CS 4100: Artificial Intelligence (Fall 2023) Lawson L.S. Wong
Northeastern University Due October 27, 2023

Exercise 3: Sequential Decision Making Under Uncertainty

Please remember the following policies:

ˆ Exercises are due at 11:59 PM Boston time (ET).

ˆ Submissions should be made electronically on Canvas, as a single .pdf file. 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.

ˆ Each exercise may be handed in up to one day late (24-hour period), penalized by 10%. 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.

ˆ 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.

ˆ You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution 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.

ˆ Contact the teaching staff if there are *extenuating* circumstances.

1. 4 points. (AIMA 16.4) In 1713, Nicolas Bernoulli stated a puzzle, now called the St. Petersburg paradox, which works as follows. You have the opportunity to play a game in which a fair coin is tossed repeatedly until it comes up heads. If the first heads appears on the *n*th toss, you win 2^n dollars.

   (a) Show that the expected monetary value of this game is infinite.

   Monetary value of the game that ends on 1 toss: 2^1\$. Expected monetary value: 0.5(heads) * 2 = 1\$

   Monetary value of the game that ends on 2 toss: 2^2\$. Expected monetary value: 0.5(tails) * 0.5(heads) * 4 = 1\$

   General: Expected monetary value of game that ends on nth toss = $1/(2^n) * 2^n = 1$.

   The expected monetary value of the game = the sum of probability * payoff for each of the infinitely many possible outcomes = 1 + 1 + 1… = infinity

   (b) How much would you, personally, pay to play the game? Give an informal explanation.

I would personally pay 2$. Even though the expected monetary value of the game is infinite, I personally would expect the game to end on the first or second toss: thus, I either break even or win two dollars.

Nicolas' cousin Daniel Bernoulli resolved the apparent paradox in 1738 by suggesting that the utility of money is measured on a logarithmic scale: $U(S_n) = a \log_2 n + b$, where S_n is the state of having $\$n$, and a and b are some constants. Use this model of utility in the following parts. You may leave your answers as unsolved expressions in terms of the constants, although you may try simplifying them using tools such as WolframAlpha.

(c) What is the expected utility of the game under this assumption?

Payoff of 1 toss = 2. Utility(2$) = $a * \log_2(2) + b = a + b$ Expected utility = ½ * (a + b)

Payoff of 2 toss = 4. Utility(4$) = a * 2 + b = 2a + b Expected utility = ¼ * (2a + b)

In general, expected utility of the game is:

EU = Σ [(a*$\log_2(2^n)$ + b) * (1/2)^n] from n=1 to ∞

Suppose a = 1, b = 0. Then the expected utility of the game would converge to around 0.60

(d) What is the expected utility if your initial wealth is $\$k$ and you had to pay $\$c$ to play the game?

With these additional conditions, the utility of a single toss changes.

Expected Utility of heads on the first round = a * $\log_2(2^1$ {reward} + k{initial} - c{cost}) + b) * ½

Utility of round n = (a * $\log_2(2^n$ + k - c) + b) * $1/2^n$

In general, expected utility of the game is:

EU = Σ [(a*$\log_2(2^n$ + k - c) + b) * (1/2)^n] from n=1 to ∞

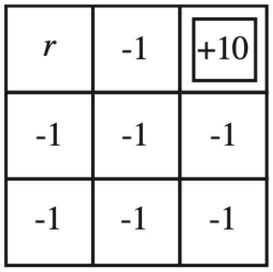
(e) Assuming the same conditions as in part (d), what is the maximum amount that it would be rational to pay to play the game?

EU = Σ [(a*$\log_2(2^n$ + k - c) + b) * (1/2)^n] from n=1 to ∞

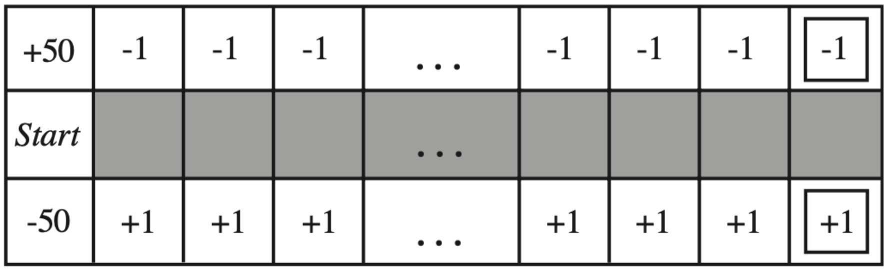
EU_not_play = U(k)

Solve for c where EU_play > EU_not_play

2. 3 points. (AIMA 17.11) Consider the 101 × 3 world shown below (right; the left sub-figure will not be used).



(a)



(b)

In the start state the agent has a choice of two deterministic actions, Up or Down, but in the other states the agent has one deterministic action, Right. Express the value of Up and Down at the start, as a function of the discount factor *γ*. For what values of the discount factor *γ* should the agent choose Up and for which Down? It is fine to leave your answer as an unsolved expression, although you can solve for the threshold value of *γ* numerically using tools such as WolframAlpha. (This simple example reflects real-world situations in which one must weigh the value of an immediate action versus the potential continual long-term consequences, such as choosing to dump pollutants into a lake.)

Utility(Up) = $50 + \Sigma\, \gamma^t * R(s_t) = 50 + \Sigma\, \gamma^t * -1 = 50 - \Sigma\, \gamma^t$ from t=2 to 101
Utility(down) = $-50 + \Sigma\, \gamma^t * R(s_t) = 50 + \Sigma\, \gamma^t * +1 = -50 + \Sigma\, \gamma^t$ from t=2 to 101

```
For all values of γ:
    If (Utility(Up) > Utility(down)):
        Optimal action = Up
    Else:
        Optimal action = Down
```

3. 5 points. In this question, we explore the relationship between search and MDPs.

   (a) Search → MDP: All deterministic search problems can be formulated as MDPs.
   Given an arbitrary search problem formulation, formulate an equivalent MDP, such that an optimal policy in the MDP will take the same sequence of actions as some optimal solution in the original search problem, if executed starting from the initial state of the search problem.
   You may assume that edge costs are all positive and finite (bounded).
   (Optional) Consider how to handle zero-cost edges, including self-loops.

Given all of the search problem parameters, an equivalent MDP can be formulated as the following:

State Space: $s \in S$(given)
Action Space: $a \in A$(given)
Transition Function: In a deterministic search problem, the transition from one state to another is deterministic, so the transition probability from one state to another, given an action, is 1 if the action takes you to the next state and 0 otherwise.

Reward: Assign a reward to each state in the MDP. In the MDP, the rewards will be assigned such that they encourage reaching the goal state. States that are closer to the goal state may have higher rewards.

Discount factor: 1 (want to find optimal path so don't discount future costs)

(b) MDP → Search: In Lecture 11, we suggested one interpretation of value iteration: one iteration of value iteration (updating all states' values using the Bellman optimality equation) is equivalent to one ply of expectimax search (with some small modifications). Starting with the pseudocode for MAX and EXP nodes (Algorithms 12 and 16 in search-notes), make appropriate modifications to show that value iteration in an MDP can be implemented as a form of expectimax search (also referred to as AND/OR search).
You may assume that the MDP reward function only depends on the state (i.e., is of form $R(s)$). (Optional) Consider how to handle general reward functions of the form $R(s, a, s')$.

```
Algorithm 12 MAX node – pseudocode version: max value(state)
    Input: state
    max value ← −∞
    best action ← None
    reward = R(state)
    for all action ∈ A(state) do
        next value, ← value(result(state, action))
        if next value > max value then
            max value ← next value
            best action ← action
    return max value + reward, best action
```

```
Algorithm 16 EXP node – pseudocode version: exp value(state)
    Input: state
    exp value ← 0
    for all next state do
        next value, ← value(next state)
        exp value ← exp value + probability(next state) * next value
return exp value * γ, None
```

```
Algorithm EX3b(Value Iteration):
Input: state, eval fn(), depth cutoff
if terminal test(state) == True then return utility(state, MAX PLAYER), None
if depth(state) >= depth cutoff then return eval fn(state, MAX PLAYER), None
if player(state) == MAX PLAYER then return max value(state)
If state == EXP_NODE then return exp value(state)
```

(c) What is the time complexity of the expectimax version of value iteration (from part (b)), versus the regular dynamic-programming version presented in Lecture 11? Provide the time complexity for both versions, along with brief justifications. Express your answers in terms of:

- $|S|$ = size of state space

ˆ $|A|$ = size of action space

ˆ b = maximum number of successors with non-zero probability, starting from any (s, a) ˆ d = number of value iteration plys being performed

Time complexity of value iteration:

O(|S| * |A| * b * d)

We perform d iterations (value iteration plies).

In each iteration, we need to consider all states (|S|) and all possible actions (|A|).

For each state-action pair, we need to consider the expected value over possible successor states, which depends on the branching factor b.

Time complexity of dynamic-programming version:

Time Complexity = O(d * |S| * |A|)

In regular dynamic programming value iteration, we assume a deterministic environment, where each action from a given state leads to a single, well-defined successor state. Therefore, there is no need to consider the branching factor because it is effectively 1 in such a deterministic setting.

(d) What accounts for the difference in time complexity between the two versions? Explain with a simple example, e.g., the 4 × 3 grid-world MDP example shown in lecture. *Hint*: Why are dynamic programming algorithms typically much more efficient?

In the case of the 4x3 grid world, the expectimax version explicitly accounts for the branching nature of uncertainty by considering multiple possible outcomes at each step(up, down, left, right), which can lead to a significantly larger number of calculations.

Dynamic programming algorithms, such as the regular dynamic-programming version, are typically more efficient because they take advantage of the optimal substructure property and eliminate redundant calculations. They don't need to explicitly consider all possible successor states at each step. Instead, they iteratively update the values of states based on the values of other states in a recursive manner, which reduces the computational complexity.

(e) When might we prefer using the expectimax version to perform value iteration? Explain with an example.

In a scenario where the environment is highly stochastic, we might prefer to use the expectimax version to perform value iteration.

Consider a board game where a player can roll a die to move. The outcome of the die roll is stochastic, and the player's position is uncertain at each turn. In this scenario, using the expectimax version of value iteration would be appropriate. Here's why:

The die roll introduces uncertainty, and the number of possible outcomes (the sides of the die) is small (b = 6 in this case).
The expectimax version can explicitly consider the different possible die roll outcomes, calculate the expected value of each, and make decisions based on this information.
It accurately models the probabilistic nature of the game, allowing the player to make informed decisions while accounting for the dice's randomness.
In such a game, the expectimax version would provide a more detailed and precise representation of the decision-making process compared to a deterministic approach. It's particularly useful when stochasticity is a significant factor in the agent's actions and outcomes.

4. 3 points. (Reinforcement Learning: An Introduction 3.14) *Bellman equation.* In Lecture 10, we introduced the Bellman equation for deterministic policies and state-only rewards:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V^\pi(s')$$

We often need to consider stochastic policies as well, which we denote by $\pi(a|s)$ instead of $\pi(s)$. $\pi(a|s)$ specifies the probability of taking action a in state s. When the policy is deterministic, exactly one action a will have probability 1, so we overload notation and refer to that action as $a = \pi(s)$. *Note*: The output types are different; $\pi(a|s)$ outputs a *probability*, whereas $\pi(s)$ outputs an *action*.

A more general version of the Bellman equation can be derived for *stochastic* policies and reward functions depending on (s, a, s'):

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^\pi(s')\right]$$

(a) Explain, in words, what the general version of the Bellman equation means. Additionally, show that it reduces to the simpler version when using deterministic policies $\pi(s)$ and state-only rewards $R(s)$.

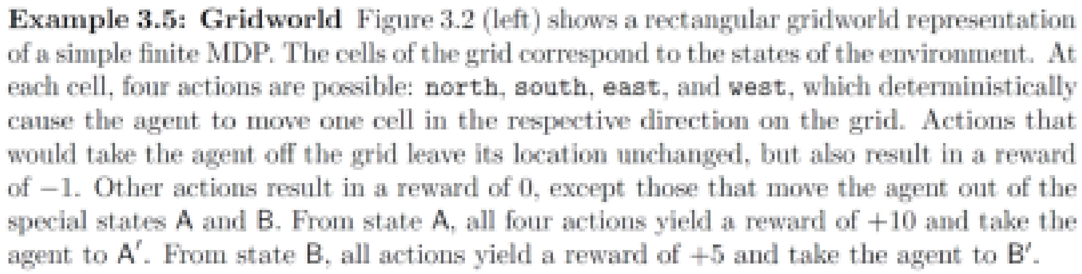
For all actions given by the policy at state s, calculate the value for all next states s' using the probability of taking action a * (probability (distribution of arriving at s' after taking action a) * general reward(dependent on s, a, and s') + discounted value of the future states).

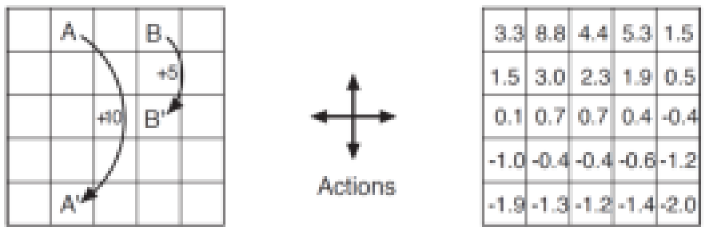
If the policy is deterministic, the first term($\Sigma_a \boldsymbol{\pi}$(a|s)) simplifies to 1, as the action given by a policy is always the same given state s).

If the reward is not general and only state dependent, it can be extracted out of the summation and the formula becomes the following Bellman equation:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s')V^{\pi}(s')$$

Now, consider the following gridworld MDP introduced in Lecture 9:

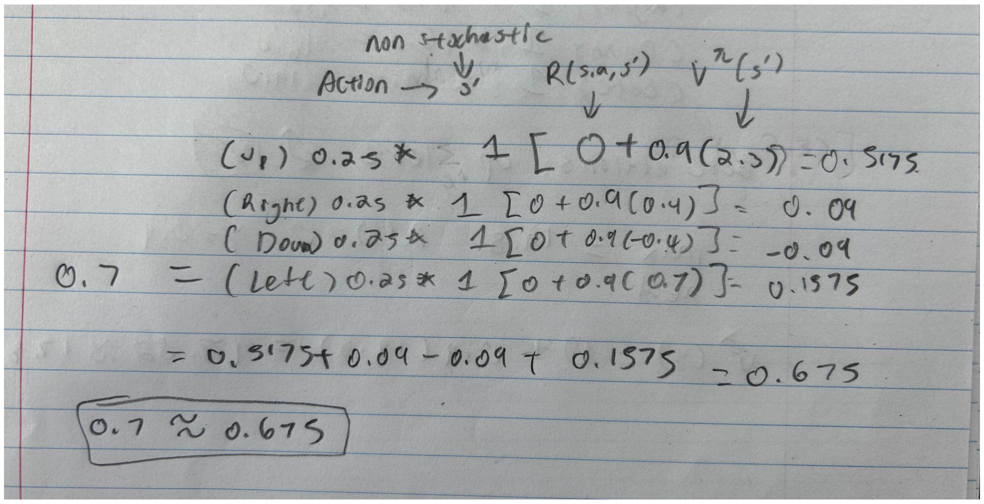

Example 3.5: Gridworld Figure 3.2 (left) shows a rectangular gridworld representation of a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: `north`, `south`, `east`, and `west`, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A, all four actions yield a reward of $+10$ and take the agent to A$'$. From state B, all actions yield a reward of $+5$ and take the agent to B$'$.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

(b) The general version of the Bellman equation must hold for each state for the value function V^{π}shown in the right figure above. Show numerically that this equation holds for the center state, valued at +0.7, with respect to its four neighboring states, valued at +2.3, +0.4, −0.4, +0.7. The discount factor is $\gamma = 0.9$. *Note*: The numbers in the figure are accurate only to one decimal place. The figure shows the value function for the equiprobable random policy, i.e., $\pi(\cdot|s) = 0.25$ for all 4 actions.


non stochastic

$\text{Action} \rightarrow \overset{\Downarrow}{s'}$ $\quad \underset{\downarrow}{R(s,a,s')} \quad \underset{\downarrow}{V^{\pi}(s')}$

(Up) $0.25 * 1[0 + 0.9(2.3)] = 0.5175$

(Right) $0.25 * 1[0 + 0.9(0.4)] = 0.09$

(Down) $0.25 * 1[0 + 0.9(-0.4)] = -0.09$

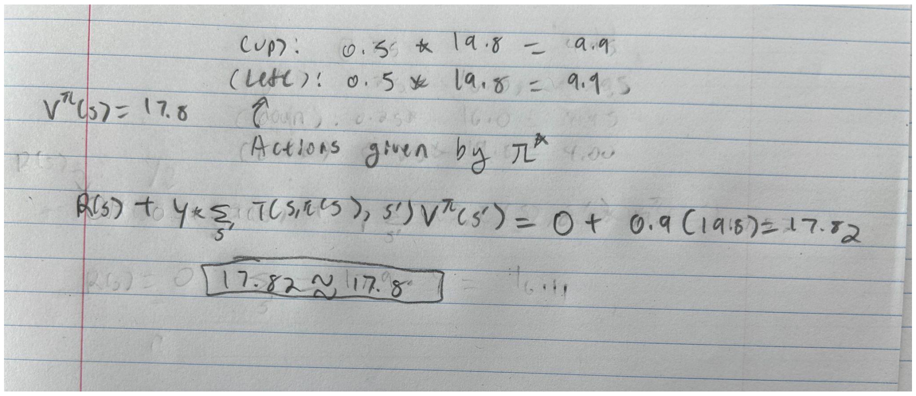
$0.7 =$ (Left) $0.25 * 1[0 + 0.9(0.7)] = 0.1575$

$= 0.5175 + 0.09 - 0.09 + 0.1575 = 0.675$

$\boxed{0.7 \approx 0.675}$

(c) The Bellman equation holds for *all* policies, including optimal policies. Consider V^{*} and π^{*}shown in the figure on the next page (middle, right respectively). Similar to the previous part, show numerically that the Bellman equation holds for the center state, valued at +17.8, with respect to its four neighboring states, for the optimal policy π^{*}shown in the figure on the next

page (right).
Also show numerically that the Bellman *optimality* equation holds for the same center state, valued at +17.8, and verify that the optimal actions at that state are indeed as shown.
Verifying Bellman Equation



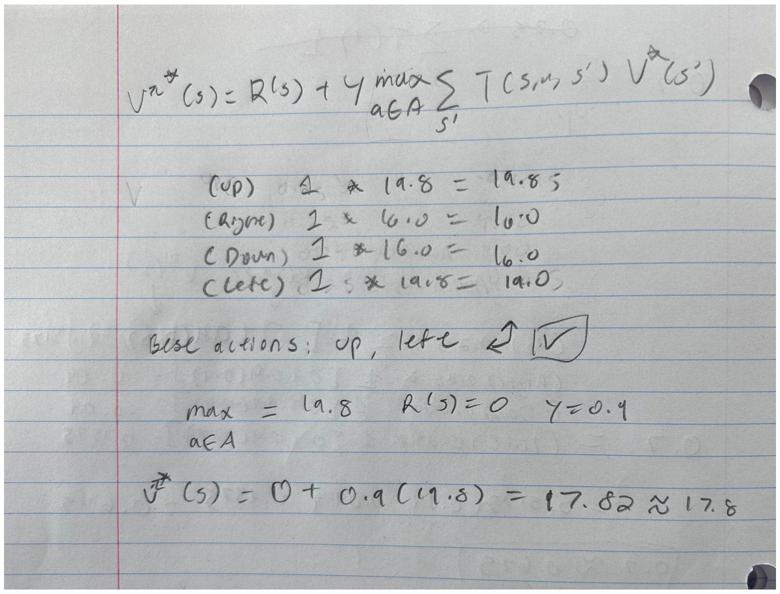
$V^{\pi}(s) = 17.8$

(up): $0.5 * 19.8 = 9.9$
(left): $0.5 * 19.8 = 9.9$
$\uparrow$
Actions given by π^*

$R(s) + \gamma * \sum_{s'} T(s, \pi(s), s') V^{\pi}(s') = 0 + 0.9(19.8) = 17.82$

$\boxed{17.82 \approx 17.8}$

Verifying Optimality Equation



$V^{\pi^*}(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} T(s, a, s') V^*(s')$

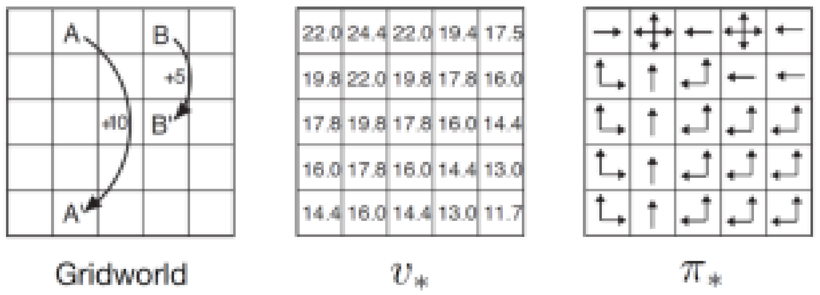
(up) $1 * 19.8 = 19.8$
(Right) $1 * 16.0 = 16.0$
(Down) $1 * 16.0 = 16.0$
(left) $1 * 19.8 = 19.0$

Best actions: up, left $\boxed{\checkmark}$

$\max_{a \in A} = 19.8 \quad R(s) = 0 \quad \gamma = 0.9$

$V^*(s) = 0 + 0.9(19.8) = 17.82 \approx 17.8$

Example 3.8: Solving the Gridworld Suppose we solve the Bellman equation for v_* for the simple grid task introduced in Example 3.5 and shown again in Figure 3.5 (left). Recall that state A is followed by a reward of +10 and transition to state A′, while state B is followed by a reward of +5 and transition to state B′. Figure 3.5 (middle) shows the optimal value function, and Figure 3.5 (right) shows the corresponding optimal policies. Where there are multiple arrows in a cell, all of the corresponding actions are optimal.



22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Figure 3.5: Optimal solutions to the gridworld example. ■

5. 6 points. *Value iteration.*
   If you need an example on formulating MDPs and value iteration, see UC Berkeley CS 188 Section 5 Q1:
   http://ai.berkeley.edu/sections/section_4_solutions_0NjcuBw70JNNGPhHucEmanLMQ1LLd4.pdf

   In a coin game, you repeatedly toss a biased coin (0.4 for head, 0.6 for tail). Each head represent 3 points and tail represents 1 point. You can either Toss or Stop if the total number of points you have tossed is no more than 7. Otherwise, you must Stop. When you Stop, your utility is equal to your total points (up to 7), or 0 if you get a total of 8 points or higher. When you Toss, you receive no utility. There is no discounting ($\gamma = 1$).

   (a) Formulate the problem as an MDP (state space, action space, transition function, reward function). Hint: The problem may be simpler to formulate using the general version of rewards: $R(s, a, s')$

State space: sum of possible points(1,2,3,4,5,6,7) and an end state *Done*

Actions are {Toss, Stop}

Transition function:

T(s, Stop, *Done*) = 1

T(s, Toss, s') = 1. 0.6 if s = {5,6} and s' = *Done*

2. 1 if s = 7 and s' = *Done*

Else:

3. 0.4 s' = s + 3

4. 0.6 s' = s + 1

4. 0 if otherwise

Reward Function:

R(s, Stop, s') = S, $S \leq 7$

R(s, a, s') = 0 otherwise

(b) Run value iteration to find the optimal value function V^* for the MDP. Show each V_k step (starting from $V_0(s) = 0$ for all states s). For a reasonable MDP formulation, this should converge in fewer than 10 steps. If you find it too tedious to do by hand, you may write a program to do this for you (and submit your code); however, there may be some benefit in seeing the calculation unfolding in front of you.

```
# Define the state space, actions, and possible points
states = [1, 2, 3, 4, 5, 6, 7, "Done"]
actions = ["Toss", "Stop"]
possible_points = [1, 2, 3, 4, 5, 6, 7]

# Define transition function T(s, a, s')
def transition_probability(s, a, s_prime):
    if a == "Stop":
        return 1
    elif a == "Toss":
        if s in [5, 6] and s_prime == "Done":
            return 0.6
        elif s == 7 and s_prime == "Done":
            return 1
        elif s + 3 == s_prime:
            return 0.4
        elif s + 1 == s_prime:
            return 0.6
        else:
            return 0

# Define the reward function R(s, a, s')
def reward(s, a, s_prime):
    if a == "Stop" and s_prime == "Done" and s in possible_points:
        return s
    else:
        return 0

# Policy Iteration
def policy_iteration(states, actions, transition_probability, reward, gamma=1, tol=1e-6):
```

```
policy = {s: "Toss" for s in possible_points}
policy["Done"] = "Stop"

while True:
    V = {s: 0 for s in states}
    while True:
        delta = 0
        for s in states:
            v = V[s]
            V[s] = sum(
                transition_probability(s, policy[s], s_prime) * (
                    reward(s, policy[s], s_prime) + gamma * V[s_prime]
                )
                for s_prime in states
            )
            delta = max(delta, abs(v - V[s]))
        if delta < tol:
            break

    policy_stable = True
    for s in possible_points:
        old_action = policy[s]
        policy[s] = max(
            actions,
            key=lambda a: sum(
                transition_probability(s, a, s_prime) * (
                    reward(s, a, s_prime) + gamma * V[s_prime]
                )
                for s_prime in states
            ),
        )
        if policy[s] != old_action:
            policy_stable = False

    if policy_stable:
        return policy, V
```

(c) Using the V^* you found, determine the optimal policy for this MDP.

Optimal policy:

State 1: Toss

State 2: Toss

State 3: Toss

State 4: Toss

State 5: Toss

State 6: Stop

State 7: Stop

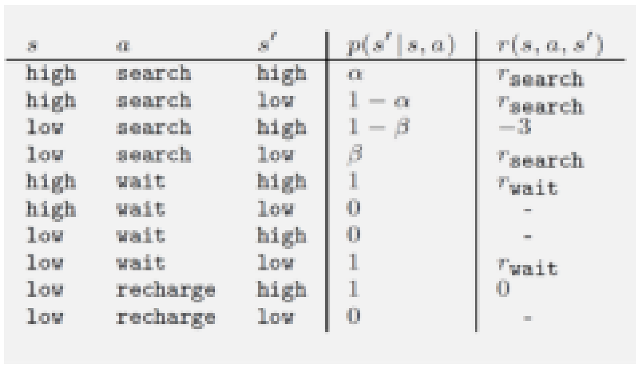
State Done: Stop

(d) Did you notice anything interesting when computing value iteration, in terms of the order in which values converged? Why did value iteration converge in the number of steps that it took?

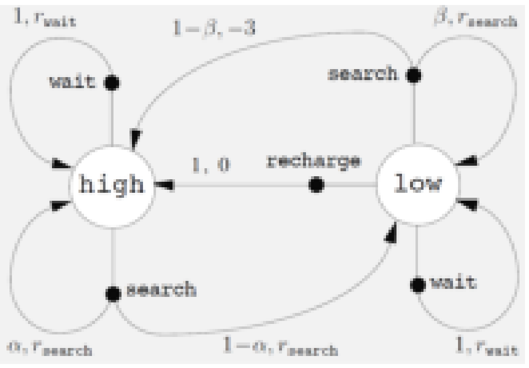
The values converged from a "bottom up" order, in which V[Done] converged first, followed by V[7], then V[6] and so on. This is because these values are "closest" to the "terminal"/reward states. The influence of the terminal states propagates first onto Done, 7, and 6 first each iteration. Because there are ~8 states, the convergence took around ~9 steps(depends on tolerance)

6. 4 points. *Solving for the value function.*

Consider a particularly simple MDP, the 2-state recycling robot described in Lecture 9. Recall that the robot is modeled with two battery states (high, low) and three actions (search, wait, recharge). The action recharge is disallowed in the high battery state. The transition and reward functions are given below:



s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-



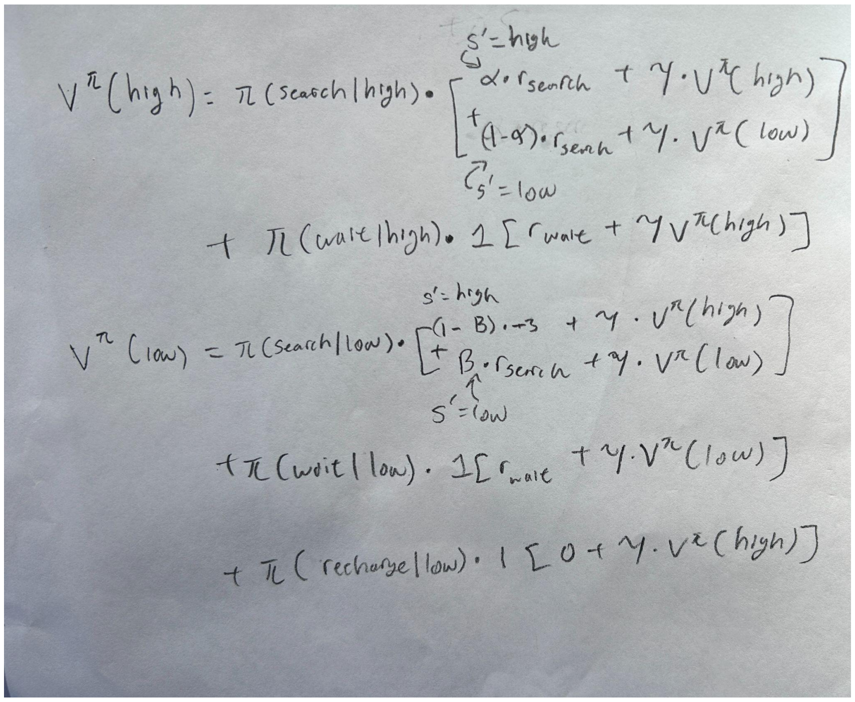
Note: For full details of this MDP, refer to Example 3.3 of *Reinforcement Learning: An Introduction* (p. 52–53; also copied on the next page for reference).

One way to solve for the value function V^π of any policy π is to write out the Bellman equation for each state, view it as a system of linear equations, and then solve for the unknowns (the value of each state).

(a) Write out the general version of the Bellman equation (see Q4) for the 2 states in the recycling robot, for an arbitrary stochastic policy $\pi(a|s)$, discount factor γ, and domain parameters $\alpha, \beta, r_{\text{search}}, r_{\text{wait}}$.

You should now have two linear equations involving two unknowns, V(high) and V(low), and various param eters. For the remainder of this question, let $\alpha = 0.8, \beta = 0.5, \gamma = 0.9, r_{\text{search}} = 10, r_{\text{wait}}$

= 3.



$$V^{\pi}(\text{high}) = \pi(\text{search}|\text{high}) \cdot \begin{bmatrix} \overset{s'=\text{high}}{\alpha \cdot r_{\text{search}}} + \gamma \cdot V^{\pi}(\text{high}) \\ + \underset{s'=\text{low}}{(1-\alpha) \cdot r_{\text{search}}} + \gamma \cdot V^{\pi}(\text{low}) \end{bmatrix}$$

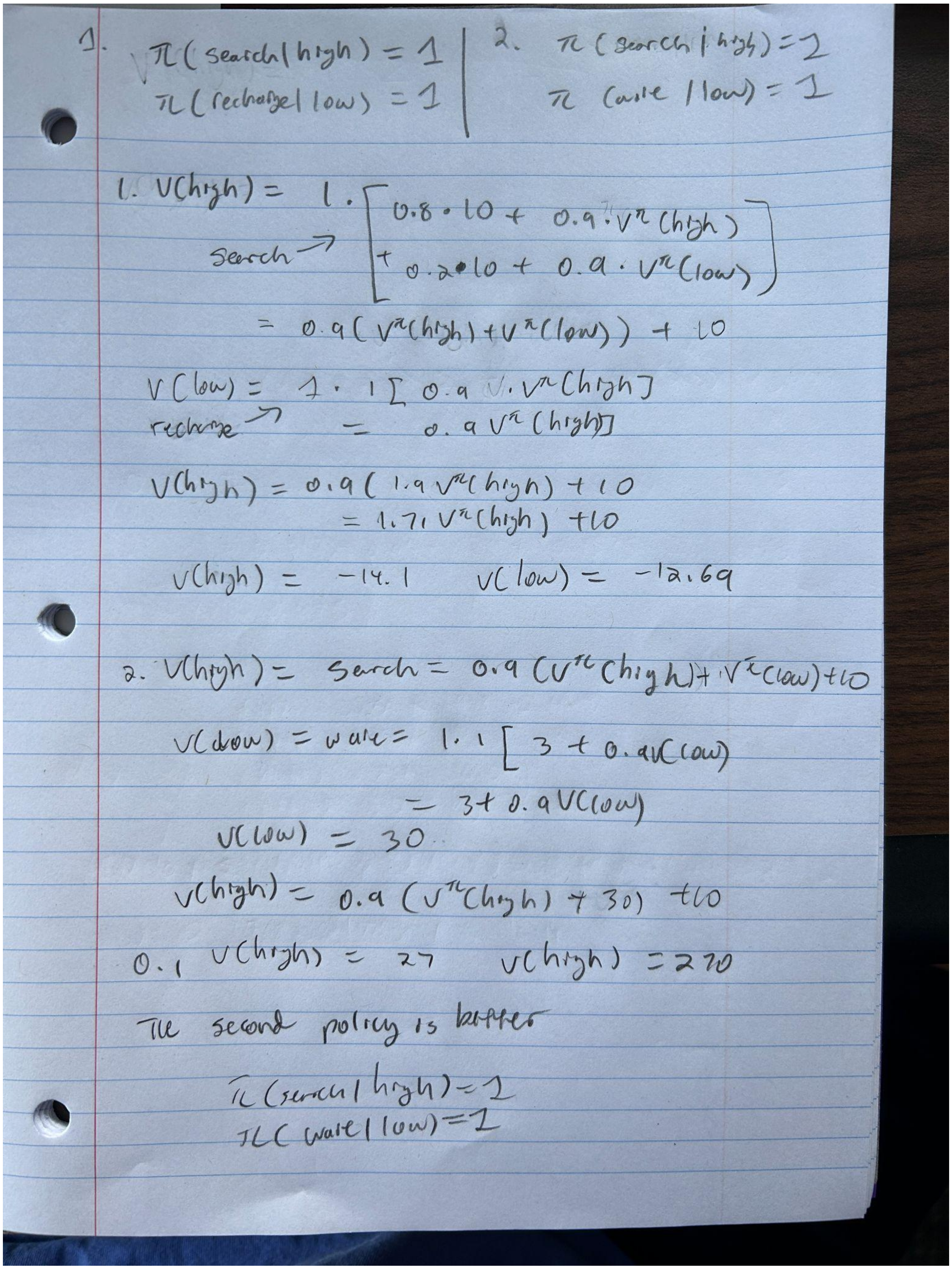
$$+ \pi(\text{wait}|\text{high}) \cdot 1 [r_{\text{wait}} + \gamma V^{\pi}(\text{high})]$$

$$V^{\pi}(\text{low}) = \pi(\text{search}|\text{low}) \cdot \begin{bmatrix} \overset{s'=\text{high}}{(1-B) \cdot -3} + \gamma \cdot V^{\pi}(\text{high}) \\ + \underset{s'=\text{low}}{B \cdot r_{\text{search}}} + \gamma \cdot V^{\pi}(\text{low}) \end{bmatrix}$$

$$+ \pi(\text{wait}|\text{low}) \cdot 1 [r_{\text{wait}} + \gamma \cdot V^{\pi}(\text{low})]$$

$$+ \pi(\text{recharge}|\text{low}) \cdot 1 [0 + \gamma \cdot V^{\pi}(\text{high})]$$

(b) There are 6 deterministic policies for this MDP, at least one of which is optimal. Pick 2 deterministic policies that you think may be optimal, and find the value functions for these policies, i.e., solve the equations for the values of V(high) and V(low).
Check that your solutions satisfy the Bellman equation. Which policy is better?



1. $\pi(\text{search}|\text{high}) = 1$

$\pi(\text{recharge}|\text{low}) = 1$

2. $\pi(\text{search}|\text{high}) = 1$

$\pi(\text{wait}|\text{low}) = 1$

1. $V(\text{high}) = 1 \cdot \left[0.8 \cdot 10 + 0.9 \cdot V^\pi(\text{high}) + 0.2 \cdot 10 + 0.9 \cdot V^\pi(\text{low}) \right]$

search $\rightarrow$

$= 0.9(V^\pi(\text{high}) + V^\pi(\text{low})) + 10$

$V(\text{low}) = 1 \cdot 1 [0.9 \cdot 1 \cdot V^\pi(\text{high})]$

recharge $\rightarrow$

$= 0.9 V^\pi(\text{high})]$

$V(\text{high}) = 0.9(1.9 V^\pi(\text{high})) + 10$

$= 1.71 V^\pi(\text{high}) + 10$

$V(\text{high}) = -14.1 \quad V(\text{low}) = -12.69$

2. $V(\text{high}) = \text{search} = 0.9(V^\pi(\text{high}) + V^\pi(\text{low})) + 10$

$V(\text{low}) = \text{wait} = 1 \cdot 1 [3 + 0.9 V(\text{low})]$

$= 3 + 0.9 V(\text{low})$

$V(\text{low}) = 30$

$V(\text{high}) = 0.9(V^\pi(\text{high}) + 30) + 10$

$0.1 \, V(\text{high}) = 27 \quad V(\text{high}) = 270$

The second policy is better

$\pi(\text{search}|\text{high}) = 1$

$\pi(\text{wait}|\text{low}) = 1$

(c) Run value iteration for 2 iterations using the MDP parameters specified above. Start from $V_0(\text{high}) = 0 = V_0(\text{low})$.

If you find it too tedious to do by hand, you may write a program to do this for you (and submit your code); however, there may be some benefit in seeing the calculation unfolding in front of you.

```
# Define the MDP parameters
alpha = 0.8
beta = 0.5
gamma = 0.9
r_search = 10
r_wait = 3

# Initialize the value functions for high and low states
V_high = 0
V_low = 0

# Perform value iteration for 2 iterations
for iteration in range(2):
    # Initialize new value functions
    new_V_high = 0
    new_V_low = 0

    # Update the value functions for the "high" state
    new_V_high = r_search + gamma * (alpha * V_high + (1 - alpha) * V_low)

    # Update the value functions for the "low" state
    new_V_low = r_wait + gamma * beta * V_low

    # Update the value functions for the "low" state when choosing "recharge" (no change)

    # Update the value functions
    V_high = new_V_high
    V_low = new_V_low

# Print the final value functions after 2 iterations
print("Value of 'high' state after 2 iterations:", V_high)
print("Value of 'low' state after 2 iterations:", V_low)
```

```
Value of 'high' state after 2 iterations: 17.740000000000002
Value of 'low' state after 2 iterations: 4.35
```

(d) What is the best policy found after 2 iterations of value iteration?
If we run value iteration to the end, the optimal action for the low state is to recharge. Comment on how this compares with the best policy found so far, and what this suggests about optimal behavior if the robot knew it had exactly two time steps / actions remaining in its lifetime.

The optimal policy:
π(high) = search

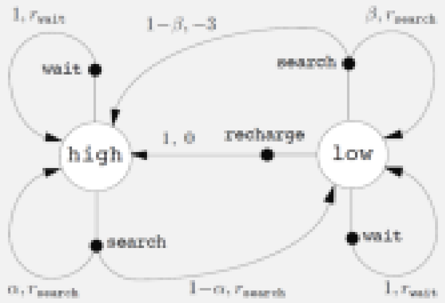
$\boldsymbol{\pi}$(low) = wait

Example 3.3 Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $\mathcal{S} = \{\texttt{high}, \texttt{low}\}$. In each state, the agent can decide whether to (1) actively **`search`** for a can for a certain period of time, (2) remain stationary and **`wait`** for someone to bring it a can, or (3) head back to its home base to **`recharge`** its battery. When the energy level is **`high`**, recharging would always be foolish, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$ and $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

The rewards are zero most of the time, but become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward). If the energy level is **`high`**, then a period of active search can always be completed without risk of depleting the battery. A period of searching that begins with a **`high`** energy level leaves the energy level **`high`** with probability α and reduces it to **`low`** with probability $1 - \alpha$. On the other hand, a period of searching undertaken when the energy level is **`low`** leaves it **`low`** with probability β and depletes the battery with probability $1 - \beta$. In the latter case, the robot must be rescued, and the battery is then recharged back to **`high`**. Each can collected by the robot counts as a unit reward, whereas a reward of -3 results whenever the robot has to be rescued. Let $r_{\texttt{search}}$ and $r_{\texttt{wait}}$, with $r_{\texttt{search}} > r_{\texttt{wait}}$, denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting respectively. Finally, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, with dynamics as indicated in the table on the left:

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-



Note that there is a row in the table for each possible combination of current state, s, action, $a \in \mathcal{A}(s)$, and next state, s'. Some transitions have zero probability of occurring, so no expected reward is specified for them. Shown on the right is another useful way of summarizing the dynamics of a finite MDP, as a *transition graph*. There are two kinds of nodes: *state nodes* and *action nodes*. There is a state node for each possible state (a large open circle labeled by the name of the state), and an action node for each state–action pair (a small solid circle labeled by the name of the action and connected by a line to the state node). Starting in state s and taking action a moves you along the line from state node s to action node (s, a). Then the environment responds with a transition to the next state's node via one of the arrows leaving action node (s, a). Each arrow corresponds to a triple (s, s', a), where s' is the next state, and we label the arrow with the transition probability, $p(s' \mid s, a)$, and the expected reward for that transition, $r(s, a, s')$. Note that the transition probabilities labeling the arrows leaving an action node always sum to 1.