Homework - Reinforcement Learning - Part A (40/100 points)

by *Todd Gureckis* and *Brenden Lake*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/ (https://brendenlake.github.io/CCM-site/)

This homework is due before midnight on March 21, 2022.

Reinforcement Learning

Reinforcement learning (RL) is a topic in machine learning and psychology/neuroscience which considers how an embodied agent should learn to make decisions in an environment in order to maximize reward. You could definitely do worse things in life than to read the classic text on RL by Sutton and Barto:

- Sutton, R.S. and Barto, A.G. (2017) Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA. [available online for free! (http://incompleteideas.net/book/the-book-2nd.html)]

The standard definition of the RL problem can be summarized with this figure:

The agent at each time point chooses an action which influences the state of the world according to the rules of the environment (e.g., spatial layout of a building or the very nature of physics). This results in a new state (s_{t+1}) and possibly a reward (r_{t+1}). The agent then receives the new state and the reward signal and updates in order to choose the next action. The goal of the agent is to maximize the reward received over the long run. In effect this approach treats life like an optimal control problem (one where the goal is to determine the best actions to take for each possible state).

The simplicity and power of this framework has made it very influential in recent years in psychology and computer science. Recently more advanced techniques for solving RL problems have been scaled to show impressive performance on complex, real-world tasks. For example, so called "deep RL" system which combine elements of deep convolutional nets and reinforcement learning algorithms can learn to play classic video games at near-human performance, simply by aiming to earn points in the game:

- Mnih, V. et al. (2015) Human-level control through deep reinforcement learning. *Nature*, 518, 529. [pdf (https://www.nature.com/articles/nature14236)]

In this homework we will explore some of the underlying principles which support these advances.

The homework is divided into two parts:

1. The first part (this notebook) explores different solution methods to the problem of behaving optimally in a *known* environment.
2. The second part (Homework-RL-B.ipynb) explores some basic issues in learning to choose effectively in an *unknown* enviroment.

References:

- Sutton, R.S. and Barto, A.G. (2017) Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.
- Gureckis, T.M. and Love, B.C. (2015) Reinforcement learning: A computational perspective. Oxford Handbook of Computational and Mathematical Psychology, Edited by Busemeyer, J.R., Townsend, J., Zheng, W., and Eidels, A., Oxford University Press, New York, NY.
- Daw, N.S. (2013) “Advanced Reinforcement Learning” Chapter in Neuroeconomics: Decision making and the brain, 2nd edition
- Niv, Y. and Schoenbaum, G. (2008) “Dialogues on prediction errors” *Trends in Cognitive Science*, 12(7), 265-72.

- Nathaniel D. Daw, John P. O'Doherty, Peter Dayan, Ben Seymour & Raymond J. Dolan (2006). Cortical

Learning and deciding in a known world

Reinforcement learning is a collection of methods and techniques for learning to make good or optimal sequential decisions. As described in the lecture, the basic definition of the RL problem (see above) is quite general and therefore there is more than one way to solve an RL problem (and even multiple ways to define what the RL problem is).

In this homework we are going to take one simple RL problem: navigation in a grid-world maze, and explore two different ways of solving this decision problem.

- The first method is going to be policy-iteration or dynamic programming.
- The second method is going to be monte-carlo simulation.

By seeing the same problem solved multiple ways, it helps to reinforce the differences between these different approaches and the features of the algorithms that are interesting from the perspective of human decision making.

The problem defintion

The problem we will consider is a grid world task. The grid is a collection of rooms. Within each room there are four possible actions (move up, down, left, right). There are also walls in the maze that the agent cannot move through (indicated in blue-grey below). There are two special states, S which is the start state, and G which is the goal state. The agent will start at location S and aims to arrive at G. When the agents moves into the G state they earn a reward of +10. If they walk off the edge of the maze, they receive a -1 reward and are returned to the S state. G is an absorbing state in the sense that you can think of the agent as never leaving that state once they arrive there.

The specific gridworld we will consider looks like this:

The goal of the agent to determine the optimal sequential decision making policy to arrive at state G.

To help you with this task we provide a simple `GridWorld()` class that makes it easy to specify parts of the gridworld environment and provides access to some of the variables you will need in constructing your solutions to the homework. In order to run the gridworld task you need to first execute the following cell:

Warning! Before running other cells in this notebook you must first successfully execute the following cell which includes some libraries.

In [210]:
```
# import the gridworld library
import numpy as np
import random
import math
import statistics
from copy import deepcopy
from IPython.display import display, Markdown, Latex, HTML
from gridworld import GridWorld, random_policy
```

The following cell sets up the grid world defined above including the spatial layout and then a python dictionary called `rewards` that determines which transitions between states result in a reward of a given magnitude.

In [211]:

```
gridworld = [
       [ 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'x', 'g'],
       [ 'o', 'x', 'x', 'o', 'x', 'x', 'o', 'x', 'o'],
       [ 'o', 'x', 'x', 'o', 'x', 'x', 'o', 'x', 'o'],
       [ 'o', 'x', 'x', 'o', 'x', 'x', 'o', 'o', 'o'],
       [ 'o', 'x', 'x', 'o', 'x', 'x', 'x', 'o', 'o'],
       [ 's', 'o', 'o', 'o', 'o', 'o', 'o', 'o', 'x']
    ] # the problem described above, 'x' is a wall, 's' is start, 'g' is
goal, and 'o' is a normal room

mygrid = GridWorld(gridworld)
mygrid.raw_print()  # print out the grid world
mygrid.index_print() # print out the indicies of each state
mygrid.coord_print() # print out the coordinates of each state (helpful
 in your code)

# define the rewards as a hash table
rewards={}

# mygrid.transitions contains all the pairwise state-state transitions a
llowed in the grid
# for each state transition intialize the reward to zero
for start_state in mygrid.transitions:
    for action in mygrid.transitions[start_state].keys():
        next_state = mygrid.transitions[start_state][action]
        rewards[str([start_state, action, next_state])] = 0.0

# now set the reward for moving up into state 8 (the goal state) to +10
rewards[str([17, 'up', 8])] = 10

# now set the penalty for walking off the edge of the grid and returning
to state 45 (the start state)
for i in [0,1,2,3,4,5,6,7]:
    rewards[str([i, 'up', 45])] = -1
for i in [0,9,18,27,36,45]:
    rewards[str([i, 'left', 45])] = -1
for i in [45,46,47,48,49,50,51,52,53]:
    rewards[str([i, 'down', 45])] = -1
for i in [8,17,26,35,44,53]:
    rewards[str([i, 'right', 45])] = -1
```

Welcome to your new Grid World!

Raw World Layout

```
o  o  o  o  o  o  o  x  g
o  x  x  o  x  x  o  x  o
o  x  x  o  x  x  o  x  o
o  x  x  o  x  x  o  o  o
o  x  x  o  x  x  x  o  o
s  o  o  o  o  o  o  o  x
```

Indexes of each grid location as an id number

```
 0   1   2   3   4   5   6   7   8
 9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26
27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44
45  46  47  48  49  50  51  52  53
```

Indexes of each grid location as a tuple

```
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7) (2,8)
(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7) (3,8)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,8)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7) (5,8)
```

In [212]:
```
#rewards[9, 'right', 8]
```

Notice that the above printouts show the grid but also an array of the indexes and coordinated of each location on the grid. You will need these to help you analyze your solution to the homework so it can be frequently helpful to refer back to these outputs.

In order to solve this problem using dynamic programming the agent needs to maintain two key representations. One is the value of each state under the current policy, V^{π}, and the other is the policy $\pi(s, a)$. The following cell initializes a new value table and a new random policy and uses functions provided in `GridWorld` to print these out in the notebook in a friendly way.

In [213]:
```
value_table=np.zeros((mygrid.nrows,mygrid.ncols))
display(Markdown("**Current value table for each state**"))
mygrid.pretty_print_table(value_table)

policy_table = [[random_policy() for i in range(mygrid.ncols)] for j in
range(mygrid.nrows)]
display(Markdown("**Current (randomized) policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

Current value table for each state

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Current (randomized) policy

↑	↓	↑	←	→	↓	←	■	↑
↑	■	■	←	■	■	←	■	→
↓	■	■	↑	■	■	→	■	←
↑	■	■	←	■	■	←	↓	↑
↑	■	■	←	■	■	■	←	↑
↑	→	←	↑	←	←	↑	↓	■

In [214]:
```
policy_table[0] # shows the first horizontal line
```

Out[214]:
```
[{'up': 1.0, 'right': 0.0, 'down': 0.0, 'left': 0.0},
 {'up': 0.0, 'right': 0.0, 'down': 1.0, 'left': 0.0},
 {'up': 1.0, 'right': 0.0, 'down': 0.0, 'left': 0.0},
 {'up': 0.0, 'right': 0.0, 'down': 0.0, 'left': 1.0},
 {'up': 0.0, 'right': 1.0, 'down': 0.0, 'left': 0.0},
 {'up': 0.0, 'right': 0.0, 'down': 1.0, 'left': 0.0},
 {'up': 0.0, 'right': 0.0, 'down': 0.0, 'left': 1.0},
 {'up': 0.0, 'right': 0.0, 'down': 0.0, 'left': 1.0},
 {'up': 1.0, 'right': 0.0, 'down': 0.0, 'left': 0.0}]
```

Note how the current policy is random with the different arrows within each state pointing in different, sometimes opposing directions. Your goal is to solve for the best way to orient those arrows.

Dynamic Programming via Policy Iteration

Problem 1 (15 points)

Remember that the Bellman equation that recursively relates the value of any state to any other state is like this: $V^{\pi}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')]$ Your job in this first exercise is to set up a dynamic programming solution to the provided gridworld problem. Your should implement two steps. The first is policy evaluation which means given a policy (in `policy_table`) update the `value_table` to be consistent with that policy. Your algorithm should do this by visiting each state in a random order and updating it's value in the `value_table` (this is known as asychronous update since you are changing the values in-place). The next step is policy improvement where you change the policy to maximize expected long-run reward in each state by adjusting which actions you should take (this means changing the values in `policy_table`). We will only consider deterministic policies in this case. Thus your algorithm should always choose one action to take in each state even if two actions are similarly valued. The algorithm you write should iterate sequentially between policy evaluate and (greedy) policy improvement for at least 2000 iterations.

(Figure from Sutton and Barto (http://www.incompleteideas.net/book/the-book-2nd.html) text) To gain some intuition about how preferences for the future impact the resulting policies, run your algorithm twice, once with γ set to zero (as in lecture) and another with γ set to 0.9 and output the resulting policy and value table using `mygrid.pretty_print_policy_table()` and `mygrid.pretty_print_table()`.

Info

The `GridWorld` class provides some helpful functions that you will need in your solution. The following code describes these features

Only some states are "valid" i.e., are not walls. `mygrid.valid_states` is a python dictionary containing those states. The keys of this dictionary are id numbers for each state (see the output of `mygrid.index_print()`) and the values are coordinates (see the output of `mygrid.coord_print()`). Your algorithm will want to iterate over this list to update the value of each "valid" state.

In [215]:
```
mygrid.valid_states  # output the indexes and coordinates of the valid tates
```

Out[215]:
```
{0: (0, 0),
 1: (0, 1),
 2: (0, 2),
 3: (0, 3),
 4: (0, 4),
 5: (0, 5),
 6: (0, 6),
 8: (0, 8),
 9: (1, 0),
 12: (1, 3),
 15: (1, 6),
 17: (1, 8),
 18: (2, 0),
 21: (2, 3),
 24: (2, 6),
 26: (2, 8),
 27: (3, 0),
 30: (3, 3),
 33: (3, 6),
 34: (3, 7),
 35: (3, 8),
 36: (4, 0),
 39: (4, 3),
 43: (4, 7),
 44: (4, 8),
 45: (5, 0),
 46: (5, 1),
 47: (5, 2),
 48: (5, 3),
 49: (5, 4),
 50: (5, 5),
 51: (5, 6),
 52: (5, 7)}
```

As the previous command makes clear, there are two ways of referencing a state: by its id number or by its coordinates. Two functions let you swap between those: - `mygrid.index_to_coord(index)` converts a index (e.g., 1-100) to a coordinate (i,j) - `mygrid.coord_to_index(coord)` takes a tuples representing the coordinate (i,j) and return the index (e.g., 1-100) Both the value table (`value_table`) and policy table (`policy_table`) are indexed using coordinates.

A key variable for your algorithm is $\mathcal{P}^a_{ss'}$, which is the probability of reaching state s' when in state s and taking action a. We assume that the world is deterministic here so these probabilities are always 1.0. However, some states do not lead to immediately adjacent cells but instead return to the start state (e.g., walking off the edge of the grid). `mygrid.transitions` contains a nested hash table that contains this information for your gridworld. For example consider state 2:

In [216]:
```
state = 2
mygrid.transitions[state]
```

Out[216]:
```
{'up': 45, 'right': 3, 'down': 2, 'left': 1}
```

The output of the above command is a python dictionary showing what next state you will arrive at if you chose the given actions. Thus `mygrid.transitions[2]['down']` would return state id 2 because you will hit the wall and thus not change state. Whereas `mygrid.transitions[2]['left']` will move to state 1. The `mygrid.transitions` dictionary thus provides all the information necessary to represent $P^a_{ss'}$. The world is deterministic so taking an action in a given state will always move the agent to the next corresponding state with probability 1.

The next variable you will need is the reward function. Rewards are delivered anytime the agent makes a transition from one state to another using a particular action. Thus this variable is written $\mathcal{R}^a_{ss'}$ in the equation above. You can access this programmatically using the python dictionary `rewards` which we ourselves defined above. The `rewards` dictionary defines the reward for taking a particular action in a particular state and then arriving at a new state s'. To look up the reward for a particular $$ triplet you create a list with these variables in index format, convert it to a string, and look it up in the dictionary. For example the reward for being in state 17, choosing up, and then arriving in state 8 is:

In [217]:
```
state = 17
next_state = 8
action = "up"
rewards[str([state, action, next_state])]
str([state, action, next_state])
```

Out[217]:
```
"[17, 'up', 8]"
```

In [218]:
```
value_table
```

Out[218]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

This should be the required ingredients to solve both the policy evaluation and policy improvement functions that you will need to write. If you need further information you can read the `GridWorld` class directly in gridworld.py (gridworld.py).

Your solution:

Implement the two major steps of your algorithm as the following two functions. Then write code that iterates between them for the specified number of steps and inspect the final solution. **Some scaffolding code has been provided for you so all you have to implement is the sections noted in the comments**

In [219]:

```
def policy_evaluate(mygrid, value_table, policy_table, GAMMA):
    valid_states = list(mygrid.valid_states.keys())
    random.shuffle(valid_states)

    for state in valid_states:
        sx,sy = mygrid.index_to_coord(state)
        new_value = 0.0
        for action in mygrid.transitions[state].keys():
            # PART 1: HOMEWORK: compute what the new value of the give s
tate should be
            # here!!!  This is your homework problem**********************
**********
            # getting the x, y coordinates for next state (s')
            x, y = mygrid.index_to_coord(mygrid.transitions[state][actio
n])
            policy = policy_table[sx][sy][action] # policy π(s,a)
            new_value += policy * (1) * (rewards[str([state, action, int
(mygrid.transitions[state][action])])] + GAMMA * value_table[x][y])
            #assert False, "Implement your solution here"

        value_table[sx][sy] = new_value

# this is a helper function that will take a set of q-values and convert
them into a greedy decision strategy
def be_greedy(q_values):
    if len(q_values)==0:
        return {}

    keys = list(q_values.keys())
    vals = [q_values[i] for i in keys]
    maxqs = [i for i,x in enumerate(vals) if x==max(vals)]
    if len(maxqs)>1:
        pos = random.choice(maxqs)
    else:
        pos = maxqs[0]
    policy = deepcopy(q_values)
    for i in policy.keys():
        policy[i]=0.0
    policy[keys[pos]]=1.0
    return policy

def policy_improve(mygrid, value_table, policy_table, GAMMA):
    # for each state
    valid_states = list(mygrid.valid_states.keys())

    for state in valid_states:
        # compute the Q-values for each action
        q_values = {}
        qval = 0 #my line
        for action in mygrid.transitions[state].keys():
            # update the q-values here for each action here
            # and store them in a variable called qval
            # PROBLEM 1: HOMEWORK: Compute the qval here***************
*****
            x, y = mygrid.index_to_coord(mygrid.transitions[state][actio
n])
```

```
            qval = (1) * (rewards[str([state, action, int(mygrid.transit
ions[state][action])])] + GAMMA * value_table[x][y])

            #assert False, "Implement your solution here"
            q_values[action]=qval
        newpol = be_greedy(q_values) # take this dictionary and convert
 into a greedy policy
        # then update the policy table printing to allow more complex po
licies
        sx,sy = mygrid.index_to_coord(state)
        for action in mygrid.transitions[state].keys():
            policy_table[sx][sy][action] = newpol[action]
```

The following code actually runs the policy iteration algorithm cycles. You should play with the parameters of this simulation until you are sure that your algorithm has converged and that you understand how the various parameters influence the obtained solutions.

In [221]:
```
mygrid.pretty_print_table(value_table)
mygrid.pretty_print_policy_table(policy_table)
```

```
0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

↑  ↓  ↑  ←  →  ↓  ←  ■  ↑

↑  ■  ■  ←  ■  ■  ←  ■  →

↓  ■  ■  ↑  ■  ■  →  ■  ←

↑  ■  ■  ←  ■  ■  ←  ↓  ↑

↑  ■  ■  ←  ■  ■  ■  ←  ↑

↑  →  ←  ↑  ←  ←  ↑  ↓  ■
```

In [222]:
```
mygrid.pretty_print_table(value_table)
mygrid.pretty_print_policy_table(policy_table)

GAMMA=0.9# run your algorithm from
          # above with different settings of GAMMA
          # (Specifically 0 and 0.9 to see how the resulting value func
tion and policy changein)
for i in range(2000):
    policy_evaluate(mygrid, value_table, policy_table, GAMMA)
    policy_improve(mygrid, value_table, policy_table, GAMMA)

mygrid.pretty_print_table(value_table)
mygrid.pretty_print_policy_table(policy_table)
```

```
0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0

0  0  0  0  0  0  0  0  0
```

```
↑  ↓  ↑  ←  →  ↓  ←  ■  ↑

↑  ■  ■  ←  ■  ■  ←  ■  →

↓  ■  ■  ↑  ■  ■  →  ■  ←

↑  ■  ■  ←  ■  ■  ←  ↓  ↑

↑  ■  ■  ←  ■  ■  ■  ←  ↑

↑  →  ←  ↑  ←  ←  ↑  ↓  ■
```

2.54187	2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	0	0
2.28768	0	0	3.13811	0	0	5.31441	0	10
2.05891	0	0	2.8243	0	0	5.9049	0	9
2.28768	0	0	3.13811	0	0	6.561	7.29	8.1
2.54187	0	0	3.48678	0	0	0	6.561	7.29
2.8243	3.13811	3.48678	3.8742	4.30467	4.78297	5.31441	5.9049	0

```
→  →  →  →  →  →  ↓  ■  ↑

↑  ■  ■  ↑  ■  ■  ↓  ■  ↑

↑  ■  ■  ↑  ■  ■  ↓  ■  ↑

↓  ■  ■  ↓  ■  ■  →  →  ↑

↓  ■  ■  ↓  ■  ■  ■  ↑  ↑

→  →  →  →  →  →  →  ↑  ■
```

Your final policy should look something like this for $\gamma = 0.0$:

and like this for $\gamma = 0.9$

Although note that your solution may not be identical because we are doing greedy action selection and randomly choosing one perferred action in the case that there are ties (partly because it is harder to display stochastic policies as a grid). However, if you consider the structure of this particular gridworld there is always one best move.

First Visit Monte-Carlo

In the previous exercise you solved the sequential decision making problem using policy iteration. However, you relied heavily on the information provided by the `GridWorld()` class, especially $\mathcal{P}^a_{ss'}$ (`mygrid.transitions`) and $\mathcal{R}^a_{ss'}$ (`rewards`). These values are not commonly known when an agent faces an environment. In this step of the homework you will solve the same grid world problem this time using Monte-Carlo.

Monte Carlo methods (https://en.wikipedia.org/wiki/Monte_Carlo_method) are ones where stochastic samples are drawn from a problem space and aggregated to estimate quantities of interest. In this case we want to average the expected rewards available from a state going forward. Thus, we will use Monte Carlo methods to estimate the value of particular actions or states.

The specific Monte-Carlo algorithm you should use is known as First-Visit Monte Carlo (described in lecture). According to this algorithm, each time you first visit a state (or state-action pair) you record the rewards received until the end of an episode. You do this many times and then average together the rewards received to estimate the value of the state or action.

Then, as you did in problem 1, you adjust your policy to become greedy with respect to the values you have estimated.

There are two significant conceptual changes in applying Monte-Carlo to the gridworld problem. First is that rather than estimate the value of each state $V^\pi(s)$ under the current policy π, it makes more sense to estimate the value of each state-action pair, $Q^\pi(s,a)$, directly. The reason is that in your previous solution, in order to determine the optimal policy, you likely had to know $\mathcal{P}^a_{ss'}$ to determine which action to perform and which state it would lead to. Since we are trying to avoid accessing any explicit knowledge about the probabilities and rewards we cannot use this variable in our solution. Thus, average the returns following the first visit to a particular action.

The second is what policy we should use for running our Monte Carlo updates. If we randomly initialize the policy as we did above and then run it forward it is very easy for the runs to get caught in cycles and loops that never visit many of the states or ever encounters any rewards. Thus, we will want to include some randomness in our simulations so that they have a non-zero probability of choosing different actions. We will consider this issue in more detail in Part B of the homework. For now use the ϵ-greedy algorithm which chooses the currently "best" action with probability $1-\epsilon$ and otherwise chooses randomly.

In addition, we will utilize the concept of **exploring starts**. Even though we designated one state as the "Start state" it can help the monte carlo algorithm explore more efficiently if we start the episodes from random starting locations. The reason is that early on the policy might have loops and other inconistencies which mean some states are rarely sampled, if at all. Exploring starts (when feasible) can help the algorithm avoid these local minima.

Problem 2 (15 points)

In this exercise you should solve the problem introduced at the start of this notebook using Monte Carlo methods. The pseudo code for your algorithm is described here: ``` Initialize, for all $s \in S, a \in A(s)$: $Q(s,a)$ <- arbitrary $\pi(s)$ <- arbitrary $Returns(s,a)$ <- empty list Repeat many times: a) Generate an episode using π with \epislon probability of choosing an action at random b) For each pair s, a appearing in the episode R <- return following the first occurence of s, a Append R to $Returns(s,a)$ $Q(s,a)$ <- discounted_average(Returns(s,a)) c) For eash s in episode: $\pi(s)$ <- arg max_a Q(s,a) ``` When you compute the average returns you should weight them by them by γ so that they reflect the discount rates described above. Run your algorithm for both $\gamma = 0.0$ and $\gamma = 0.9$ and compare the resulting `policy_table` to the one you obtained in Problem 1. They should work out to the same optimal policies, obtained using a quite different method, and one that in particular doesn't need an explicit model of the environment. Keep in mind that in cases where there are two equally good actions which one is selected and shown in your policy table is arbitrary. If correctly implemeneted the dynamic programming soluion then you should be aware of when these cases happen. It is thus fine if the policies you get from monte-carlo and dynamic programming are not **identical** but are still **correct**.

There are a couple of hints that you will need to implement your solution which are provided by the `GridWorld` class. The first is that you will still need to use the `rewards` dictionary from your solution to Problem 1 to compute when the rewards are delievered. However instead of consulting this function arbitrarily you are using it just to sample the rewards when the correct event happens in your Monte Carlo simulation. Second, you will need to find out what state you are in after taking an action in a given state. The one-step transition dynamics of the gridworld can be simulated from the GridWorld class. For example, to determine the state you would be in if you were in state 45 (the start state) and chose the action "up", "down", "left", or "right" is given by:

In [223]:
```
[mygrid.up(45), mygrid.down(45), mygrid.left(45), mygrid.right(45)]
```

Out[223]:
```
[36, 45, 45, 46]
```

Note that in this example, down and left walk off the edge of the environment and thus return the agent to the start state.

The following two functions implement the epsilon-greedy Monte Carlo sample from your gridworld task using a recursive function. Although this is provided to you for free, you should try to understand the logic of these functions.

In [224]:
```
def epsilon_greedy(actions, epsilon):
    if random.random() < epsilon:
        return random.choice(list(actions.keys()))
    else:
        if actions['up']==1.0:
            return 'up'
        elif actions['right']==1.0:
            return 'right'
        elif actions['down']==1.0:
            return 'down'
        elif actions['left']==1.0:
            return 'left'

#recursively sample state-action transitions using epsilon greedy algori
thm with a maximum recursion depth of 100.
def mc_episode(current_state, epsilon, goal_state, policy_table, depth=0
, max_depth=100):
    if current_state!=goal_state and depth<max_depth:
        sx, sy = mygrid.index_to_coord(current_state)
        action = epsilon_greedy(policy_table[sx][sy],epsilon)
        if action == 'up':
            new_state = mygrid.up(current_state)
        elif action == 'right':
            new_state = mygrid.right(current_state)
        elif action == 'down':
            new_state = mygrid.down(current_state)
        elif action == 'left':
            new_state = mygrid.left(current_state)
        r = rewards[str([current_state,action,new_state])]
        return [[r, current_state, action]] + mc_episode(new_state, epsi
lon, goal_state, policy_table, depth=depth+1, max_depth=max_depth)
    else:
        return []
```

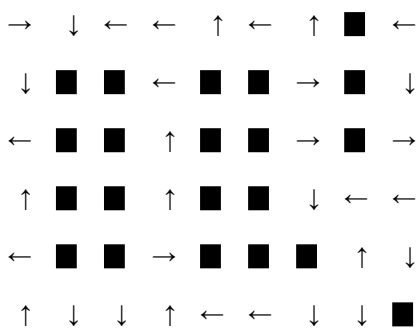
Some initial data structures for managing the q-values, policy, and returns have been defined for you here:

In [263]:

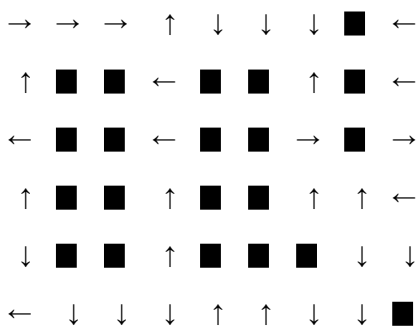
```
from collections import defaultdict

starting_state = 45
goal_state = 8 # terminate the MC roll out when you get to this state
GAMMA=0.9
EPSILON = 0.2 # more exploration is often better
ITERATIONS = 50000 # this may need to be 100,000 or more!
PRINT_EVERY = 1000 # how often to print out our progress
random.seed(5000) # try multiple random seed to verify your code works

# set up initial data strucutres that might be useful for you
# q(s,a)
def init_q_values(init):
    qvals = {"up": init, "right": init, "down": init, "left": init}
    return qvals
INIT_VAL = -99999.0  #initialize unsampled q values to a very small numb
er (pessimitic intiaizliation)
q_value_table = [[init_q_values(INIT_VAL) for i in range(mygrid.ncols)]
for j in range(mygrid.nrows)]

# pi
policy_table = [[random_policy() for i in range(mygrid.ncols)] for j in
range(mygrid.nrows)]
display(Markdown("**Initial (randomized) policy**"))
mygrid.pretty_print_policy_table(policy_table)

# dictionary for returns, can be filled in as more info is encountered
#returns = {}

# using default dictionary so we can append right away without initializ
ing
returns = defaultdict(list)

for i in range(ITERATIONS):  # you probably need to take many, many step
s here and it make take some time to run
    # instead of always starting at the start state, this algorithm will
use the concept of an
    # "exploring start" so that it starts in a random valid state
    # this can help a lot
    ss = random.choice(list(mygrid.valid_states.keys()))  # select and e
xploring start state
    episode = mc_episode(ss, EPSILON, goal_state, policy_table) # mc_epi
sode creates the random walk from state to state in the form [reward for
action, current state, action]

    visited = {}
    for idx in range(len(episode)):
        item = episode[idx]
        qkey = str((item[1],item[2]))
        if qkey not in visited:
            # PROBLEM 2- update the returns dictionary to include the di
scounted average rewards according to
            # the first visit algorithm
            #assert False, "Implement your solution here"
            # how do i find the return?
            visited[qkey] = 0
```

```
            R = 0 # R = discounted return = sum(GAMMA*rewards)

            # calculating sum of discounted return from subsequent step
s' rewards
            GAMMA_exp = 0
            for index in range(idx,len(episode)):
                R += GAMMA ** (GAMMA_exp) * episode[index][0] # updating
discounted return R
                GAMMA_exp += 1

            # append the discounted return value to the (s,a) pair
            returns[qkey].append(R)

    # update q-value-table
    for ret in returns.keys():
        s,a = eval(ret)
        sx, sy = mygrid.index_to_coord(s)

        #assert False, "Implement your solution here"
        # getting the average return from the return dictionary and assi
gning to q_value_table
        avg_return = sum(returns[ret])/len(returns[ret])
        q_value_table[sx][sy][a] = avg_return # PROBLEM 2- UPDATE your a
verage returns here, depends on how you implement the above


    # improve policy
    for sx in range(len(q_value_table)):
        for sy in range(len(q_value_table[sx])):
            policy_table[sx][sy] = be_greedy(q_value_table[sx][sy])

    if i%PRINT_EVERY==0:
        display(Markdown(f"**Improved policy interation {i}**"))
        mygrid.pretty_print_policy_table(policy_table)

display(Markdown("**Improved policy**"))
mygrid.pretty_print_policy_table(policy_table)
```

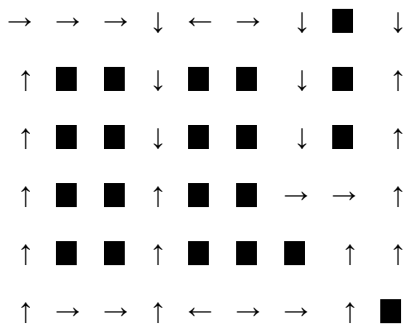
Initial (randomized) policy



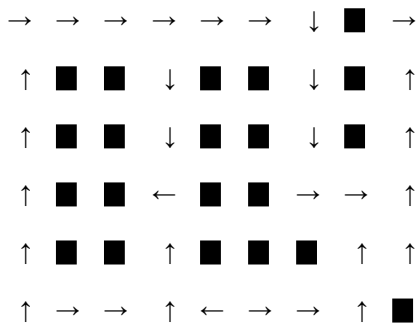
Improved policy interation 0



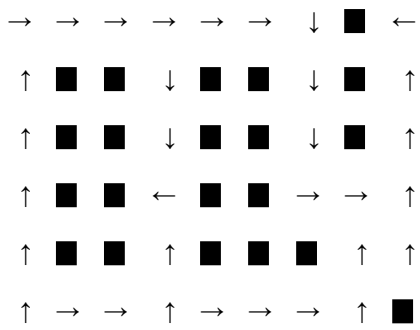
Improved policy interation 1000



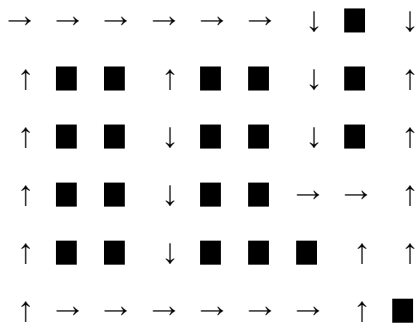
Improved policy interation 2000



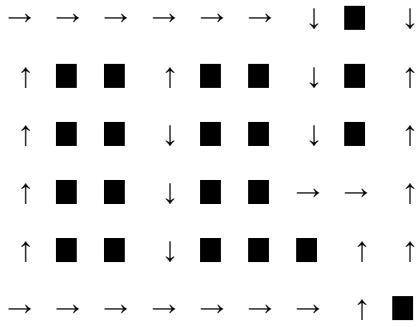
Improved policy interation 3000



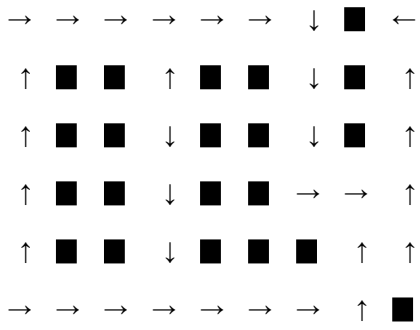
Improved policy interation 4000



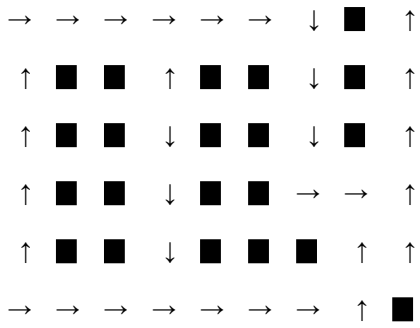
Improved policy interation 5000



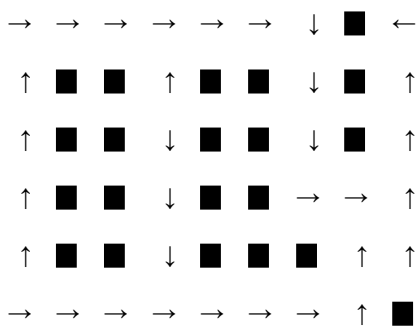
Improved policy interation 6000



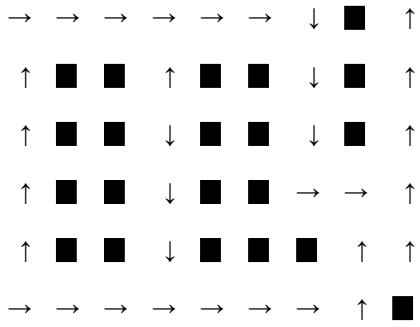
Improved policy interation 7000



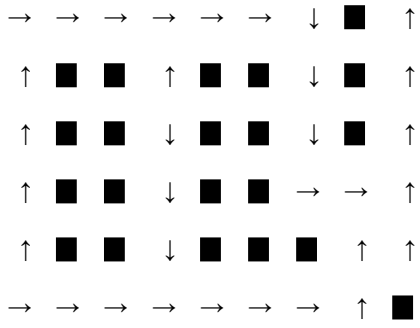
Improved policy interation 8000



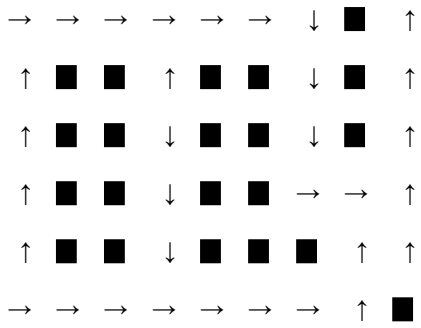
Improved policy interation 9000



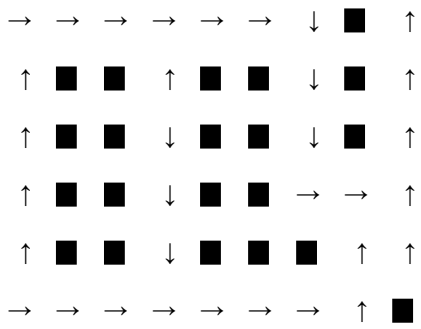
Improved policy interation 10000



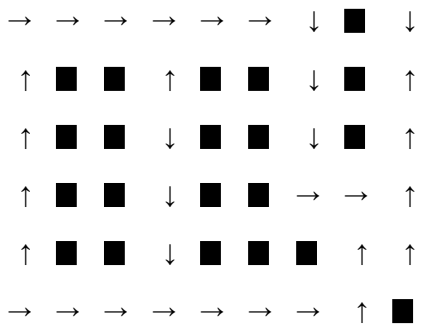
Improved policy interation 11000



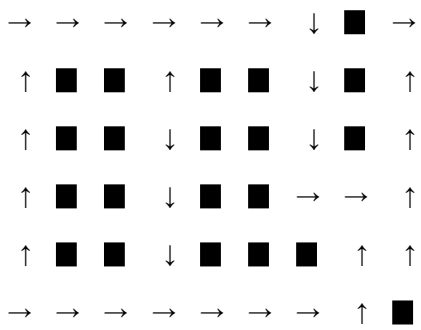
Improved policy interation 12000



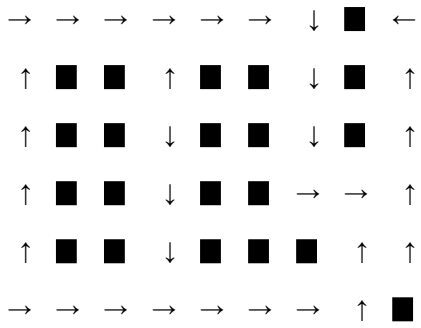
Improved policy interation 13000



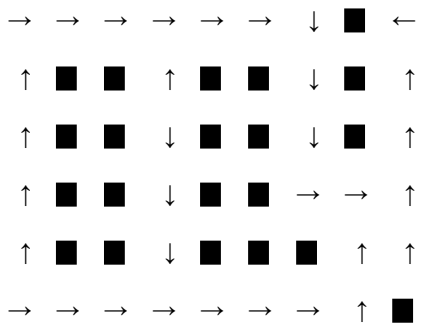
Improved policy interation 14000



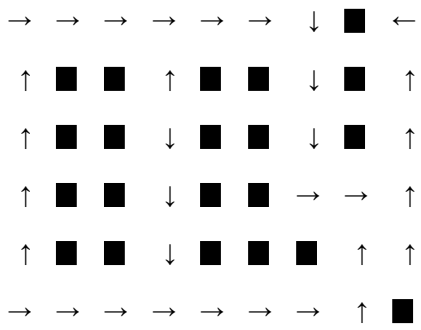
Improved policy interation 15000



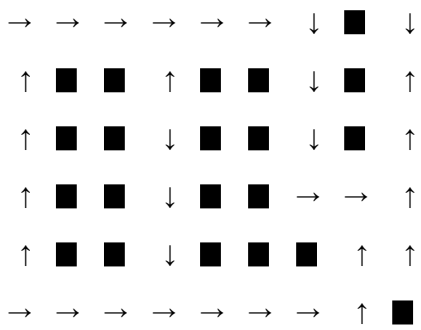
Improved policy interation 16000



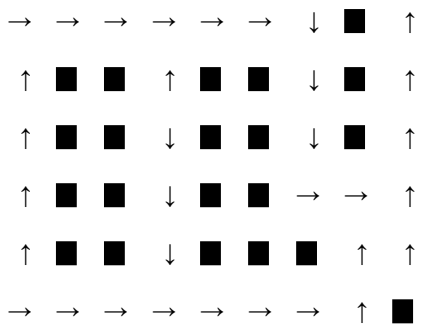
Improved policy interation 17000



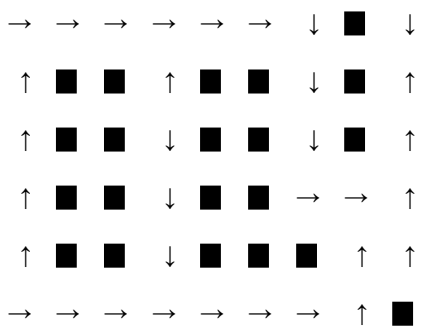
Improved policy interation 18000



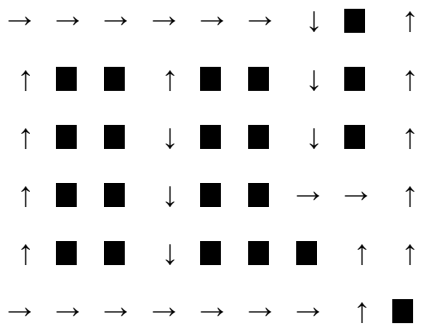
Improved policy interation 19000



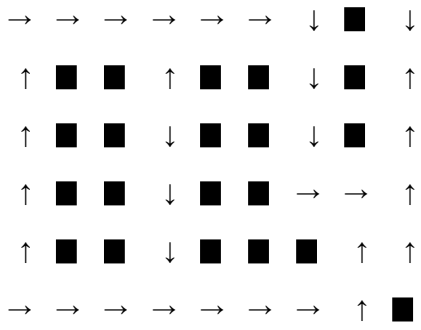
Improved policy interation 20000



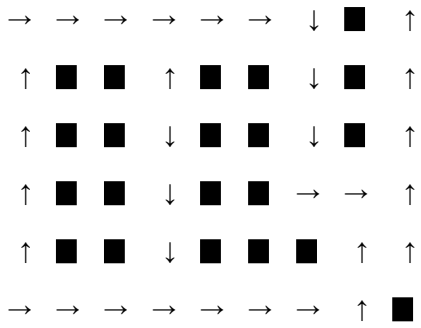
Improved policy interation 21000



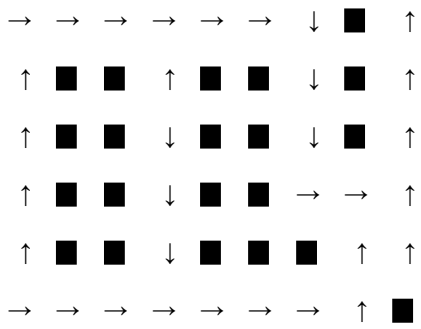
Improved policy interation 22000



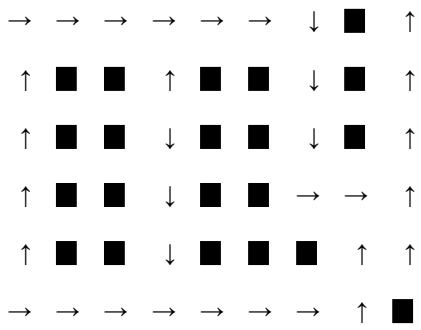
Improved policy interation 23000



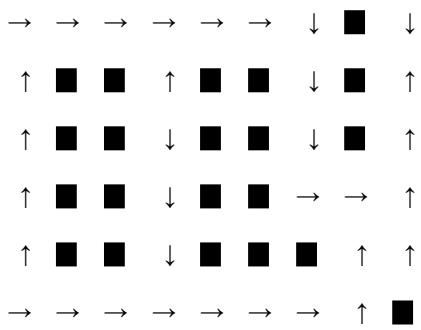
Improved policy interation 24000



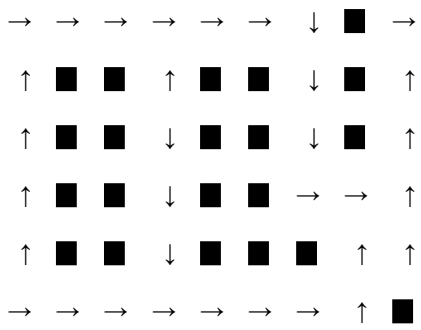
Improved policy interation 25000



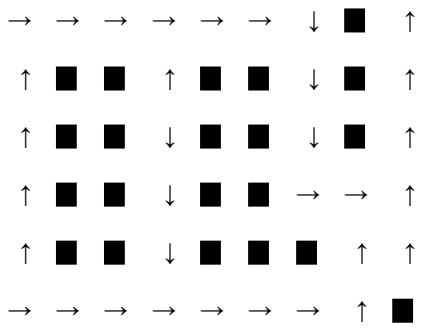
Improved policy interation 26000



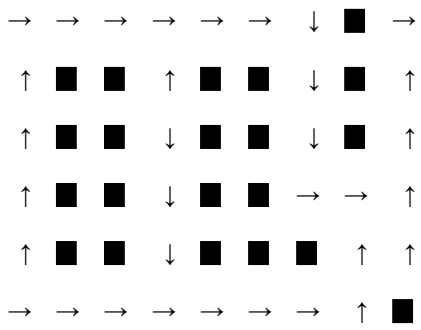
Improved policy interation 27000



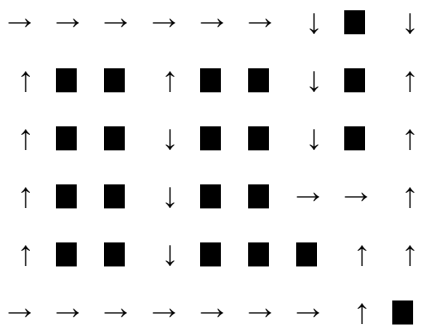
Improved policy interation 28000



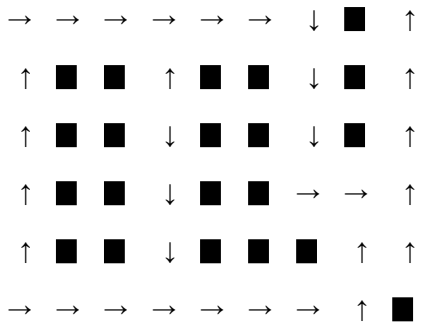
Improved policy interation 29000



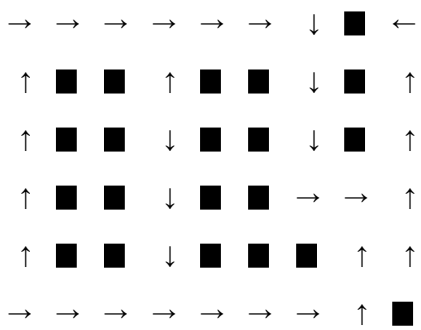
Improved policy interation 30000



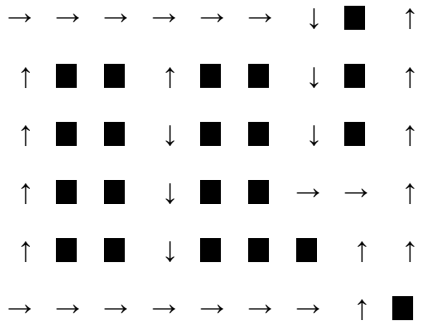
Improved policy interation 31000



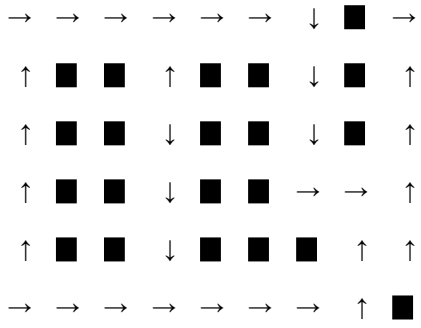
Improved policy interation 32000



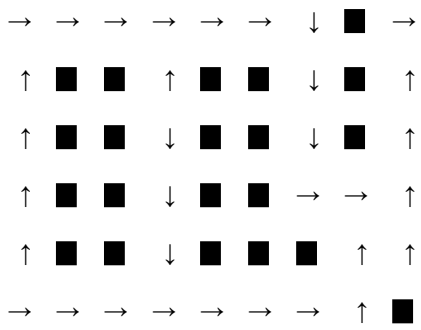
Improved policy interation 33000



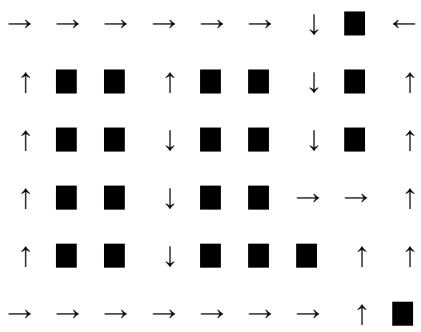
Improved policy interation 34000



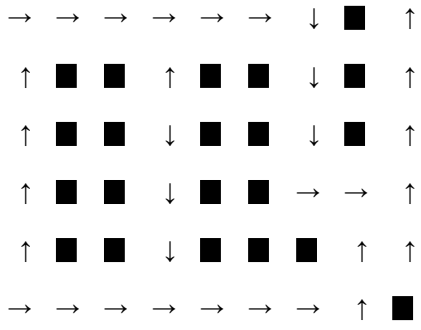
Improved policy interation 35000



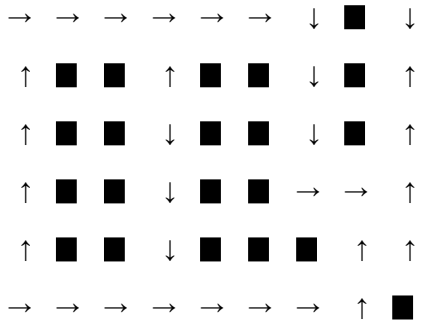
Improved policy interation 36000



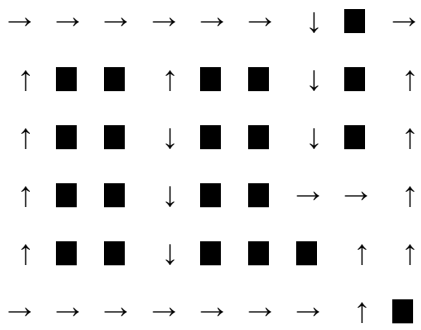
Improved policy interation 37000



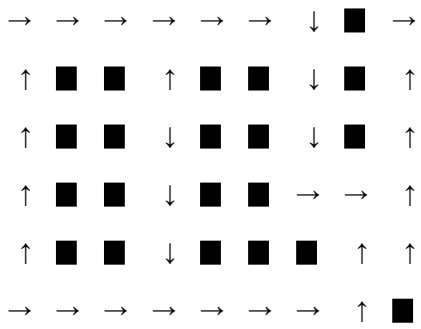
Improved policy interation 38000



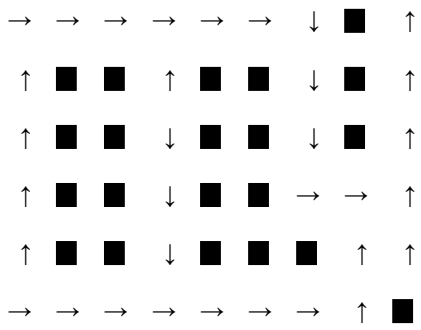
Improved policy interation 39000



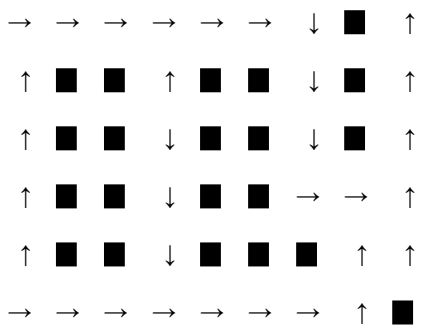
Improved policy interation 40000



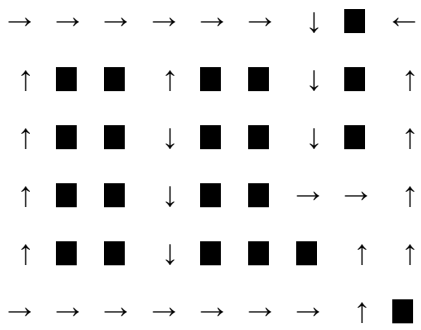
Improved policy interation 41000



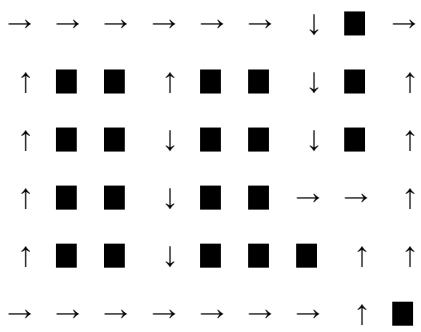
Improved policy interation 42000



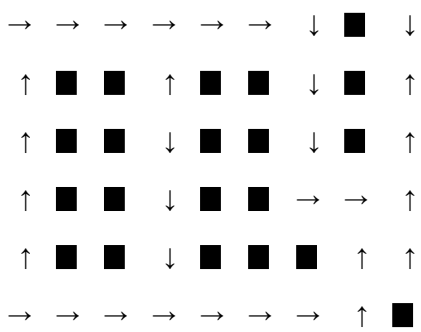
Improved policy interation 43000



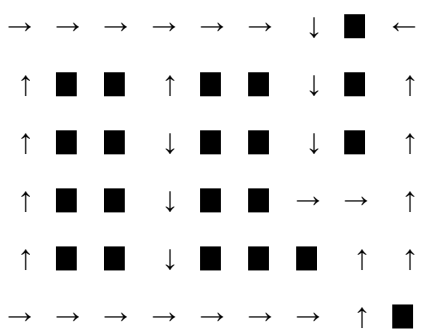
Improved policy interation 44000



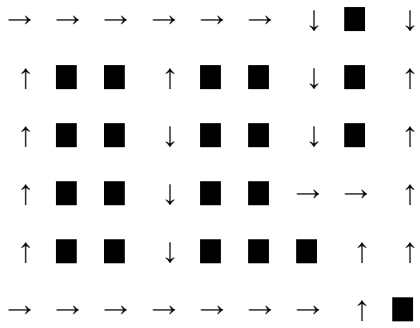
Improved policy interation 45000



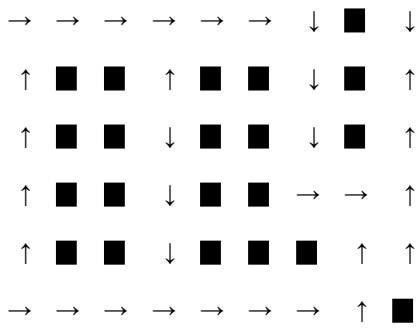
Improved policy interation 46000



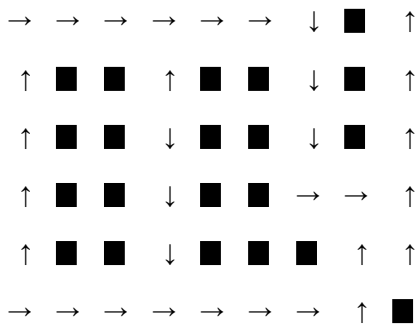
Improved policy interation 47000



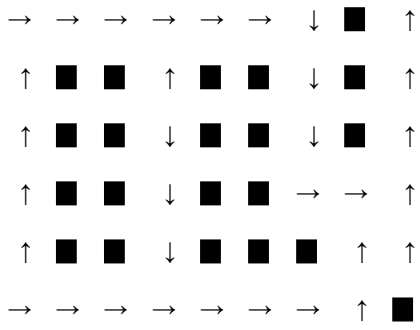
Improved policy interation 48000



Improved policy interation 49000



Improved policy



Problem 3 (10 points)

The two solution methods we just considered have different strengths and weaknesses. Describe in your own words the things that make these solutions methods better or worse. Your response should be 2-3 sentences and address both the computational efficiency of the algoriths, the amount of assumed knowledge of the environment, and the relationship between these methods to how humans might solve similar sequential decision making problems. Are either algorithm plausible models of human cognition?

- A key variables for dynamic programming algorithm are $\mathcal{P}^{a}_{ss'}$ which is the probability of reaching state s' when in state s and taking action a and rewards $\mathcal{R}^{a}_{ss'}$, which are delivered anytime the agent makes a transition from one state to another using a particular action. This requirement makes it worse than Monte Carlo method that doesn't depend on the existence of the environmental knowledge. However, if we do have knowledge of the environment, dynamic programming is better than Monte Carlo because the calculation for optimal solution can be easily driven using the bellman equation and it could be less resource intensive and driven with less number of simulations.
- Monte Carlo method is better when we are not given Rss' and Pss' that gives an exact behavior/information about the environment. Instead, we rely on the different random iterations to learn and adapt the algorithm to find the best choice of action. This is very useful because we often do not have the complete knowledge of the environment. Monte Carlo may take longer to train and require more computational resources if the environment is large and complex and this is a disadvantage of Monte Carlo.
- The Monte Carlo method is a plausible model of human cognition because we usually do not have the perfect knowledge of the environment. Information such as Rss' and Pss' are very uncommon in the real world, where we are often forced to experiment without a clear answer or feedback. Humans are more likely to use a first-visit Monte Carlo method where we experiement and learn from our actions, and then we adjust our action next time for better reward.

```
In [230]:
```

```
Out[230]: "(36, 'right')"
```

```
In [231]:
```

Out[231]:
```
[[0.0, 36, 'right'],
 [0.0, 36, 'down'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'right'],
 [0.0, 46, 'up'],
 [-1, 46, 'down'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [-1, 45, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [-1, 45, 'down'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'right'],
 [-1, 46, 'down'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [0.0, 36, 'down'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
 [0.0, 45, 'up'],
 [-1, 36, 'left'],
```

```
            [0.0, 45, 'up'],
            [0.0, 36, 'down'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'right'],
            [-1, 46, 'down'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [0.0, 36, 'up'],
            [0.0, 27, 'up'],
            [-1, 18, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [0.0, 36, 'down'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [-1, 45, 'down'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [-1, 45, 'left'],
            [-1, 45, 'down'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [-1, 45, 'down'],
            [0.0, 45, 'up'],
            [0.0, 36, 'right'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [-1, 36, 'left'],
            [0.0, 45, 'up'],
            [0.0, 36, 'up']]
```

In [232]:

Out[232]: 0.0

In [249]:

Out[249]: []

In [250]:

In [251]:

In [252]:

```
(36, 'right')
```

In [253]:

Out[253]: defaultdict(list, {"(36, 'right')": [1, 2]})

In [ ]: