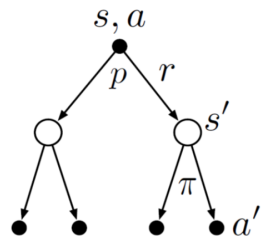
CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2024) Lawson L.S. Wong
Northeastern University Due February 9, 2024

Exercise 3: Dynamic Programming

Please remember the following policies:

- Exercises are due at **11:59 PM** Boston time (ET).
- **Submissions should be made electronically on Canvas.**
  Please ensure that your solutions for both the written and programming parts are present.
  Upload all files in a single submission, keeping the files individual (do not zip them).
  You can make as many submissions as you wish, but only the latest one will be considered, and late days will be computed based on the latest submission.
- Each exercise may be handed in up to two days late (24-hour period), penalized by 5% per day.
  Submissions later than this will not be accepted. There is no limit on the total number of late days used over the course of the semester.
- Written solutions may be handwritten or typeset. For the former, please ensure handwriting is legible. If you write your answers on paper and submit images of them, that is fine, but please put and order them correctly in a single .pdf file. One way to do this is putting them in a Word document and saving as a PDF file.
- Programming solutions should be in Python 3, either as `.py` or `.ipynb` files.
  For the latter (notebook format), please export the output as a PDF file and submit that together. To do so, first export the notebook as an HTML file (File: Save and Export Notebook as: HTML), then open the HTML file in a brower, and finally print it as a PDF file.
- You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution **and code** yourself, *and* indicate who you discussed with (if any). If you are collaborating with a large language model (LLM), acknowledge this and include your entire interaction with this system. We strongly encourage you to formulate your own answers first before consulting other students / LLMs.
- Typically, each assignment contains questions designated [CS 5180 only.], which are required for CS 5180 students. Students in CS 4180 can complete these questions for extra credit, for the same point values.
  Occasionally, there will also be [Extra credit.] questions, which can be completed by everyone. Point values for these questions depend on the depth of the extra-credit investigation.
- Contact the teaching staff if there are *extenuating* circumstances.

1. **1 point.** (RL2e 3.12, 3.13, 3.17) *Action-value function.*



q_π backup diagram

Written:

(a) Give an equation for v_π in terms of q_π and π.

(b) Give an equation for q_π in terms of v_π and the four-argument p.

(c) What is the Bellman equation for action values, that is, for q_π? It must give the action value $q_\pi(s, a)$ in terms of the action values, $q_\pi(s', a')$, of possible successors to the state–action pair (s, a).
Hint: The backup diagram above corresponds to this equation. Show the sequence of equations analogous to Equation 3.14, but for action values.

2. **1 point.** (RL2e 3.25 – 3.29) *Fun with Bellman.*
Written:

(a) Give an equation for v_* in terms of q_*.

(b) Give an equation for q_* in terms of v_* and the four-argument p.

(c) Give an equation for π_* in terms of q_*.

(d) Give an equation for π_* in terms of v_* and the four-argument p.

(e) Rewrite the four Bellman equations for the four value functions (v_π, v_*, q_π, q_*) in terms of the three-argument function p (Equation 3.4) and the two-argument function r (Equation 3.5).

3. **1 point.** (RL2e 4.4) *Fixing policy iteration.*
Written:

(a) The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is okay for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed.

(b) Is there an analogous bug in value iteration? If so, provide a fix; otherwise, explain why such a bug does not exist.

4. **1 point.** (RL2e 4.5, 4.10) *Policy iteration for action values.*
Written:

(a) How would policy iteration be defined for action values? Give a complete algorithm for computing q_*, analogous to that on page 80 for computing v_*. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book.

(b) What is the analog of the value iteration update Equation 4.10 for action values, $q_{k+1}(s, a)$?

5. **3 points.** *Implementing dynamic programming algorithms.*
Code/plot: For all algorithms, you may use any reasonable convergence threshold (e.g., $\theta = 10^{-3}$).

(a) Implement the 5×5 grid-world in Example 3.5 (i.e., implement its dynamics function, to be used in dynamic programming). Implement *iterative policy evaluation* and verify that you obtain the value function for the equiprobable random policy, given in Figure 3.2.

(b) Implement *value iteration* to output both the optimal state-value function and optimal policy for a given MDP (dynamics function). Use your implementation to verify the optimal value function and policy for the 5×5 grid-world (v_* and π_* are given in Figure 3.5).

(c) Implement *policy iteration* to output both the optimal state-value function and optimal policy for a given MDP (dynamics function). You should include the fix you proposed in Q2(a). Use your implementation to verify the optimal value function and policy for the 5×5 grid-world (v_* and π_* are given in Figure 3.5). Consider reusing your code from part (a) in policy iteration.

6. **[CS 5180 only.] 5 points.** (RL2e 4.7) *Jack's car rental problem.*

   (a) **Code/plot:** Replicate Example 4.2 and Figure 4.2. Implement the dynamics function for Jack's car rental problem, and use your policy iteration implementation (ideally from Q5) to solve for the optimal policy and value function. Reproduce the plots shown in Figure 4.2, showing the policy iterates and the final value function – your plots do not have to be in exactly the same style, but should be similar.

   (b) **Code:** Re-solve Jack's car rental problem with the following changes.
   Written: Describe how you will change the dynamics function to reflect the following changes.
   Plot: Similar to part (a), produce plots of the policy iterates and their respective value functions.
   Written: How does your final policy differ from Q6(a)? Explain why the differences make sense.

   - One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs 2, as do all cars moved in the other direction.
   - In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then a total additional cost of 4 must be incurred to use a second parking lot (independent of how many cars are kept there). (Each location has a separate overflow lot, so if both locations have > 10 cars, the total additional cost is 8.)

   These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming.

Some clarification and guidance for completing Q6:

- The description of Jack's car rental problem in Example 4.2 is detailed, but some extra details are needed to reproduce the results shown in Figure 4.2. Assume the following daily schedule for the problem:
  - 6 PM: "End of day": Close of business; this is when move actions are decided.
    From the description: "The state is the number of cars at each location at the end of the day."
  - 8 PM: Cars to be moved (if any) have arrived at their new location, including (in part b) by the employee going from location 1 to 2. The new location may have max $20 + 5$ cars after the move.
  - 8 PM – 8 AM: Overnight parking; in part b, need to pay \$4 for each location that has > 10 cars.
  - 8 AM: "Start of day": Open of business; one location may have up to 25 cars.
  - 9 AM: All requests come in at this time (before any returns).
  - 5 PM: All cars are returned at this time, i.e., a returned car cannot be rented out on the same day.
  - 5:59 PM: Excess cars (> 20) are removed at each location; each location has max 20 cars.
- Because of the somewhat larger state space and numerous request/return possibilities, a number of enhancements will likely be necessary to make dynamic programming efficient.
  - The four-argument *dynamics function* $p(s', r|s, a)$ is the most general form, but also the most inefficient form. In this case, using the three-argument *transition function* $p(s'|s, a)$ (Equation 3.4) and the two-argument *reward function* $r(s, a)$ will be much more efficient. Use the Bellman equations for v_π and v_* in terms of $p(s'|s, a)$ and $r(s, a)$, derived in Q1(e), to replace the relevant lines in policy iteration.
  - Consider pre-computing and saving your transition and reward functions so that you only have to do it once (assuming it is done correctly). This step is likely more costly than policy iteration (in our implementation, computing the dynamics function takes ~ 3 minutes, versus ~ 1 minute for all iterations of policy iteration). To save and load your pre-computed functions, consider using `pickle`.
  - Even computing the transition and reward functions might require some care to ensure that it is efficient. One observation is that, once the cars have been moved, the two locations evolve independently until the end of the next day. Therefore, we do not need to consider their transitions (and the myriad of request/return possibilities at each location) jointly.
  - In particular, consider writing an "open to close" function that computes, for a single location, the probability of ending the day with $s_{\text{end}} \in [0, 20]$ cars, given that the location started the day with $s_{\text{start}} \in [0, 20 + 5]$ cars. The function should also compute the average reward the location experiences during the day, given that the location started the day with s_{start} cars. This "open to close" function can be pre-computed for all 26 possible starting numbers of cars for each location. Then, to compute the joint dynamics between the two locations, all that is necessary is to consider the (deterministic) overnight dynamics, and then combine the appropriate "open to close" dynamics for each location.
- Useful Python tools: `pickle`, `scipy.stats.poisson`, `matplotlib.pyplot.imshow` (visualizing policy)

7. **[Extra credit.] 2 points.** (AIMA 17.6) *Proving convergence of value iteration.*
Written:

(a) Show that, for any functions f and g,

$$\left|\max_a f(a) - \max_a g(a)\right| \leq \max_a |f(a) - g(a)|$$

Hint: Consider the cases $\max_a f(a) - \max_a g(a) \geq 0$ and $\max_a f(a) - \max_a g(a) < 0$ separately.

(b) Let V_k be the k'th iterate of value iteration (assume the two-array version for simplicity, i.e., there is no in-place updating). Let $\mathcal{B}$ denote the *Bellman backup operator*, i.e., the value iteration update can be written as (in vector form):

$$V_{k+1} \leftarrow \mathcal{B}V_k$$

which is equivalent to applying the following assignment for all $s \in \mathcal{S}$:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_k(s')\right]$$

Let V_i and V_i' be any two value function vectors. Using part (a), show that:

$$\left\|\mathcal{B}V_i - \mathcal{B}V_i'\right\|_\infty \leq \gamma \left\|V_i - V_i'\right\|_\infty$$

(The ℓ_∞-norm of a vector v is the maximum absolute value of its elements: $\|v\|_\infty \triangleq \max\{|v_1|, |v_2|, \ldots, |v_n|\}$)

(c) The result from part (b) shows that the Bellman backup operator is a *contraction* by a factor of γ. Prove that value iteration always converges to the optimal value function v_*, assuming that $\gamma < 1$. To do this, first assume that the Bellman backup operator $\mathcal{B}$ has a fixed point x, i.e., $\mathcal{B}x = x$. (The proof of this is beyond this scope of this course; see Banach fixed-point theorem for details.) You should then show three things: value iteration converges to this assumed fixed point x, the fixed point is unique, and this unique fixed point is the optimal value function v_*.