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

Approximate Dynamic Programming & Reinforcement Learning

Programming Assignment

December 13, 2018

Introduction

In this programming assignment we will implement a simple 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. We now want to find the optimal policy for maneuvering the agent safely through the maze.

Important: You have to use Python for the implementation. We use Python 2.7 for testing your submissions. Most likely you can use Python 3.x without any problem, the only major issue should be the outcome of integer divisions: e.g. 1 / 4 results in 0 (integer, Python 2) instead of 0.25 (float, Python 3). Use 1.0 / 4.0 to avoid this problem.

Provide "plain" python scripts, that can be run by the interpreter from the command line. We will **not** touch Jupyter Notebooks.

You are allowed to use basic modules like Numpy for representing vectors or Matplotlib for plotting, but no existing RL/DP framework (e.g. the Reinforcement Learning Toolkit by Richard Sutton).

The performance will not be graded, but your code should finish within a minute.

The Environment

As the maze is a very simple environment, we can represent it by nested lists. 0 indicates an empty field in the maze where an agent can go. A 1 represents a wall. 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 $C = \{up, down, left, right, idle\}$. The four directions move the agent around in the maze (if possible). 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 at no cost. Your tasks:

- Adapt the maze for your programming environment. For the grading a random maze will be used. Provide the functionality 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.
- 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 and so on.

```
# This is the definition of a maze
# Lines starting with # must be ignored
# 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
```

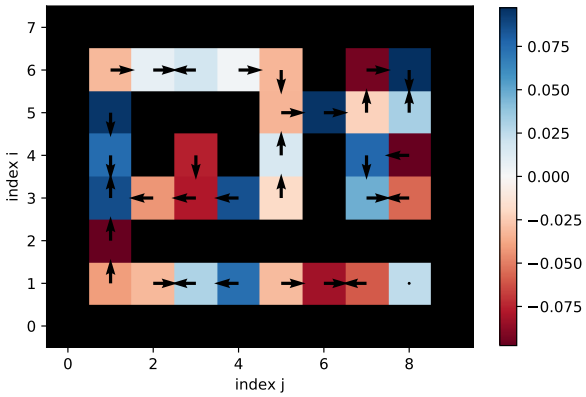

Figure 1: The maze as text file and a possible visualization of a random policy and cost function.

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, which can be used to evaluate expectations over the successor states j given the current state i and control u. If you want you can also use the system equation model $f(x, u, w)$ and encode the stochasticity in the parameter w.
- describe why you decided to implement it in the way you did. What are possible problems of a naive 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 adjacent states that are perpendicular to the moving direction is reached. See Figure 2 for a visual explanation.

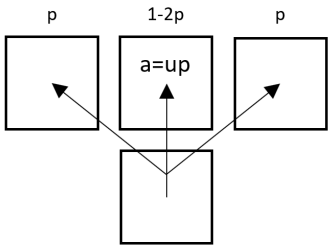

Figure 2: The transition model for the action *up*.

If the agent walks along 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.

One-step-cost

We have to model the goal-finding with the help of the one-step-costs $g(i, u, j)$. There are typically several possibilities to communicate the desired behavior via the costs, e.g.:

1. walking through the maze is at no cost, transitioning into the terminal goal state gets rewarded with a treasure, i.e. a negative cost of -1.
2. punishing the wast of energy, i.e. each move in the maze produces a fixed cost of 1. Only the self loop in the goal state is for free.

In any case the transition into the trap results in a strong punishment, $g(\cdot, \cdot, trap) = 50$. Implement both on-step-costs $g_1(i, u, j)$ and $g_2(i, u, j)$, which reflect the behavior described above.

Policy

In order to learn a optimal path from a start state to the goal, we need a policy to map each state to an (optimal) action. You have to:

- think of a way to represent a modifiable deterministic policy $\mu(i)\ \forall i \in \mathcal{S}$ in your code.
- implement and describe it.

Solution with Dynamic Programming

Now we have all the ingredients ready for solving the problem with Dynamic Programming. The last thing that is missing is the cost function J itself. To achieve this, think of a way to linearize the two dimensional and unstructured state space to represent the cost function J as a simple 1-D vector. Your tasks:

- Find a mapping to linearize the state space into a vector. What is your solution?
- Implement Value Iteration
- Implement Policy Iteration, i.e.:
  - Policy Evaluation
  - Policy Improvement
- How can you determine if the DP algorithms have converged? Think of a sensible numerically stable criteria and describe it.

- Implement a visualization for a policy and cost function in a 2d plot. Colored boxes and a heat map should show the entries of J for each cell in the maze and a quiver plot should render the actions produced by the policy. See the example in Figure 1.
- Derive the optimal policy and its cost function with both algorithms for $\alpha = 0.9$ and for both costs g_1 and g_2. Visualize J and μ.
- Do the two methods generate the same policy?
- Vary the discounting factor α and test two additional values (e.g. 0.5 and 0.01) an repeat the steps from before. Has the changed α an influence on the final policy?

A Study of Algorithm Properties

In this section we will study further the behavior of the algorithms when changing the discount:

- Run Policy Iteration and Value Iteration until each have converged with the discount factor $\alpha = 0.99$. Regard the resulting cost functions of those runs as the ground truth for each algorithm.
- Now run the algorithms again and plot the error (i.e. the squared distance) to the ground truths with respect to the iterations.
- Create these error plots for three suitable values of α and for both one-step-cost functions (= 6 graphs for VI + 6 for PI). Suitable α are those where its impact becomes visible and the comparison of PI and VI is easy. Which values for α did you use?
- Is VI or PI better / faster? Why? Describe your findings.

Final Instructions

Summarize your results in a **short** report and save it as a PDF file. The report has to contain the answers to the all the sentences ending in a question mark. This should result in approximately two pages. 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. Zip all your python scripts and the report to a single `.zip` archive. As filename use *lastname-firstname-matriculationnumber.zip*. and hand this in on Moodle. And concerning your code:

You have to provide one file called *main.py*, that can be run from the command line to produce the plots that you have in your report. You will receive a single command line argument with the path to the maze to load. The first maze will be the one from Figure 1, the second one will be a random one. We will use the following command:

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

Add to all plots a title to identify them. You can either show all the plots directly on the screen, or you can use savefig. In the second case make sure to use a format that can be openend direclty and without effort, like a simple `.png`.