The MADP Toolbox 0.3.1

Frans A. Oliehoek
University of Amsterdam, Amsterdam, The Netherlands
University of Liverpool, Liverpool, United Kingdom

Matthijs T. J. Spaan
Delft University of Technology
Delft, The Netherlands

Philipp Robbel
Massachusetts Institute of Technology
Cambridge, MA, USA

João V. Messias
University of Amsterdam, Amsterdam, The Netherlands

April 10, 2015

Abstract

This is the user and developer guide accompanying the version 0.3.1 release of the Multiagent Decision Process (MADP) Toolbox. It is meant as a first introduction to the organization of the toolbox, and tries to clarify the approach taken to certain implementation details. In addition, it covers a few typical use cases and provides an installation guide. This document complements the automatically generated API reference.

Contents

1 Introduction

This text describes the Multiagent Decision Process (MADP) Toolbox version 0.3.1, which is a software toolbox for scientific research in decision-theoretic planning and learning in multiagent systems (MASs). We use the term MADP to refer to a collection of mathematical models for multiagent decision making: multiagent Markov decision processes (MMDPs) [6], decentralized MDPs

(Dec-MDPs), decentralized partially observable MDPs (Dec-POMDPs) [2], partially observable stochastic games (POSGs) [12], etc.

The toolbox is designed to be rather general, potentially providing support for all these models, although so far most effort has been put in planning algorithms for discrete Dec-POMDPs. It provides classes modeling the basic data types of MADPs (e.g., action, observations, etc.) as well as derived types for planning (observation histories, policies, etc.). It also provides base classes for planning algorithms and includes several example applications using the provided functionality. Additionally, the toolbox also provides a number of more mature 'solvers' that can be used to solve certain MADPs. For instance, applications that use JESP or brute-force search to solve problems (specified as `.dpomdp` files) 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.

Here we highlight some of the functionality of the MADP toolbox:

- (Multiagent) MDP solvers and learning:
  - value iteration
  - Q-learning
  - export to SPUDD
- (Multiagent) POMDP solvers:
  - Perseus [32]
  - Monahan [16]
  - Incremental Pruning [8]
- Dec-POMDP solvers:
  - JESP [18]
  - DP-LPC [5]
  - (Generalized) Multiagent A* (GMAA) and variants [35, 24, 29, 33]
  - GMAA-ELSI for factored Dec-POMDPs [25]
- Parsers:
  - .pomdp (Tony Cassandra's POMDP file format [7])
  - .dpomdp (a Dec-POMDP extension of Tony Cassandra's format, see Section 6.2)
  - .pgmx (a file format for factored models, see Section 7)

A more detailed description of features can be found in the developer guide (see Section 8).

This document is split in two parts. The first part is intended for people that intend to use MADP as a "out-of-the-box" tool without writing any code themselves. It presents a mathematical model of the family of MADPs, which also introduces notation, gives a high-level overview of the toolbox and an example of how to use it. The second part, is intended for people that do want to use MADP in their own code. It gives pointers to useful classes for typical functionality and some more specific design choices and mechanisms are explained.

Part I
User Guide

Even though MADP is designed as a collection of libraries, with as the primary goal to use this in one's own code, there is also much functionality that can be used "out of the box". This part of the document is intended for people that want to use MADP to perform simulations, reinforcement learning or solve a planning problem, without writing code themselves.

2 For the Impatient: Compiling, and Running an MADP Program

You should be able to compile the MADP toolbox on any recent Linux distribution (for details see Appendix A). It makes use of GNU autotools and therefore, a typical installation is as follows:

```
tar xfz madp-0.3.1.tar.gz
cd madp-0.3.1
./configure
make
```

It is not required to `make install` to use the toolbox. Please see Appendix A for more information about installation.

Now in order to run your first program do

```
src/examples/example_BFS DT -h2 -v
```

which should solve the horizon $h = 2$ `DecTiger` problem and produce the following output:

```
ArgumentUtils: Problem DecTiger instantiated.
BruteForceSearchPlanner initialized
value=-4
JointPolicyPureVector:
JPolComponent_VectorImplementation index 0
Policy for agent 0 (index 0):
Oempty,  --> a00:Listen
Oempty, o00:HearLeft,  --> a00:Listen
Oempty, o01:HearRight,  --> a00:Listen
Policy for agent 1 (index 0):
Oempty,  --> a10:Listen
Oempty, o10:HearLeft,  --> a10:Listen
Oempty, o11:HearRight,  --> a10:Listen
```

This tells you that the problem is solved and that the optimal value of this problem (a Dec-POMDP [2]), is -4. The optimal policy is the one with index 0 (and is represented by the class `JointPolicyPureVector`). The rest of the output specifies the behavior of this optimal joint policy: both agents perform the action `Listen` for all their observation histories. ("`Oempty`" is the empty observation that agents have at the start of the problem, and "`Oempty, o00:HearLeft`" encodes the history where the agent has gotten the observation `HearLeft`).

If you have your own `.dpomdp` file, you can try to solve it using, e.g., GMAA*-ICE [33, 29]:

```
src/solvers/GMAA-ICE <PATH-TO-YOUR-.dpomdp-FILE>  -h2
```

MADP includes a number of .dpomdp and .pgmx problem files, others can be found online at `http://masplan.org/problem_domains`.

3 Theory: MADPs and Basic Notation

Even though MADP is a toolbox aimed at finding numerical solutions for all kinds of multiagent planning and learning problems, these problems themselves have formal definitions. As such, this section provides some theoretical background here with respect to the models that can be represented and solved in MADP.

As mentioned, MADPs encompass a number of different models. Here we briefly introduce the components of these mathematical models and some basic notation. For a more extensive introduction to these models, see, e.g., [31, 20].

3.1 Discrete Time MASs

An MADP is often considered for a particular finite number of discrete time steps t. When searching policies (*planning*) that specify h actions, this number is referred to as the (planning-)*horizon*. So typically we look at time steps:

$$t = 0, 1, 2, ..., h-2, h-1.$$

At each time step:

- The world is in a specific state $s \in \mathcal{S}$.
- Each agent receives an individual observation: a (noisy) observation of the environment's state.
- The agents take an action.

The individually selected actions form a joint action. After such a joint action, the system jumps to the next time step. In this jump the system's state may change stochastically, and this transition is influenced by the taken joint action. In MADPs (such as the Dec-POMDP), there are transition and observation functions describing the probability of state transitions and observations.

3.2 Basic MADP Components

More formally, a multiagent decision process (MADP) consists of a subset of the following components:

- $\mathcal{D} = \{1, \ldots, n\}$, a finite set of n agents.
- $\mathcal{S}$ is a finite set of world states.
- The set $\mathcal{A} = \times_i \mathcal{A}_i$ is the set of *joint actions*, where $\mathcal{A}_i$ is the set of actions available to agent i. Every time step, one joint action $\boldsymbol{a} = \langle a_1, ..., a_n \rangle$ is taken. Agents do not observe each other's actions.
- $\mathcal{O} = \times_i \mathcal{O}_i$ is the set of joint observations, where $\mathcal{O}_i$ is a finite set of observations available to agent i. Every time step one joint observation $\boldsymbol{o} = \langle o_1, ..., o_n \rangle$ is received, from which each agent i observes its own component o_i.
- $b^0 \in \triangle(\mathcal{S})$, is the initial state distribution at time $t = 0$.[1]
- A transition function that specifies the probabilities $\Pr(s'|s, \boldsymbol{a})$.
- An observation function that specifies the probabilities $\Pr(\boldsymbol{o}|\boldsymbol{a}, s')$.
- A set of reward functions $\{R_i\}$ that specify the payoffs of the agents.

The partially observable stochastic game (POSG) is the most general model in the MADP family. Dec-POMDPs are similar, but all the agents receive the same reward, so only 1 reward function is needed.

Unless stated otherwise, we use superscript for time indices. I.e., a_i^2 denotes the agent i's action at time $t = 2$.

3.3 Histories

Let us more formally consider what the history of the process is. An MADP history of horizon h specifies h time steps $t = 0, ..., h-1$. At each of these time steps, there is a state s^t, joint observation $\boldsymbol{o}^t$ and joint action $\boldsymbol{a}^t$. 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:

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

Here s^0 is the initial state, drawn according to the initial state distribution b^0. The initial joint observation $\boldsymbol{o}^0$ is usually assumed to be the empty joint observation: $\boldsymbol{o}^0 = \boldsymbol{o}_\emptyset = \left\langle o_{1,\emptyset}, ..., o_{n,\emptyset} \right\rangle$. (In

[1] We use $\triangle(X)$ to denote the infinite set of probability distributions over the finite set X.

multiagent decision making, it customary to assume that all information that is available before to process is started is collected in the initial distribution over states, and consequently in the MADP toolbox there is no initial observation.)

In many MADPs, an agent can only observe his own actions and observations. Therefore we introduce notions of histories from the perspective of an agent. We start with the *action-observation history* of agent i at time step t:

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

note that the choice points for the agents are right before the action:

$$\bar{\theta}_i^k = \left(\uparrow_{t=0} a_i^0, o_i^1, \uparrow_{t=1} a_i^2, ..., o_i^{k-1}, \uparrow_{t=k-1} a_i^{k-1}, o_i^k \right)$$

Therefore, when we write $\bar{o}_i^t = \left(o_i^1, , ..., o_i^{t-1}, o_i^t\right)$ for agent i's *observation history* at time step t and $\bar{a}_i^t = \left(a_i^0, a_i^1, , ..., a_i^{t-1}\right)$ for the *action history* of agent i at time step t. We can thus redefine the action-observation history as: $\bar{\theta}_i^t \triangleq \langle \bar{o}_i^t, \bar{a}_i^t \rangle$. For time step $t = 0$, we have that $\bar{a}_i^0 = (()) = \bar{\boldsymbol{a}}_\emptyset$ and $\bar{o}_i^0 = (()) = \bar{\boldsymbol{o}}_\emptyset$ are empty sequences.

3.4 Policies, Planning & Learning

The overall goal of the toolbox is to support algorithms that allow agents to behave intelligently in MADPs. Roughly, we can discriminate two main types of approaches:

Planning When given a complete specification of the environment (i.e., the MADP), 'all' there is left to do is to compute policies that seem reasonable for that particular model. For instance, for a Dec-POMDP we may want to compute an optimal joint policy, while for a POSG we may want to find a joint policy that forms a Nash equilibrium.

(Reinforcement) Learning When the model is not completely known in advance, the agents will not need to merely execute a policy that is computed for them. Instead, the agents will need to interact repeatedly in the environment to learn about the environment (and possibly each other), updating their policy as a result of these interactions.

Clearly, the problems in learning settings are even bigger than those in planning settings, but planning settings are already very hard by themselves. Both planning and learning revolves around finding 'policies'.

For instance, when planning for ('solving') a finite-horizon Dec-POMDP, we typically try and find a joint policy, i.e., a tuple of policies

$$\boldsymbol{\pi} = \langle \pi_1, \ldots, \pi_n \rangle$$

with individual policies π_i that deterministically map observations histories to actions (i.e., $\pi_i(\bar{o}_i) = a_i$). The goal in a Dec-POMDP typically is to optimize the expected expected cumulative reward:

$$V(\boldsymbol{\pi}) = \mathbf{E}\left[\sum_{t=0}^{h-1} R(s, \boldsymbol{a}) \mid \boldsymbol{\pi}, b^0\right],$$

where the expectation is over the realization of sequences of states and observations.

Similarly, learning settings also aim at finding policies. For instance Q-learning [34] is a standard RL method for single-agent fully observable problems. It finds a policy that is implicitly represented by Q-values $Q(s, a_i)$: the represented policy is the one that is greedy with respect to these values. That is:

$$\pi_i(s) \triangleq \arg\max_{a_i} Q(s, a_i).$$

Since Q-learning continually adapts the Q-values it effectively searches through the space of policies.

These two examples show that in very different settings there is still a common ground: both settings are looking for policies and these policies map from a type of internal state (observation

history and global state respectively) to actions. The MADP Toolbox is designed to be flexible enough to allow such differences by making explicit the domain of policies (e.g., see Section 14 for more details). Moreover, the question of what the environment is like (e.g., stochastic and/or partially observable) is for a large part orthogonal to whether we are in a learning or planning setting. As such the MADP toolbox's philosophy is to separate the representation of these environments from the planning and learning algorithms, making them usable for both types of algorithms.

4 Finding Things: Useful Directories

When using MADP out of the box, many of the directories can be ignored. The following directories, however, do contain useful information and/or programs:

path	description
`/`	Package root.
`/doc`	Documentation.
`/doc/html`	The html documentation generated by doxygen. (perform 'make htmldoc' in root.)
`/problems`	A number of problems in .dpomdp file format.
`/src`	C++ code
`/src/solvers`	Code for a number of executables that use the MADP libs to implement (Dec-)POMDP solvers.
`/src/utils`	Code for a number of executables that perform auxiliary tasks (e.g., printing problem statistics or evaluating a joint policy through simulation).

Directories for Results and Problem Specification In addition, MADP is coded to use the following default locations:

```
~/.madp/results
~/.madp/problems
```

for writing results, and reading problem descriptions respectively. Here the tilde (~) denotes your home directory. In order to let MADP find problem description files without specifying the full path, it is recommended to create a symbolic link from MADP's problems directory to `~/.madp/problems`. Alternatively, one can copy the problems one wants to use to `~/.madp/problems`.

5 Using the Toolbox: Some Examples

Here we show how to use some of the command-line tools MADP provides.

5.1 General Options

Before we start discussing particular methods, we point out that nearly all provided tools expect standard unix style arguments. In particular, they typically support to following standard options:

```
General options
 -q, -s, --quiet, --silent  Don't produce any output
 -v, --verbose              Produce verbose output. Specifying this option
                            multiple times increases verbosity.
 -?, --help                 Give this help list
     --usage                Give a short usage message
 -V, --version              Print program version
```

We have tried to make the help messages as clear as possible, so when in doubt on the usage of a program, a first good step is to run it with the `--help` option.

In addition, there are some options that most tools in the toolkit support, such as:

`--sparse` Specifies that the program should make use of sparse datastructures (e.g., to represent transition matrices, etc.)

`--dry-run` Specifies that the program should not actually try to write the results to an output file.

`--horizon` Specifies the *finite* horizon over which we want to solve the problem.

`--inf` Used to specify an infinite horizon.

5.2 Solving a Dec-POMDP

Much of the functionality currently present in MADP is directed at solving finite-horizon Dec-POMDPs. Here we show how to use some of them, focusing on GMAA*-ICE, a state-of-the-art optimal solution method. We also briefly discuss other methods.

Generalized MAA* One of the most comprehensive solvers in the MADP toolbox is the program `GMAA` which implements a whole range of algorithms in the 'generalized multiagent A*' family [20]. All these algorithms are variants of heuristic search (i.e., MAA* [35]) that use collaborative Bayesian games (CBGs, also referred to as Bayesian Games with Identical Payoffs, BGIPs, in many parts of the code) to represent the one-stage node expansion problems.

The `GMAA` solver has two main options that determine the working (what algorithm is performed):

- The BGIP_SOLVERTYPE parameter, specified with `-B` or `--BGIP_Solver`, specifies the type of Bayesian game solver used:
  - `BFS`, solve the CBGs using brute-force search.
  - `AM`, approximate solution via alternating maximization.
  - `CE`, approximate solution via Cross-Entropy optimization.
  - `MP`, approximate solution via Max-Plus.
  - `BnB`, Branch-and-Bound (see BnB options)
  - `Random`, gives random solutions, for testing purposes
- The GMAA parameter is specified using `-G` or `--GMAA` and specifies whether the method performs a full backtracking heuristic search, samples just one path from root to leaf, or does something in between [24]. In particular, you can specify the following values:
  - `MAAstar`, this is the option to select full backtracking (MAA*) search. It performs incremental expansion [29] of the search nodes.
  - `FSPC`, this selects forward-sweep policy computation: at every node in the search tree it only expands the most promising child.
  - `kGMAA`, this uses an argument (specified using option `-k`) to expand only the k most promising children at each node.
  - `MAAstarClassic`, this is an older version that uses a built-in BFS solver (and does not do incremental expansion).

Not all combination of the above two options are possible. For instance, `MAAStar` requires an optimal CBG solver that can (incrementally, in order of the heuristic value) deliver all children. Nevertheless, many combinations are useful. For instance

```
./GMAA -G FSPC -B AM -h5 DT
```

will run FSPC with alternating maximization on the horizon $h = 5$ Dec-Tiger problem. (Essentially the BAGA approximation method [10] without pruning or clustering). While

```
./GMAA --GMAA=MAAstar --useBGclustering --BGIP_Solver=BnB -h4 DT
```

runs GMAA*-ICE. (Note that there is also a `GMAA-ICE` shell script provided for convenience.)

Finally, an important option of all `GMAA` flavors is the `QHEUR` option (specified with `-Q`) which determines the heuristic. Currently, the supported heuristics are:

```
QMDP                (defined on joint beliefs)
QPOMDP              (defined on joint history tree)
QBG                 (defined on joint history tree)
QMDPc               (cached for each joint AO history)
QPOMDPav            (uses alpha vectors over joint beliefs)
QBGav               (uses alpha vectors over joint beliefs)
QHybrid             (hybrid between vector and trees, customizable)
QPOMDPhybrid        (QPOMDP hybrid between vector and trees, no options)
QBGhybrid           (QBG hybrid between vector and trees, no options)
QBGTreeIncPrune     (vector-based QBG using tree-based inc. pruning with
                     memoization)
QBGTreeIncPruneBnB(vector-based QBG using tree-based inc. pruning with
                     branch-and-bound)
```

These heuristics can also be pre-computed and stored to disk using `utils/calculateQheuristic`, in which case they can be loaded from disk by GMAA by specifying `--useQcache`.

Other Provided Methods A number of other methods for solving Dec-POMDPs are also provided:

BFS Brute-force search. Very slow, but instructive.

JESP Joint Equilibrium-based Search for policies [18]. Performs alternating maximization in the space of entire policies. This is the dynamic-programming version of JESP.

GMAA_ELSI Exploits factored structure in the last stage of factored Dec-POMDPs [25].

DICEPS Direct CE Policy Search [23]. An approximate method based on CE optimization. Not always the best method, but tends to give you at least an answer on bigger problems.

DP-LPC Dynamic Programming with Lossless Policy Compression [5]. This is a port of Boularias' code and is limited to two-agent problems. It requires CPLEX.

5.3 Solving a (Multiagent) POMDP with Perseus

MADP provides exact POMDP solving algorithms such as Monahan's algorithm [16] with incremental pruning [8] (discussed below in Section 5.4) but here we show how a (multiagent) POMDP can be approximately solved by the Perseus algorithm [32]. It operates on POMDP models with a single agent or with multiple agents, in which case it considers the centralized POMDP defined over joint actions, observations and states. Currently, Perseus only supports infinite-horizon problems, which means that you must specify the `--inf` command line flag, as well as a discount factor (`--discount=XXX`).

The most important options of the `Perseus` program are the following:

- `-n` specifies how many beliefs should be sampled for the belief set on which Perseus operates. Add `-u` if you want unique beliefs (by default duplicates are allowed). Other options related to belief sampling are `-H`, indicating the horizon for the belief sampling process (after this many steps, the sampling will be restarted from the initial belief), `-Q` to indicate to follow a QMDP policy when sampling beliefs instead of taking actions uniformly at random. The `-x` option allows you to specify the probability of taking a random action instead of the QMDP one when using `-Q`.
- `-b` specifies which type of backup to use, which defaults to the standard POMDP backup. Other possibilities are the BG backup [24] or the backup for event-driven models (see 7.2).

For instance,

```
Sampling 50 beliefsWarning: sampling beliefs for an infinite horizon without reset.
PerseusPOMDP: max reward in beliefset is 17.8859
PerseusPOMDP: iteration      0 |V| 1 sumV/nrB -1010 V0 -1010 (best -1.79769e+308)
Added vector for 2 (V -911 improved 50)
...
...
PerseusPOMDP: iteration    169 |V| 3 sumV/nrB 60.5307 V0 59.8165 (best 59.8165)
Added vector for 26 (V 59.8166 improved 47)
Added vector for 30 (V 71.7207 improved 2)
Added vector for 6 (V 71.7207 improved 1)
```

Figure 1: Sample Perseus output.

```
$ cd src/solvers
$ ./Perseus ../../problems/dectiger.dpomdp --inf --discount=0.9 -d  -n50
```

should run Perseus on dec-tiger, i.e., treating it as a (multiagent) POMDP, rather than a Dec-POMDP and output something like the output shown in Figure 1.

5.4 Other POMDP Methods

MADP incorporates more POMDP machinery, albeit not implemented as fully equipped solvers; much of the POMDP functionality has actually been used in order to provide heuristics for Dec-POMDPs. Nevertheless, this can still be used to solve (finite-horizon) POMDPs exactly.[2]

In particular, one can use the `calculateQheuristic` utility to perform the Monahan algorithm [16] with incremental pruning [8].[3] For instance, it can executed as follows:

```
$ cd src/utils
$ ./calculateQheuristic DT -h6 -Q QPOMDPav
MonahanPOMDPPlanner: t 5 contains < 1 1 1 1 1 1 1 1 > vectors (total 9)
MonahanPOMDPPlanner::BackupStage < 7 1 1 1 1 1 1 1>
MonahanPOMDPPlanner: t 4 contains < 7 1 1 1 1 1 1 1 1 > vectors (total 24)
MonahanPOMDPPlanner::BackupStage < 11 1 1 1 1 1 1 1 1>
MonahanPOMDPPlanner: t 3 contains < 11 1 1 1 1 1 1 1 1 > vectors (total 43)
MonahanPOMDPPlanner::BackupStage < 15 1 1 1 1 1 1 1 1>
MonahanPOMDPPlanner: t 2 contains < 15 1 1 1 1 1 1 1 1 > vectors (total 66)
MonahanPOMDPPlanner::BackupStage < 19 1 1 1 1 1 1 1 1>
MonahanPOMDPPlanner: t 1 contains < 19 1 1 1 1 1 1 1 1 > vectors (total 93)
MonahanPOMDPPlanner::BackupStage < 19 1 1 1 1 1 1 1 1>
MonahanPOMDPPlanner: t 0 contains < 19 1 1 1 1 1 1 1 1 > vectors (total 120)
MonahanPOMDPPlanner[h=6]: Vjb0=19.7164
Wallclock: from 1421058982.981798 until 1421058983.7764 which took 2 clock
ticks
Q saved to /home/frans/.madp/results/GMAA/DecTiger/QAVMonahanPOMDPheuristic_h6
ComputeQ: 0.03 s in 1 measurements, max 0.03, avg 0.03, min 0.03
Overall: 0.03 s in 1 measurements, max 0.03, avg 0.03, min 0.03
Parsing: 0 s in 1 measurements, max 0, avg 0, min 0
PlanningUnit: 0 s in 1 measurements, max 0, avg 0, min 0
Save: 0 s in 1 measurements, max 0, avg 0, min 0
WallclockTime: 0.02 s in 1 measurements, max 0.02, avg 0.02, min 0.02
Timings saved to /home/frans/.madp/results/GMAA/DecTiger/calculateQheuristicQAV
```

[2]Infinite-horizon POMDPs can also be specified using the, e.g., `--inf --discount=0.9` switches, but are effectively treated as a 1 million stage finite-horizon POMDP.

[3]Since there typically is no need to do so, incremental pruning is enabled by default and cannot be disabled on the command line. If one really needs to disable incremental pruning, it is possible to do so by modifying the code.

```
MonahanPOMDP_h6_Timings
Value of jaohI 0 = 19.7164
```

This shows that the `DecTiger` problem is solved for $h = 6$ leading to a value of 19.7164 for the initial belief (corresponding to the joint action-observation history with index 0). It also reports timing results, and gives statistics on the number of 'α-vectors' used to represent each stage (it shows the number of α-vectors associated with each joint action, e.g., for $t = 0$ we have 19 vectors for $\langle Listen, Listen \rangle$ and 1 vector for all other joint actions).

Alternatively, the (M)POMDP solution can be computed by performing dynamic programming over the tree of joint action-observation histories (using `-Q QPOMDP`) or making use of a hybrid (using `-Q QPOMDPhybrid)` representation [29].

`calculateQheuristic` can also be used to use Monahan's algorithm compute the so-called 'QBG' value function [24] (using `-Q QBGav`), which gives the optimal solution of a Dec-POMDP under 1-step-delayed synchronizing communication. It also supports the *tree-based incremental pruning* algorithms (using `-Q QBGTreeIncPrune` and `-Q QBGTreeIncPruneBnB`) [21].

5.5 Planning: Solving a (Multiagent) MDP

MADP provides an implementation of value iteration (VI) for finite and infinite-horizon problems to compute an optimal (joint) policy. As for the multiagent POMDP solver, VI operates on MDPs with a single agent, or, alternatively, multiagent (MMDP) models in which case the centralized MDP in joint action and state space is solved. As usual, infinite-horizon problems are indicated with the `--inf` parameter (and require a discount factor of less than 1) whereas finite-horizon problems can be specified using the `--horizon` switch.

Note that the MMDP_VI solver also simulates the resulting optimal policy and outputs statistics (e.g., average reward) on the command-line and log files. For simulation, two options of relevance are:

- `--runs` to specify the number of iterations over which the (joint) policy is simulated.
- `--seed` to specify the random number generator seed for simulations.

For instance,

```
$ cd src/solvers
$ ./MMDP_VI ../../problems/dectiger.dpomdp --inf --discount=0.9
```

runs VI on dec-tiger, i.e., treating it as a fully-observable (multiagent) MDP, rather than a Dec-POMDP.

5.6 Learning in a (Multiagent) MDP

For *infinite-horizon* problems, MADP comes with an implementation of the Q-learning algorithm. Q-learning Sutton and Barto [34] is a single-agent learning method and hence in principle operates on MDPs with a single agent. However, it is also possible to run it on MMDPs in which case learning is run on the centralized MDP in joint action and state space. Only infinite-horizon problems are supported, therefore a discount factor has to be specified via the `--discount` switch.

The most important options of the MMDP_QLearner program are the following:

- `--nrRuns` number of learning episodes, each consisting of a reset to a random initial state followed by h learning iterations (where $h = 999999$ is chosen inside the solver for infinite horizon problems). Each learning iteration is performed in joint action and state space using the well-known Q-learning update rule.
- `--seed` random number seed.
- `--verbose` run value iteration (VI) on identical (M)MDP in addition to Q-learning and output first row of Q-table for both VI and Q-learning to the command-line. Both results should be comparable if `--nrRuns` is set large enough.

For instance,

```
$ cd src/solvers
$ ./MMDP_QLearner DT --discount 0.9 --verbose --runs 100
```

runs Q-learning (and VI, since `--verbose` is specified) on `Dec-Tiger`, i.e., treating it as a fully-observable (multiagent) MDP, rather than a Dec-POMDP. Note that for MMDP_QLearner, `--inf` is implicit and does not need to be specified.

6 Specifying Problems: File Formats, etc.

There are three main ways to specify problems in MADP: in the `ProbModelXML` format, as a `.dpomdp` file, or as a sub-class of a suitable a `MultiAgentDecisionProcess`.

6.1 Using the OpenMarkov Graphical Editor

OpenMarkov (`http://www.openmarkov.org/users.html`) is a GUI editor for creating Factored Models in the ProbModelXML format. Using this tool is treated in detail in Section 7.

6.2 Specifying & Parsing `.pomdp` & `.dpomdp` files

For single-agent POMDPs, it is possible to use Tony Cassandra's `.pomdp` file format, which is specified at `http://www.pomdp.org/code/pomdp-file-spec.shtml`, to specify problems. For non-factored (i.e., typically smaller) multiagent problems, an easy way to specify them is to create a `.dpomdp` file. It is easiest to get an understanding of this format by example: the code for the (in)famous decentralized tiger benchmark is illustrated in Figure 2. The figure clearly shows that there is are parts to specify the number of agents, the states, the possible actions and observations, and finally the transition-, observation- and reward model. A version of this file including comment is available in the `problems/` directory. This also contains `example.dpomdp` which provides even further information.

6.3 Specifying Problems as a Sub-Class

Another way to specify problems is to actually program them. This has some advantages in terms of space and time requirements. Please refer to the second part of this document (Section 11) for more information.

7 The ProbModelXML Format

MADP can parse problem files written in the *ProbModelXML* format [1]. This format is useful for the definition of Factored MDPs / POMDPs / Dec-POMDPs, or any other class of problems that can be represented graphically as a two-time-slice Dynamic Bayesian Network (DBN). You can find the complete specification of the ProbModelXML format at: `http://www.cisiad.uned.es/ProbModelXML/`. You should save your ProbModelXML files with the ".pgmx" extension.

Since ProbModelXML supports many different probabilistic graphical models and concepts that are outside of the scope of MADP, there are some constraints on what can be interpreted by the MADP ProbModelXML parser:

Network Types: The `MDP`, `POMDP`, and `DEC_POMDP` formats are supported. The network type is actually inferred by the parser, so you can always safely specify `DEC_POMDP` as the network type, even if you are designing a less general model. This is also valid for centralized multiagent models (MMDPs, MPOMDPs).

Variables: Only discrete, finite-domain variables (state factors, actions, observations and reward factors) are currently supported. You can only specify variables for time "0" and "1", which respectively represent time steps t and $t+1$ in the two-time-slice DBNs. In ProbModelXML terminology, model variables can be defined as follows:

```
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

T: * :
uniform
T: listen listen :
identity

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-left : hear-right hear-left : 0.1275
O: listen listen : tiger-left : hear-right hear-right : 0.0225
O: listen listen : tiger-right : hear-right hear-right : 0.7225
O: listen listen : tiger-right : hear-left hear-right : 0.1275
O: listen listen : tiger-right : hear-right hear-left : 0.1275
O: listen listen : tiger-right : hear-left hear-left : 0.0225

R: listen listen: * : * : * : -2
R: open-left open-left : tiger-left : * : * : -50
R: open-right open-right : tiger-right : * : * : -50
R: open-left open-left : tiger-right : * : * : +20
R: open-right open-right : tiger-left : * : * : 20
R: open-left open-right: tiger-left : * : * : -100
R: open-left open-right: tiger-right : * : * : -100
R: open-right open-left: tiger-left : * : * : -100
R: open-right open-left: tiger-right : * : * : -100
R: open-left listen: tiger-left : * : * : -101
R: listen open-right: tiger-right : * : * : -101
R: listen open-left: tiger-left : * : * : -101
R: open-right listen: tiger-right : * : * : -101
R: listen open-right: tiger-left : * : * : 9
R: listen open-left: tiger-right : * : * : 9
R: open-right listen: tiger-left : * : * : 9
R: open-left listen: tiger-right : * : * : 9
```

Figure 2: The `dectiger.dpomdp` file.

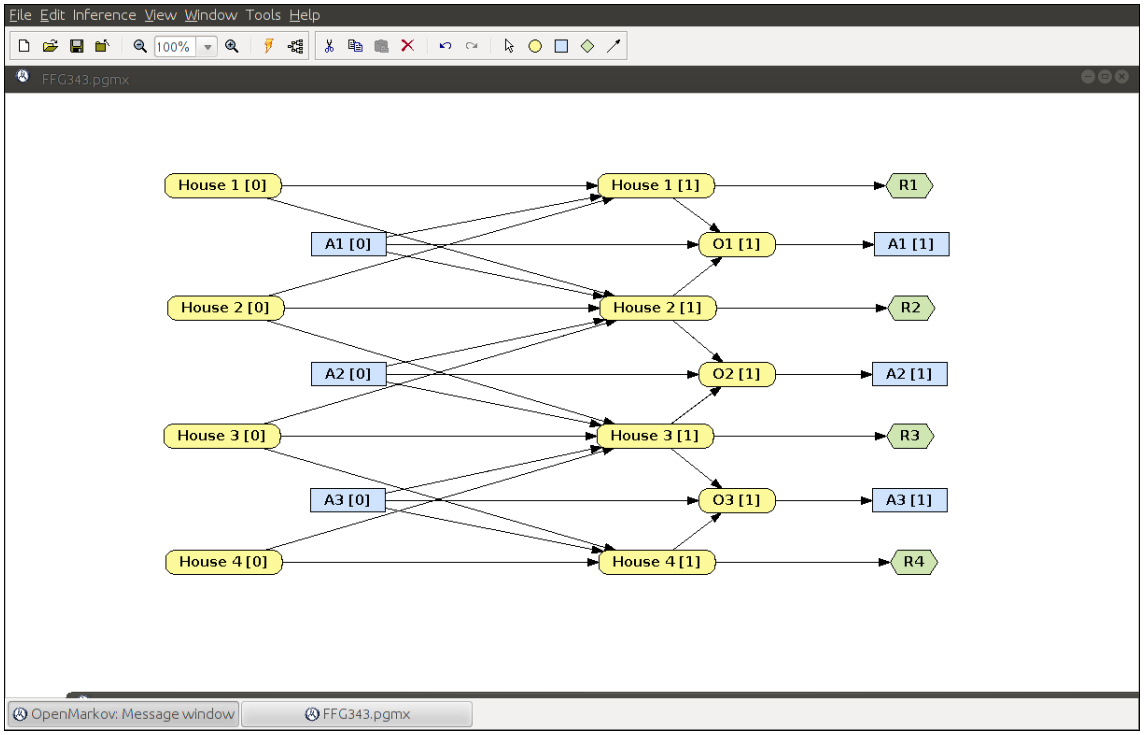


Figure 3: The Firefighting Graph (FFG) problem [19] in OpenMarkov.

- State Factors: *Chance* nodes, that should be defined both at time 0 and 1. The *Potential* of a state factor node at time 0 is its initial distribution, and at time 1 it is its Conditional Probability Distribution (CPD);
- Actions: *Decision* nodes, that should be defined both at time 0 and 1.
- Reward Factors: *Utility* nodes, that can be defined either at time 0 or at time 1 (but not both), depending on whether you want to represent $R(s^t, \mathbf{a})$ or $R(\mathbf{a}, s^{t+1})$.
- Observations: ProbModelXML does not explicitly recognize "observations" as a separate type of variable. Instead, observations are defined implicitly as time 1 "chance" nodes that link to the actions (at time 1) of their respective agents. Examples are shown in the following section. The *Potential* of an observation node defined in this way is its CPD.

CPDs: CPDs can only be defined as `Table`, `Tree/ADD`, or `Uniform`. Note that, internally, MADP only supports CPDs defined as tables, but you can still use decision trees or ADDs in the ProbModelXML representation - just keep in mind that they'll be "flattened" into tables.

Inference options: Even though ProbModelXML provides some options for probabilistic inference, this is ignored in MADP, since belief propagation is handled internally.

The MADP problems folder contains some examples of problem files written in ProbModelXML. The "Dec-Tiger" problem file ("DTPGMX.pgmx") can be referred to as a starting point to understand the general layout of this file format, as it includes additional descriptive comments.

7.1 Using OpenMarkov to Design Factored Problems

OpenMarkov (`http://www.openmarkov.org/users.html`) is a Java-based graphical editor for ProbModelXML files. Although using OpenMarkov is not strictly necessary to design ProbModelXML factored problems, it is the recommended option for this purpose, since it is far more intuitive and less time-consuming than coding the `.pgmx` file directly. As of the time of writing, OpenMarkov requires Java 7. The MADP ProbModelXML parser has been tested with the latest

(0.1.4) version of OpenMarkov. Note that since OpenMarkov is an independent project, and it is not authored or maintained by the MADP community, later versions are not guaranteed to be immediately compatible.

After you download OpenMarkov, move it to your MADP folder and rename it to something simpler (e.g. "openmarkov.jar"). Then you can start it by typing:

```
~/madp$ java -jar openmarkov.jar
```

After OpenMarkov is loaded, try to open one of the `.pgmx` problem files in the MADP `problems/` folder, for instance, `FFG343.pgmx`. Your view should then be similar to Figure 3.

In this view, *chance* nodes, shown in yellow, correspond to the state and observation variables of the problem; *decision* nodes, shown in blue, correspond to the actions of each agent; and *utility* nodes, shown in green, correspond to local reward factors. The bracketed number next to each node (`[0]` or `[1]`) represents the time-slice that it belongs to (t or $t+1$ respectively). Notice that observation variables (`O1`, `O2`, `O3`) are simply chance nodes, but they are only defined at time 1 and they link only to the actions of the agents that they belong to (`A1`, `A2`, `A3`, respectively).[4] This symbolizes that the actions of those agents at time $t+1$ "depend" on the values of these variables, although the way through which that dependency is manifested is not explicitly represented in the model. For fully observable problems, there is no need to specify observations, since all chance nodes are assumed to be observable. In that case, you can either ommit the time 1 action nodes or represent them as "orphaned" nodes.

Right-click a chance node and view its "Node properties". There, you can view and edit its name, time slice, and domain values (Figure 4). Other options are not relevant for MADP.

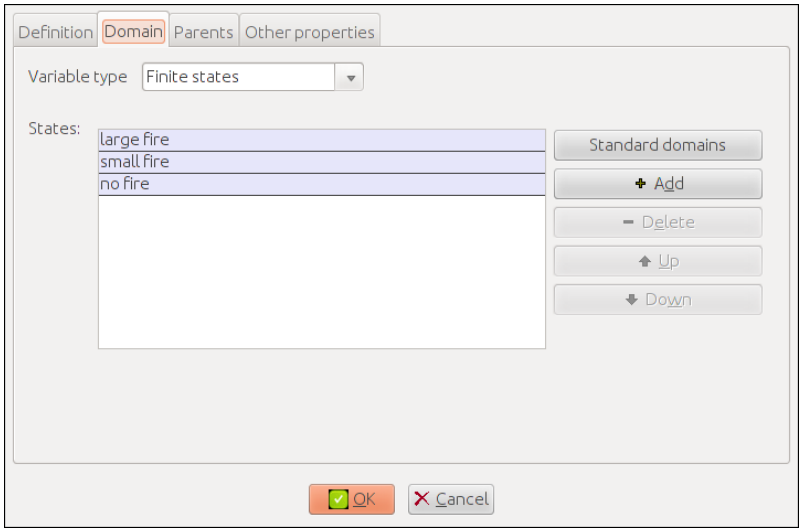


Figure 4: The domain values of a chance node in OpenMarkov. The variable type should always be "Finite states". You can add, delete, or rearrange the domain values. The bottom-most value will have index 0.

Likewise, right-click a time 1 chance node and select "Edit probability". Your view should be similar to Figure 5. In the Tree/ADD view, branches are represented by the white labeled boxes under each of the colored "variable" elements, each of them assigned to a particular value or set of values of that variable. You can expand or contract each branch by clicking to the left of its box. You can also right-click each branch to add or remove values (*a.k.a.* "states" in OpenMarkov), and right-click variable nodes to change their assignment (only possible if there are valid alternatives). In leaf nodes, you can right-click to "edit potential", i.e. define the CPD at that point. To define a

[4]That is, observation nodes are interpreted as observations by the MADP toolbox because they are the parent of an action node. In general, the ProbModelXML format would support multiple observation factors per agent (i.e., multiple parents for an action), but due to limitations of the internal representation employed by OpenMarkov, such more complex structures will not be exported.

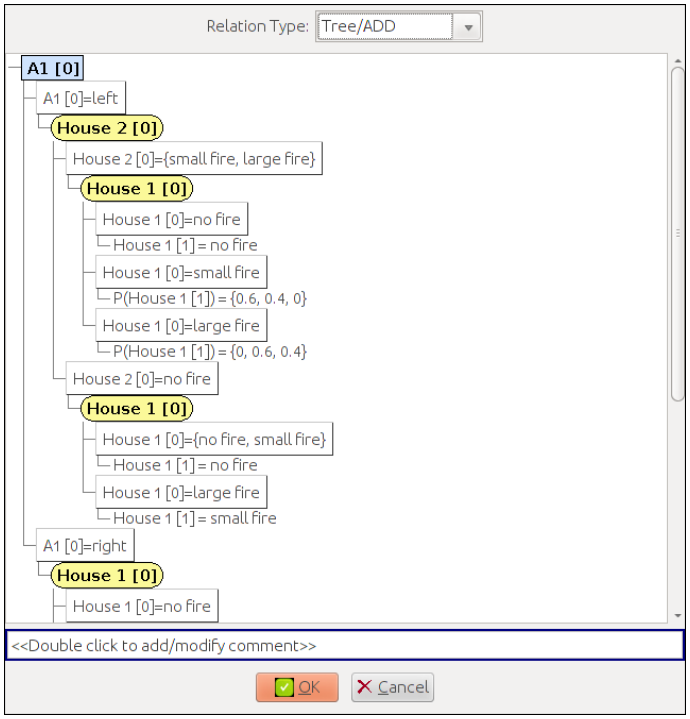


Figure 5: A $t+1$ CPD specified as a tree.

CPD as an ADD, you can assign a label to a branch ("set label"), and then you can bind subsequent equivalent branches to that label, so that you don't have to re-define them ("set reference").

A CPD can also be specified as a table (Figure 6). In that case, the various combinations of the parent variables will be shown at the top, and each row of the table corresponds to one specific value of the dependent variable (shown on the left). This implies that all columns should sum to 1.

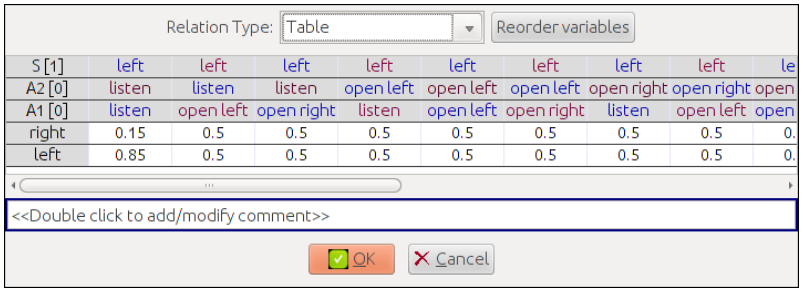


Figure 6: A $t+1$ CPD specified as a table (from "problems/DTPGMX.pgmx").

Again, in order to define the initial state distribution for your problem, you should define the *potentials* for all state variables in time slice 0. The potentials at time slice 1 encode the CPDs of all relevant variables, which are assumed to be stationary (do not depend on the absolute time index).

Rewards (*a.k.a. utilities*) can be defined in the same way as CPDs – the only difference is that they can have real-valued outcomes outside of the $[0, 1]$ interval.

To create a new problem file ("File→ New"), select "Dec-POMDP" as the network type, and simply follow the above guidelines. Here's what you **can't** do:

- You can't have time slices with indexes greater than 1;
- You can't have variables at time 1 that influence variables at time 0;
- You can't have non-stationary (dynamic) CPDs;

- You can define ADDs by "labeling" certain branches and linking to those labels elsewhere in the graph, but you can't link to labels of other ADDs;
- In partially-observable problems, you can't have a different number of observations and decision nodes. Both these should always be equal to the number of agents (an exception is discussed in the next subsection).

7.2 Designing Event-Driven Models

A special kind of model that is supported in MADP is the Event-Driven MPOMDP model [15]. In these models, state factor transitions are typically *asynchronous*, that is, only a small subset of state variables change between t and $t+1$ (typically only one). The change that occurs is called an *event*. Event-Driven MPOMDPs are not as straightforward to represent as two-time-slice DBNs as standard (Dec-)POMDPs due to this property. However, they can still be modeled by considering a "virtual" state-factor that encodes the underlying cause of the event and its probability of occurring. These factors are special in that they are influenced by time t variables, but they subsequently influence time $t+1$ variables, that is, they establish intra-slice dependencies at time $t+1$. An example of an Event-Driven POMDP model, with an associated virtual state factor "Cause of Events", is shown in (Figure 7).[5]

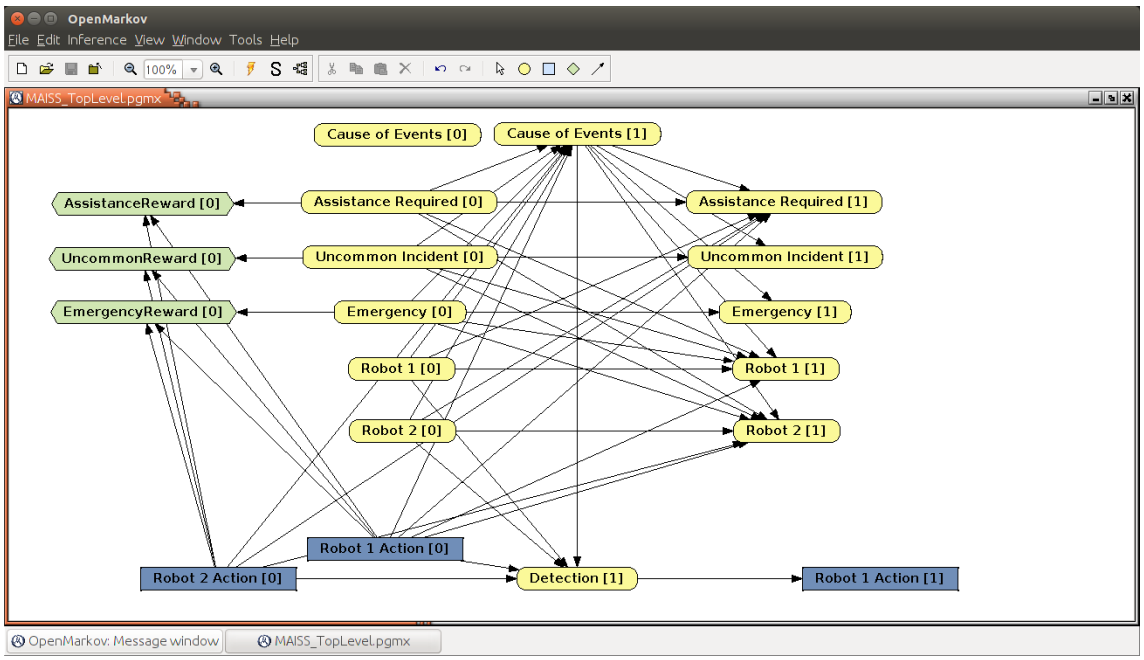


Figure 7: The "MAIS+S" problem, an Event-Driven MPOMDP, modeled in OpenMarkov.

Another characteristic of event-driven models is that observations depend on *transitions*, as opposed to *states*. This means that the (only) observation node, *Event[1]*, at time $t+1$ contains both time $t+1$ and time t parents.

Event-Driven POMDPs are useful to represent decision-making problems in which random amounts of "real-time" can elapse between decision steps. However, note that these models are typically associated with a stochastic "temporal" model for each transition, a probability distribution over event firing instants. In OpenMarkov (and at the time of writing), it is not possible to represent these temporal models, and so it is assumed that all events follow the same temporal distribution.

Event-Driven POMDPs are the only type of model that can accept a different number of action nodes and observation nodes (since there is always only one observation node). If you include more

[5] As is illustrated, the model also needs to specify an ***Cause of Events [0]*** variable, since it is not possible to specify an ***Cause of Events [1]*** without it.

than one decision node at time 0, you should warn the MADP parser by including the following element in your `.pmgx` file (under the `ProbNet` element):

```
<AdditionalProperties>
  <Property name="EventDriven" value="1" />
</AdditionalProperties>
```

Part II
Developer Guide

This part of the documentation is intended for people interested in using MADP in their own coding projects. It tries to give an overview of some typical functionality and how this is organized in classes. The documentation here is only intended as a starting point. For a reference style documentation, please refer to the documentation generated by doxygen (via "make htmldoc").

8 Overview of the MADP Toolbox Libraries

The MADP framework consists of several parts, grouped in different libraries. A brief overview of these 'MADP libraries' is given in Section 8.1. Also, there are a number of other libraries and software included to realize compilation with a minimum of effort. Therefore, Section 8.2 gives an overview of the entire directory structure.

8.1 MADP Libraries

The main part of MADP is the set of core libraries. These are briefly discussed here.

8.1.1 The Base Library (`libMADPBase`)

The base library is the core of the MADP toolbox. It contains:

- A representation of the basic elements in a decision process such as states, (joint) actions and observations.
- A representation of the transition, observation and reward models in a multiagent decision process. These models can also be stored in a sparse fashion.
- A uniform representation for MADP problems, which provides an interface to a problem's model parameters.
- Auxiliary functionality regarding manipulating indices, exception handling and printing: `E`, `IndexTools`, `PrintTools`, `StringTools`, `TimeTools`, `VectorTools`. Some project-wide definitions are stored in the `Globals` namespace.

8.1.2 The Parser Library (`libMADPParser`)

The parser library depends on the base library, and contains a parser for Tony's `.pomdp` file format as well as for `.dpomdp` files, which is a file format for problem specifications of discrete Dec-POMDPs, as well as a parser for models specified in the `ProbModelXML` format. A set of benchmark problem files in both formats can be found in the `problems/` directory.

The `.dpomdp` syntax is documented in `problems/example.dpomdp`. The format is based on Tony's `.pomdp` file format, and the formal specification is found in `src/parser/dpomdp.spirit`. The parser uses the Boost Spirit library. Also, parsers for several transition-observation independent models are provided, which are derived from the `.dpomdp` parser.

The `ProbModelXML` format is an XML format and parsed using `libXML2`. The format is covered in detail by Arias et al. [1], and a detailed introduction is given in Section 7.

8.1.3 The Support Library (`libMADPSupport`)

The support library contains basic data types and support useful for planning, such as:

- A representation for (joint) histories, for storing and manipulating observation, action and action-observation histories.
- A representation for (joint) beliefs, both stored as a full vector as well as a sparse one.
- Functionality for representing (joint) policies, as mappings from histories to actions.
- Shared functionality for discrete MADP planning algorithms, collected in `PlanningUnitMADPDiscrete` and `PlanningUnitDecPOMDPDiscrete`. These classes compute, e.g., (joint) history trees, joint beliefs, and value functions.
- Implementation for various problems:
  - An implementation of the DecTiger problem [18] which does not use `dectiger.dpomdp`, see `ProblemDecTiger`.
  - Also an implementation of the Fire Fighting problem (`ProblemFireFighting`), as well as a factored version (`ProblemFireFightingFactored`) [24].
  - Implementation of factored problem domains `ProblemFireFightingGraph` and `ProblemAloha` [25, 30].
  - Fully observable versions of the firefighting problem: `ProblemFOBSFireFightingFactored` and `ProblemFOBSFireFightingGraph`.
- Functionality for handling command-line arguments is provided by `ArgumentHandlers`.

8.1.4 The Planning Library (`libMADPPlanning`)

The planning library depends on the other libraries and contains functionality for planning algorithms, as well as some solution methods. In particular, it contains

- MDP solution techniques: value iteration [34].
- POMDP solution techniques: Monahan's algorithm [16] with incremental pruning [8], as well as Perseus [32].
- Dec-POMDP solution algorithms:
  - Brute Force Search.
  - JESP (exhaustive and dynamic programming variations) [18].
  - Direct Cross-Entropy (DICE) Policy Search [23].
  - GMAA* type algorithms, in particular:
    - MAA* [35],
    - k-GMAA* (as well as forward sweep policy computation) [24],
    - GMAA*-ELSI [25].
    - GMAA*-Cluster [26] (also called GMAA*-IC [33])
    - GMAA*-ICE [33].
    - DP-LPC [5]. (implementated mostly inside `solvers/DP-LPC.cpp`).
- Functionality for building and solving collaborative Bayesian Games:
  - Random, Brute force search, Alternating Maximization, Cross-entropy optimization, BaGaBaB [27], and Max-Plus for regular CBGs.
  - Random, Non-serial dynamic programming (a.k.a. variable elimination) [25] and Max-Plus [28] for collaborative *graphical* BGs (CGBGs).
- Heuristic Q-functions: Q_{MDP}, Q_{POMDP}, and Q_{BG} [24]. Including 'hybrid' representations [33] and tree-based pruning for Q_{BG} [21].
- A simulator class to empirically test the control quality of a solution, or perform evaluation of particular types of agents (e.g., reinforcement learning agents).

```
1  #include "ProblemDecTiger.h"
2  #include "JESPExhaustivePlanner.h"
3  int main()
4  {
5      ProblemDecTiger dectiger;
6      JESPExhaustivePlanner jesp(3,&dectiger);
7      jesp.Plan();
8      cout << jesp.GetExpectedReward() << endl;
9      cout << jesp.GetJointPolicy()->SoftPrint() << endl;
10     return(0);
11 }
```

Figure 8: A small example program that runs JESP on the DecTiger problem.

8.2 MADP Directory Structure

path	description
`/`	Package root.
`/config`	
`/doc`	Documentation.
`/doc/html`	The html documentation generated by doxygen. (perform 'make htmldoc' in root.)
`/m4`	M4 macros used by configure.
`/problems`	A number of problems in .dpomdp file format.
`/src`	C++ code
`/src/base`	The MADP base lib.
`/src/boost`	Included parts of the boost library
`/src/include`	This contains configuration .h files.
`/src/libpomdp-solve`	Parts of Tony Cassandra's 'pomdp-solve' library. Used for pruning POMDP value vectors.
`/src/libDAI`	Library for Discrete Approximate Inference by Joris Mooij [17]. Used for max-plus implementation for Collaborative (graphical) Bayesian Games.
`/src/parser`	The MADP parser lib.
`/src/planning`	The MADP planning lib.
`/src/solvers`	Code for a number of executables that use the MADP libs to implement (Dec-)POMDP solvers.
`/src/support`	The MADP support lib.
`/src/utils`	Code for a number of executables that perform auxiliary tasks (e.g., printing problem statistics or evaluating a joint policy through simulation).

9 Using the MADP Toolbox: An Example

Here we give an example of how to use the MADP toolbox. Figure 8 provides the full source code listing of a simple program. It uses exhaustive JESP to plan for 3 time steps for the 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.

This is a simple but complete program, and in the distribution (in `src/examples`) more elaborate examples are provided which, for instance, demonstrate the command-line parsing functionality and the use of the `.dpomdp` parser. Furthermore, for each of the solution methods provided there is a program to use it directly.

10 Typical Use Cases

In this section we elaborate on and provide pointers to examples of typical ways in which the MADP toolbox can be used.

10.1 One-Shot Decision Making

MADP implements one-shot team decision making via Bayesian games with identical payoffs, also referred to as *collaborative Bayesian games.* These are implemented as different sub-classes of `BayesianGameIdenticalPayoffInterface`. These Bayesian games can trivially also model strategic (i.e., 'normal-form') games by defining just one type per agent.

The current version of MADP includes a number of solvers for such collaborative BGs. For instance:

- `BGIP_SolverRandom` provides a random solution.
- `BGIP_SolverBruteForceSearch` a naive enumeration of all joint policies.
- `BGIP_SolverAlternatingMaximization` implements alternating maximization (a best-response hill-climbing).
- `BGIP_SolverCE` implements a cross-entropy optimization procedure [4] for optimizing the joint BG policy (see also [23]).
- `BGIP_SolverBranchAndBound`, the BaGaBaB method from [27] (note that it really performs A*).
- `BGIP_SolverMaxPlus` the max-plus solver from [28].

The usage of these solvers is illustrates in `examples/example_RandomBGs.cpp`. Additionally, there also are solvers for collaborative graphical Bayesian games:

- `BGCG_SolverRandom` provides a random solution.
- `BGCG_SolverNonserialDynamicProgramming` provides the exact solution via non-serial dynamic programming [3] (a.k.a. value iteration [11, 14] and bucket elimination [9]).
- `BGCG_SolverMaxPlus` provides a approximate solution via max-plus message passing [28]. This is based on (a older version of) the `LibDAI` library [17] which is included with MADP.

MADP also includes a (non-identical payoff) `BayesianGame` class. But so far, there is no solver for this class.

10.2 Sequential Planning Algorithms

Even though one-shot decision making is interesting on its own, the focus of the MADP toolbox lies on sequential decision making. Here we give a concise overview of the main components for planning for sequential decision settings.

10.2.1 MultiAgentDecisionProcessInterface and PlanningUnits

Two important sets of classes are those that represent actual multiagent decision process problems and those that represent planners.

The former classes inherit from `MultiAgentDecisionProcessInterface`, as illustrated in Figure 9. The figure indicates the relations between different models such as Dec-POMDPs, and POSGs, and shows that the toolbox separates interface classes from implementation. The figure also illustrates that the code offers opportunities to develop MADPs with continuous states, actions and observations, even though so far development has focused on problems with discrete sets (e.g., as represented by the `DecPOMDPDiscrete` class). Note that a number of included problems of type `FactoredDecPOMDPDiscrete` are shown in Figure 10.

The second important collection of classes pertain to planning. These classes all derive from the `PlanningUnit` base class. Part of this hierarchy is shown in Figure 11, centered around the

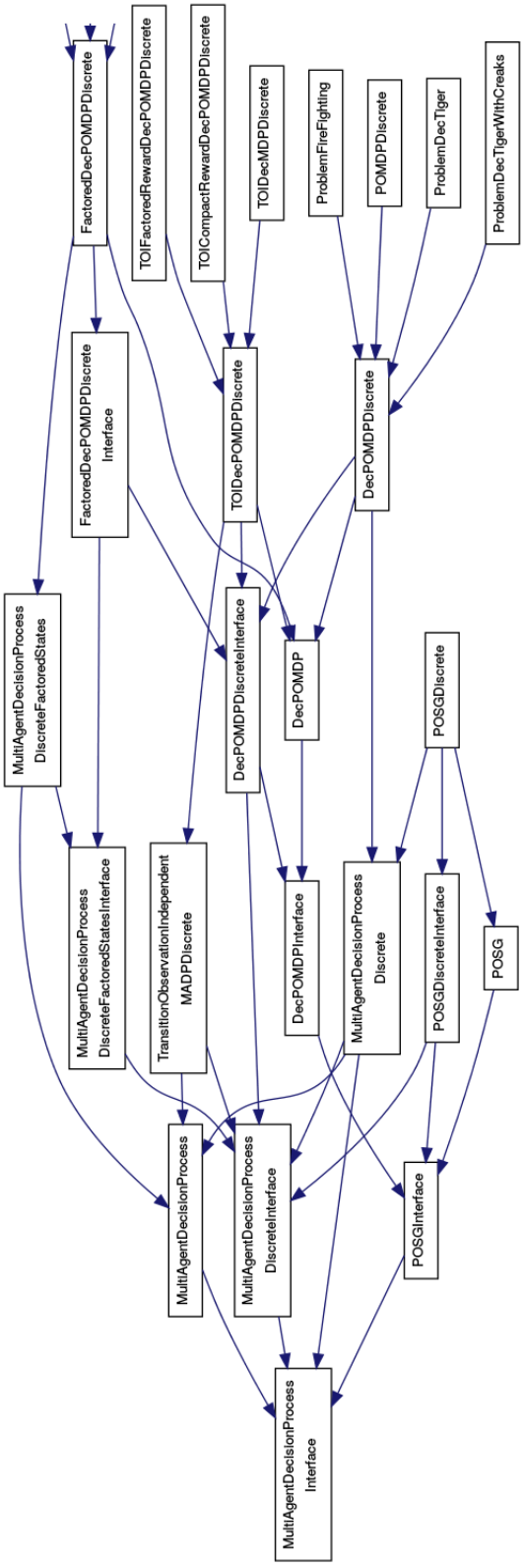


Figure 9: The `MultiAgentDecisionProcess` hierarchy.

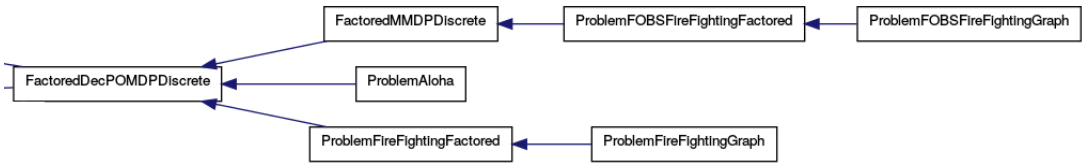


Figure 10: Included problems of type `FactoredDecPOMDPDiscrete`, the top branch are fully observable (a factored MMDP is the fully-observable special case of a factored Dec-POMDP), while the bottom two branches are actual Dec-POMDPs.

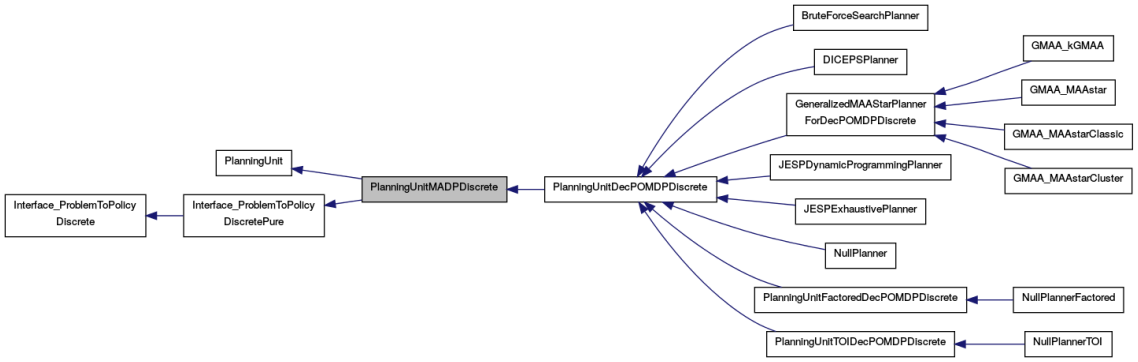


Figure 11: A part of the `PlanningUnit` class hierarchy.

`PlanningUnitMADPDiscrete` class. This class provides auxiliary functionality (e.g., generation of histories and conversion of history indices) for planners for discrete problems. The figure also shows that the class implements the so called '`Interface_ProblemToPolicyDiscretePure`'. This is the mechanism by which the planner gets to know certain basic information about the problem it will be planning for, for more details, see Section 14.

10.2.2 Multiagent Planning

A typical program that performs multiagent planning has three main components: 1) first, there is an 'experiment' (or 'solver') file which contains `main()`. This experiment instantiates both 2) the MADP (e.g., a Dec-POMDP), and the planner (e.g., GMAA*), and 3) subsequently performs the actual planning by calling `Plan()`.

An Example: example_decTigerJESP.cpp. For instance, let's look a the `example_decTigerJESP.cpp` file, of which the important content was already shown in Figure 8. The file it self is the 'experiment' file that contains main. It instantiates an MADP—a `ProblemDecTiger`, see also Figure 9—on line 5. Next, it instantiates a planning unit—a `JESPExhaustivePlanner`, see Figure 11— and calls the `Plan()` method.

Solving fully-observable MADPs. MADP currently implements value iteration in `MDPValueIteration` (for flat models of limited scope) and supports finite and infinite horizon problems. Usage is best demonstrated by an example, such as in the file `./src/examples/MMDP_SolveAndSimulate.cpp`. This file also shows how an offline policy can then be simulated on a specific problem.

For larger, generally factored settings, *every* existing problem, be it a `FactoredMMDPDiscrete` or the fully-observable subset of any class derived from `FactoredDecPOMDPDiscrete`, can be exported into SPUDD format with a call to `FactoredDecPOMDPDiscrete::ExportSpuddFile("filename")`.[6]

[6]There currently exists no functionality to load and simulate policies from SPUDD in MADP, however (see the SPUDD package for this functionality [13]).

10.2.3 Planning for a Single Agent

Of course, it is also possible to perform single-agent planning. From a modeling point of view, a single-agent model is just a multiagent one in which there happens to be just one agent. From a solution method perspective, however, this is different: most multiagent planning algorithms are not particularly suited for single agent planning. Methods suitable for single agents currently include `MDPValueIteration` (for MDPs) and `PerseusPOMDPPlanner, MonahanPOMDPPlanner` (for POMDPs).

10.3 Simulation and Reinforcement Learning

The toolbox also provides functionality to perform simulations for (teams of) agents interacting in a environment, as well as doing reinforcement learning. For instance, the following command performs 10000 simulations of the MMDP solution for the horizon 5 `GridSmall` problem:[7]

```
src/example$ ./example_MMDP_SolveAndSimulate ../../problems/GridSmall.dpomdp -h5 \
--runs=10000
Instantiating the problem...
...done.
Avg rewards: < 3.00999 >
```

10.3.1 Simulations

MADP provides functionality for doing simulations for a wide range of models. Simulations are performed using the following classes:

- `Simulation` — base class for all simulations.
- `SimulationDecPOMDPDiscrete` — actually implements simulations.
- `SimulationResult` — class that stores the results of simulations.
- `SimulationAgent` — base class for agents that can interact in a simulation.

Currently, there is just one class, `SimulationDecPOMDPDiscrete`, that actually implements simulations, but (contrary to what the name suggests) it can work for many type of models. There are two modes in which it works:

1. By giving it a joint policy. In this mode, a Dec-POMDP policy will be simulated (e.g., to empirically test its quality).
2. By giving it a vector of `SimulationAgent` objects. In this mode, the `SimulationDecPOMDPDiscrete` will give all relevant information to each agent, and the agents return back an action.

The second mode can be used to simulate also environments that are not Dec-POMDPs. The trick is that `SimulationDecPOMDPDiscrete` simply provides all relevant information (e.g., state, taken joint action, and/or received joint observation) to each `SimulationAgent`. For instance, it knows (via function overloading) that if it is dealing with agents of the type `AgentFullyObservable` it should provide them with the entire current state, while it will will only give the individual observations to agents of type `AgentLocalObservations`. (e.g., see the different `GetAction` functions in `SimulationDecPOMDPDiscrete.cpp`). Since the simulator does not dictate anything about the inner working of the agents, this framework directly supports (reinforcement) learning agents.

10.3.2 The Agents Hierarchy

As may be clear by now, MADP provides a hierarchy of some different types of `SimulationAgent`. The current hierarchy is shown in Figure 12. It shows the class `AgentDecPOMDPDiscrete`, which is

[7] Note that this demonstrates the flexibility of the toolbox. Even though `GridSmall` is a Dec-POMDP benchmark, it can be treated as a fully observable (MMDP) problem.

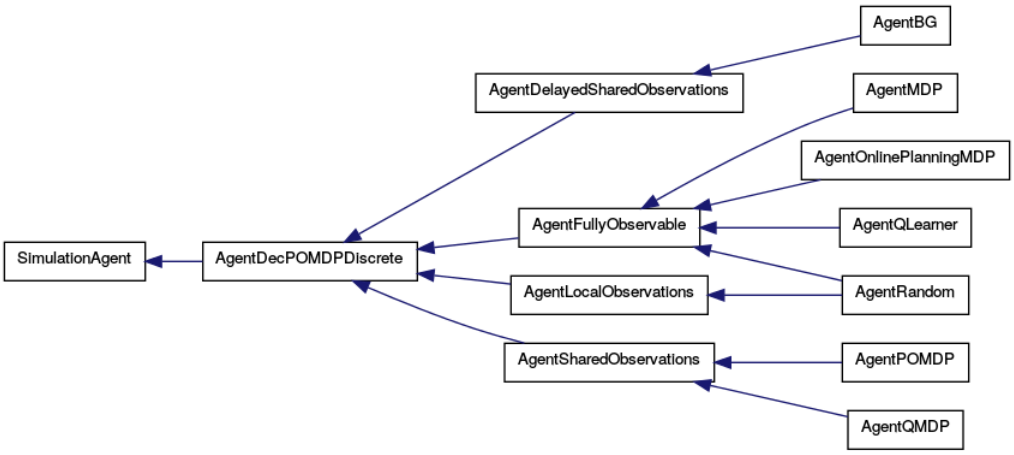


Figure 12: The `SimulationAgent` Hierarchy.

a superclass for all agents that make use of a (special case of a) `PlanningUnitDecPOMDPDiscrete`. It also shows that there are a number of different subclasses of agent: one for teams of agents with shared observations (i.e., for a POMDP or a 'multiagent POMDP'), one for agents with just local observations (i.e., the 'real Dec-POMDP setting'), one for agents with full observability, and one for teams of agents with delayed shared observations (i.e., the one-step delayed communication setting [22]).

10.3.3 Reinforcement Learning

For *fully-observable* problems, this release of MADP includes a Q-learning agent which learns a joint policy in the joint state & action space. Both ϵ-greedy and Boltzmann exploration methods are currently implemented.

If multiple agents are specified in a problem, team learning can be performed without having to replicate the learning in each individual agent: a single agent can be designated as the only learning agent with the `SetFirstAgent()` call. In this case, only the specified agent learns a (sparse) Q-table while the other agents look up their respective action in the joint table.

Note that we currently do not handle (reinforcement learning style) episode ends. MADP simulations are always run as many times as specified in the `SimulationDecPOMDPDiscrete` horizon parameter. However, episode ends can be modeled with special sink states in the problem formulation, i.e., absorbing states that generate no more rewards until simulation end.

The program `./src/examples/example_MMDP_OnlineSolve.cpp` illustrates the use of the simple (joint) Q-learning agent. Only agent 0 is designated as the learning agent with the `SetFirstAgent()` call so that all other agents refer to its learned Q-table for action selection. The program first computes the off-line policy using value iteration and then performs Q-learning for a number of iterations. Finally, a single row from both resulting Q-tables is compared and displayed. An example run (e.g. on the fully-observable Tiger problem for a discount factor of 0.99) is as follows: `./example_MMDP_OnlineSolve -g 0.99 DT`

11 Specifying Problems as a Sub-Class

While less portable, and arguably more complex, specifying your own problem as a sub-class is the most (run-time and space) efficient and gives you the most flexibility. In MADP, one typically implements a problem by deriving from the appropriate base class. We give a few examples here.

11.1 Dec-POMDPs.

For instance, to specify a Dec-POMDP with discrete states, actions and observations, one would inherit from `DecPOMDPDiscrete`. This class specifies the state, action and observation spaces, as

well as the transition, observation and reward model. All that the derived class needs to do is actually construct these. For an example on how this works, see the `ProblemDecTiger` class.

11.2 Factored Dec-POMDPs.

Similarly, factored Dec-POMDPs derive from `FactoredDecPOMDPDiscrete`, shown in Figure 10. The class `ProblemAloha` is a good example of a Factored Dec-POMDP model implemented as a class, and could serve as a template for new classes.

11.3 Fully-observable problems.

Factored fully-observable problems can directly derive from the `FactoredMMDPDiscrete` class (which in turn derives from the partially-observable `FactoredDecPOMDPDiscrete` class). The benefit is that `FactoredMMDPDiscrete` includes convenience functions that shield the user from having to define an observation model. That is, construction of a factored MMDP is then no different than constructing a factored Dec-POMDP in MADP, except that observations do not have to be considered. [8]

The file `./src/tests/test_mmdp.cpp` gives an example of how a fully-observable version of the `ProblemFireFightingFactored` would be modeled in MADP. The code is equivalent to the partially-observable version, except that `ComputeObservationProb` and `SetOScopes` are omitted from the class declaration and that no calls to either `ConstructObservations` or `ConstructJointObservations` are performed. See the classes `./src/support/ProblemFOBS*` for further examples of fully-observable, factored problem domains already implemented. For demonstration, `test_mmdp.cpp` prints the entire observation model and exports the problem specification to the SPUDD file format as well.

Currently, there is no convenience class provided to specify *flat, non-factored* (but fully-observable) problems in MADP. Note, however, that it is easy to use the fully-observable subsets of already existing flat Dec-POMDP problems by simply ignoring the observation model. Alternatively, a factored problem with one (joint) agent could be set up according to the description above.

12 IndexTools: Indices for Discrete Models

Although the design allows for extensions, the MADP toolbox currently only provides implementation for discrete models. I.e., models where the sets of states, actions and observations are discrete. For such discrete models, implementation typically manipulates indices, rather than the basic elements themselves. The MADP toolbox provides such index manipulation functions. In particular, here we describe how individual indices are converted to and from joint indices.

12.1 Enumeration of Joint Actions and Observations

As a convention, joint actions $\boldsymbol{a} = \langle a_1, ..., a_n \rangle$ are enumerated as follows

$$\begin{aligned}
\langle 0, \ldots, 0, 0 \rangle &\quad — \quad 0 \\
\langle 0, \ldots, 0, 1 \rangle &\quad — \quad 1 \\
\vdots \quad &\quad \vdots \quad \vdots \\
\langle 0, \ldots, 0, |\mathcal{A}_n| \rangle &\quad — \quad |\mathcal{A}_n| - 1 \\
\langle 0, \ldots, 1, 0 \rangle &\quad — \quad |\mathcal{A}_n| \\
\vdots \quad &\quad \vdots \quad \vdots \\
\langle |\mathcal{A}_1|, \ldots, |\mathcal{A}_n - 1|, |\mathcal{A}_n| \rangle &\quad — \quad |\mathcal{A}_1| \cdot \ldots \cdot |\mathcal{A}_n| - 1.
\end{aligned}$$

[8] Internally, an observation model with one certain observation per (joint) state is implicitly maintained: Recall that in the general case $\mathcal{O} = \times_i \mathcal{O}_i$ is the set of joint observations. In the fully-observable MMDP, $\mathcal{O}_i = \mathcal{S}$ and $P(\mathbf{o}|\mathbf{a}, s')$ maps o_i to s' deterministically: all agents know the true state of the world with certainty.

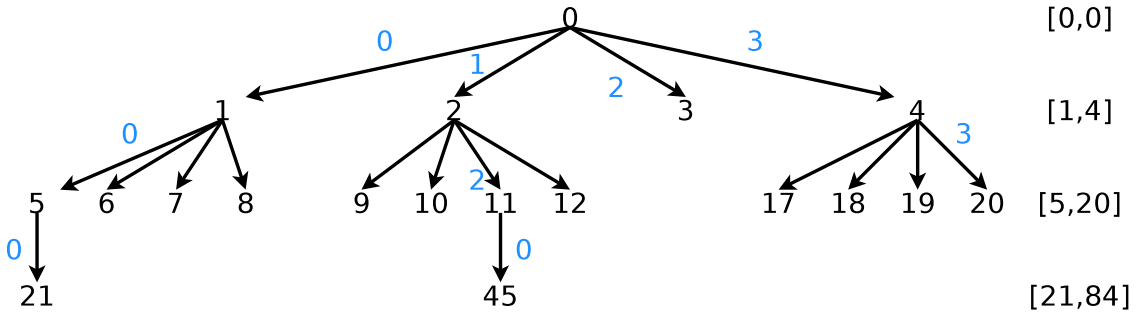


Figure 13: Illustration of the enumeration of (joint) observation histories. This illustration is based on a MADP with 4 (joint) observations.

This enumeration is enforced by `ConstructJointActions` in `MADPComponentDiscreteActions`. The joint action index can be determined using the `IndividualToJointIndices` functions from`IndexTools.h`. This file also lists functions for the reverse operation.

Joint observation enumeration is analogous to joint action enumeration (and therefore the same functions can be used).

12.2 Enumeration of (Joint) Histories

Most planning procedures work with indices of histories. For example, `PolicyPureVector` implements a mapping not from observation histories to actions, but from indices (of typically observation-) histories to indices (of actions).

It is important to be able convert between indices of joint/individual action/observation histories and therefore that the method by which the enumeration is performed is clear. This is what is described in this section.

The number of such histories is dependent on the number of observations for each agent, as well as the planning history h. As a result the auxiliary functions for histories have been included in `PlanningUnitMADPDiscrete`. This class also provides the option to generate and cache joint (action-) observation histories, so that the computations described here do not have to be performed every time.

12.2.1 Observation Histories

Figure 13 illustrates how observation histories are enumerated. This enumeration is

- based on the indices of the observations of which they consist.
- breadth-first, such that smaller histories have lower indices and histories for a particular time step t occupy a closed range of indices (also indicated in figure 13).

We will now describe the conversion between observation history indices and observation indices in more detail.

Observation indices to observation history index. Let I_{o_i} denote the index of observation o_i. In order to convert a sequence of observation indices up to time step k for agent i

$\left(I_{o_i^1}, I_{o_i^2}, ..., I_{o_i^k}\right)$ [9] to an observation history index, the following formula can be used:

$$I_{\bar{o}_i^k} = \text{offset}_k + \left(I_{o_i^1} \cdot |\mathcal{O}_i|^{k-1} + I_{o_i^2} \cdot |\mathcal{O}_i|^{k-2} + ... + I_{o_i^{k-1}} \cdot |\mathcal{O}_i|^1 + I_{o_i^k} \cdot |\mathcal{O}_i|^0\right),$$

I.e., $\left(I_{o_i^1}, I_{o_i^2}, ..., I_{o_i^k}\right)$ is interpreted as a base-$|\mathcal{O}_i|$ number and offset by

$$\text{offset}_k = \sum_{j=0}^{k-1} |\mathcal{O}_i|^j - 1 = \frac{|\mathcal{O}_i|^k - 1}{|\mathcal{O}_i| - 1} - 1.$$

(One is subtracted, because the indices start numbering from 0.)

As an example, the sequence leading to index 45 in figure 13 is $(1, 2, 0)$ with $k = 3$ and $|\mathcal{O}_i| = 4$. We therefore get:

$$\begin{aligned}
&\left[\frac{4^3 - 1}{4 - 1} - 1\right] + \left[1 \cdot 4^2 + 2 \cdot 4^1 + 0 \cdot 4^0\right] = \\
&\frac{64 - 1}{3} + 16 + 8 + 0 = \\
&21 + 24 = 45.
\end{aligned}$$

This conversion is performed by `GetObservationHistoryIndex`.

Observation history index to observation indices The inverse is given by a standard division procedure:

$$\begin{aligned}
45 - 21 = 24 \overset{\%4}{\rightarrow} 0 \\
\overset{/4}{\rightarrow} 6 \overset{\%4}{\rightarrow} 2 \\
\overset{/4}{\rightarrow} 1
\end{aligned}$$

(Here % denotes modulo.)

12.2.2 Action Histories

Individual action histories are enumerated exactly the same way as observation histories: a sequence of actions indices up to time step k $\left(I_{a_i^0}, I_{a_i^1}, ..., I_{a_i^{k-1}}\right)$ can be converted to an action history index by:

$$I_{\bar{a}_i^k} = \text{offset}_k + \left(I_{a_i^0} \cdot |\mathcal{A}_i|^{k-1} + ... + I_{a_i^{k-1}} \cdot |\mathcal{A}_i|^0\right).$$

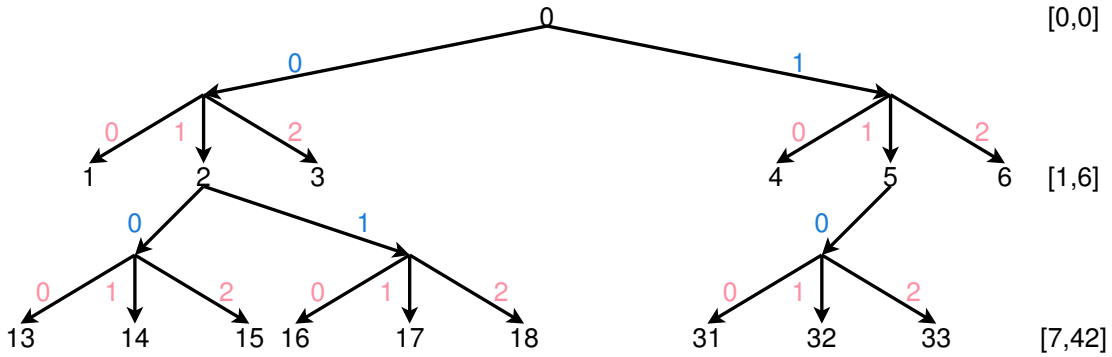
12.2.3 Action-Observation Histories

Enumeration of action-observation histories follows the same principle as for observation histories (and action histories), but have a complicating factor. 'Action-observations' are no data type and indices are not clearly defined.

Therefore, in order to implement action-observation histories an enumeration of action-observation is assumed. Let an action observation history $\bar{\theta}_i^k = \left(o_i^0, a_i^0, o_i^1, a_i^1, o_i^2, ..., a_i^{k-1}, o_i^k\right)$ be characterized by its indices $\left(I_{a_i^0}, I_{o_i^1}, I_{a_i^1}, I_{o_i^2}, ..., I_{a_i^{k-1}}, I_{o_i^k}\right)$ (again we assume no initial observation). We can group these indices as $\left(\left\langle I_{a_i^0}, I_{o_i^1}\right\rangle, \left\langle I_{a_i^1}, I_{o_i^2}\right\rangle, ..., \left\langle I_{a_i^{k-1}}, I_{o_i^k}\right\rangle\right)$, such that each $\left\langle I_{a_i^{t-1}}, I_{o_i^t}\right\rangle$ corresponds to an action-observation. Clearly, there are $|\mathcal{A}_i| \cdot |\mathcal{O}_i|$ action-observations. Let's denote an action-observation with θ_i and its index with I_{θ_i}, corresponding to action a_i and observation o_i, we then have that:

$$I_{\theta_i} = I_{a_i} \cdot |\mathcal{O}_i| + I_{o_i}.$$

[9] Note, we assume the initial observation o_i^0 to be empty. I.e. the sequence of indices $\left(I_{o_i^1}, I_{o_i^2}, ..., I_{o_i^k}\right)$ corresponds to the following sequence of observations: $\left(o_{i,\emptyset}, o_i^1, o_i^2, ..., o_i^k\right)$.



1,2,... (joint) action indices

1,2,... (joint) observation indices

1,2,... (joint) action-observation history indices

Figure 14: Illustration of the enumeration of (joint) action-observation histories. This illustration is based on a MADP with 2 (joint) actions and 3 (joint) observations.

`ActionAndObservation_to_ActionObservationIndex` from `IndexTools.h` performs this computation. The inverse operation is performed by `ActionObservation_to_ActionIndex` and `ActionObservation_to_ObservationIndex`,

Now these indices are defined, the same procedure for observation histories can be used as illustrated in fig. 14. I.e.,

$$\begin{aligned} I_{\bar{\theta}_i^k} =& \text{offset}_k + I_{\theta_i^1} \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^{k-1} + I_{\theta_i^2} \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^{k-2} + ... \\ & + I_{\theta_i^{k-1}} \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^1 + I_{\theta_i^k} \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^0 , \end{aligned}$$

Note that

$$\begin{aligned} I_{\theta_i^t} \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^{k-t} &= \left(I_{a_i^t} \cdot |\mathcal{O}_i| + I_{o_i^t}\right) \cdot (|\mathcal{A}_i| \cdot |\mathcal{O}_i|)^{k-t} \\ &= I_{a_i^t} \cdot |\mathcal{O}_i|^{k-t+1} \cdot |\mathcal{A}_i|^{k-t} + I_{o_i^t} |\mathcal{O}_i|^{k-t} \cdot |\mathcal{A}_i|^{k-t} \end{aligned}$$

As an example, index 32 is corresponds to index sequence $(1, 1, 0, 1)$ which, in action-observation indices, corresponds with $(4, 1)$ and thus:

$$\begin{aligned} \left(6^0 + 6^1\right) + 4 \cdot 6^{2-1} + 1 \cdot 6^{2-2} = \\ 7 + 24 + 1 = 32 \end{aligned}$$

Alternatively we can use the sequence $(1, 1, 0, 1)$ directly:

$$\begin{aligned} 7 + \left(1 \cdot 3^{2-1+1} \cdot 2^{2-1} + 1 \cdot 3^{2-1} \cdot 2^{2-1}\right) + \left(0 \cdot 3^{1-1+1} \cdot 2^{1-1} + 1 \cdot 3^{1-1} \cdot 2^{1-1}\right) = \\ 7 + \left(1 \cdot 3^2 \cdot 2^1 + 1 \cdot 3^1 \cdot 2^1\right) + \left(0 \cdot 3^1 \cdot 2^0 + 1 \cdot 3^0 \cdot 2^0\right) = \\ 7 + (1 \cdot 9 \cdot 2 + 1 \cdot 3 \cdot 2) + (0 \cdot 3 \cdot 1 + 1 \cdot 1 \cdot 1) = \\ 7 + 18 + 6 + 1 = \\ 7 + 25 = 32 \end{aligned}$$

12.2.4 Joint Histories

Joint observations are enumerated in the same way as individual observation histories, only now using the indices of joint observations rather than individual observations.

I.e., figure 13 also illustrates how joint observation histories are enumerated. And in order to convert a sequence of joint observations indices up to time step k $(I_{\boldsymbol{o}^1}, ..., I_{\boldsymbol{o}^k})$ to an observation history index, the following formula can be used:

$$I_{\bar{\boldsymbol{o}}} = \text{offset}_k + \left(I_{\boldsymbol{o}^0} \cdot |\mathcal{O}|^{k-1} + I_{\boldsymbol{o}^1} \cdot |\mathcal{O}|^{k-1} + ... + I_{\boldsymbol{o}^{k-1}} \cdot |\mathcal{O}|^1 + I_{\boldsymbol{o}^k} \cdot |\mathcal{O}|^0 \right),$$

I.e., $(I_{\boldsymbol{o}^1}, ..., I_{\boldsymbol{o}^k})$ is interpreted as a base-$|\mathcal{O}|$ number and offset by

$$\text{offset}_{k+1} = \sum_{j=0}^{k-1} |\mathcal{O}|^j - 1 = \frac{|\mathcal{O}|^k - 1}{|\mathcal{O}| - 1} - 1.$$

Indices for joint action histories and joint action-observation histories are computed in the same way. The action-observation functions (`ActionObservation_to_ActionIndex`, etc.) can also be used for joint action-observations.

`PlanningUnitMADPDiscrete` also provides functions to convert joint to individual history indices `JointToIndividualObservationHistoryIndices`, etc.

13 Joint Beliefs and History Probabilities

Planning algorithms for MADPs will typically need the probabilities of particular joint action observation histories, and the probability over states they induce (called joint beliefs). `PlanningUnitMADPDiscrete` provides some functionality for performing such inference, which we discuss here.

13.1 Theory

Let $\Pr_{\boldsymbol{\pi}}(\boldsymbol{a}^t|\bar{\boldsymbol{\theta}}^t)$ denote the probability of $\boldsymbol{a}$ as specified by $\boldsymbol{\pi}$, then $\Pr(s^t, \bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0)$ is recursively defined as

$$\Pr(s^t, \bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0) = \sum_{s^{t-1} \in \mathcal{S}} \Pr(s^t, \bar{\boldsymbol{\theta}}^t|s^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}) \Pr(s^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}|\boldsymbol{\pi}, b^0). \tag{1}$$

with

$$\Pr(s^t, \bar{\boldsymbol{\theta}}^t|s^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}) = \Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, s^t) \Pr(s^t|s^{t-1}, \boldsymbol{a}^{t-1}) \Pr_{\boldsymbol{\pi}}(\boldsymbol{a}^{t-1}|\bar{\boldsymbol{\theta}}^{t-1}).$$

For stage 0 we have that $\forall_{s^0} \ \Pr(s^0, \bar{\boldsymbol{\theta}}_\emptyset|\boldsymbol{\pi}, b^0) = b^0(s^0)$.

Since we tend to think in joint beliefs $b^{\bar{\boldsymbol{\theta}}^t}(s^t) \triangleq \Pr(s^t|\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0)$, we can also represent the distribution (1) as:

$$\Pr(s^t, \bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0) = \Pr(s^t|\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0) \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0) \tag{2}$$

The joint belief $\Pr(s^t|\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0)$ The joint belief $\Pr(s^t|\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0)$ is given by:

$$\begin{aligned} \Pr(s^t|\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0) &= \frac{\Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, s^t) \sum_{s^{t-1}} \Pr(s^t|s^{t-1}, \boldsymbol{a}^{t-1}) \Pr(s^{t-1}|\bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)}{\sum_{s^t} \Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, s^t) \sum_{s^{t-1}} \Pr(s^t|s^{t-1}, \boldsymbol{a}^{t-1}) \Pr(s^{t-1}|\bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)} \\ &= \frac{\Pr(s^t, \boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)}{\Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)} \end{aligned} \tag{3}$$

where $\Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) = \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)$.

The probability of an history $\Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0)$ The second part of (2) is given by

$$\begin{aligned} \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0) &= \Pr(\bar{\boldsymbol{\theta}}^t|\bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) \Pr(\bar{\boldsymbol{\theta}}^{t-1}|\boldsymbol{\pi}, b^0) \\ &= \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) \Pr_{\boldsymbol{\pi}}(\boldsymbol{a}^{t-1}|\bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) \Pr(\bar{\boldsymbol{\theta}}^{t-1}|\boldsymbol{\pi}, b^0) \\ &= \Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) \Pr_{\boldsymbol{\pi}}(\boldsymbol{a}^{t-1}|\bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0) \Pr(\bar{\boldsymbol{\theta}}^{t-1}|\boldsymbol{\pi}, b^0) \end{aligned} \tag{4}$$

where $\Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)$ is the denominator of (3).

Algorithm 1 $[b^{\bar{\boldsymbol{\theta}}^t}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}})] = \texttt{GetJAOHProbs}(\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^{\bar{\boldsymbol{\theta}}^{t'}}, \bar{\boldsymbol{\theta}}^{t'})$

if $\bar{\boldsymbol{\theta}}^t = \bar{\boldsymbol{\theta}}^{t'}$ **then**
  return $[b^{\bar{\boldsymbol{\theta}}^t} = b^{\bar{\boldsymbol{\theta}}^{t'}}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}}) = 1]$
end if
if $\bar{\boldsymbol{\theta}}^t$ not an extension of $\bar{\boldsymbol{\theta}}^{t'}$ **then**
  return $[b^{\bar{\boldsymbol{\theta}}^t} = \vec{0}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}}) = 0]$
end if
$\bar{\boldsymbol{\theta}}^{t''} = (\bar{\boldsymbol{\theta}}^{t'}, \boldsymbol{a}^{t'}, \boldsymbol{o}^{t'+1})$ {consist. with $\bar{\boldsymbol{\theta}}^t$}
$[b^{\bar{\boldsymbol{\theta}}^{t''}}, \Pr(\bar{\boldsymbol{\theta}}^{t''}|\boldsymbol{a}^{t'}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}})] = b^{\bar{\boldsymbol{\theta}}^{t'}}.\texttt{Update}(\boldsymbol{a}^{t'}, \boldsymbol{o}^{t'+1})$ {belief update, see (3)}
$[b^{\bar{\boldsymbol{\theta}}^t}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t''}, b^{\bar{\boldsymbol{\theta}}^{t''}})] = \texttt{GetJAOHProbs}(\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^{\bar{\boldsymbol{\theta}}^{t''}}, \bar{\boldsymbol{\theta}}^{t''})$
$\Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}}) = \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t''}, b^{\bar{\boldsymbol{\theta}}^{t''}}) \Pr(\bar{\boldsymbol{\theta}}^{t''}|\boldsymbol{a}^{t'}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}}) \Pr_{\boldsymbol{\pi}}(\boldsymbol{a}^{t'}|\bar{\boldsymbol{\theta}}^{t'}, \boldsymbol{\pi})$
return $[b^{\bar{\boldsymbol{\theta}}^t}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}})]$

13.2 Implementation

Since computation of $\Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0)$ is interwoven with the computation of the joint belief through $\Pr(\boldsymbol{o}^t|\boldsymbol{a}^{t-1}, \bar{\boldsymbol{\theta}}^{t-1}, \boldsymbol{\pi}, b^0)$, it is impractical to separately evaluate (3) and (4).

Rather we define a function

$$[b^{\bar{\boldsymbol{\theta}}^t}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, b^0)] = \texttt{GetJAOHProbs}(\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^0)$$

Because in many situations an application might evaluate similar $\bar{\boldsymbol{\theta}}^t$ (i.e., ones with an identical prefix), a lot of computation will be redundant. To give the user the possibility to avoid this, we also define

$$[b^{\bar{\boldsymbol{\theta}}^t}, \Pr(\bar{\boldsymbol{\theta}}^t|\boldsymbol{\pi}, \bar{\boldsymbol{\theta}}^{t'}, b^{\bar{\boldsymbol{\theta}}^{t'}})] = \texttt{GetJAOHProbs}(\bar{\boldsymbol{\theta}}^t, \boldsymbol{\pi}, b^{\bar{\boldsymbol{\theta}}^{t'}}, \bar{\boldsymbol{\theta}}^{t'})$$

which returns the probability and associated joint belief of $\bar{\boldsymbol{\theta}}^t$, *given* that $\bar{\boldsymbol{\theta}}^{t'}$ (and associated joint belief $b^{\bar{\boldsymbol{\theta}}^{t'}}$) are realized (i.e., given that $\Pr(\bar{\boldsymbol{\theta}}^{t'}) = 1$).

14 Policies

Here we discuss some properties and the implementation of policies. Policies are plans for agents that specify how they should act in each possible situation. As a result a policy is a mapping from these 'situations' to actions. Depending on the assumption on the observability in an MADP, however, these 'situations' might be different. Also we would like to be able to reuse the implementations of policies for problems with a slightly different nature, for instance (Bayesian) games.

In the MADP toolbox, the most general form of a policy is a mapping from a domain (`PolicyDomain`) to (probability distributions over) actions. Currently we have only considered policies for discrete domains, which are mappings from indices (of these histories) to indices (of actions). It is typically still necessary to know what type of indices a policy maps from in order to be able to reuse our implementation of policies. To this end a discrete policy maintains its `IndexDomainCategory`. So far there are four types of index-domain categories: `TYPE_INDEX`, `OHIST_INDEX`, `OAHIST_INDEX` and `STATE_INDEX`.

As said `PolicyDiscrete` class represents the interface policies for discrete domains. `PolicyDiscretePure` is the interface for a pure (deterministic) policy. A class that actually implements a policy is `PolicyPureVector`. This class also implements a function to get and set the index of the policy (pure policies over a finite domain are enumerable). Joint policies are represented by similarly named classes `JointPolicyDiscrete`, `JointPolicyPureVector`, etc.

In order to instantiate a (joint)policy, it needs to know several things about the problem it is defined over. We already mentioned the index domain category, but there is other information needed as well (the number of agents, the sizes of their domains, etc.). To provide this

information, each problem for which we want to construct a (joint) policy has to implement the `Interface_ProblemToPolicyDiscretePure`.

A Installation Guide

This section elaborates on the quick start instructions presented in Section 2.

A.1 System requirements

The MADP Toolbox has mainly been developed on Debian GNU/Linux (64 bit, amd64 architecture), but should work on any recent Linux distribution. It has been known to compile on Debian 5, 6, 7 & 8, Ubuntu 12.04lts, 13.10 & 14.04lts, Linux Mint 17, Fedora 17 & 20 and OpenSuSe 13.1. For details on experimental Mac OSX support see Section A.5.

MADP requires the following software to compile (as Debian package names):

- libtool (libtool)
- GCC, version >=4.2 (g++)

Optional software:

- Doxygen (doxygen) [for generating documentation]
- Graphviz (graphviz) [for dependency graphs in the generated documentation]
- libxml2 (libxml2-dev) [for using the XML-based factored model parser]

The software also uses part of the Boost C++ libraries, but due to potential compatibility issues we ship the relevant parts in `src/boost`.

A.2 Compiling, installing and linking

Execute the following:

```
tar xfz madp-0.3.1.tar.gz
cd madp-0.3.1
./configure
make
make install [optional]
```

If you need to install MADP locally in your home directory (e.g., because you are no root on your machine), you can use the following configure line:

`./configure --prefix=<PATHTOMADP>`

(E.g., "`./configure --prefix=$HOME/madp-local`" will install into a directory called '`madp-local`' in your home dir.)

The installation will create the following directories:

`<PATHTOMADP>/bin`
`<PATHTOMADP>/include`
`<PATHTOMADP>/lib`

In order to be able to execute the binaries, you may need to add the lib dir to the `LD_LIBRARY_PATH` (E.g.: `export LD_LIBRARY_PATH=<PATHTOMADP>/lib:$LD_LIBRARY_PATH)`. Also, for conveniently running the binaries, you may want to add the bin dir to your PATH variable (E.g.: `export PATH=<PATHTOMADP>/bin:$PATH)`.

If you intend to use MADP as a library and link your own code against it, you only have to add `<PATHTOMADP>/include` to your compiler flags (e.g., `-I <PATHTOMADP>/include`) and link to `libMADP.so` (e.g., `-L <PATHTOMADP>/lib -lMADP -lxml2 -lm`).

A.3 Using CPLEX

For some functionality, such as the DP-LPC algorithm, CPLEX is required. In order to compile against CPLEX, the `configure` script should be called with options to indicate the location of those (see also `configure --help`). The easiest approach is to create a script that specified these options and calls configure. For instance, one of the authors has been using a bash script with the following contents:

```
CPLEX_LOCATION=/opt/ibm/ILOG/CPLEX_Studio125
CPLEX_STRING="--with-cplex-includes=\"$INCLUDE -I$CPLEX_LOCATION/cplex/include/ \
   -I$CPLEX_LOCATION/concert/include/\" "
CPLEX_STRING="$CPLEX_STRING \
   --with-cplex-ldflags=\"-L$CPLEX_LOCATION/cplex/lib/x86-64_sles10_4.1/static_pic \
   -L$CPLEX_LOCATION/concert/lib/x86-64_sles10_4.1/static_pic\" "
CPLEX_STRING="$CPLEX_STRING --with-cplex-libs=\"-lilocplex -lconcert -lcplex \" "
configure $CPLEX_STRING <OTHER_OPTIONS>
```

A.4 Specifying problem and result directories

By default, MADP assumes problems to be located in `$HOME/.madp/problems`, as discussed in Section 4. That is, problem descriptions can be loaded without specifying a path if `~/.madp/problems` is a symlink to the `problems` subdir in the MADP tree (or if you simply copy the problem descriptions to `~/.madp/problems`). Similarly, results are saved in (subdirs of) `~/.madp/results`, so will be convenient to make a symlink to the desired results locations.

The following commands should get most people started:

```
mkdir ~/.madp
mkdir ~/.madp/results
cd ~/.madp
ln -s <PATH-TO-EXTRACTED-MADP-TAR>/problems
```

Alternatively, one can adapt these locations in the code (`src/planning/directories.cpp`) and recompile. If you get an error like: "ERROR: mkdir error for `/home/*/.madp/results/<METHODNAME>/...`" or "failed to open file `/home/*/.madp/results/<METHODNAME>/...`" you should also create a sub-directory for `<METHODNAME>`. E.g., in case of GMAA: `mkdir ~/.madp/results/GMAA` .

A.5 Mac OSX support (experimental)

Support for Mac OSX is experimental, but the following steps have been reported to work:

1. Install XCode 6
2. Compile / install boost 1.56 from source
3. Install argp-standalone from MacPorts or Homebrew
4. Create symbolic links from (or move the files) `/opt/local/include/argp.h` to `/usr/local/include/argp.h` and `/opt/local/lib/libargp.a` to `/usr/local/lib/libargp.a`.

Acknowledgments

We like to thank all contributors: Abdeslam Boularias, Julian Kooij, Tiago Veiga, Francisco Melo, Timon Kanters, Philipp Beau. Also, we are grateful for numerous suggestions and bug-reports we have received to improve the MADP Toolbox.

F.O. is funded by NWO Innovational Research Incentives Scheme Veni #639.021.336.

References

[1] M. Arias, F. J. Díez, and M. P. Palacios. ProbModelXML. A format for encoding probabilistic graphical models. Technical report cisiad-11-02, UNED, Madrid, Spain, 2011.

[2] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[3] Umberto Bertele and Francesco Brioschi. *Nonserial Dynamic Programming*. Academic Press, Inc., 1972.

[4] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

[5] Abdeslam Boularias and Brahim Chaib-draa. Exact dynamic programming for decentralized POMDPs with lossless policy compression. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2008.

[6] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proc. of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 195–210, 1996.

[7] A. R. Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Brown University, 1998.

[8] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 54–61. Morgan Kaufmann, 1997.

[9] Rina Dechter. Bucket elimination: a unifying framework for processing hard and soft constraints. *Constraints*, 2(1):51–55, 1997.

[10] Rosemary Emery-Montemerlo, Geoff Gordon, Jeff Schneider, and Sebastian Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *Proceedings of the International Conference on Autonomous Agents and Multi Agent Systems*, pages 136–143, 2004.

[11] Carlos Guestrin, Daphne Koller, and Ronald Parr. Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems 14*, pages 1523–1530, 2002.

[12] Eric A. Hansen, Daniel S. Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the National Conference on Artificial Intelligence*, pages 709–715, 2004.

[13] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of Uncertainty in Artificial Intelligence*, 1999.

[14] Jelle R. Kok and Nikos Vlassis. Collaborative multiagent reinforcement learning by payoff propagation. *Journal of Machine Learning Research*, 7:1789–1828, 2006.

[15] Joao V. Messias, Matthijs T. J. Spaan, and Pedro U. Lima. Asynchronous execution in multiagent POMDPs: Reasoning over partially-observable events. In *AAMAS'13 Workshop on Multi-agent Sequential Decision Making under Uncertainty (MSDM)*, pages 9–14, May 2013.

[16] George E. Monahan. A survey of partially observable Markov decision processes: theory, models and algorithms. *Management Science*, 28(1), January 1982.

[17] Joris M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, August 2010.

[18] Ranjit Nair, Milind Tambe, Makoto Yokoo, David V. Pynadath, and Stacy Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 705–711, 2003.

[19] Frans A. Oliehoek. *Value-Based Planning for Teams of Agents in Stochastic Partially Observable Environments*. PhD thesis, Informatics Institute, University of Amsterdam, February 2010.

[20] Frans A. Oliehoek. Decentralized POMDPs. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State of the Art*, volume 12 of *Adaptation, Learning, and Optimization*, pages 471–503. Springer Berlin Heidelberg, Berlin, Germany, 2012.

[21] Frans A. Oliehoek and Matthijs T. J. Spaan. Tree-based solution methods for multiagent POMDPs with delayed communication. In *Proc. of the AAAI Conference on Artificial Intelligence*, pages 1415–1421, 2012.

[22] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. Dec-POMDPs with delayed communication. In *Proceedings of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM)*, May 2007.

[23] Frans A. Oliehoek, Julian F.P. Kooi, and Nikos Vlassis. The cross-entropy method for policy search in decentralized POMDPs. *Informatica*, 32:341–357, 2008.

[24] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32: 289–353, 2008.

[25] Frans A. Oliehoek, Matthijs T. J. Spaan, Shimon Whiteson, and Nikos Vlassis. Exploiting locality of interaction in factored Dec-POMDPs. In *Proceedings of the International Conference on Autonomous Agents and Multi Agent Systems*, pages 517–524, May 2008.

[26] Frans A. Oliehoek, Shimon Whiteson, and Matthijs T. J. Spaan. Lossless clustering of histories in decentralized POMDPs. In *Proceedings of the International Conference on Autonomous Agents and Multi Agent Systems*, pages 577–584, May 2009.

[27] Frans A. Oliehoek, Matthijs T. J. Spaan, Jilles Dibangoye, and Christopher Amato. Heuristic search for identical payoff Bayesian games. In *Proceedings of the International Conference on Autonomous Agents and Multi Agent Systems*, pages 1115–1122, May 2010.

[28] Frans A. Oliehoek, Shimon Whiteson, and Matthijs T. J. Spaan. Exploiting structure in cooperative Bayesian games. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pages 654–664, August 2012.

[29] Frans A. Oliehoek, Matthijs T. J. Spaan, Christopher Amato, and Shimon Whiteson. Incremental clustering and expansion for faster optimal planning in decentralized POMDPs. *Journal of Artificial Intelligence Research*, 46:449–509, 2013.

[30] Frans A. Oliehoek, Shimon Whiteson, and Matthijs T. J. Spaan. Approximate solutions for factored Dec-POMDPs with many agents. In *Proceedings of the Twelfth International Conference on Autonomous Agents and Multiagent Systems*, pages 563–570, 2013.

[31] Sven Seuken and Shlomo Zilberstein. Formal models and algorithms for decentralized decision making under uncertainty. *Autonomous Agents and Multi-Agent Systems*, 17(2):190–250, 2008.

[32] Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[33] Matthijs T. J. Spaan, Frans A. Oliehoek, and Chris Amato. Scaling up optimal heuristic search in Dec-POMDPs via incremental expansion. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2027–2032, 2011.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998.

[35] Daniel Szer, François Charpillet, and Shlomo Zilberstein. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In *Proceedings of Uncertainty in Artificial Intelligence*, pages 576–583, 2005.