TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Datenverarbeitung
Prof. Dr.-Ing. K. Diepold

Approximate Dynamic Programming & Reinforcement Learning

Assignment 5
December 2, 2019

General Information

Submit your code as `zip` archive via Moodle. Detailed instructions are available at the end of this document.

The deadline is 19. December 2019 at the end of the lecture (14:45 pm)

Write in every file your full name and your student ID. The fulfillment of all formalities will also be part of the points for this assignment.

Making Optimal Decisions in a Maze - (Points: 40)

In this programming assignment you will implement a simple stochastic environment and learn to make optimal decisions inside a maze. The task in this environment is to reach a goal state G from some starting state S while taking care of a trap T. Find the optimal policy for maneuvering an agent safely through the maze. This time you will make use of the Markov Decision Process (MDP) formulation (expectation directly over the successor states) instead of the system equation $f(x, u, w)$ approach.

The Environment

As the maze is a very simple environment, you can represent it by nested lists. 0 indicates an empty field in the maze where an agent can go. The symbol 1 represents a wall. There are always walls at the border. The starting position is denoted by S, the goal by G. Furthermore, there is a trap T which should be avoided. The agent has five actions to chose from $C = \{up, down, left, right, idle\}$.

```
# This is the definition of a maze
# Ignore lines starting with #
# 1: Wall 0: Free
# S: Start G: Goal T: Trap
#
1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 1 0 0 1
1 0 1 1 1 0 0 0 0 1
1 0 1 T 1 0 1 0 S 1
1 0 0 0 0 0 1 0 0 1
1 0 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 G 1
1 1 1 1 1 1 1 1 1 1
```

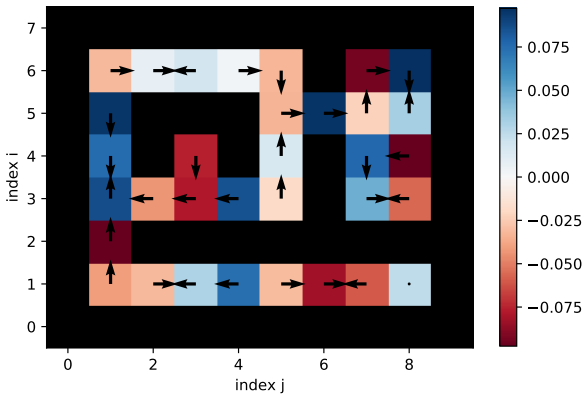

Figure 1: The maze as text file and a possible visualization of a random policy and value function.

The four directions move the agent around in the maze (if allowed). Once the agent transits into the goal state, it has to stay there and cannot move any further. Transitions from the goal state to itself are always at no cost. Your tasks:

- Adapt the maze for your programming environment. For grading a random maze will be used, so provide code to load a maze from a text file. The path to the text file will be the single command line argument. An example is given in Figure 1, use this maze for the assignment. Copy the text (keep spaces) or use the file from Moodle.
- Implement what you need to work with such a maze. E.g. you should be able to get the successor states x' for a current state x and control u. You may want to have a look-up table $U(x)$ to retrieve the allowed actions in a certain state x (the agent cannot move towards walls) and so on.

Probabilistic Formulation of the Problem

In order to solve arbitrary Markov Decision Processes with your implementation of the Dynamic Programming algorithms you must allow for a probabilistic formulation of the problem. Thus you have to:

- represent state transition probabilities $p_{ij}(u)$ in your code, i.e. the probability that a transition from state i to j under control u occurs. These can be used to evaluate expectations over the successor states j given the current state i and control u.
- describe why you decided to implement it in the way you did. What are possible problems of a naïve implementation of $p_{ij}(u)$ for all possible combinations of i,j and u?

In this maze the ground is a bit slippery, meaning that the outcome of an action varies. Because an oracle has told you the model of the environment, you know that applying an action will result with a probability $1 - 2p$, in the desired next state (e.g. executing *up*, if available, will result in the state above the current one). With probability p one of the two adjacent states, which are perpendicular to the moving direction, is reached. See Figure 2 for a visual explanation. If the agent walks along

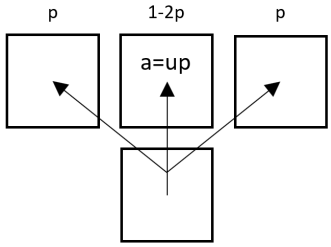

Figure 2: The transition model for the action *up*. Other actions work in a similar way.

a wall the corresponding adjacent state is not available, thus its probability to be reached is added to the desired state, $1 - 2p + p = 1 - p$. If the agent is in a corridor with walls on both sides the transition becomes deterministic, $1 - 2p + p + p = 1$. Use $p = 0.1$ in the assignment.

Local Cost-to-Go

You have to model the goal-finding with the help of the local cost-to-go $g(i, u, j)$, i.e. the cost obtained in state i when executing control u and reaching state j. In this task assume the cost is deterministic, that means the cost only depends on i and u. There are typically several possibilities to communicate the desired behavior via the costs, e.g.:

1. the states you encounter while walking through the maze are boring and give no cost, however the goal state contains a little treasure, i.e. a negative cost of -1 (that means a reward).
2. punishing the wast of energy: each control in any state of the maze produces a fixed cost of 1. Only the self loop in the goal state (which is the only allowed control in that state) is for free.

In both variations of the local cost-to-go function the state containing the trap results in a strong punishment for any control, $g(trap, \cdot) = 50$. Implement both local cost-to-go $g_1(i, u)$ and $g_2(i, u)$ as described above.

Solution with Dynamic Programming

Now all ingredients for solving the problem with Dynamic Programming are ready. The last thing that is missing is the value function $V(x)$ for all $x \in \mathcal{X}$ itself. To achieve this, think of a way to linearize the two dimensional and unstructured state space to represent the value function V as a simple 1-D vector. Your tasks:

- Find a method for representing the state space in your program. What is your solution? Why did you choose this solution, are there other possibilities?
- Write the code for the optimal Bellman operator

$$(T_g V)(i) = \min_{u \in \mathcal{U}(x)} \mathop{\mathbb{E}}_{j} [g(i, u) + \gamma V(j)] \quad \forall i \in \mathcal{X}$$

  and the Bellman Operator under policy π

$$(T_\pi V)(i) = \mathop{\mathbb{E}}_{j} [g(i, \pi(i)) + \gamma V(j)] \quad \forall i \in \mathcal{X}.$$

  Also the greedily induced policy must be calculated, thus you need to implement

$$\pi(i) = \operatorname*{argmin}_{u \in \mathcal{U}(x)} \mathop{\mathbb{E}}_{j} [g(i, u) + \gamma V(j)] \quad \forall i \in \mathcal{X}.$$

- Use these Bellman Operators to realize:
  - Value Iteration (VI)
  - Policy Iteration (PI)
  - Optimistic Policy Iteration (OPI) with $m = 50$ iterations for Policy Evaluation
- How can you determine if the algorithms have converged? Think of sensible and numerically stable criteria and describe them.
- Implement a visualization for a policy and cost function in a 2d plot. Colored boxes and a heat map could show the entries of V for each cell in the maze and a quiver plot can render the controls produced by the policy. An example is given in Figure 1.
- Derive the optimal policy and its value function with the three algorithms for both local cost-to-go g_1 and g_2. Use as discount factor $\gamma = 0.9$. Visualize V and π (6 plots in total).
- Do g_1 and g_2 always generate the same policy?

A Study of Algorithm Properties

In this section you will study further the behavior of the algorithms. First, you will investigate the discount factor:

- For VI and g_1 only, vary the discounting factor γ over the interval $[0.01, 0.99]$ with 15 subdivisions and let the algorithm run till "convergence" (long enough, since you cannot apply infinite many times T_g).
- Use the policy corresponding to V^* and the largest discount factor as reference.
- Count how many entries in the policy differ from the reference for other values of γ.
- Plot the result: γ on x-axis and the amount of entries on the y-axis.
- Plot the policy for the smallest γ as before. Has a change in γ a meaningful impact on the final policy? Argue why this happens.

Second, you will compare the different algorithms:

- Again for g_1 only, run VI, PI and OPI until "convergence" for $\gamma = 0.99$.
- Measure the runtime until each algorithm terminates and compare these values.
- What do you observe? What did you expect? Argue why this happens.

Finally, based on your finding, you will test OPI further:

- In the same setting as above, run OPI until "convergence" for different m.
- Pick $m \in \{1, \ldots, 100\}$ with suitable subdivision and measure the runtime.
- Plot the required time for each m and describe what you can see.

Final Instructions

Summarize your results in a short report and save it as a PDF file.

- The report has to contain the answers to all the sentences ending in a question mark.
- This should result in two pages of text.
- Then attach the plots, they do not count to the page limit.
- Try to put as many on one page without destroying the visual quality, typically a 3x2 grid is fine.

And concerning your code:

- You have to use Python for the implementation, other languages are not accepted.
- Python 3.x on a standard Ubuntu System is used for testing your submissions.
- Provide "plain" python scripts, we will **not** evaluate Jupyter Notebooks.
- You are allowed to use basic modules like Numpy for representing arrays or Matplotlib for plotting, but no existing RL/DP frameworks (e.g. the Reinforcement Learning Toolkit by Richard Sutton).

- The performance will not be graded, but your code must finish within a reasonable time.
- Provide one file called *main.py*, that can be run from the command line to produce **exactly** the plots you have in your report.
- You will receive a single command line argument with the path to a text file containing a maze to load, thus use the following command to start your code:

  ```
  python main.py /absolute/path/to/maze.txt
  ```

- The first maze will be the one from Figure 1, the second one will be a bigger and randomized one (the border is always covered by walls).
- You can either show the plots directly on the screen, or you can use `savefig()`. In the second case make sure to use a file format, that can be opened without effort (e.g. a simple `.png`).
- Add proper titles to identify them.
- Zip all your python scripts and the report to a single `.zip` archive. As filename use *lastname-firstname-matriculationnumber.zip*. Hand in your archive on Moodle in time.