02465 Project: Part 1

Tue Herlau
tuhe@dtu.dk

June 10, 2022

Formalities

- The deadline for this report is March 4, 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/project1/Latex/02465project1_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/project1/project1_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/project1/` 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/project1/project1_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 Avoid the droid (pacman.py)

R2D2, who in this problem bears a remarkable resemblance to pacman, must first find the data-discs with the plans for the death-star (these are illustrated as small dots) while avoiding the blue patrol ghost-droids. Since this is rather important, R2D2 must determine the absolutely optimal plan which we will do using dynamical programming.

Your job is to help R2D2 by carrying out a sequence of increasingly difficult tasks culminating in the full solution. It is highly recommended you make sure each task is completed correctly before you progress to the next one (use the UNITGRADE test-scripts). The first tasks are easier, since R2D2 does not have to account for the patrol-droids and just has to find the optimal way to pick up data-discs.

1.1 Getting set up

Maps are specified as strings such as

```
# pacman.py

east = """
%%%%%%%%
% P   .%
%%%%%%%% """
```

The maps are loaded using an openai gym environment. The environments `reset()` method can then be used to get a state corresponding to this map configuration

```
# pacman_demo.py
    # Instantiate the map 'east' and get a GameState instance:
    env = GymPacmanEnvironment(layout_str=east)
    x = env.reset() # x is not a GameState object. See irlc/pacman/gamestate.py for
    ↪  the definition if you are curious.
    print("Start configuration of board:")
    print(x)
    # The GameState object `x` has a handful of useful functions. These are
    # x.A()       # Action space
    # x.f(action) # State resulting in taking action 'action' in state 'x'
    # x.players() # Number of agents on board (at least 1)
    # x.player()  # Whose turn it is (player = 0 is us)
    # x.isWin()   # True if we have won
    # x.isLose()  # True if we have lost
    # You can check if two GameState objects x1 and x2 are the same by simply doing
    ↪  x1 == x2.
```

The state `x` is a bit special as it is actually an instance of a `GameState` object, which means that it contains a number of useful functions to get the available actions and so on. These functions, which are shown in the comment above, are *all* you need to use to complete this project. The above output shows one of the things we can do with the

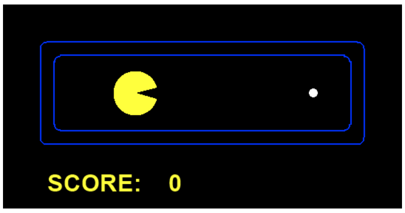

Figure 1: The corridor-map

`GameState`, namely print the game state corresponding to the `GameState`. The rest of the code also shows how you can render the `GameState` in a nicer way as shown in fig. 1.

```
Start configuration of board:
%%%%%%%%
% <   .%
%%%%%%%%
Score: 0
```

The first question will help familiarize ourselves with the `Gamestate` by simply walking east to get to the datadisc.

Problem 1 *Go east*

Help R2D2 get to the datadisc by completing the function `go_east`. The function should take a map (a string) as an input, and return a list of `GameState` objects corresponding to the path to the datadisc, starting in the initial configuration of the map. The last `GameState` in the list should correspond to the winning configuration.

You should assume the map has the form of an east-bound corridor (as in fig. 1), but of arbitrary length – R2D2 can therefore solve the problem by just going east until victory.

Info:

- This is supposed to be an easy question. Don't get too creative, just go east!
- Check out the code in `pacman_demo.py` to see how to instantiate the gamestate `x`. After you have instantiated `x`, play around with it in the console interpreter
- Use the action-space function `x.A()` to figure out what the east-action is
- Use the next-state function `x.f(action)` to move towards the datadisc
- Use `print(x)` whenever you are confused about what `x` is
- Append all the `x`'s to a list and return it
- Make sure the function works for arbitrary-length corridors (still, you only have to consider the east-bound direction)
- **Use the Unitgrade test script to debug your code. Remember you can re-run individual tests to limit the junk-output**

Problem 2 *Describe the go-east problem*

In [Her21, chapter 4] we were introduced to a lot of technical words to describe both controllers and environments. Try to use the terminology to describe (i.e., classify) the go-east environment and the controller (agent). Can the environment be solved by an open-loop controller?

Answer:
The environment is an example of a ....
The controller is an example of a ... **YOUR SOLUTION HERE**

Info:

- Describe just mean in words; some variability is obviously expected
- Only consider the simple version of the problem, and not the full pacman-problem we will consider later.

1.2 No droid planning

To apply dynamical programming, we must first determine the $N+1$ state-spaces $\mathcal{S}_0, \ldots, \mathcal{S}_N$. This question will break this problem into smaller tasks which, once complete, will make applying DP trivial. This also means the task formulation may seem a little counter-intuitive at first.

Problem 3 *Predict consequence of actions*

First, R2D2 must know the consequences of a single action. Do this by completing the `p_next(x,u)` function, which takes a `GameState` and an action as input, and return a dictionary which has the next possible outcomes as keys, and their probability as values.

Info:

- The problem is deterministic. There is a single outcome. What is the probability of something happening with certainty?
- Your solution will likely just be a single, short line
- What did f do?

We can now construct the state-spaces $\mathcal{S}_0, \dots, \mathcal{S}_N$. This will be done using a single function, which should therefore return a list of lists of states, such that the k'th element is $\mathcal{S}_k$, which corresponds to the states R2D2 can reach in exactly k moves starting from the initial state

Problem 4 *Possible future states*

Complete the function `get_future_states(x, N)`, which return a list of length $N+1$ such that the k'th element correspond to $\mathcal{S}_k$ (the states R2D2 can reach in k steps starting from `x`.

Info:

- I recommend using `p_next` when solving this problem
- The first set $\mathcal{S}_0$ is just the singleton list `[x0]`
- Use a loop over actions
- If you are stuck, consider how you would make $\mathcal{S}_1$ given $\mathcal{S}_0$?
- You don't have to account for ghost-droids yet: Everything is deterministic
- Check your solution using UNITGRADE
- The function should return a list of lists of `GameState` objects
- Remember to avoid duplicated states in the list; you can do this by removing them afterwards or by checking for duplicates before adding elements. I added a test to unitgrade a few weeks back to check for duplicates.

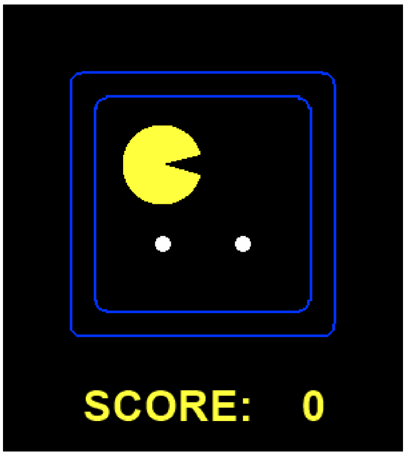

Figure 2: The state-counting map

Problem 5 *Reachable states*

Use pen-and-paper to account for the final number of possible states for $N = 10$, $\mathcal{S}_{10}$, in the map given in fig. 2. Do this by dividing the total number of states into different categories and count them; i.e. you should provide a justification for your result.

Answer:
Account for all states here. Provide a brief justification for your results. **YOUR SOLUTION HERE**

Info:

- The previous exercise allows you to confirm your answer.
- Remember you can use `print(str(s))` where `s` is a `GameState` to get a string representation of a state. Use this, and the previous hint, if you are stuck

Problem 6 *Shortest path*

You now have all the ingredients to find the datadiscs in the least amount of time. Do this by completing the function `shortest_path(map_layout, N)`. It should return two values (i) a list of optimal actions (ii) the list of states R2D2 will thereby traverse. Doing this, R2D2 should be able to find the optimal path in the no-ghost map shown in fig. 3.

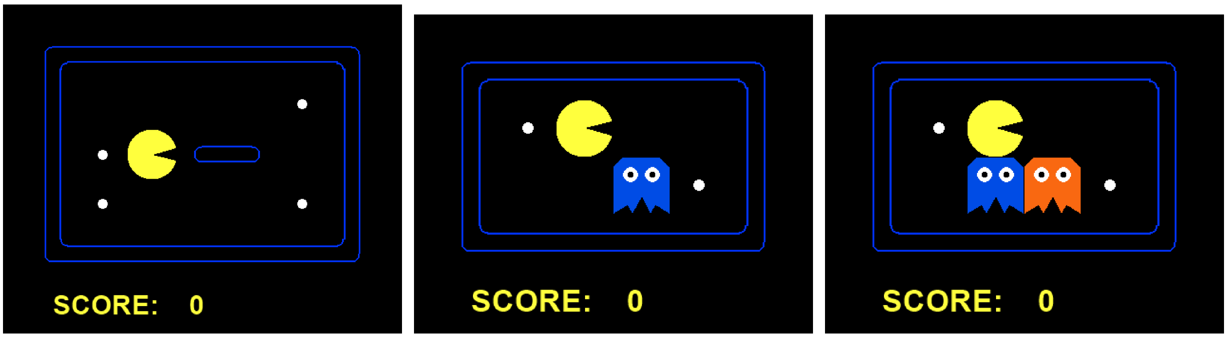

Figure 3: Example map with no ghosts (`datadiscs`), one ghost (`SS1tiny`) and two ghosts (`SS2tiny`)

Info:

- Solve the problem with regular (backward) DP
- To do so, build a DP-problem class corresponding to the problem
- The function in the DP problem class should be very short. Use the two functions you have just defined.
- What is an appropriate cost function to find the shortest path? Make a relevant choice. Remember that all the cost function need to do is to assign a higher cost the longer it takes for the agent to win.
- First check the optimal cost agrees with what it should be, then focus on the optimal action/state sequences
- The sequence of optimal actions/states should be constructed using the optimal policy found by the DP-problem
- In continuation of the above, there is in fact a problem from the first week which is highly relevant for showing how to simulate a policy and obtain the states/actions from a starting state.
- Check your solution using Unitgrade

1.3 One ghost-droid problems

In this problem R2D2 has to account for one ghost-droid (see map in fig. 3). We have to account for the randomness in the problem using the w_k-disturbance terms in the DP problem. To this end, remember that a ghost-droid selects between the available actions with uniform probability.

Problem 7 *Predict consequence of actions with one ghost*

Update the `p_next(x,u)` function to account for a single ghost-droid. The input it still an action, and the output is a dictionary where the keys are the states obtained after: (i) one deterministic move by Pacman and (ii) one random move of the ghost, and the values are their probability.

Info:

- Remember probabilities should sum to 1. Check this is the case as a sanity-check
- Since a ghost (usually) have three available actions, the dictionary will (usually) have three elements
- Note sometimes a ghost has less than three available actions, adjust the probabilities accordingly
- If you have troubles figuring out how to choose w_k, look at the exercises and lecture notes for inspiration
- Check the output of the function manually and make sure it looks okay
- The code should be a natural extension of the code you have already written; i.e. the previous exercise should keep working, and the changes will properly be fairly minimal
- Remember the move of the ghost is made in the game state that arises after pacman has made his move. Use the functions mentioned in the `pacman_demo.py` file.
- Check your solution using Unitgrade

Next, we will compute the state-spaces $\mathcal{S}_0, \dots, \mathcal{S}_N$ when there is one ghost-droid

Problem 8 *Possible future states with one ghost*

Update the function `get_future_states(x, N)`, which return a list of length $N + 1$ such that the k'th element correspond to $\mathcal{S}_k$ (the states R2D2 can reach in k steps starting from `x`, i.e. when it is R2D2's turn to move).

Info:

- The code should be a small update to your existing function assuming you used `p_next`. In fact, the code you have already written may in fact work
- If not, consider how you can use `p_next` to solve this problem
- Check your solution using Unitgrade

Problem 9 *Optimal one-ghost planning*

You now have all the ingredients to find the datadisc, while avoiding being caught by the ghost. To implement this, we assume that the reward is $g_N(x_N) = -1$ if the terminal state x_N is a won configuration and $g_N(x_N) = 0$ if it is lost, and that at other time steps $g_k(x_k, u_k, w_k) = 0$. In other words, the function you implement should just return the probability of winning within N steps; how soon it happens within the N steps is irrelevant.

Info:

- Solve this using a new DP model, nearly identical to the old one, and using backward-DP
- We are not adding a living-cost since we are only interested in the chance of success, not how long time it takes.
- I recommend you only compute each state-space once and store the result. They do not change, and your implementation can become very slow if you re-compute them many times.
- Check your solution using Unitgrade

1.4 Any-ghost planning

We are finally ready to tackle an arbitrary number of ghosts and still plan completely optimally. The most difficult part of this task is the following:

Problem 10 *Predict consequence of actions with several ghosts*

Update the `p_next(x,u)` function to account for any number of ghost-droids. I.e, in the case of two ghost-droids, the function is given a state and action, and returns a dictionary where the keys are the states obtained after (i) one deterministic move by pacman (ii) one (random) move by the first ghost and (iii) another random move by the second ghost, and the values are the probability of the resulting state.

Info:

- Focus on understanding what happens when e.g. `x` is the starting configuration and `u` is a specific action (such as going east). What can the ghosts do? You can in principle list all possible states and count their probability, and then check the result with your code.
- Remember probabilities should sum to 1. Although not required, I recommend checking if this is the case automatically as a sanity-check to see if you are on the right track. I suspect most bugs will be found in this way.
- Remember the ghosts take turns to move. Implement this as a for-loop over the number of ghosts
- For each ghost, you still have to loop over all actions it can take
- $P(A, B) = P(A)P(B)$ for independent events. The ghost-movements are independent.
- Errors are probably due to either not updating probabilities correct, or that you are not accounting for certain moves the ghosts can make. Use the ghosts action spaces to get all available moves for the ghosts.
- Although the difference between 2 and 5 ghosts are is likely to be minuscule, your code will only be tested with 0, 1 or 2 ghosts.
- Check your solution using Unitgrade.

Next, verify you compute your state spaces correctly:

Problem 11 *Future states*

Update `get_future_states(x, N)` to work for any number of ghosts.

Info:

- If you used `p_next` in your implementation, it is very likely your implementation just works
- Check your solution using Unitgrade

If you pass all tests in unitgrade, it is time to move on to the final test: R2D2 should now be able to steal the plans for the death-star and evade the ghost-droids using optimally planning over N steps.

Problem 12 *Optimal planning*

Solve the missing optimal-planning problems for two ghost-droids to compute the chance R2D2s mission will succeed.

Info:

- Use same cost function as before
- Very likely, no additional code is necessary. If you find yourself writing a lot of code, you may be on the wrong track
- All unitgrade tests relating to this exercise should pass.

2 The kiosk (`kiosk.py`)

In this problem, you take the role of a blaster salesman on the desert planet of Tatoine. You sell blasters to Jawas and Tusken Raiders and, as we will see, this is a surprisingly well-regulated line of work with some unique challenges.

Your job in this problem is to determine how many blasters to buy each day in order to maximize your expected profit. Let's first establish the basic rules:

- You can have $0, 1, 2, \ldots, n_s$ blasters in the kiosk.
- You can order $0, 1, 2, \ldots, n_o$ blasters to restock your inventory
- The cost of ordering a single blaster is 1.5 credits
- The sale price of a blaster is 2.1 credits
- You will plan on a horizon of $N = 14$ days

Running a kiosk resembles the inventory-control problem which we saw in [Her21, section 5.1.2]. Each day, starting in day $k = 0$, it goes as follows:

- Very early in the morning you have an initial inventory
- Based on this, you submit an order of a given number of blasters to the orbital merchant ship
- The number of blasters you ordered are delivered before your shop opens
- Customers buy a number of blasters during the day
- Based on the initial inventory, the number of ordered blasters, and the number of purchases you obtain a daily reward, and a final inventory

As an example, suppose over the full $N = 14$-day period you sell a total of 6 blasters and buy 5. The total (accumulated) profit will then be

$$6 \times 2.1 - 5 \times 1.5.$$

How you should plan will depend on your assumptions. These are going to change from problem to problem as we take more effects of local life into account: We start from a simple model, and then make it more realistic. Therefore, although the problems can be solved independently, I recommend solving the simple problems first and modifying the solutions.

Problem 13 *A basic blaster-business*

To get started, we make the following assumptions:

- The empire does not allow you to store more than $n_s = 20$ blasters overnight for safety reasons. In other words, suppose in the morning you have $x = 15$ blasters, you order an additional 10, and you sell a total of 3. Then in the evening you will have 15+10-3 = 22 blasters, and you have to discard two blasters so the inventory the following morning will be $x' = 20$.
- Discarding blasters does not cost anything
- Your cost-function is solely determined by your profit; the more profit, the better
- you can order up to $n_o = 15$ blasters at a time

In addition to this, suppose that Tusken raiders always buy blasters 3 at a time. Therefore, the chance there is a demand of 0 blasters in a day is 30%, 3 blasters in a day is 60% and 6 blasters in a single day is 10%; all other demands can be ruled out (the demand otherwise work as for the inventory environment: if you have 2 blasters and the demand is 6, you will sell 2 blasters). Given this information, formulate the blaster-business as standard decision problem below:

Answer:
YOUR SOLUTION HERE To get you started:

$$N = 14 \tag{1}$$
$$\textbf{for } k = 0, \dots, N\textbf{:} \quad \mathcal{S}_k = \dots \tag{2}$$
$$\textbf{for } k = 0, \dots, N-1\textbf{:} \quad \mathcal{A}_k(x_k) = \dots \tag{3}$$
$$\vdots \tag{4}$$

Problem 14 *Warmup*

Now that we have gotten this far, complete the functions `warmup_states()` and `warmup_actions()` which should return $\mathcal{S}_0$ and $\mathcal{A}(x_0)$ respectively. Since $\mathcal{A}(x_0)$ is independent of x_0, you can ignore the x_0-value.

Info:

- These should be around one line each – just return the state and action spaces as sets
- Read the description of the problem above (or look at the previous exercise)
- Look at the inventory-control environment for inspiration

Problem 15 *Manually computing J_{N-1}*

Consider the expected optimal cost-to-go just before the last action is taken, $J_{N-1}(x_{N-1})$. Suppose x_{N-1} corresponds to a completely full inventory. Calculate the value of $J_{N-1}(20)$ below and explain your calculation.

Answer:
YOUR SOLUTION HERE

$$J_{N-1}(20) = \dots$$

Info:

- Beware of the signs $\pm$!

Problem 16 *Compute optimal policy and value function*

Implement the function `solve_kiosk_1()` which returns the optimal value function and policy in the usual format. That is:

- Value functions are a list-of-dictionaries such that `J[k][10]` is $J_k(x = 10)$
- Policies are a list-of-dictionaries such that `pi[k][10]` is $\mu_k(x = 10)$ (i.e. the number of blasters to buy if the inventory is $x = 10$ blasters in planning round k)

Info:

- The recommended way to solve this problem is by using the DP algorithm just like the inventory-control environment. You can call the code you used in the exercises.
- If you take this approach, you should implement a DP-model corresponding to the problem
- The format of `J` and `pi` discussed above is compatible with the output of the DP algorithm
- Check your results using UNITGRADE

Problem 17 *Kiosk2*

As it turns out, the previous plan was way to to naive and failed to take two important factors into account

- The demand for blasters actually resemble a binomial distribution. The chance that $w = 0, \ldots, n_s$ blasters are bought in a given day is

  $$p(w) = \binom{n_s}{w} p^w (1-p)^{n_s - w}.$$

  Where n_s is the storage space. Using historical data you determine that $p = \frac{1}{5}$.
- When you dispose of excess blasters (i.e., blasters that you are not allowed to store over night) you have to obey the pesky imperial environmental protection act regarding safe handling of dedlanite and bla bla bla. Anyway, it costs 3 credits to dispose of a *single* excess blaster.

Implement these rules in `solve_kiosk_2()` and check how it affects your profits.

Info:

- You probably need to change two functions
- `scipy.stats` has a build-in binomial distribution function. Alternatively, just implement the function yourself using the above equation.
- Manually check if you compute the binomial probabilities correctly; this is an easy way to avoid one potential source of problems
- Check your results using UNITGRADE

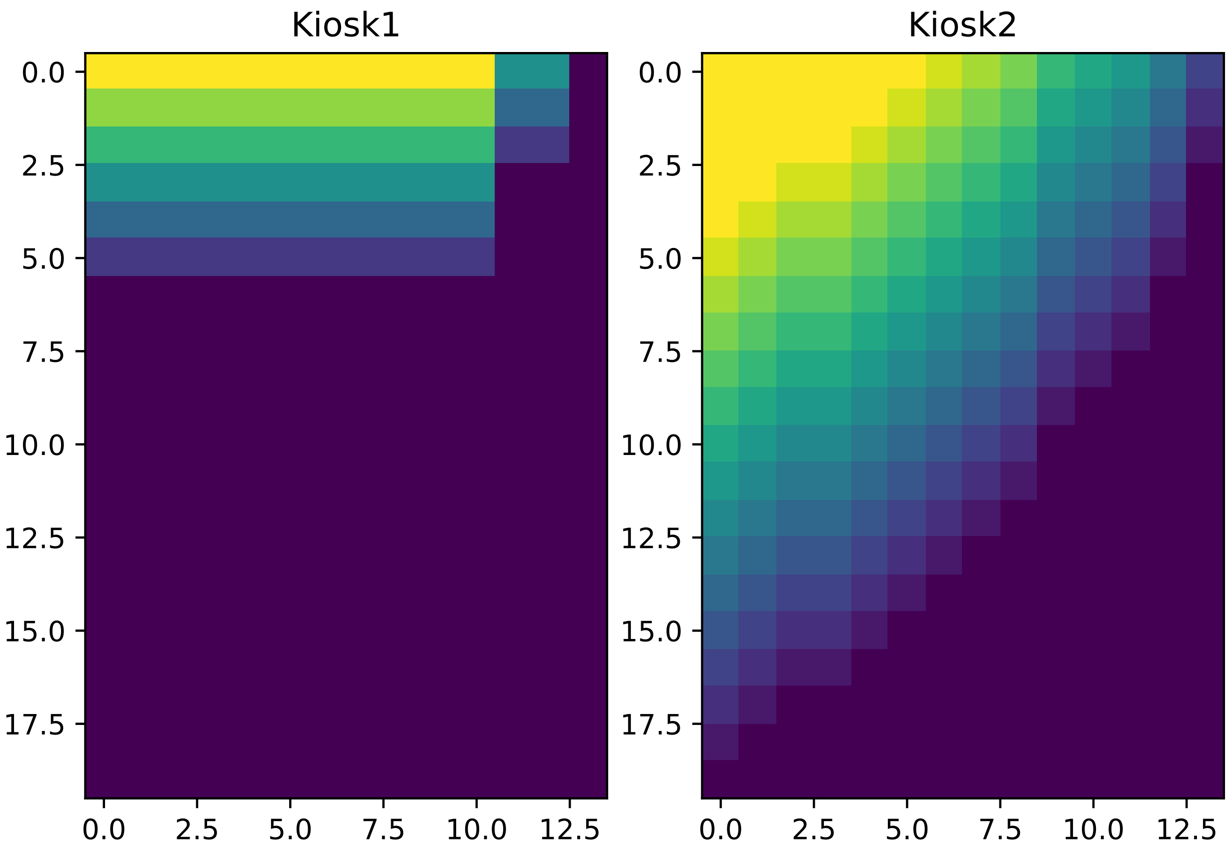

Figure 4: Plot of policies for the two first questions (kiosk1 and kiosk2)

2.1 Explaining policies

If you implemented the previous methods correctly the policies will be plotted (see fig. 4). The format of the figures will not be explained here – you have to look at the code and figure it out. As you can tell from the figure, the changes between kiosk1 and kiosk2 caused a change in the overall behavior of the policy. Your job is to explain this change.

Problem 18 *Explaining the policy*

Explain the overall change in policy shown in fig. 4. In other words, what do you see? How would you describe what the blaster-policies attempt to do in words (especially the first policy tries to do something very natural considering the problem). What aspect of the cost function do you attribute this change to? How does the policy suggest we should change our blaster-restocking policy as time elapses?

Answer:
The first policy... this can be explained by noting ... **YOUR SOLUTION HERE**

Info:

- You don't have to give a mathematical argument.

Problem 19 *Policy explanation continued*

The two policies differ as to how many blasters should be bought on day $N-1$ (i.e., the last day where a decision is made) assuming the inventory is empty. Provide a mathematical argument to show how many blasters the first policy prefers to buy on the last day given the inventory is empty, i.e. compute $\mu_{N-1}(0)$ manually.

Answer:

$$\mu_{N-1}(0) = \dots$$

YOUR SOLUTION HERE

Info:

- Remember, this is what you code does right now; it just does not tell you the intermediate calculations.

References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.