# EXERCISE 2
# Dynamical Programming

Tue Herlau
tuhe@dtu.dk

11 February, 2022

**Objective:** The goal of this exercise is to introduce the dynamical programming (DP) algorithm in its most general form (backward-dp). To practically implement it, we need to introduce a model-class to represent the various terms in the DP problem such as $f_k$ and $g_k$. (39 lines of code)
**Material:** Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

## Contents



# 1 Deterministic environments and graphs (`graph_traversal.py`)

Our first problem will examine the dynamical programming model in the context of graphs. Consider the graph problem in [Her21, fig. 4.4] where the goal is to go from a node $x_0$ to a destination node $t$. A connected graph can be represented as a collection of edges, which are naturally represented using a dictionary, where the keys are edges and the values are their weights:

```python
# graph_traversal.py
G222 = {(1, 2): 6,  (1, 3): 5, (1, 4): 2, (1, 5): 2,
        (2, 3): .5, (2, 4): 5, (2, 5): 7,
        (3, 4): 1,  (3, 5): 5, (4, 5): 3}
```

I.e. the weight between vertex $i$ and $j$ is `G222[(i,j)]`. If dictionaries are at all unfamiliar, I greatly recommend reading [Her21, chapter 2].

To evaluate a deterministic environment, what we need is the transition function $f_k$ and cost functions $g_N$ and $g_k$. For generality, we have given their signature in the following abstract class which all instance of the basic dynamical programming problem inherits from (the code can be found in `dp_model.py`)

```python
# dp_model.py
class DPModel:
    def __init__(self, N):
        self.N = N

    def f(self, x, u, w, k):
        raise NotImplementedError("Return f_k(x,u,w)")

    def g(self, x, u, w, k):
        raise NotImplementedError("Return g_k(x,u,w)")

    def gN(self, x):
        raise NotImplementedError("Return g_N(x)")

    def S(self, k):
        raise NotImplementedError("Return state space as set S_k = {x_1, x_2, ...}")

    def A(self, x, k):
        raise NotImplementedError("Return action space as set A(x_k) = {u_1, u_2,
        ↪  ...}")

    def Pw(self, x, u, k):
        """
        At step k, given x_k, u_k, compute the set of random noise disturbances w
        and their probabilities as a dict {..., w_i: pw_i, ...} such that

        pw_i = P_k(w_i | x, u)
        """
        return {'w_dummy': 1/3, 42: 2/3}  # P(w_k="w_dummy") = 1/3, P(w_k =42)=2/3.
```

to implement the graph environment, we will inherit this class and implement the specific transition/cost functions we need for our problem.

## 1.1 Implementing the graph environment

We translate graph transversal into an instance of the basic decision problem using the recipe given in [Her21, section 5.1.1]. Briefly, states are vertices. Actions are also vertices, but only those which we can traverse to by crossing a single edge, and cost is the cost along each edge. Since the goal is to reach a certain terminal state the terminal cost is infinite if we do not reach it. What about the noise disturbances `def pW`? Since the environment is deterministic, we can simply ignore them: Just keep the code as it is.

**Problem 1** *Deterministic graph traversal*

Implement the missing functions in the `SmallGraphDP` class. You will not have to implement `def S` or `def Pw` as we will not need them for this exercise.

## 1.2 Policies

Recall that a policy is a function which takes as input the current state $x$ and time $k$ and returns the next action $u$. An example of a policy is:

```python
# graph_traversal.py
def pi_silly(x, k):
    if x == 1:
        return 2
    else:
        return 1
```

this policy is fairly silly as it just goes back and forth between vertex $1$ and $2$.

**Problem 2** *A smart policy*

Complete the function `pi_smart` to implement a policy that you think can go from node 1 to node 5 in the graph at the smallest cost.

## 1.3 Evaluating policies

Our next task is to write a function to evaluate the policy. The function should, given $\pi$ and $x_0$, compute the rest of the trajectory $x_1, x_2, \ldots, x_{N-1}, x_N$, as well as the cost of this path.

**Problem 3** *Policy rollout*

Implement the function `def policy_rollout(env, pi, x0)` to do one rollout of the policy and compute the cost $J_\pi(x_0)$ of this path

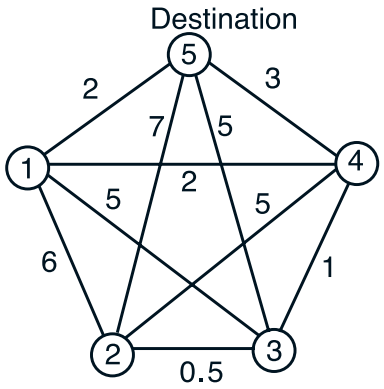



Figure 1: Figure reproduced from the lecture notes

**Info:** To match the description of the basic decision problem we still generate random noise disturbances $w$ and pass them to the transition functions which then ignores them.

Problem 4 *Putting it all together*

To put it all together, implement a policy `def pi_inc` which traverse the vertices in order, i.e. $1 \to 2 \to 3 \cdots \to 5$ and compute the cost associated with each of the three policies you just implemented.

**Info:** The script should produce output:

```
Cost of pi_silly inf
Cost of pi_inc 10.5
Cost of pi_smart 2
```

## 2 Implementing deterministic DP (`dp.py`)

The main goal today is to implement the dynamical programming algorithm. The implementation will be applied to the shortest path problem we also considered in the last exercise, and which is further detailed in [Her21, section 6.2.1].

The graph we will apply DP to is given in fig. 1. It is in this case so simple we can easily track the states of the agent. Since you already implemented the DP model last week, all we need to do this week is to implement the DP algorithm; furthermore, since the problem is deterministic, it is possible to ignore the noise terms $w$ which simplifies the algorithm significantly.

Problem 5 *Deterministic DP*

Implement the DP algorithm as described in [Her21, algorithm 1], using comments in the exercise and in the pseudo code. If you are having troubles, I strongly recommend you follow the hints in [Her21, section 6.2.1] and implement the version without a loop over $w$, i.e. `w = None`. Carefully verify the first steps of the solution agrees with the expected output; if one of the cost function terms $J_k(x)$ differ, one of us have made a mistake! When the version without $w$ works, implement the full version; it should produce the same output.

**Info:** Since the DP algorithm starts at $k = N$ and proceeds backwards you should focus on the first $k$ where the output differs from the expected output in [Her21, section 6.2.1]. Carefully follow the standard debugging recipe of using breakpoints/stepping the code to find the first time a quantity is updated wrongly, then figure out why it is updated wrongly, and fix the problem. When done, you should obtain the following output:

```
J_0(1) = 2.0, J_0(2) = 4.5, J_0(3) = 4.0, J_0(4) = 3.0, J_0(5) = 0.0
J_1(1) = 2.0, J_1(2) = 4.5, J_1(3) = 4.0, J_1(4) = 3.0, J_1(5) = 0.0
J_2(1) = 2.0, J_2(2) = 5.5, J_2(3) = 4.0, J_2(4) = 3.0, J_2(5) = 0.0
J_3(1) = 2.0, J_3(2) = 7.0, J_3(3) = 5.0, J_3(4) = 3.0, J_3(5) = 0.0
J_4(1) = inf, J_4(2) = inf, J_4(3) = inf, J_4(4) = inf, J_4(5) = 0.0
Actual cost of rollout was 4.5 which should obviously be similar to J_0[2]
Path was [2, 3, 4, 5, 5]
```

**Any** differences means you have a bad implementation and the following exercises will fail.

# 3 Implementing stochastic DP (`inventory.py`)

Once done, we can increase the complexity slightly by including the noise distribution (obviously, you might have implemented this already, but now we will test it). Recall we represent the noise distribution as a dictionary: `{..., w:pw, ...}`. The implementation should be quite similar to the above, except it should include an extra loop to account for the average over the noise parameters, see [Her21, algorithm 1]. If you are having problems debugging the code, consult [Her21, section 6.2.3] for a detailed example of the intermediate states of the algorithm.

Problem 6 *Stochastic DP*

Ensure your DP algorithm implementation in `dp.py` includes the loop over noise terms. When done, run the code and inspect the output to get a sense of how it represents the optimal policy and value function.

**Info:** The code should produce the following output

```
Inventory control optimal policy/value functions
 J_0(x_0=0) = 3.70,  J_0(x_0=1) = 2.70,  J_0(x_0=2) = 2.82
 J_1(x_1=0) = 2.50,  J_1(x_1=1) = 1.50,  J_1(x_1=2) = 1.68
 J_2(x_2=0) = 1.30,  J_2(x_2=1) = 0.30,  J_2(x_2=2) = 1.10
pi_0(x_0=0) = 1, pi_0(x_0=1) = 0, pi_0(x_0=2) = 0
```

# 4 The DP agent (`dp_agent.py`)

We are now ready to build the first serious agent, namely an agent which plan using the DP algorithm. In other words, we need both an environment, and a DP model that corresponds to that environment. The agent should then plan using the dp model to obtain an optimal policy $\pi* = \{\mu_k^*\}_{k=0}^{N-1}$, and then in step $k$ use policy function $\mu_k^*$. Since the Inventory control problem is the only one where we both have a model and an environment it will provide a good testbed. Once done, the interaction will look as follows:

```
# dp_agent.py
    env = InventoryEnvironment(N=3)
    inventory = InventoryDPModel(N=3)
    agent = DynamicalProgrammingAgent(env, model=inventory)
    stats, _ = train(env, agent, num_episodes=5000)
```

Problem 7 *Dynamical Programming Agent*

Implement the missing functionality from the DP agent. Once done, verify the (sample estimate) of the optimal value function agrees with the (exact) DP result.

**Info:** You should expect the following output:

```
Estimated reward using trained policy and MC rollouts -3.7054
Reward as computed using DP -3.6999999999999997
```

Having to specify both an agent and an environment, when it is quite apparent we can derive the environment from the agent, is obviously not ideal. Subsequent exercises will fix this problem.

# 5 Chessmatch (`chessmatch.py`)

We will now turn our attention to the chess match example [Her21, section 6.2.2], which illustrates all the steps required to apply the DP algorithm for a relatively non-trivial decision problem. To summarize, if we choose the action to play bold, we win with probability $p_w$ (and otherwise we lose) and if we choose the action to play timid, we draw with probability $p_d$ (and lose otherwise). Therefore:

- The states $x_k$ is the match score (an integer, positive if we are ahead)
- The action is binary (timid play and bold play)
- The $w_k$ is whether we won (+1), drew (+0) or lost (-1); in other words, the actions affect $P(w_k|x_k, u_k)$
- For $g_k$, even though the match technically continues, if after $N = 2$ rounds the match is tied and we enter sudden-death mode only bold play wins, and the reward in this case is therefore $p_w$. Otherwise, at $N = 2$ the reward is $1$ if we win the match and $0$ if we lose.
- Is there a reward-per-round $g_k$?

Finally (and I managed to mess this one up on the first go), the problem is formulated in terms of reward. Simply define cost as $c = -r$ where $r$ is reward and everything should be good.

Problem 8 *Chess match*

Implement the chess match class and the policy `pi_smart`. What the policy should do is to implement the optimal policy as given in the book: Play timid when ahead, otherwise play bold when the match is tied or we are behind.

When this is done the first part of the code should run and produce the output given below.

**Info:**

```
Expected reward (-cost) when starting from a match score of 0: 0.511908 (true
↪  value 0.511875)
```

# 6 DP and solving Frozen lake (`frozen_lake_dp.py`)★

The frozen lake environment is a classic problem in reinforcement learning. We already discussed the frozen lake environment quite extensively last week, so look at that exercise for further references (they are also given in the code).

Normally, the frozen lake environment is used to illustrate $Q$ or $TD$ learning, and based on these examples it might seem that finding the optimal policy is time-consuming or difficult. In this example we will simply solve the environment, and in fact we will solve all environments which uses a comparable representation.

First things first: The frozen-lake environment is not really time-limited (nothing prevents us from goofing around forever), but fortunately the environment itself imposes an artificial time limit as

```
env._max_episode_steps
```

Next, we need the reward/transition format. They are described in the data structure

```
env.env.P
```

which you need to explore a bit. Pay close attention to how it is used in
`https://github.com/openai/gym/blob/8e5a7ca3e6b4c88100a9550910dfb1a6ed8c5277/gym/envs/toy_text/discrete.py`
and you should be able to work it out. As usual, the environment defines a reward, so we have to multiply it by $-1$ to get a cost.

**Problem 9** *Solving frozen lake*

Implement the frozen-lake wrapper environment and test the first part of the code. It should output both the expected reward (as computed using DP) and by using MC sampling using the DP agent. Obviously these two should agree fairly closely.

**Info:** What I get is the following:

```
Estimated reward using trained policy and MC rollouts 0.7436
Reward as computed using DP 0.7441902878292659
```

## 6.1 Visualizing the value function

Recall the value function is the expected cost (or if we multiply by $-1$, expected reward) starting in each possible location. In other words, we can use the value function to get useful information about how good each location $(x, y)$ in the frozen-lake environment is.

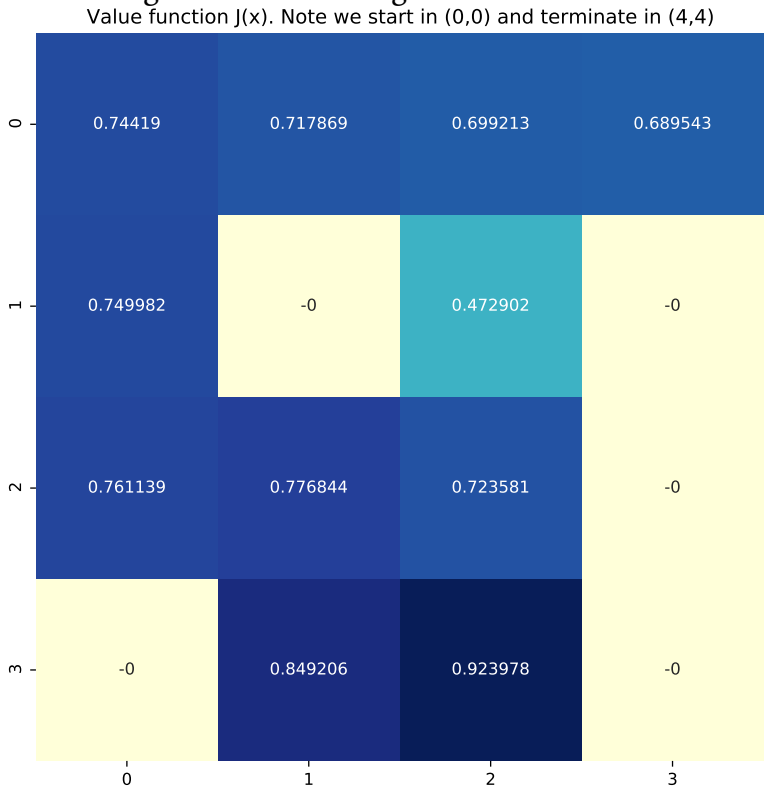
To do so, we plot the value function as a grid. This is a common technique and it is for instance done here:

`https://www.deeplearningwizard.com/deep_learning/deep_reinforcement_learning_pytorch/dynamic_programming_frozenlake/` (scroll to the very bottom) using another technique called value iteration and policy improvement we will explore in reinforcement learning. As you can tell, this is a rather lengthy procedure, so let's check our few lines of code can compete:

**Problem 10** *Frozen lake value function*

Plot the value function (expected reward, so $-J_0(x)$) on a $4x4$ grid. The result, if you look carefully, will not completely agree with `https://www.deeplearningwizard.com/deep_learning/deep_reinforcement_learning_pytorch/dynamic_programming_frozenlake/`. Did we mess up or not?

**Info:** What I get is the following:




# References

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