Decision Making under Uncertainty

Lab course for Module 2 of the MAS Course in Leuven 2012-2013

Matthijs Spaan
m.t.j.spaan@tudelft.nl
Delft University of Technology

1 Introduction

This manual describes the lab course of Module 2 of the 2012-2013 MAS course in Leuven. The goal of the lab course is to become acquainted with computing decision making strategies for agents inhabiting stochastic environments. You will be asked to experiment with several dynamic programming and reinforcement learning techniques introduced in the lectures.

This assignment for the MAS course consists of 2 parts: one about single-agent planning under uncertainty, and one about multiagent planning. In order to pass this module, you need to submit a single report for both parts combined. The assignment should be completed in pairs, i.e., two students work together on a solution and submit a single report. Deliverables:

- Your code produced during the assignment (Matlab source files).
- Your report of 6 to 10 pages describing your findings, incorporating the questions mentioned in the assignments (pdf file).

Please send your submission to `m.t.j.spaan@tudelft.nl` by **March 29, 23:59**.

2 Getting started

We will use Matlab and a set of functions which implement several (PO)MDP domains. The software and additional material (such as the lecture slides) are available at Toledo.

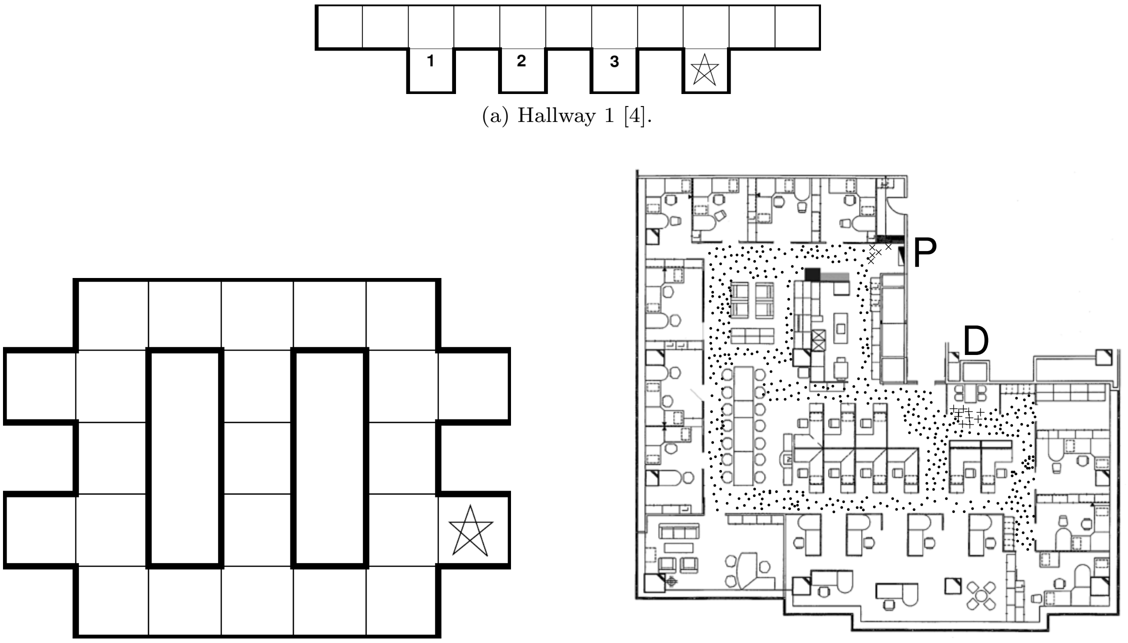
To get started:

1. Download `dmuu.zip` and unzip it.
2. Start Matlab.
3. Add the `dmuu/generic` and `dmuu/yourcode` subdirectories to Matlab's search path (File, Set Path, Add folder), or try running `setpaths`.
4. Matlab: `cd dmuu/problems/hallway2`
5. Matlab: `initProblem; global problem; problem`

Now you should see a struct defining the Hallway2 problem printed on your screen, which will be the input for your implementation, as we will discuss next.

3 Assignments Part 1: single-agent planning under uncertainty

The single-agent part consists of three assignments. In the first two we will assume full knowledge of the environment (i.e., all the information in the `problem` struct is available), while in the last



(a) Hallway 1 [4].

(b) Hallway 2 [4].

(c) TRC [5].

Figure 1: Problem domain maps for the single-agent problems.

we will treat the transition and reward model as unknown. For background information consult the slides of the lecture and [6][1].

3.1 Problem domains

We will be using three test domains: Hallway, Hallway2 and TRC. The TRC problem is the robotic delivery task discussed in the lecture (more details in [5]). Hallway and Hallway2 are POMDP navigation problems, in which the objective is to reach a designated goal state as quickly as possible. The robot's state is defined as its location in the maze combined with its orientation (the four cardinal directions). At each step the agent can take one out of five actions: {*stay in place, move forward, turn right, turn left, turn around*}. The robot observes each possible combination of the presence of a wall in four directions plus a unique observation indicating the goal state; in the Hallway problem three other landmarks are also available. Both the transition and the observation model are noisy. The map of each maze can be found in Figure 1.

The software can load many more problem descriptions, downloadable from `http://www.pomdp.org/pomdp/examples/`. To run them, put each `.POMDP` in a directory whose name corresponds to the base part of POMDP file (e.g., `cit.POMDP` goes in `cit/`) and run `initProblem` in that directory.

3.2 Software

The algorithms you will implement in the assignments in Section 3 need a model of the (PO)MDP, which are stored in a global struct called `problem`. Table 1 gives a description of the parameters relevant for your implementation. The software is designed to be run from a problem directory, i.e., `dmuu/problems/hallway`, `dmuu/problems/hallway2`, and `dmuu/problems/trc`. Problem-independent code which should not have to touch is in `dmuu/generic`, and your own code should go in `dmuu/yourcode`, where stubs have been provided.

[1]HTML version at `http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html`

Struct member	Description
`unixName`	String identifying the problem.
`nrStates`	Number of states.
`nrActions`	Number of actions.
`gamma`	Discount factor γ.
`start`	Starting belief (typically the initial state is drawn from this distribution).
`reward`	Immediate reward function: `reward`(s, a) gives the reward for taking action a in state s.
`transition`	Transition function: `transition`(s', s, a) gives the probability $p(s'\|s, a)$.
`state`	Current state of the system.
The following parameters are only relevant in a POMDP setting:	
`nrObservations`	Number of observations.
`observation`	Observation function: `observation`(s', a, o) gives the probability $p(o\|s', a)$, i.e., the probability of receiving observation o in state s' after taking action a (in the previous time step).
`belief`	Current belief of the system.

Table 1: Relevant (PO)MDP model parameters stored in a `problem` struct.

3.3 Value iteration in MDPs

We will start by considering the fully observable case, i.e., at each time step the robot knows its state, which means we can ignore the observation model for the moment.

1. Implement value iteration starting from the skeleton in `vi.m`. The function should return a matrix Q, where $Q(s, a)$ is the optimal values for each state and action. You can test your implementation by comparing the resulting Q table on the `loadunload` problem with the values on the lecture slides.

2. To test your solution, write a function that takes as input Q and simulates trajectories through the MDP, starting from `sampleTrajectories.m`. The initial state should be drawn from the `start` distribution. To know what action a Q-table prescribes for a particular state use `getActionForState`. To sample a successor state s' given a current state s and action a you can use `sampleSuccessorState`. This function should also calculate the discounted reward that the agent receives, in order to be able to verify the results.

3. Run value iteration on the three test problems, and see how the robots perform their task. Note that for large problems such as TRC, it might be more efficient to use a sparse matrix representation (see `transitionS` and `observationS` in the `problem` struct). You can use the `plotState` function to visualize their trajectories.

4. (Optional) For a simple domain such as Hallway, implement a hand-coded policy and compare its performance to the solution computed by value iteration. Do you expect your hand-coded solution to perform better?

5. (Optional) The three domains included are essentially all path-planning problems (although TRC contains the twist of carrying mail or not). Go to `http://www.pomdp.org/pomdp/examples/` and select some other types of (PO)MDP problems to test your implementation on.

In your report, you should at least provide performance results for your value-iteration implementation for all three problems in Figure 1, detailing both the value (assuming the initial state is drawn according to the `start` distribution) and the computation time. Average the sampled value (i.e., $\sum_{t=0}^{h} \gamma^t R_t$) over a sufficient number of runs and provide the standard deviation over the value. Compare with the value of the initial starting distribution using your Q table ($\sum_s b(s) \max_a Q(s,a)$). Also include a number of qualitative results, graphically illustrating the trajectories chosen by the agents.

3.4 Partially observable environments: heuristic methods

Next, we will assume that the environment is partially observable to the robots. As discussed in the lecture, we can tackle the resulting POMDP by planning over a belief state instead of the state of the system. For more background on the POMDP model see [3].

1. Implement the belief update (starting from `beliefUpdate.m`), and experiment with heuristic control strategies, such as Q_{MDP} and MLS, that use the MDP Q-values you computed in the previous assignment. The belief should be initialized using the `start` distribution. Sample observations using a similar function (that you need to create) to how successor states are sampled.

2. Check the performance on the different test domains. Use `plotSingleBelief` to plot a belief.

In your report, compare the performance results of your Q_{MDP} and MLS implementations with the results in the fully observable case in the previous section. Explain the differences. Also include a belief evolution of some runs of the system.

3.5 Partially observable environments: point-based methods

Finally, we will compare the performance of MDP-based heuristics vs. true POMDP methods.

1. Download and install the Perseus algorithm from
   `http://staff.science.uva.nl/~mtjspaan/pub/pomdpSoftware_0.1.tar.gz`

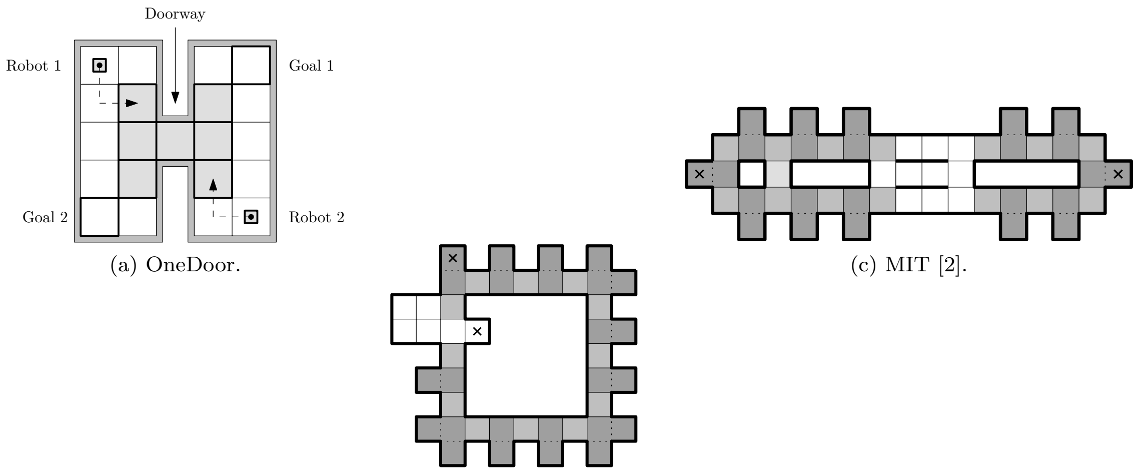
2. Run the solver on test problems and compare the performance and behavior with respect to the MDP-based methods. Vary the parameter that controls the number of beliefs sampled, for instance try 10, 100, 1000, 10000.

3. (Optional) At `pomdp.org` you can find code for several optimal POMDP solvers, which uses the same problem description format. You can experiment with the performance of optimal methods such as Incremental Pruning, comparing the run time and value to the point-based method that you tested.

In your report, compare the performance of the point-based algorithms (with different parameter settings) with the heuristic methods, both in terms of value and run time, and by comparing actual runs of the system. Explain the differences by referring to the theory discussed in class.

4 Assignments Part 2: multiagent planning under uncertainty

In the multiagent assignment you will test your value iteration code on multiagent problems, formalized as Multiagent MDPs [1]. We will use 3 two-agent domains, depicted in Figure 2. In ISR and MIT, the agents have to navigate to the states marked with an 'x', while in the OneDoor problem they have to navigate to their respective goals. While the agents can navigate independently in some parts of the environments, if they are located both in certain cells (representing narrow passages), they collide and receive a large penalty.



(a) OneDoor.

(b) ISR.

(c) MIT [2].

Figure 2: Problem domain maps for the multiagent problems.

1. Run your value iteration code on these MMDP models, and test the solution in a similar as in the single-agent case: by comparing the value and the trajectories the robots take. You can use the `plotState` function for the latter. Other useful functions are `splitMMDPagentStates` and `splitMMDPagentActions`, that separate joint states (actions) into the individual states (actions) for each agent. Experiment with different starting distributions. Note that, depending on your implementation, it might be beneficial to create and use sparse models.

2. In the `problem` struct, you can also find the individual models of these agents in `transitionInd` and `rewardInd`. Solve the individual models, and extend your `sampleTrajectories` function to handle the case of two agents choosing actions separately, but combining them to allow the system to evolve according to the joint model (use `combineMMDPagentActions`). What are the differences you see in performance? Can you come up with a way to avoid that the robots collide?

3. (Optional) Define an observation model for these robots (e.g., based on the number of walls surrounding a robot), to be able to address a corresponding Multiagent POMDP problem. Test the performance of Q_{MDP} and MLS on these problems, using the belief update you implemented for the single-agent case.

References

[1] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Theoretical Aspects of Rationality and Knowledge*, 1996.

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

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

[4] M. L. Littman, A. R. Cassandra, and L. Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *International Conference on Machine Learning*, 1995.

[5] Matthijs T. J. Spaan and Nikos Vlassis. A point-based POMDP algorithm for robot planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2399–2404, New Orleans, Louisiana, 2004.

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