_j is supposed to follow the initial policy. t_i's policy is revised assuming t_j's initial policy. Once the policy of t_i is revised, t_j will be considered and its policy may change. t_j's new policy is computed assuming the new policy of t_i (thanks to EOC). Nonetheless, given the new policy of t_j, t_i may find a better policy. Using the iteration process, revision of t_i's policy is possible.

Theorem 3 *The complexity of the iterative is polynomial time in $(N+1) \cdot |S|$ where N is the number of iterations.*

Proof: At each iteration step, the transition function is computed and the revision algorithm is executed. The complexity of the revision algorithm is polynomial time in $|S|$. The transition function is updated before each iteration step by propagating temporal constraints through the mission graph whose complexity is less than $O(|S|)$. Then, the overall complexity of the algorithm is $O((N+1) \cdot |S|)$. □

Experiments

So that our iterative algorithm can be used to solve realistic decision problems, it must be able to consider large problems and to find good approximate solutions. The following experiments first test the scalability of our approach and then describe its performance.

Scalability The same space is required to iterate once or several times. We have proved that large problems can be solved by the revision algorithm. This assertion remains true while considering the iterative process.As in OC-DEC-MDP, this new approach can deal with large problems, for instance we can consider missions of 200 taks and 15 agents.

The running time of the iterative algorithm relies on the state space size of each agent, the branching factor of the mission graph and the number of iterations. The branching factor of the mission graph defines the number of tasks per graph level. The iterative algorithm revises all the tasks of a same level at the same time. The higher the branching factor is, the more tasks are revised at the same time. Running time is also influenced by the number of iterations. Most of the time, the final solution is reached within four iterations. Moreover, a solution is always available even if the algorithm has not finished its execution. Indeed, the initial policy of the last ended iteration can stand for a temporary solution. Experimental results show that for a mission graph of 20 tasks and 2 agents, the final solution is obtained within 25 seconds. For larger graphs, for instance 60 tasks and 2 agents, the final solution is returned within one minute. Graphs composed of hundreds of tasks can also be considered. Our approach can therefore deal with large problems that cannot be solved by other existing approaches.

Performance Other experiments have been developed to test the performance and convergence of the iterative algorithm. The number of iteration steps to converge has been studied. The performances obtained at each iteration step have been compared by running mission executions. The benchmark is the same as the one used for OC comparisons. EST-policy (select the Earliest Possible Start Time) is supposed to be the initial policy of the first iteration.

First, the number of iteration steps needed to converge has been studied. Experiments show that the algorithm always converges. Most of the time, it takes less than four iterations to converge. Figure 3 relates the number of iteration steps to the initial resource rate. With large or unlimited resources, only one iteration step is needed to converge. As the initial resource rate decreases, the number of iteration steps increases since it reaches a maximum which corresponds to a critical resource rate. If the agents initially have less resources than this critical rate, they will not be able to execute all the tasks. Whatever their policy, the latest tasks cannot be executed because of a lack of resources. Then, all the possible policies of these tasks are equivalent and there is no strictly better policy than the initial EST-policy.

Figure 4 describes the relationship between the initial re-

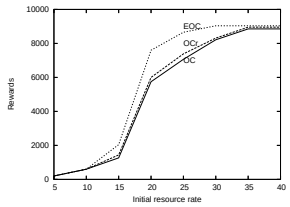

Figure 2: Influence of OC

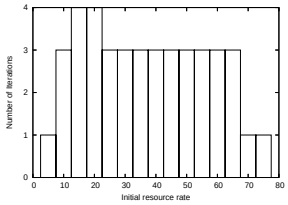

Figure 3: Influence of resources

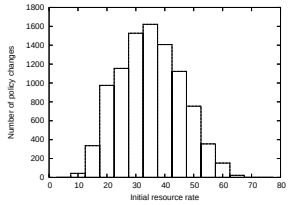

Figure 4: Influence of resources

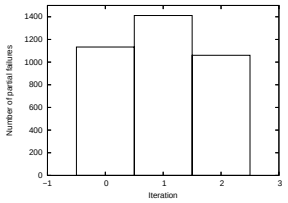

Figure 5: Nb. of Partial failures

source rate and the number of policy changes of each iteration. If resources are large or unlimited, there is no change. Indeed, it can be proved that the initial policy is an optimal policy. As the initial resource rate decreases, more and more changes are needed to obtain the solution. If initial resources are low, the policy of the latest task remains unchanged and few changes are necessary. The number of policy changes per iteration has also been studied. Experiments demonstrate that the number of policy changes decreases at each iteration step. Most changes are made during the earliest iteration. The number of policy changes per iteration diminishes until it becomes null and convergence is reached.

Experiments demonstrate that the performance of the agents increases with the number of iterations. Moreover, it can be shown that first iteration achieves largest improvements. Subsequent iterations reduce the number of partial failures and lead to maximum gain. By iterating the process, the likelihood the agents fail because of lack of resources decreases. The resulting policy is safer than policies of previous iterations and the gain of the agents is steady over executions: it does not fall when partial failures arise. Figure 5 plots the number of partial failures of the agents over 1000 executions. A near optimal policy is obtained at the end of the first iteration. Second iteration leads to small improvements but it diminishes the number of partial failures. When resources are tight, more iterations are needed to obtain a near optimal policy. Now, more benefits are gained from re-iterating. As soon as the solution produces the maximum gain, re-iterating reduces the number of partial failures.

Related work

It is difficult to compare the performance of our approach with other works. We have in fact developed the first approach that can deal with large problems and several kinds of constraints. Other existing approaches can only solve small problems so, we were unable to compare our performance on large problems. Despite this lack of comparison, experiments show that most of the time, the iterative algorithm allows the agents to obtain the maximum reward. We may assume that our solutions are closed to the optimal.

Our results have been compared on smaller problems which can be solved by other approaches. The Coverage Set Algorithm (CSA) (Becker *et al.* 2003) is the only algorithm that is able to deal with complex constraints and solve these problems. Despite the wide variety of problems formalized by CSA, it suffers from exponential complexity. In practice, only small problems with restricted constraints can be solved which have been proved to be solved optimally by our algorithm. We deduce that our approach performs as well as CSA, on the problems CSA can solve. Other existing approaches do not consider temporal and precedence constraints. We were therefore unable to compare our performance with algorithms such as JESP (Nair *et al.* 2003). Nonetheless, it can be mentioned that JESP improves only one local policy at a time while our algorithm allows the agents' policies to be improved at the same time.

Conclusion

Most recent multiagent applications such as rescue teams of robots or planetary rovers demonstrate the need for cooperative decision making systems that can deal with large problems and complex constraints. Previous work provided a framework for solving large decision problems with temporal, precedence and resource constraints. Introduction of EOC in Bellman Equation improves the coordination of local policies. With EOC, each agent can accurately compute the consequences of its decision on the other agents. We proposed an iterative version of the previous algorithm for solving such problems in a decentralized way. This polynomial algorithm consists in improving each agent's local policy at the same time. This new algorithm does not reduce the size of the problems that can be solved. Moreover, higher quality policies are obtained and best improvements are obtained with tight resources and tight temporal constraints. It has been shown that the quality of the policies increases at each iteration step. Experiments show that our algorithm has anytime properties. Future work will explore this issue.

References

Becker, R.; Zilberstein, S.; Lesser, V.; and Goldman, C. 2003. Transition-independent decentralized markov decision processes. In *AAMAS*, 41–48.

Becker, R.; Lesser, V.; and Zilberstein, S. 2004. Decentralized markov decision processes with event-driven interactions. In *AAMAS*, 302–309.

Bernstein, D.; Zilberstein, S.; and Immerman, N. 2002. The complexity of decentralized control of mdps. In *Mathematics of Operations Research*, 27(4):819–840.

Beynier, A., and Mouaddib, A.I. 2005. A polynomial algorithm for decentralized markov decision processes with temporal constraints. *AAMAS* 963–969.

Goldman, C., and Zilberstein, S. 2004. Decentralized control of cooperative systems: Categorization and complexity analysis. *JAIR* 22:143–174.

Nair, R.; Tambe, M.; Yokoo, M.; Marsella, S.; and Pynadath, D.V. 2003. Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In *IJCAI*, 705–711.

Peshkin, L.; Kim, K.; Meuleu, N.; and Kaelbling, L. 2000. Learning to cooperate via policy search. In *UAI*, 307–314.

Pynadath, D., and Tambe, M. 2002. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *JAIR* 389–423.