The MultiAgent Decision Process toolbox: Software for decision-theoretic planning in multiagent-systems

Spaan, M.T.J.; Oliehoek, F.A.

Published in:
Multiagent Sequential Decision Making (MSDM), 2008

The MultiAgent Decision Process toolbox: software for decision-theoretic planning in multiagent systems

Matthijs T.J. Spaan
Institute for Systems and Robotics
Instituto Superior Técnico
Lisbon, Portugal
mtjspaan@isr.ist.utl.pt

Frans A. Oliehoek
Intelligent Systems Lab Amsterdam
University of Amsterdam
The Netherlands
faolieho@science.uva.nl

Abstract

This paper introduces the MultiAgent Decision Process software toolbox, an open source C++ library for decision-theoretic planning under uncertainty in multiagent systems. It provides support for several multiagent models, such as POSGs, Dec-POMDPs and MMDPs. The toolbox aims to reduce development time for planning algorithms and to provide a benchmarking platform by providing a number of commonly used problem descriptions. It features a parser for a text-based file format for discrete Dec-POMDPs, shared functionality for planning algorithms, as well as the implementation of several Dec-POMDP planners. We describe design goals and architecture of the toolbox, and provide an overview of its functionality, illustrated by some usage examples. Finally we report on current and future work.

1 Introduction

Sequential decision making has been a central topic in the Artificial Intelligence community since the beginning of the field, as endowing agents with intelligent behavior in general requires them to consider what course of action to take. Such a planning process requires an agent to consider its task as well as the impact of its actions on its environment. Decision-theoretic planning provides a framework for agent planning in which the environment is allowed

to be stochastic, and the agent's goal is to maximize its expected utility. For single agents with imperfect sensing, partially observable Markov decision processes (POMDPs) form a popular planning model [7]. We will consider several multiagent decision-theoretic planning models, denoted as multiagent decision processes (MADPs). Well-known MADPs include partially observable stochastic games (POSGs) [6], decentralized POMDPs [1], multiagent team decision problems (MTDPs) [11], and multiagent MDPs (MMDPs) [2].

In this paper, we present MultiAgent Decision Process, a software toolbox for research in decision-theoretic planning (and learning) in multiagent systems. It has been designed to be rather general and extensible and provides a uniform representation for several MADP models. As a result of our particular research interest, most effort has been put in planning algorithms for discrete Dec-POMDPs. By releasing the software as open source, we hope to lower the threshold for researchers new to this area, allowing rapid development of new algorithms. Furthermore, we would like to encourage benchmarking of existing algorithms, by providing a repository of problem domain descriptions. The software consists of a set of C++ libraries, as well as several applications. It is being developed as free software on GNU/Linux and is available at [9].

The POMDP community has benefitted greatly from the availability of a repository of problem descriptions and POMDP solving software, provided by Anthony Cassandra [3] (resulting from his PhD thesis [4]). In particular, the file format proposed to define POMDP models has found widespread use. For MADPs such as POSGs and Dec-POMDPs, on the other hand, less software is available and a lower level of standardization has been established. This comes as no surprise as computational approaches for such models have only more recently been receiving a growing amount of attention. To our knowledge, only two collections of MADP-related software are available [12, 16]. Their scope is more limited than the proposed toolbox, and at the moment considerably less functionality is provided.

The remainder of this paper is organized as follows. Section 2 introduces a mathematical formulation of several MADP models and related concepts. Section 3 introduces the software architecture, including its design goals and the provided functionality. Next, the file format for discrete Dec-POMDPs is highlighted in Section 4. Section 5 introduces some of the example applications, and in Section 6 we briefly describe some implementation details and the computing platform. Finally, in Section 7 we draw some short conclusions and highlight in which directions we are extending the MADP toolbox.

2 Multiagent decision processes

We first introduce the mathematical framework behind MADPs, at the same time introducing notation. We start by formally defining the POSG model [6], the most general case we consider, after which we will discuss other MADP models as special cases.

2.1 Partially observable stochastic games

A *partially observable stochastic game (POSG)* is a tuple $\langle \mathcal{A}g, \mathcal{S}, \mathcal{A}, \mathcal{O}, T, O, R \rangle$, where

- $\mathcal{A}g = \{1, \ldots, n\}$ is the set of n agents.
- $\mathcal{S}$ is a set of states the environment can be in.
- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ is the set of joint actions $\mathbf{a} = \langle a_1, \ldots, a_n \rangle$, where an individual action of an agent i is denoted $a_i \in \mathcal{A}_i$.
- $\mathcal{O} = \mathcal{O}_1 \times \cdots \times \mathcal{O}_n$ is the set of joint observations $\mathbf{o} = \langle o_1, \ldots, o_n \rangle$. An individual observation of agent i is denoted $o_i \in \mathcal{O}_i$.
- T is the transition function which provides $P(s'|s, \mathbf{a})$, the probability of a next state s' given that joint action $\mathbf{a}$ is executed from state s.
- O is the observation function that specifies $P(\mathbf{o}|\mathbf{a}, s')$, the probability that the agents receive joint observation $\mathbf{o}$ of state s', when they reached this state through joint action $\mathbf{a}$.
- $R = \{R_1, \ldots, R_n\}$ is a collection of individual reward functions. $R_i(s, \mathbf{a})$ specifies the individual reward for agent i, represented as a real number.

At every time step, or *stage*, the agents simultaneously take an action. The resulting joint action causes a transition to the next stage. At that point the environment the system emits a joint observation, of which each agent observes its own component. The initial state distribution b^0 specifies the probability of each state at stage $t = 0$.

2.2 MADP models

Now we will discuss different types of MADPs as special cases of a POSG. In particular, these special cases make assumptions on the reward functions and/or the observability of states.

In a POSG each agent has its own reward function, indicating that each agent has its own preferences. As a result POSGs model *self-interested* multiagent systems. When all agents have the same reward function (i.e., $\forall i,j\ R_i(s,\mathbf{a}) = R_j(s,\mathbf{a})$) all agents have the same preferences and the system is *cooperative*. This special case is referred to as a *decentralized partially observable Markov decision process (Dec-POMDP)* [1], and is equivalent to an MTDP [11].

When assuming that the agents in a POSG can observe the state s, a POSG reduces to a (fully observable) *stochastic game (SG)*. In such a SG, O and $\mathcal{O}$ are omitted. A SG that is cooperative (i.e., with only one reward function for all agents) is also referred to as a *multiagent MDP (MMDP)* [2]. A final special case is when there is only 1 agent. In that case the above models reduce to a (PO)MDP [7].

2.3 Planning

An MADP is typically considered over some number of time steps or *stages*, referred to as the *horizon* h (which can also be infinity). The main task we address is *planning*: finding an (near-)optimal plan, or *policy*, that specifies what to do in each situation that can occur during h stages. In multiagent planning the goal is to find a *joint policy*; a policy for each agent.

In such a planning context, we assume that the necessary models (T, O and R) are available. Using these models we compute a joint policy in an off-line phase. Once this plan is computed it is given to the agents, who then execute the plan in an on-line phase. Note that this is in contrast to the setting of reinforcement learning (RL) [14], where agents have to *learn* how the environment reacts to their actions in the on-line phase[1]. Exactly which (joint) policies are optimal depends on the optimality criterion considered. A typical choice is the maximization of the expected *cumulative future reward*, also called expected *return*: $E[\sum_{t=0}^{h-1} \gamma^t R(s^t, \mathbf{a}^t)]$, where $0 < \gamma \leq 1$ is a

[1] Although the MADP toolbox currently does not provide functionality specific to reinforcement learning, the uniform problem representation and the provided process simulator should prove useful for reinforcement-learning settings as well.

discount factor. Note that when $h = \infty$ the discount should be < 1 to ensure a finite sum.

2.4 Histories and policies

An agent's policy tells it how to act at any moment in time. The form of a policy of an agent depends heavily on the observability assumptions imposed by the particular kind of MADP. In the most general case, however, a policy is a mapping from the observed history of the process to an agent's actions. A process of horizon h specifies h time-steps $t = 0, \ldots, h-1$. At each of these time-steps, there is a state s^t, joint observation $\mathbf{o}^t$ and joint action $\mathbf{a}^t$. Therefore, when the agents will have to select their k-th actions (at $t = k-1$), the history of the process is a sequence of states, joint observations and joint actions, which has the following form:

$$\left(s^0, \mathbf{o}^0, \mathbf{a}^0, s^1, \mathbf{o}^1, \mathbf{a}^1, \ldots, s^{k-1}, \mathbf{o}^{k-1}\right).$$

The initial joint observation $\mathbf{o}^0$ is assumed to be the empty joint observation and will be omitted from now on. An agent can only observe his own actions and observations. The *action-observation history* of agent i at time step t is defined as

$$\vec{\theta}_i^t = \left(a_i^0, o_i^1, a_i^2, \ldots, o_i^{t-1}, a_i^{t-1}, o_i^t\right)$$

and the *observation history for agent i,* $\vec{o}_i$, as the sequence of observations an agent has received: $\vec{o}_i^t = (o_i^1, \ldots, o_i^t)$. The set of all possible action-observation histories for agent i is $\vec{\Theta}_i$. The *joint action-observation history,* $\vec{\boldsymbol{\theta}}$, is the action-observation history for all agents: $\vec{\boldsymbol{\theta}}^t = \langle \vec{\theta}_1^t, \ldots, \vec{\theta}_n^t \rangle$. Notation of joint observations and observation histories are analogous ($\mathbf{o}$,$\vec{\mathcal{O}}_i$).

In the most general case, we can formally define a policy π_i for agent i as a mapping from action-observation histories to probability distributions over actions $\pi_i : \vec{\Theta}_i \rightarrow \mathcal{P}(\mathcal{A}_i)$. Such policies are also called *stochastic policies.* For deterministic or *pure* policies, this reduces to $\pi_i : \vec{\mathcal{O}}_i \rightarrow \mathcal{A}_i$. Similar to previous notation we use $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ to denote a *joint policy.*

3 MADP Toolbox architecture

Now that we briefly introduced the decision-theoretic mathematical framework in which our toolbox operates, we discuss its global architecture and components.

3.1 Design goals and assumptions

The development of the MADP toolbox has two main goals. The first is to provide a library for rapid development and evaluation of planning algorithms for MADPs. As shown in Section 2, the considered MADPs consist of many components: states, (joint) actions, (joint) observations, transition models, etc., and in the planning process we encounter even more data types (histories, policies, etc.). We aim to reduce the amount of work needed to implement a working MADP description and a corresponding planning algorithm. The second goal is to provide a benchmark set of test problems and planning algorithms for Dec-POMDPs (and potentially other MADPs).

Before elaborating on the design objectives of the toolbox, first let us state our main assumptions. In particular, throughout the software we assume that time has been discretized, i.e., that agents take decisions at predefined intervals. Such an assumption is ubiquitous in the literature on the MADP models we consider. Furthermore, the POSG definition of Section 2.1 does not specify whether the sets $\mathcal{S}$, $\mathcal{A}_i$, and $\mathcal{O}_i$ are continuous or discrete, which is reflected in our design. However, as the majority of algorithms tackle the discrete case, most effort (until so far) has been spent in providing functionality for discrete models.

Based on the goals of the toolbox, we have identified the following design objectives. (1) A principled and object-oriented representation of all MADP *elements*, such as (joint) actions, observations, etc., such that they can be recombined in new MADP variants; and of derived data types such as histories, policies, etc., such that they can be easily used in new planning algorithms. (2) To provide typical functionality, for example construction of joint actions from individual actions, generation of histories, manipulation of indices, and computation of derived probabilities. (3) The classes provided by the library should be customizable: the user should be able to select which of the functionality (e.g., generation of observation histories) he would like to use, to allow a trade-off between speed and memory. (4) The design should be extensible. It should be easy to write and use new implementations of particular classes.

In accordance with these design goals, the MADP toolbox has a separation of interface and implementation. Although C++ by itself does not distinguish between interface classes and their implementation, we have chosen to make such a distinction, clearly recognizable by their naming. In this text, however, we will not specifically emphasize this separation. The MADP

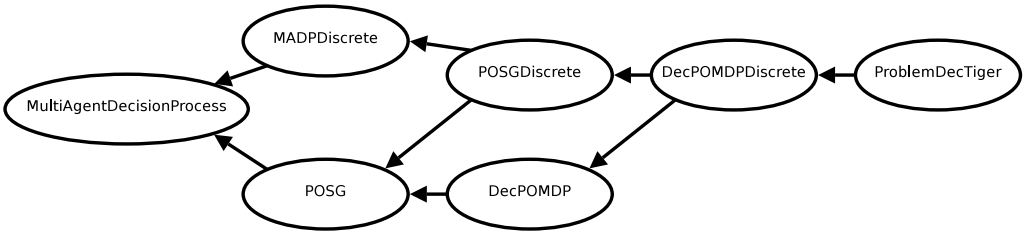

Figure 1: The hierarchy of the different MADP models.

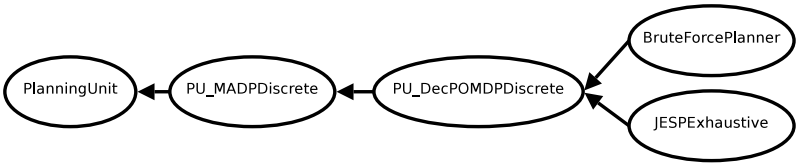

Figure 2: The hierarchy of planning units.

toolbox also has a strict separation between on the one hand problem representations and on the other hand (planning) algorithms. The former are represented by a hierarchy of MADP classes, while the latter are represented by various derivations of the abstract class PlanningUnit. Because planning algorithms typically only refer to an interface of a particular problem, a user is able to provide his own implementations for specific problems, and still run general planning algorithms on this class. In order to provide some intuition on the design of the toolbox, we will give a brief overview of these two different components.

3.2 Representing MADP models

A set of classes is dedicated to representing MADPs, currently implementing a hierarchical representation of POSGs and Dec-POMDPs as illustrated in Figure 1. In particular the figure shows the interfaces for various MADP classes. `MultiAgentDecisionProcess`[2] is the most abstract level of the hierarchy in which only the number of agents is defined. `MultiAgentDecisionProcessDiscrete` assumes that world consists of discrete states and that agents have discrete actions and observations. This means the functions in this class can be based on indices (of states, actions and observations).

[2]Capitalized nouns in typewriter font refer to C++ classes.

On the other hand `POSG` defines functions to set a reward function and a discount factor for each agent. `POSGDiscrete` inherits from both these classes. A `DecPOMDP` is a special case of, and thus extends, `POSG` by providing get/set reward functions that can be called without specifying an agent. `DecPOMDPDiscrete` inherits both from `POSGDiscrete` and `DecPOMDP`, while finally `ProblemDecTiger` provides a full implementation of a discrete DecPOMDP, implementing all the interfaces it inherited.

3.3 Planning Algorithms

In the MADP toolbox planning algorithms are represented by a family of so-called *planning unit (PU)* classes as illustrated in Figure 2. Each class in the hierarchy of PUs is associated with a corresponding class from the MADP hierarchy. Intermediate classes in the hierarchy provide auxiliary functionality for the associated MADP, while the leaves of this hierarchy represent complete planning algorithms. In particular `PlanningUnit` forms the most abstract level, which defines the horizon h and an abstract function `Plan()`, which needs to be implemented by all planning algorithms.

One of the major classes is `PlanningUnitMADPDiscrete` which represents a PU for a discrete MADP, and provides a multitude of auxiliary functions. For instance, it provides functions to construct (joint) (action-) observation histories. The generation of these elements is controlled by parameters, giving the user the control to optimize between memory and speed requirements. For instance, for particular algorithms it will be beneficial to generate and cache all joint observation histories, but this might not be feasible due to limited memory. The class also provides functionality for conversions between joint and individual history indices. Finally, many planning algorithms typically will require probabilities of joint action-observation histories and the corresponding probability distribution over states (given a joint policy). I.e., they will often require the following, recursively defined joint probability distribution:

$$P(s^t, \vec{\boldsymbol{\theta}}^t | b^0, \pi) = \sum_{s^{t-1}} P(\mathbf{o}^t | \mathbf{a}^{t-1}, s^t) P(s^t | s^{t-1}, \mathbf{a}^{t-1}) \\ P(\mathbf{a}^{t-1} | \vec{\boldsymbol{\theta}}^{t-1}, \pi) P(s^{t-1}, \vec{\boldsymbol{\theta}}^{t-1} | b^0, \pi). \quad (1)$$

The class also provides functions to compute these probabilities.

`PlanningUnitDecPOMDPDiscrete` provides some forwarding functions to get rewards. Currently, the MADP Toolbox includes two basic planning

algorithms for the solutions of discrete Dec-POMDPs, namely brute force search, which valuates each joint policy, and exhaustive JESP [8].

There is also a set of classes that provides general functionality for planning algorithms. Examples of such classes are (joint) beliefs and policies and the implementation of the data types uses to represent all types of histories. Furthermore, the MADP toolbox also provides efficient command-line option parsing.

3.4 Overview

We present a short overview of the different libraries in which the functionality of the MADP toolbox has been organized.

Base library The base library is the core of the MADP toolbox. It contains the classes that represent the MADPs and their basic elements such as states, (joint) actions and observations. Is also includes a representation of the transition, observation and reward models in a multiagent decision process and auxiliary functionality regarding manipulating indices, exception handling and printing.

Parser library The parser library only depends on the base library, and contains a parser for .dpomdp problem specifications, which is a file format for discrete Dec-POMDPs (see Section 4). The format is based on Cassandra's POMDP file format. A set of benchmark problem files is included.

Support library The support library contains basic data types useful for planning, such as: a representation for (joint) observation, action and action-observation histories and a representation for (joint) beliefs. It also contains functionality for representing (joint) policies, as mappings from histories to actions, and provides easy handling of command-line arguments.

Planning library The planning library depends on the other libraries and contains functionality for planning algorithms. In particular it provides shared functionality for discrete MADP planning algorithms, such as computation of (joint) history trees, joint beliefs, and value functions. It also includes a simulator to empirically test the quality of a solution. Currently two simple Dec-POMDP solution algorithms are included: `JESPExhaustivePlanner` and `BruteForceSearchPlanner`.

4 Problem specification

In the MADP toolbox there are basically two ways of specifying problems. The first is to derive a class from the MADP hierarchy, as is done for `ProblemDecTiger` illustrated in Figure 1. For (discrete) Dec-POMDPs, there is a second option: create a `.dpomdp` file and use the parser library to load the problem. The first option has the advantage of speed, while the second option makes it easier to distribute a test problem.

The `.dpomdp` file format is based on Cassandra's `.pomdp` format [3], which has been extended to handle multiple agents. An example of the format is shown in Figure 3, which shows a major part of the DecTiger problem specification. Illustrated is how the transition and observation model can be compactly represented by first declaring them uniform, and then overriding the values that are not uniform. E.g., only when the agents perform ⟨`listen,listen`⟩, the state does not reset with uniform probability, but remains unchanged (the transition matrix is the identity matrix). A full specification of the `.dpomdp` file format is available in the MADP toolbox distribution [9].

5 Example applications

Apart from the libraries presented in Section 3.4, the MADP toolbox also includes several applications using the provided functionality. For instance, applications which use JESP or brute-force search to solve `.dpomdp` for a particular planning horizon. In this way, Dec-POMDPs can be solved directly from the command line. Furthermore, several utility applications are provided, for instance one which empirically determines a joint policy's control quality by simulation.

To provide a concrete example, Figure 4(a) shows the full source code listing of an application. It is a simple but complete program, and in the distribution more elaborate examples are provided. The program uses exhaustive JESP to compute a plan for the horizon 3 DecTiger problem, and prints out the computed value as well as the policy. Line 5 constructs an instance of the DecTiger problem directly, without the need to parse `dectiger.dpomdp`. Line 6 instantiates the planner, with as arguments the planning horizon and a pointer to the problem it should consider. Line 7 invokes the actual planning and lines 8 and 9 print out the results.

```
agents: 2
discount: 1
values: reward
states: tiger-left tiger-right
start:
uniform
actions:
listen open-left open-right
listen open-left open-right
observations:
hear-left hear-right
hear-left hear-right
# Transitions
T: * :
uniform
T: listen listen :
identity
# Observations
O: * :
uniform
O: listen listen : tiger-left : hear-left hear-left : 0.7225
O: listen listen : tiger-left : hear-left hear-right : 0.1275
[...]
O: listen listen : tiger-right : hear-left hear-left : 0.0225
# Rewards
R: listen listen: * : * : * : -2
R: open-left open-left : tiger-left : * : * : -50
[...]
R: open-left listen: tiger-right : * : * : 9
```

Figure 3: Specification of the DecTiger problem as per `dectiger.dpomdp` (abridged).

```
#include "ProblemDecTiger.h"
#include "JESPExhaustivePlanner.h"
int main()
{
    ProblemDecTiger dectiger;
    JESPExhaustivePlanner jesp(3,&dectiger);
    jesp.Plan();
    std::cout << jesp.GetExpectedReward() << std::endl;
    std::cout << jesp.GetJointPolicy()->SoftPrint() << std::endl;
    return(0);
}
```

(a) Example program.

```
src/examples>  ./decTigerJESP
Value computed for DecTiger horizon 3: 5.19081
Policy computed:
JointPolicyPureVector index 120340 depth 999999
Policy for agent 0 (index 55):
Oempty,  --> a00:Listen
Oempty, o00:HearLeft,  --> a00:Listen
Oempty, o01:HearRight,  --> a00:Listen
Oempty, o00:HearLeft, o00:HearLeft,  --> a02:OpenRight
Oempty, o00:HearLeft, o01:HearRight,  --> a00:Listen
Oempty, o01:HearRight, o00:HearLeft,  --> a00:Listen
Oempty, o01:HearRight, o01:HearRight,  --> a01:OpenLeft
Policy for agent 1 (index 55):
Oempty,  --> a10:Listen
Oempty, o10:HearLeft,  --> a10:Listen
Oempty, o11:HearRight,  --> a10:Listen
Oempty, o10:HearLeft, o10:HearLeft,  --> a12:OpenRight
Oempty, o10:HearLeft, o11:HearRight,  --> a10:Listen
Oempty, o11:HearRight, o10:HearLeft,  --> a10:Listen
Oempty, o11:HearRight, o11:HearRight,  --> a11:OpenLeft
src/examples>  ../utils/evaluateJointPolicyPureVector -h 3 \
-r 10000 ../../problems/dectiger.dpomdp 120340
Horizon 3 reward: 5.1
```

(b) Program output.

Figure 4: (a) C++ code of a small example program that runs JESP on the DecTiger problem, and (b) shows its output (with enhanced printout).

The output of the program is shown in Figure 4(b). Lines 6 through 12 show the resulting policy for the first agent, and lines 14 through 20 the second agent's policy. For instance, lines 12 and 20 show that both policies specify that the agents should open the left door after hearing the tiger on the right twice. Line 21 shows how to empirically evaluate the computed joint policy (index 120340) on the parsed DecTiger description for horizon 3 using 10000 sample runs. As such, it briefly demonstrates the command-line parsing functionality and the use of the `.dpomdp` parser.

6 Implementation and platform

As mentioned before, the MADP toolbox is implemented in C++ and is licensed under the GNU General Public License (GPL). We develop it on Debian GNU/Linux, but it should work on any (recent) Linux distribution. It uses a standard GNU autotools setup and the GCC compiler. The software uses the Boost C++ libraries, in particular the Spirit library to implement the `.dpomdp` parser, and the uBLAS library for representing sparse vectors and matrices. A reference manual is available, and is generated from the source by doxygen. The manual can be consulted on-line at [9], where the source code can also be obtained.

7 Conclusions and future work

We introduced the design of the MADP toolbox, which aims to provide a software platform for research in decision-theoretic multiagent planning. Its main features are a uniform representation for several popular multiagent models, a parser for a text-based file format for discrete Dec-POMDPs, shared functionality for planning algorithms, as well as the implementation of several Dec-POMDP planners. The software has been released as free software, and as special attention was given to the extensibility of the toolbox, our hope is that it will be easy for others to use. Moreover, because the toolbox includes a set of benchmark problems and parser, we think it provides a useful tool to the research community.

The MADP toolbox remains under active development. Our focus has been partially observable models, however, we also would like to include observable models (MMDPs, SGs) and other special cases such as transition-

and observation independent Dec-(PO)MDPs in the future. In particular, we are currently working to incorporate support for factored (Dec-POMDP) models. Also, we are planning to add more advanced algorithms, such as GMAA* [10] (which includes MAA* [15]) and forward sweep policy computation [5]. We also hope that we will be able to include other algorithms such as memory bounded dynamic programming [13], DP-JESP [8], SPIDER [17], so as to provide a true MADP algorithm library. To accompany these planning algorithms, a more elaborate set of benchmark problems should be provided. Finally, contributions from the community are welcomed.

Acknowledgments

This work was partially supported by Fundação para a Ciência e a Tecnologia (ISR/IST pluriannual funding) through the POS_Conhecimento Program that includes FEDER funds, and through grant PTDC/EEA-ACR/73266/2006. The research reported here is part of the Interactive Collaborative Information Systems (ICIS) project, supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024.

References

[1] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The complexity of decentralized control of Markov decision processes. *Math. Oper. Res.*, 27(4):819–840, 2002.

[2] C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *TARK '96: Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, 1996.

[3] A. Cassandra. The POMDP page. `http://pomdp.org/pomdp/`.

[4] A. R. Cassandra. *Exact and approximate algorithms for partially observable Markov decision processes*. PhD thesis, Brown University, 1998.

[5] R. Emery-Montemerlo, G. Gordon, J. Schneider, and S. Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *Proc. of Int. Joint Conference on Autonomous Agents and Multi Agent Systems*, 2004.

[6] E. A. Hansen, D. S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proc. of the National Conference on Artificial Intelligence*, 2004.

[7] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[8] R. Nair, M. Tambe, M. Yokoo, D. V. Pynadath, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proc. Int. Joint Conf. on Artificial Intelligence*, 2003.

[9] F. A. Oliehoek and M. T. J. Spaan. MultiAgent Decision Process page. `http://www.science.uva.nl/~faolieho/madp`.

[10] F. A. Oliehoek, M. T. J. Spaan, and N. Vlassis. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 2008. To appear.

[11] D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.

[12] Resource-Bounded Reasoning Lab, University of Massachusetts at Amherst. Dec-POMDP web page. `http://rbr.cs.umass.edu/~camato/decpomdp/`.

[13] S. Seuken and S. Zilberstein. Memory-bounded dynamic programming for DEC-POMDPs. In *Proc. Int. Joint Conf. on Artificial Intelligence*, 2007.

[14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998.

[15] D. Szer, F. Charpillet, and S. Zilberstein. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In *Proceedings of the Twenty First Conference on Uncertainty in Artificial Intelligence*, 2005.

[16] TEAMCORE Research Group, University of Southern California. Distributed POMDPs web page. `http://teamcore.usc.edu/pradeep/dpomdp_page.html`.

[17] P. Varakantham, J. Marecki, Y. Yabu, M. Tambe, and M. Yokoo. Letting loose a SPIDER on a network of POMDPs: Generating quality guaranteed policies. In *Proc. of Int. Joint Conference on Autonomous Agents and Multi Agent Systems*, 2007.