02465 Project: Part 3

Tue Herlau
tuhe@dtu.dk

June 10, 2022

Formalities

- The deadline for this report is April 29, 2022 before 23:59.
- Submission of reports happen on DTU learn
- You can work in groups of 1, 2 or 3 students (but not 4)
- Collaboration policy: It is not allowed to collaborate with other groups on this project, except for discussing the text of the project with teachers and fellow students enrolled on the course in the same semester. Under no circumstances is it allowed to exchange, hand-over or in any other way communicate solutions or parts of solutions to the project to other people. It is not allowed to use solutions from previous years, or solutions found on the internet or elsewhere.
- You can (and probably should!) use code from the *exercises* when you solve the project, for instance the dynamical programming algorithm. The exercises may be solved with help from teachers or fellow students, and you can make use of the solutions I make available. However, you are not allowed to copy or share exercise code directly between groups or make solutions publicly available. This will ensure there is no accidental copying of projects.
- Your overall evaluation will be based on your written answers and your UNITGRADE score. They will be weighted based on an assessment of the required work.

Preparing the hand-in:

Hand in these three files (please do not hand in a `.zip` file as this confuse DTU learn):

A `.tex` file with your written answers: Prepared this by modifying the template in `irlc/project3/Latex/02465project3_handin.tex` . Simply write your answers where it says **YOUR SOLUTION HERE**. I recommend keeping the layout as it is.

A `.pdf` file corresponding to this `.tex` file

A `.token` file containing your python-solutions: Generate this file by running the script `irlc/project3/project3_grade.py` . It is very important you do not modify this file.

Contribution table

To make sure your final grade is properly individualized, each students contribution to the report must be clearly specified. Therefore, for each section or problem (or part of a problem), specify which student was responsible for it in the table in the template. A report must contain this documentation to be accepted. The responsibility assignment must be individualized. This means for reports made by 3 students: Each section must have a student who is 40% or more responsible. For reports made by 2 students: Each section must have a student who is 60% or more responsible. Please keep in mind this is an external requirement and it has to be that way. Ask me if you are in doubt about how to do this.

Code hand-in:

- Please keep the structure of the `irlc`-folder. All of your code which is specific to this report should be in the `irlc/project3/` directory. Solutions which use code outside the `irlc` folder cannot be verified and therefore cannot be evaluated. You can (of course) call, re-use or re-purpose any exercise code, including my solutions.

- If you wish to use additional third-party libraries please discuss them with me first to ensure you are on the right track, and make sure I can verify your solutions

- Breaking or tampering with the UNITGRADE framework, for instance by reporting a false number of points or making your solution unverifiable, is potentially cheating. Code build by reverse engineering specific tests and simply returning the values which makes them pass will not get credit and may also need to be treated as a cheating-attempt.

- That asides, this is not a programming course: Strange, long, undocumented and downright disturbing solutions will be evaluated simply based on whether they work or not.

Overall hints:

- You have free reins when it comes to solving the problems. However, they are often easier if you look at (and use) code from the exercises.

- You can put your code in multiple files if you feel like it.

- Although the script `irlc/project3/project3_grade.py` is used to generate the `.token`-file, I recommend running the normal version when you develop the code. You can run it by right-clicking on it from pycharm. I have tried to include tests which examine if you return the right data structures and so on. Use the debugger on the tests that fail, starting with the simplest tests.

- **I have added a video on `https://video.dtu.dk` on how to easily integrate the tests with the UI in Microsoft VSCode**.

1 Jar-Jar at the battle of Naboo (`jarjar.py`)

In this problem, you have the ungrateful job of ordering Jar-Jar around during the battle of Naboo. Experience has taught you to keep things simple. You can order Jar-Jar either left or right one step, and you want him to simply stay near a particular location $s = 0$ [1]

Specifically, the states Jar-Jar can be in are all integers:

$$\mathcal{S} = \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}.$$

In each turn, you can tell Jar-Jar to move one step left ($a = -1$) or right ($a = 1$). Although Jar-Jar always does as you tell him, he will not stand still, and so you have to give him orders in each turn. If for instance Jar-Jar is in state $s = 10$, and you tell him to move left $a = -1$, he will transition to state $s + a = 9$, and in the next turn you need to give him a new order so he transitions to either $s = 8$ or back to $s = 10$.

The reward in each step is (minus) the distance to 0, i.e. $r = -|s|$. In the previous example, the reward you would obtain would be $r = -|10| = -10$. The environment does not terminate.

You are particularly concerned with the $0 < \gamma < 1$-discounted return you will obtain over time, and you want to be able to predict what it will be for any state s Jar-Jar may be in. The following exercises will highlight how we can both compute this approximately and exactly.

As a warm-up, you will implement the optimal policy for ordering Jar-Jar around. I.e., the policy which minimize the accumulated reward:

Problem 1 *Optimal policy*

Implement the optimal policy function π^* as the function `optimal_pi`.

Info:

No theory required: Simply think about what we want Jar-Jar to do.

Actions are encoded as $a = -1$ (left) and $a = 1$ (right)

There are in fact two optimal policies (why?). Either one is acceptable.

Next, let's consider the optimal value function:

Problem 2 *Simulating a finite approximation of the optimal action-value function*

Simulate an approximate version of the optimal action-value function $Q^*(s, a)$ in the state $s = 0$ and using action $a = 1$.

To do this, simulate the optimal policy for N steps, starting in $s_0 = 0$, to get $N + 1$ states $s_0, s_1, \dots, s_N$ and the corresponding N rewards $r_1, r_2, \dots, r_N$. Use these to approximate $Q^*(0, 1)$ for an arbitrary discount factor γ. Implement the result as `Q0_approximate`.

[1] coincidentally, the landing spot of the trade federations landing vessel

Info:

Think about what Jar-Jar will do under the optimal policy you just defined; it is something very simple and predictable.

The result should probably involve a loop.

Problem 3 *Analytically computing the optimal action-value function*

Next, we will analytically compute an expression for the optimal action-value function when computed on an infinite horizon, i.e. the same as the previous exercise but for $N \to \infty$. Your job is to derive a simple, closed-form expression for the optimal value function. You should derive the result for just two values: $Q^*(0,1)$ and $Q^*(1,-1)$.

The phrase *simple, closed form expression* means it should be expressed **without** using the sum-sign Σ.

Answer:

Using that ... we obtain

$$Q^*(0,1) = \cdots \tag{1}$$
$$Q^*(1,-1) = \cdots \tag{2}$$

therefore...

Info:

- The two values of (s,a) are not chosen at random: First use the Bellman optimality condition for $Q^*(0,1)$ to express $Q^*(0,1)$ in terms of $Q^*(1,-1)$.
- Then use the Bellman optimality condition for $Q^*(1,-1)$ to express $Q^*(1,-1)$ in terms of $Q^*(0,1)$.
- This should give you two equations with two unknowns.
- The phrase *simple, closed-form expression* refers to an expression such as $Q^*(0,1) = \gamma^3 - 1$ or $Q^*(0,1) = \frac{1}{\gamma}e^\gamma$ (neither of these are the correct solution, obviously).
- You can check your result by selecting a very large value of N in the previous exercise.

Problem 4 *Extend solution to all states and actions*

Extend the solution above, which works for two combinations of s and a, to get an *exact* method for computing $Q^*(s, a)$ for all values of s, a. The method can involve for-loops or recursions, but should not simply correspond to selecting a large value of N in problem 2. Implement this as `Q_exact`.

Info:

- You should extend your previous solution. Start by considering $Q(2, -1)$ and use Bellmans optimality equation to express this in terms of a known Q-value. Then consider $Q(3, -1)$ and get a good idea
- Similarly, for actions that point *in the wrong way*, use Bellmans optimality condition to express $Q^*(1, 1)$ in terms of a (now) known quantity
- Since the problem is symmetric around $s = 0$, the Q^*-functions also obey symmetries
- It is fine to use a (finite) for loop or recursion when computing $Q(s, 1)$ in general. However, your expression must be exact – i.e. do not use an infinite sum that you truncate etc.
- If you really want, you can avoid for loops and recursion completely by using a bit of math to obtain a closed-form expression for $Q^*(s, a)$. However, this is not required or really that advisable.
- Unitgrade will test values that I consider simple to obtain first, and then more complicated values.

2 Finding the rebels using UCB-exploration (`rebels.py`)

Things have been tense after the rebels blew up the boss death-star, and finding them has become a top priority. This is done using search-droids, who must explore the various star systems the rebels may be hiding in. Practically, a search droid moves around on a grid, and gets a positive reward depending on how many rebels it finds, or a negative reward for flying into a dangerous zone and being destroyed. The problem, therefore, resembles the standard grid-world problem. You are particularly interested in the two scenarios show in the top row of fig. 1 (*basic* and *bridge*).

The existing exploration-approach (random ϵ-greedy; i.e. what the current Q-learning method does) was designed by the same person who made the curriculum at the marksmanship school for storm troopers, and your job is to see if you can do better – specifically, you want to use an UCB-inspired strategy.

To elaborate on this, it is useful to first refresh how regular Q-learning uses the regular ϵ-greedy bandit exploration method (see [SB18, Equation (2.5)]) when making decisions as explained by [SB18]:

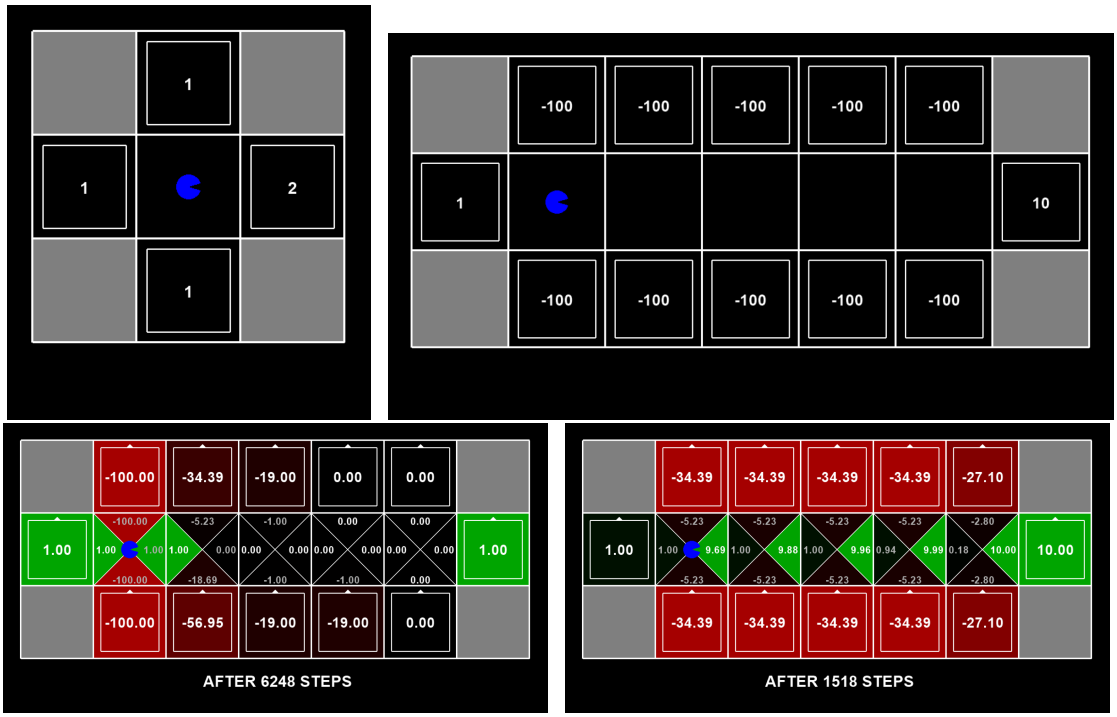

Figure 1: Top row: The basic and bridge rebel-locating scenario. Check the `rebels_demo.py` file to see how they are plotted and try to play in them yourself. Bottom row: The Q-values as obtained using a Q-learning agent for 3000 episodes (!) and an UCB-exploration agent for just 300 episodes.

- You have as many bandit-problems as there are states
- The actions for the bandit-problem in state s is $\mathcal{A}(s)$
- When the bandit, in state s, takes an action $a \in \mathcal{A}(s)$, the bandit-algorithm obtains a reward:

$$\tilde{R}_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'). \tag{3}$$

- The Q-learning agent then learns by updating the Q-values using this reward:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\tilde{R}_t - Q(s,a))$$

  just as in the regular bandit-learning algorithm [SB18, Equation (2.5)].

- Finally, the bandit in state s implements the ϵ-greedy rule: i.e. select the action with the highest Q-value with probability $1-\epsilon$ and otherwise a random action in $\mathcal{A}(s)$

To *use UCB for exploration* simply modify the above as follows:

- You have as many UCB-bandit-problems as there are states
- The actions for the bandit-problem in state s is $\mathcal{A}(s)$

- When the bandit, in state s, takes an action $a \in \mathcal{A}(s)$, the UCB-bandit-algorithm once again obtains a reward of $\tilde{R}_t$ (see eq. (3)).

- The bandit (specific for state s) is trained and selects actions using the UCB-algorithm (see [SB18, Eq. (2.10)]) using the reward $\tilde{R}_t$.

- In this equation, note that Q_t has the same meaning as before (and should therefore be updated the same way), but you need to define N_t and t to be specific for this state. I.e., keep track of how many times we have seen the state t, and how many times we have played a given action $N_t(a)$ in the state.

- The UCB-algorithm in [SB18] has the following corner-cases:

  - If two actions have the same priority (i.e., same value of the UCB-bound according to what is inside the $\arg\max_a$ in [SB18, Eq. (2.10)], we prefer the action with the *lowest* index (i.e. we prefer $a = 1$ over $a = 3$ etc.)

  - If an actions have not been tried before, so that $N(a) = 0$, it has infinitely high priority and will always be selected. I.e. the method will first try all actions once in order: $a = 0, 1, 2, \ldots$.

In the first of the two scenarios shown in fig. 1 (basic), the agent can take one of four actions in the starting state, and from there it must take a single extra action to (deterministically) transition to the terminal state with the reward shown as the numbers in the figure. Since there is one relevant state (the starting state), the problem will closely resemble the standard UCB bandit problem. One slight annoyance is that the last (dummy) action in an episode have no effect in the gridworld-environment and we will therefore discard it. You can find an example (see the `rebels_demo.py`)-file which illustrates how you can generate actions using an agent and discard the last (dummy) action:

```
Trajectory 0: States traversed [(1, 1), (0, 1), 'Terminal state'] actions taken [3, 3]
Trajectory 1: States traversed [(1, 1), (2, 1), 'Terminal state'] actions taken [1, 3]
All actions taken in 16 episodes, excluding the terminal (dummy) action [[3], [1], [0], [0], [3], [2], [2], [2], [0], [2], [3], [3], [1],
↪  [3], [3], [2]]
```

The file also contains code for visualizations, and how to turn on keyboard-inputs so you can play with your UCB-agent.

Problem 5 *UCB-based exploration*

Implement the function `get_ucb_actions`. The function should accept a gridworld layout, the parameter α controlling the learning rate in the Q-learning algorithm, the exploration parameter c (the constant which control exploration in the UCB algorithm), and a number of episodes to simulate. You can assume that $\gamma = 1$ in all experiments.

The function should return a list simply consisting of all actions the agent takes but *excluding* the last dummy action. i.e., simply remove the last action in each episode and concatenate the action-sequences together. Example code for doing this is included in the `rebels_demo.py` file. The return value should thus be a list of integers, and in case of the basic grid, it should have the same length as the number of episodes.

Info:

Overall hints:

- Since the method will be based on Q-learning, the code for the Q-learning agent is probably a good place to start (by start I mean copy-paste). Note in particular the Q-values are updated the same in the UCB and ϵ-greedy method.
- You can turn the grid layout (a list of lists of strings) into a gym `Env` using the methods from the toolbox. See the `rebels_demo.py` file for more details.
- If you write an agent, you can use the `train`-function to get a list of trajectories, and you can simply concatenate them together to get the list of all actions. See the `rebels_demo.py`-script for details.
- Note that to implement the UCB-algorithm, you must keep track of how many times each state has been visited, and how many times an action has been taken in that state. In other words, you must keep track of the N-values (see [SB18, Eq. (2.10)]). Note there are as many of these as there are Q-values...
- Again, if you look at [SB18, Eq. (2.10)], the t-parameter must actually be equal to: $t = \sum_a N_t(a)$. This can potentially reduce what you need to keep track of.
- Remember that in the basic-environment, with a single state, there is a single UCB-bandit problem and it will therefore be quite simple to reason about/keep track of: it is just doing UCB-learning.

Implementation and self-check hints: UCB is a simple algorithm and you can reason about what the method will do. Two cases are particularly interesting:

- The first four episodes will compute the same action sequence regardless of the rewards. Figure out why and check your result.
- Actually, the first 8 episodes will always be the same independent of the rewards. To explain this, you should consider eq. (3), and that each episode in the basic-environment consists of two actions and three states.
- From here, you can manually run the UCB algorithm and figure out the actions taken.

Figure 2: The inside of the great Sarlacc

3 Individual contribution: The great sarlacc (`sarlacc.py`)

Individual contribution: **This is an individual part of the project. This means that you have to work out a solution yourself without collaborating with your other team members. To do this, work on the code in the `irlc/project3i/`-directory, and create a `.token` file as usual using the test-script in this directory. You can upload the `.token` to the (individual) assignment on DTU learn.**

Oh no, you took a job on a sail barge and next thing you know, someone pushed you overboard and you are now in the belly of the great Sarlacc. It is said people who fall into the great Sarlacc are slowly eaten over 1000 years, however, you plan to climb out. The only question is how long this will take?

Climbing out of a Sarlacc a random process where in each turn, you can take 1, 2, 3, 4, 5 or 6 steps forward (each selected with probability $\frac{1}{6}$ as a random roll). Small Sarlacc worms (illustrated as snakes) make you slide back, and easily-traversible parts (illustrated as ladders) allows you to climb up quickly. In other words, it resembles the snakes-and-ladders game, and you draw up the diagram in fig. 2 representing the location of the snakes and ladders.

We consider the states of the game as integers s. You start in the lower-right corner ($s = 0$, indicated by the Zebra the Sarlacc once ate). To win, you need to exactly land on the winning tile, and excess moves beyond the winning tile makes you move backwards. Some examples of the game rules:

- If in the first turn you roll a 2, you go to the base of the ladder and transition to state $s = 16$
- If in the next turn you roll a 1, you go to state 27
- The game terminates when you land on tile 55. That is, if you are in state 54, roll a 1, you win

- Excess moves make you bounce and move backwards. If you are in state 54, roll a 3, you go to state 53, and if you had rolled a 2, you would have stayed in state 54.

We can represent the game rules as a dictionary where each key is a state, and the value is the state the player is teleported to. We include a special transition to state -1 to correspond to winning (compare the rules dictionary in the code to fig. 2).

Problem 6 *Sarlacc rules*

Implement the `game_rules` function. It takes a rules-dictionary, a state and a roll of the die, and return the next state. Note that the function return -1 when the game is won.

Info:

- First consider what happens in the starting state. This will be fairly easy. Work out the result in specific situations and check your function agrees with your answer
- The states near the goal are slightly harder but not very hard. Consider concrete examples, for instance the $s = 54$ and $s = 53$, and use these to generalize to the complete solution.

To work out the time it will take to escape we consider the game as a Markov Decision process. We are interested in the time it will take to complete, and therefore we associate each step (i.e., each of the four examples above) with a reward of 1. Your job is to compute the expected return given a set of game rules and a discount factor γ; the expected return when $\gamma = 1$ will then correspond to the number of steps until winning.

Problem 7 *Escape the Sarlacc*

Compute the value function for the Sarlacc escape-problem. The value function should work for arbitrary game rules (i.e., dictionaries) and for arbitrary values of the discount factor γ. It should return a dictionary where the keys are tile numbers and the values the γ-discounted value function. The dictionary should include $s = 0$ and all states the player can be in (but not $s = -1$ as this is a terminal state).

Info:

- What about the actions? You already saw a problem in which the actions had no effect (the chess problem all the way back in week 1, se `chess.py`). How was that situation handled?
- Unitgrade will check if your dictionary contains the right keys. Make sure this is the case.
- The problem resembles the Gambler problem from [SB18]. Look at the approach to this problem to get you started
- The `game_rules` function will likely be of help.
- Since the problem is a Markov decision process the problem will likely involve probabilities. It is always a good idea to check the probabilities you compute sum to 1 using an automatic test.

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (See **sutton2018.pdf**).