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/)](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 embboddied 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]:
```python
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 allowed 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)](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.transitions[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 policies
        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)](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 inconsistencies 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 \episilon 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'],
 [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 [ ]:

Homework - Reinforcement Learning - Part B (60/100 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/)](https://brendenlake.github.io/CCM-site/)

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

Learning and deciding in an unknown world

Part A (Homework-RL-A.ipynb) of the homework explored various solution to a sequential decision making problem in a case where considerable information about the environment was known or was provided (e.g., the probabilities of transitioning between different states and the magnitude of rewards available in particular states). However, if reinforcement learning could only be applied to cases with full, explicit knowledge than it would be much less of a compelling model of human learning. In many cases, humans and other animals learn even when there is much more ambiguity, and as a result, a good model of experiential learning for humans would apply to cases where less is known a-priori about the world.

In this part of the homework, we will shift to think about learning and deciding in a unknown environment. This is a fairly complex topic with many different solutions and types of problems. However, we will focus on one particular problem class known as the **n-armed bandit**. N-armed bandits are optimization problems that mimic many real-world problems faced by humans, organizations, and machine learning agents. The term "bandit" comes from the name of the casino games where you pull a lever to enter a lottery. The bandits have one arm (the arm you pull down) and they steal your money (see above).

An N-armed bandit is a problem where a decision maker is presented with a bandit with n arms instead of just one (see Octopus cartoon). The task for the agent is, on each trial or moment in time, to choose bandits that are good while avoiding those that are less good. Since nothing may be known about the bandits a-priori, the problem is difficult and requires a balance of *exploration* (trying new things in order to learn) and *exploitation* (choosing options known to be good).

If each bandit paid out a fixed amount every time it was selected, then the problem would be solved with very simple exhaustive search process (visit each bandit once and then select the best one for the remaining time). However, the sequential search strategy just described doesn't capture the *opportunity cost* of exploration. For example, imagine that there is 100 armed bandits. Further assume that you know that 98 give zero reward, one gives a reward of 10, and one gives a reward of 20. If on the first pull you receive 10 units of reward then you are lucky and landed on a good one. However, is it worth going searching for the 20 point bandit? Given that you will have to pull a lot of zero reward bandits, it might actually be more rewarding over a finite period to continue to pull the 10 point bandit arm. Thus, exploration and exploitation act more like a tradeoff depending on the structure of the problem.

In addition, when the reward received from each bandit is probabilistic or stochastic, and furthermore the quality of the bandits might change over time, the problem becomes much more difficult. These cases require the agent to learn from the past but also be willing to adjust their beliefs based on more recent information.

Bandit tasks come up in many areas of cognitive science and machine learning. For example, there is a way to view A/B testing on websites as a particular type of bandit problem (https://www.amazon.com/Bandit-Algorithms-Website-Optimization-Developing/dp/1449341330) (your goal is to ensure conversions or purchases on your website, and your bandit arms are the different web designs you might try out). Similarly, the very real human problem of deciding where to eat lunch is a bit like a bandit problem -- should you return to your favorite restuarant or try a new one? Is the exploration worth giving up a reliably good meal?

In this part of the homework you will explore different simple algorithms for bandit problems.

Starter code

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

In [114]:

```
# The typical imports
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import random
import math
import seaborn as sns
```

A simple bandit environment

The first class provided here creates a set of simple, stationary multi-arm bandits. The bandits are stateless in that the reward from choose each action is simply a probabilistic function of the bandit itself, and there are no other cues you can use to decide which action to take. The parameters to the constructor of the bandit environment are:

- `mus` : the mean of the distribution from which each bandit is drawn from. This should have k numbers (for the k-armed bandit)
- `sds` : the standard deviation of the distribution from which the bandit means are drawn from (also k numbers for k bandits)

In [49]:

```
class KArmBanditStationary():
    def __init__(self, mus, sds):
        self.action_means = mus
        self.action_sds = sds
        self.optimal = np.argmax(self.action_means)
        self.k = len(mus)

    def step(self, action):
        return np.random.normal(self.action_means[action], self.action_s
ds[action]), action==self.optimal
```

Your job in this first exercise is to write a simple RL agent which samples from these bandits and attempts to earn as much reward as possible. The following cell gives an example of how to initialize the bandit and how to draw from it

In [50]:
```
bandit = KArmBanditStationary([0,50,10,4],[10,10,10,10])
action = 0 # pull the 0th bandit
bandit.step(action)  # return values are the reward, and if the action i
s actually optimal or not
```

Out[50]: (-11.639803475640203, False)

When we initialize the `KArmBanditStationary` we in some sense know exactly which arm is optimal (the one with the higher mean), and also how hard the problem is (the standard deviation of the rewards on each arm determines the difficulty... low SD generally is a easier task due to signal-noise relationship).

However, we are going to be implementing agents that do not have access to this information. For example, this first cell implements a simple random agent. You will want to modify this class to create an agent that can learn.

In [51]:
```
class RandomAgent():
    def __init__(self, k):
        self.num_actions = k
        # you could add parameters to your agent here
        pass

    def choose(self):
        return np.random.randint(self.num_actions)

    def learn(self, reward, action): # this agent doesn't learn
        pass
```

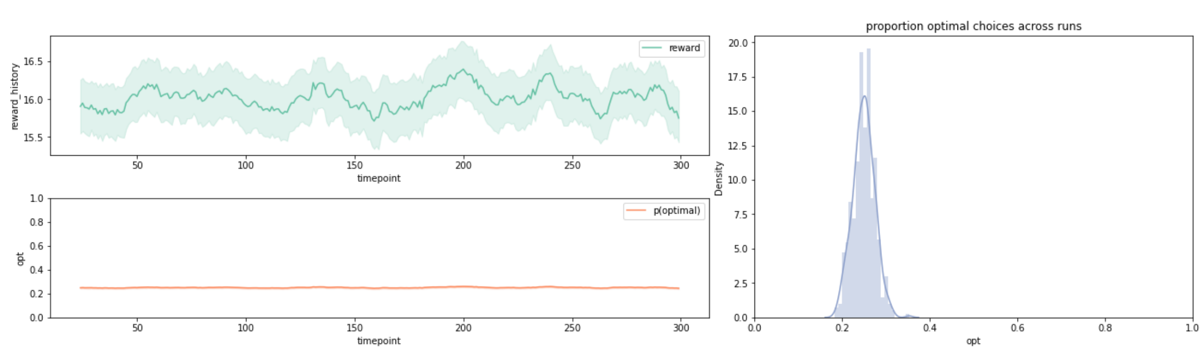
This cell helps you plot the reward history including a smoothed average reward earned by the agents over the last 30 trials

In [52]:
```
def plot_results(results_df, window=25):
    # set up figiure
    palette = iter(sns.color_palette("Set2"))
    fig = plt.figure(constrained_layout=True,figsize=(18,5))
    gs = fig.add_gridspec(2,5)

    # add three axes
    rew_ax  = fig.add_subplot(gs[0,:-2])
    opt_ax  = fig.add_subplot(gs[1,:-2])
    runs_ax = fig.add_subplot(gs[:,-2:])

    #fig, (rew_ax,opt_ax) = plt.subplots(nrows=2,ncols=1,figsize=(18,8))
    smooth_df = results_df.groupby('run').rolling(window,on='timepoint')
.mean()
    sns.lineplot(x='timepoint',y='reward_history', data=smooth_df, ax=re
w_ax, label='reward', color=next(palette))
    rew_ax.legend(loc="upper right")

    sns.lineplot(x='timepoint',y='opt', data=smooth_df, ax=opt_ax, label
='p(optimal)', color=next(palette))
    opt_ax.set_ylim(0,1)
    opt_ax.legend(loc="upper right")

    sns.distplot(results_df.groupby('run')['opt'].mean(), ax=runs_ax, co
lor=next(palette))
    runs_ax.set_title("proportion optimal choices across runs")
    runs_ax.set_xlim(0,1)
```

Finally, this is an example of the random agent's performance in the environment.

In [53]:
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of the experiment
    agent = RandomAgent(n_bandits)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep, "reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



Here the plot shows the average reward earned for each trial fo the task across the repeated runs/experiments. The lower panel shows the proportion of optimal choices made by the agent across trials. The error bars on both of these plots are bootstrapped confidence intervals across runs of the agent. Finally, the right panel shows a histogram of the proportion of optimal choices made across the runs.

Everything looks as expected here. The random agent makes about the average reward of the task and chooses the optimal choice about 25% of the time.

Can you use what we have learned in lecture to do better?

Problem 4 (15 points)

Create a new agent class based on `RandomAgent` called `EpsilonGreedyAgentIncremental()` which keeps track of the average reward earned from each draw of the bandit. This agent will include a parameter `epsilon` which will determine the probability of choosing a random action, otherwise it should choose the best so far. To update the value of each bandit use the incremental implementation of mean-tracking which was first introduced on the slides blending monte carlo methods (this is a incremental approach to calculating the mean as a new observation arrive). Make a plot similar to the ones above showing the performance of the agent on the three measures we have considered when the environment is initialized in the same way (i.e., means are a random shuffling of [0.,50.,10.,4.] with sd=1). Next, show with a couple examples how changes in epsilon determine the shape of that plot. You should show the final code for your agent and the plots along with a single markdown cell describing your solution (1-2 paragraphs). In your answer be sure to explain if your new agent does better than the random agent and why you think that is the case. In all cases run your agent for 300 time steps and average over 500 runs.

- Change in Epsilon shifts the proportion of optimal choice distribution graph left as Epsilon value gets closer to 1. Opposingly, as Epsilon gets closer to 0, proportion of optimal choice distribution graph to right. This means when Epsilon is closer to 0, the proportion that the algorithm makes the optimal choice is higher. This trend is shown with below plots for epsilon = 0.1, 0.5, 0.9.
- Below is the code for EpsilonGreedyAgentIncremental where I initialize variables k num_actions, epsilon, Q table for value of each action, and N table for counting the number of visits to each action. Then, choose function incorporates epsilon greedy algorithm where less than epsilon random probability chooses the action randomly and greater than epsilon chooses the action with maximum Q value. Finally, learn function takes in reward and action as a parameter; then, N table is incremented by 1 everytime an action is performed (i.e. pulling the nth arm). New EpsilonGreedyAgentIncremental() agent does better than the random agent because it incorporates epsilon greedy algorithm as well as exploring random action when choosing the action of the agent. Also, learn function allows the agent to learn from the action choice using the Q(a) table that updates the value of an action with formula Q(a) = Q(a) + 1/N(a) * (reward - Q(a)). On the other hand, RandomAgent() randomly chooses the action and does not learn from the action. As a result, random agent does not improve proportion of optimal choices even after many iterations.

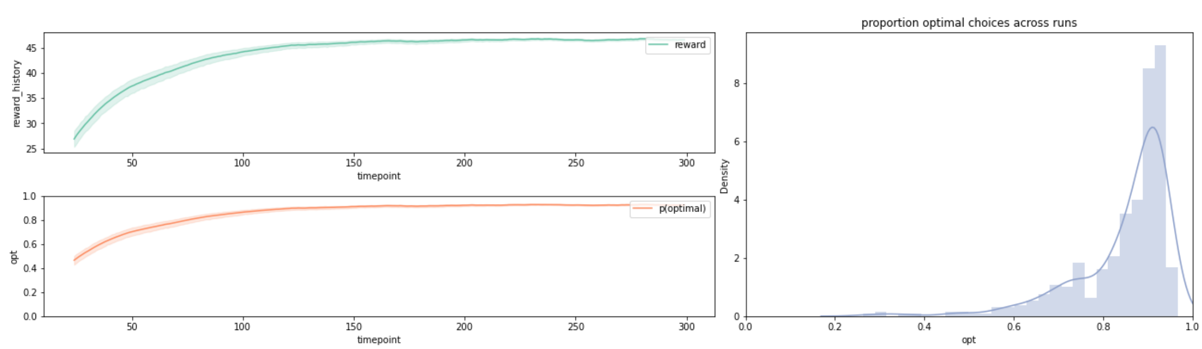
In [54]:
```
class EpsilonGreedyAgentIncremental():
    def __init__(self, k, epsilon):
        self.num_actions = k
        self.epsilon = epsilon # determines the prob of choosing a rando
m action/ otherwise choose best one so far.
        # you could add parameters to your agent here

        # list of 0's with k elements; value table for each action/ nth
 arm
        self.Q = list(0 for i in range(0, k))
        # list of 0's with k elements; # of times nth arms are pulled/ch
osen
        self.N = list(0 for i in range(0, k))

    def choose(self): # this method returns integer that represent the a
rm in N-bandit
        probability = random.random()
        if probability < self.epsilon:
            return np.random.randint(self.num_actions)
        else:
            # return the argmax action from Q
            max_index = np.argmax(self.Q)
            return max_index

    def learn(self, reward, action):
        # increment nth arm index's value every time it is chosen/visite
d
        self.N[action] += 1
        # update Q-table (aka V(s) value table for each action/nth arm p
ulled)
        self.Q[action] += (1/self.N[action]) * (reward - self.Q[action])
```

With epsilon = 0.1

In [55]:

```python
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits
epsilon = 0.1

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentIncremental(n_bandits,epsilon)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a ne
w bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

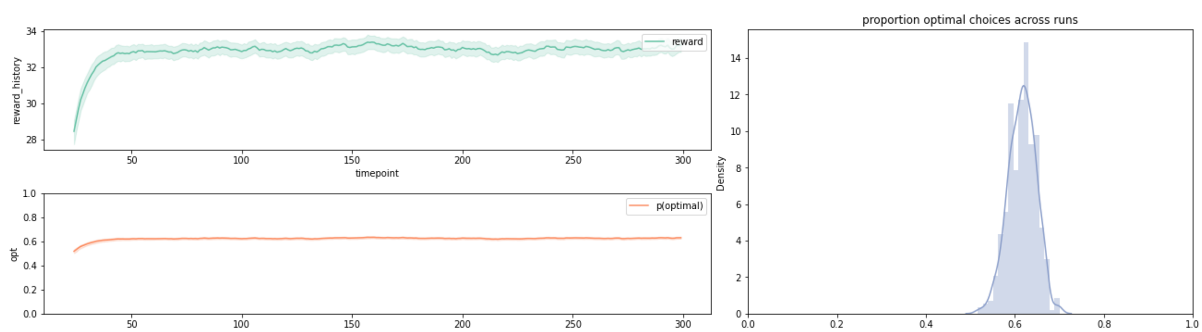


With epsilon = 0.5

In [56]:

```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits
epsilon = 0.5

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentIncremental(n_bandits,epsilon)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a ne
w bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

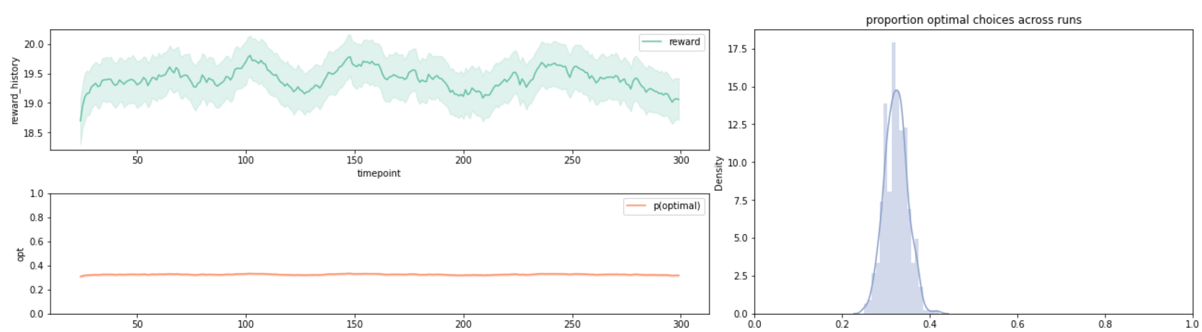
```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



With epsilon = 0.9

In [57]:
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits
epsilon = 0.9

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentIncremental(n_bandits,epsilon)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a ne
w bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



Problem 5 (15 points)

Create a new agent class based on `RandomAgent` called `EpsilonGreedyAgentConstant()` which keeps track of the average reward earned from each draw of the bandit. This agent will be nearly identical to `EpsilonGreedyIncremental()`. However, in addition to the parameter `epsilon` which will determine the probability of choosing a random action, this agent should use the "constant step size" update rule related to temporal-different learning to update the value of each action. The step size parameter (`alpha`) should be added as a new input parameter to your agent (hint: small values of this parameter are often better). Make a plot similar to the ones above showing the performance of the agent on the three measures we have considered. Is the performance of this agent the same or different than the previous agent you coded for this environment? Try a few parameter combinations with your agent and in your response show 1-2 examples to help make your point. You should show the final code for your agent and the plots along with a single markdown cell describing your solution (1-2 paragraphs). Be sure that your answer includes the answer to this key question: Does the constant agent out perform the incremental agent? And does it do better than the random agent? You don't need to do statistics but just a general visual comparison of the quality of the algorithsm is enough. In all cases run your agent for 300 time steps and average over 500 runs.

- The performance of this agent is a little different than the previous agent you coded for this environment because setting same epsilon value = 0.1 and alpha value = 0.7, 0.5 and plotting the graphs, I got similar proportion of optimal choices near 90% which is very close to the EpsilonGreedyAgentIncremental(). However, there was a very slightly more left skewedness with alpha = 0.7. Nontheless, previous incremental agent with epsilon = 0.1 also has very high proportion of optimal choices if not better than constant agent, so it is hard to define exactly whether constant agent is better than incremental agent.
- My Constant agent solution has constructor with parameters num actions k, epsilon, alpha, Q table for each action values, and N table for counting number of visits to action. Choose function follows the exact same epsilon greedy algorithm as the incremental agent. Finally, the Q table update algorithm uses alpha in place of 1/N(a) from incremental agent.
- The constant agent does not necessarily outperform the incremental agent because they are different methods of updating Q table. It may be more adapatable to environments that favors equal impact/update of action values across all nth actions. Unlike constant agent, incremental agent decreases the impact of actions with large # of visits from the 1/N(a) term. Nonetheless, constant agent is definitely better than random agent which does not converge to policy solution due to agent not learning from its actions.

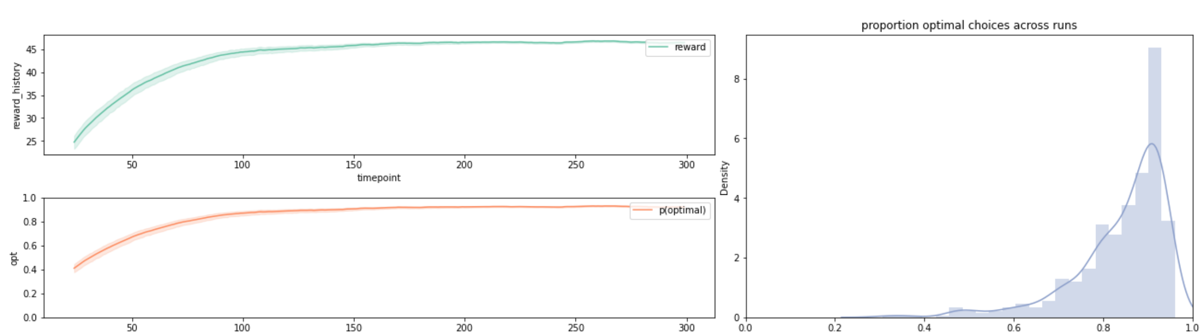
In [77]:
```
class EpsilonGreedyAgentConstant():
    def __init__(self, k, epsilon, alpha):
        self.num_actions = k
        self.epsilon = epsilon # determines the prob of choosing a rando
m action/ otherwise choose best one so far.
        self.alpha = alpha
        # you could add parameters to your agent here

        # list of 0's with k elements; value table for each action/ nth
 arm
        self.Q = list(0 for i in range(0, k))
        # list of 0's with k elements; # of times nth arms are pulled/ch
osen
        self.N = list(0 for i in range(0, k))

    def choose(self): # this method returns integer that represent the a
rm in N-bandit
        probability = random.random()
        if probability < self.epsilon:
            return np.random.randint(self.num_actions)
        else:
            # return the argmax action from Q
            max_index = np.argmax(self.Q)
            return max_index

    def learn(self, reward, action): # this agent doesn't learn
        self.N[action] += 1
        self.Q[action] += self.alpha * (reward - self.Q[action])
```

epsilon = 0.1, alpha = 0.7

In [81]:

```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits
epsilon = 0.1
alpha = 0.7 # step-size

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentConstant(n_bandits,epsilon, alpha)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a ne
w bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

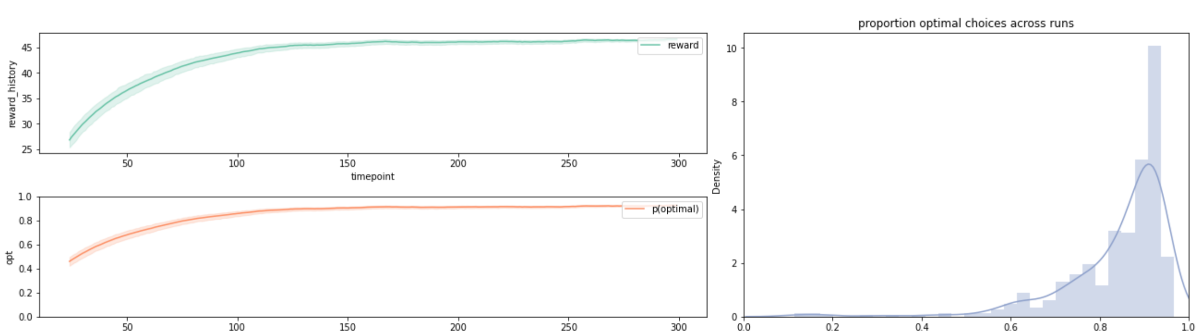
```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



epsilon = 0.1, alpha = 0.5

In [82]:
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits
epsilon = 0.1
alpha = 0.5 # step-size

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentConstant(n_bandits,epsilon, alpha)
    np.random.shuffle(bandit_means) # randomize location of "best"
    bandit = KArmBanditStationary(bandit_means,bandit_sds) # create a ne
w bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



Problem 6 (15 points)

Below is a new type of bandit environment based on `KArmBanditStationary` class where the reward probabilities of each bandit change over time. This is sometimes known as a "restless bandit" (see the Daw et al. 2013 paper on explore exploit mentioned in Lecture). The idea is that on each time step the mean reward of each action should be modified up or down by a sample from a Gaussian distribution (e.g., $\mu_{t+1} = \mu_t + \mathcal{N}(0, 20)$). I have called the new class `KArmBanditRestless`. The mean of the arms is itself drawn intiailly from a random normal distribution as well. Using this environment (with the number of arms set to 4), test the `RandomAgent()`, `EpsilonGreedyAgentConstant()` and `EpsilonGreedyAgentIncremental()` agents. You may want to play with the alpha parameter of the incremental agent to see if you can find a particularly good setting. Show the final code for your agent, plots showing the average reward the agent earns over time, along with a markdown cell describing your solution in 1-2 paragraphs. Which agent performs better in this environment? Is this different than the conclusion you made from the previous environment? Be sure to answer these two questions in your response. In all cases run your agent for 300 time steps and average over 500 runs.

- My solution includes 3 different agents (Random, Constant, and Incremental) with KArmBanditRestless as a bandit. For each of them, I set up parameters of simulation including time_steps and number of runs. Also, I set up bandit options, such as # of bandits, bandit means, and bandit standard deviations and any necessary parameters. In the for loop, I created the agent and bandit, and each run, agent chose an action and learned from the action to update the Q table for better rate of choosing optimal action.
- Average reward the agent earns over time is plotted for each agent methods
- In this environment, contant agent with epsilon = 0.1 and alpha = 0.5 performed the best based on the proportion of optimal choices distribution plot. This agent has the most left skewed plot of optimal choices graph, meaning it had the highest rate of choosing the optimal choice out of the three agents. Also, the width of the distribution was narrower than incremental agent, meaning less variance in the proportion.
- This is different than the conclusion of previous environment because in the previous stationary bandit, incremental agent (with epsilon = 0.1) in general had a very high performance almost better than that of constant agent. At the least, in the stationary bandit environment, incremental agent and constant agent had similar proportion of choosing optimal action with same exact parameters as in the restless bandit case. Instead, restless bandit environment makes it obvious that constant agent performs better than incremental agent because of the dynamic environment. Mean reward of each action is modified from Gaussian distribution impacting the update of Q(a) values, and this kind of dynamic environment works better with constant step-size alpha, where each discrepancy term (R-Q(a)) will have same influence.

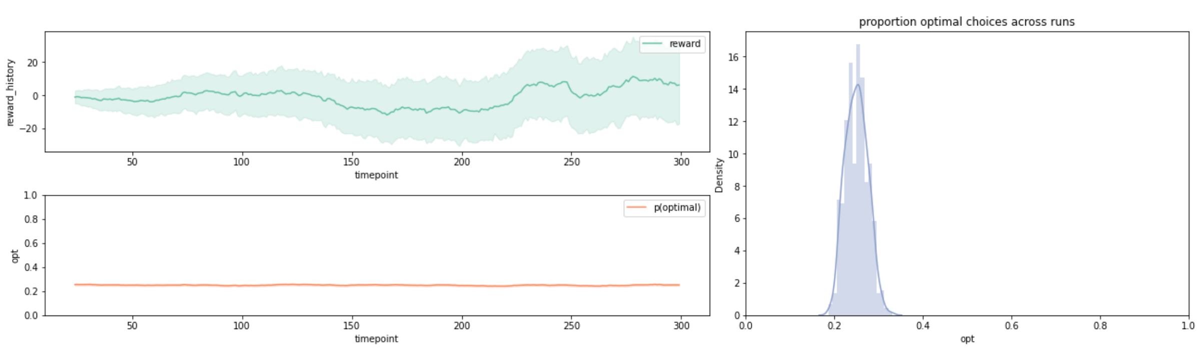
In [101]:
```
class KArmBanditRestless():
    def __init__(self, k, mu=0, sigma=2, sd=2, walk_sd=30):
        self.k = k
        self.action_means = np.random.normal(mu, sigma, k)
        self.action_sds = sd
        self.walk_sd = walk_sd
        self.optimal = np.argmax(self.action_means)

    def step(self, action):
        out = np.random.normal(self.action_means[action], self.action_sd
s), action==self.optimal
        self.action_means = self.action_means + np.random.normal(0.0, se
lf.walk_sd, self.k)
        self.optimal = np.argmax(self.action_means)
        return out
```

In [102]:
```
# intialize this bandit like this
n_bandits = 4
bandit = KArmBanditRestless(n_bandits)
```

RandomAgent()

In [122]:
```python
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
bandit_means = [0,50,10,4]
bandit_sds = [1]*n_bandits

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = RandomAgent(n_bandits)
    bandit = KArmBanditRestless(n_bandits) # create new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

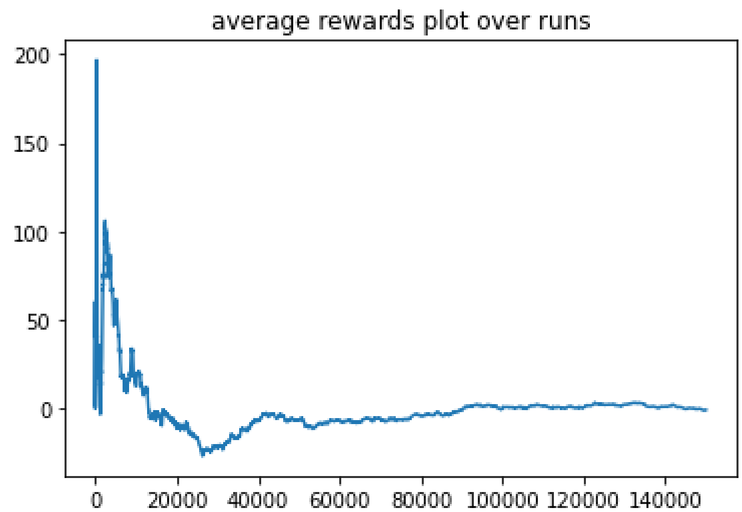


In [123]:

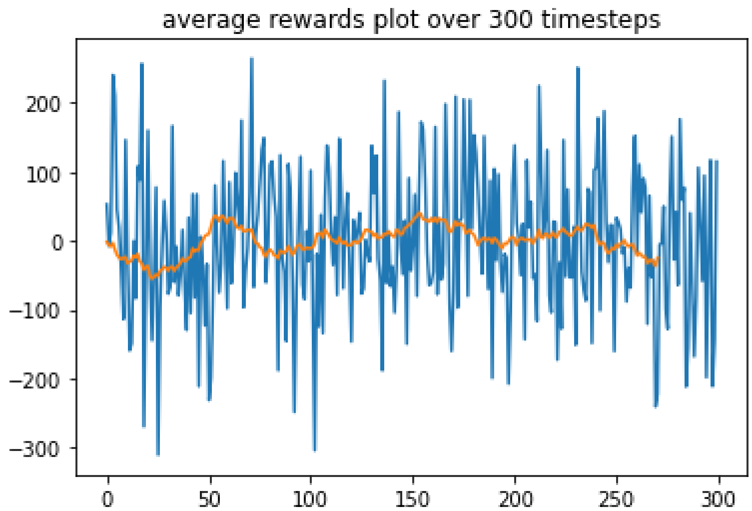
```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    avg_reward_list.append(sum_reward/(i+1))
plt.plot(avg_reward_list)
plt.title("average rewards plot over runs")
```

Out[123]:

```
Text(0.5, 1.0, 'average rewards plot over runs')
```



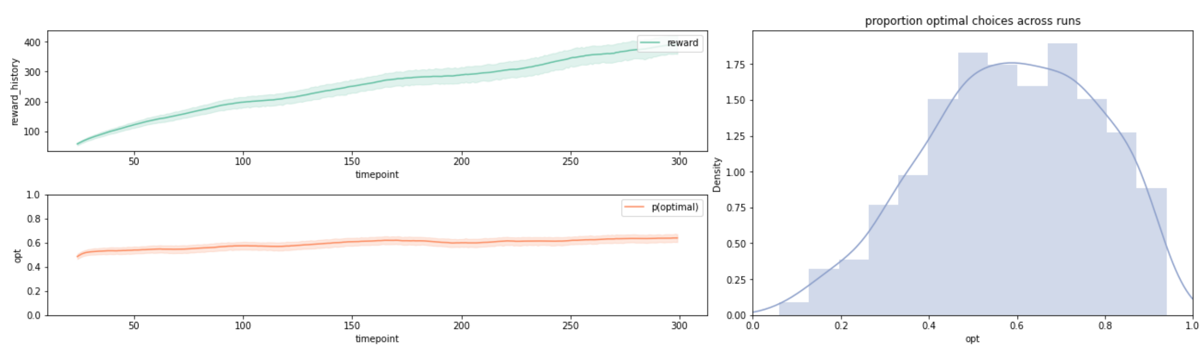
In [124]:
```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    if (i+1) % n_runs == 0:
        avg_reward_list.append(sum_reward/n_runs)
        sum_reward = 0
def running_mean(x, N):
    #x == an array of data. N == number of samples per average
    cumsum = np.cumsum(np.insert(x, 0, 0))
    return (cumsum[N:] - cumsum[:-N]) / float(N)
moving_avg = running_mean(avg_reward_list, 30)
plt.plot(avg_reward_list)
plt.plot(moving_avg)
plt.title("average rewards plot over 300 timesteps")
plt.show()
```



EpsilonGreedyAgentIncremental()

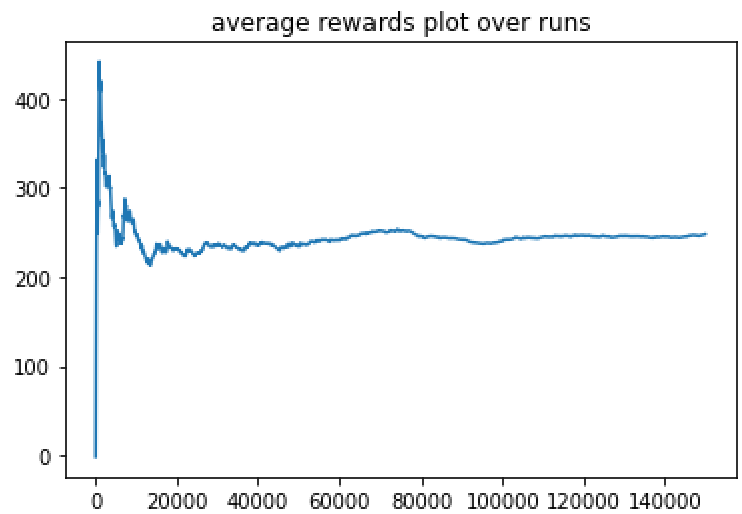
In [125]:
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
epsilon = 0.1

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentIncremental(n_bandits, epsilon)
    bandit = KArmBanditRestless(n_bandits)# create a new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



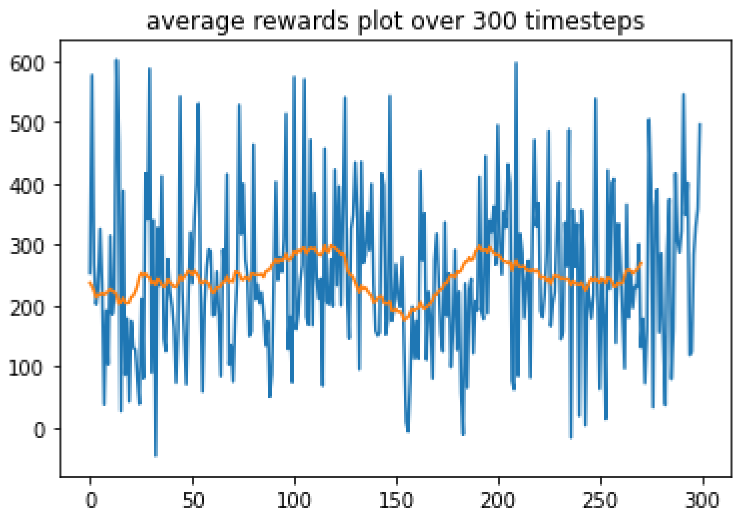
In [126]:
```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    avg_reward_list.append(sum_reward/(i+1))
plt.plot(avg_reward_list)
plt.title("average rewards plot over runs")
```

Out[126]:
```
Text(0.5, 1.0, 'average rewards plot over runs')
```



In [127]:

```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    if (i+1) % n_runs == 0:
        avg_reward_list.append(sum_reward/n_runs)
        sum_reward = 0
def running_mean(x, N):
    #x == an array of data. N == number of samples per average
    cumsum = np.cumsum(np.insert(x, 0, 0))
    return (cumsum[N:] - cumsum[:-N]) / float(N)
moving_avg = running_mean(avg_reward_list, 30)
plt.plot(avg_reward_list)
plt.plot(moving_avg)
plt.title("average rewards plot over 300 timesteps")
plt.show()
```

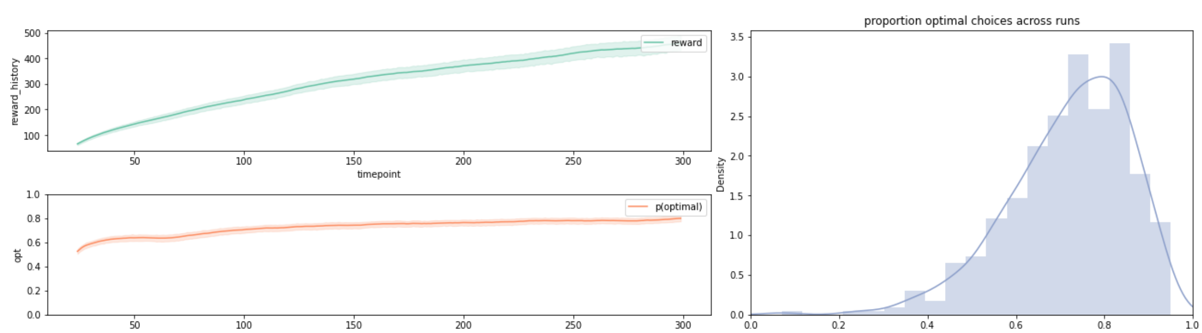


EpsilonGreedyAgentConstant() with epsilon = 0.08, alpha = 0.5

In [128]:

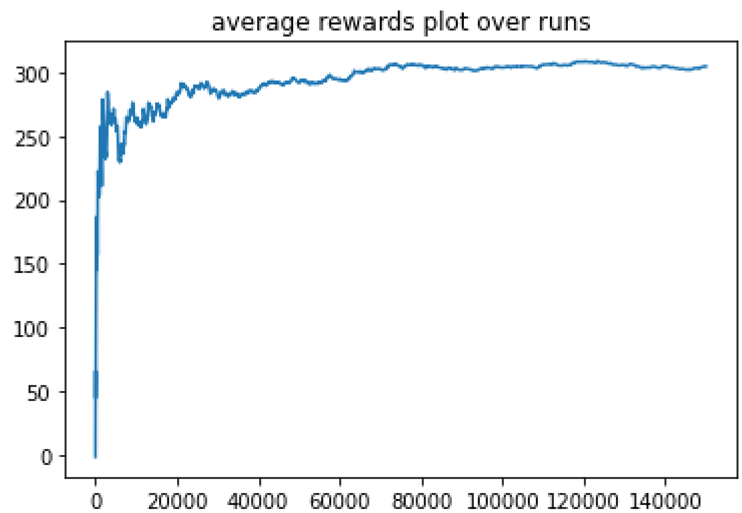
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
epsilon = 0.08
alpha = 0.5

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentConstant(n_bandits, epsilon, alpha)
    bandit = KArmBanditRestless(n_bandits) # create new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



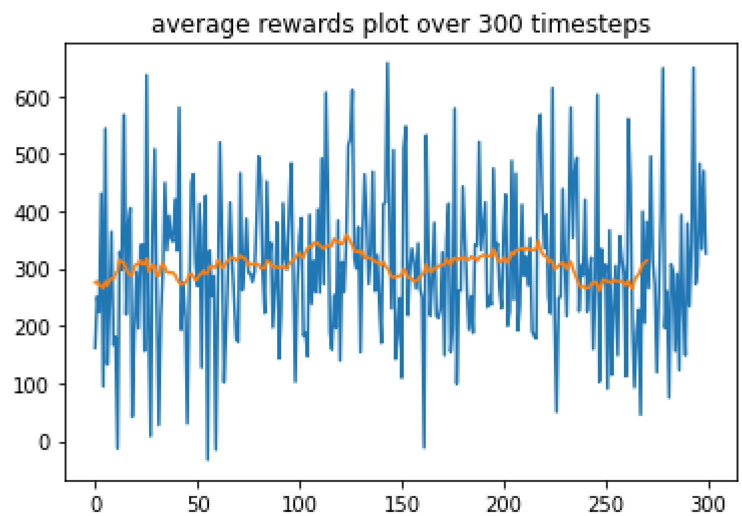
In [129]:
```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    avg_reward_list.append(sum_reward/(i+1))
plt.plot(avg_reward_list)
plt.title("average rewards plot over runs")
```

Out[129]:
```
Text(0.5, 1.0, 'average rewards plot over runs')
```



In [130]:

```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    if (i+1) % n_runs == 0:
        avg_reward_list.append(sum_reward/n_runs)
        sum_reward = 0
def running_mean(x, N):
    #x == an array of data. N == number of samples per average
    cumsum = np.cumsum(np.insert(x, 0, 0))
    return (cumsum[N:] - cumsum[:-N]) / float(N)
moving_avg = running_mean(avg_reward_list, 30)
plt.plot(avg_reward_list)
plt.plot(moving_avg)
plt.title("average rewards plot over 300 timesteps")
plt.show()
```

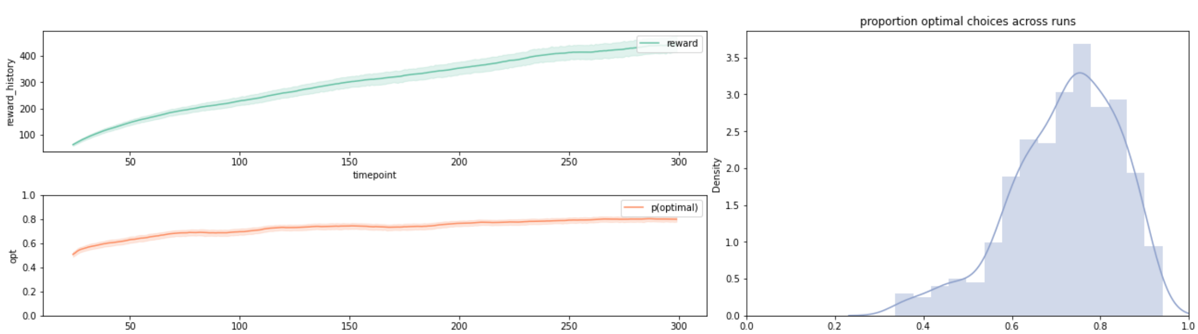


EpsilonGreedyAgentConstant() with epsilon = 0.1, alpha = 0.5

In [140]:

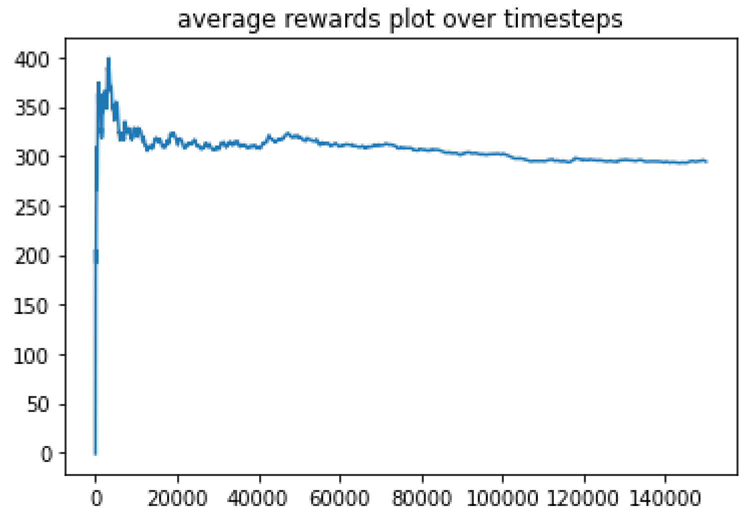
```
np.random.seed(100) # fix a seed for repeatable experiments

# parameters of simulation
n_timesteps = 300
n_runs = 500

# set up bandit options
n_bandits = 4
epsilon = 0.1
alpha = 0.5

reward_history = []
opt_history = []
run_history = []
timestep = []

for run in range(n_runs):  # repeat a given number random repetitions of
the experiment
    agent = EpsilonGreedyAgentConstant(n_bandits, epsilon, alpha)
    bandit = KArmBanditRestless(n_bandits) # create new bandit
    for i in range(n_timesteps): # for a certain number of time steps
        choice = agent.choose() # let the agent choose
        reward, opt = bandit.step(choice)
        agent.learn(reward, choice)
        reward_history.append(reward)
        opt_history.append(opt)
        run_history.append(run)
        timestep.append(i)

# plot the results
sim_result_df=pd.DataFrame({"run": run_history, "timepoint": timestep,
"reward_history":reward_history, "opt": opt_history})
plot_results(sim_result_df)
```

```
/Users/changhyunlee/opt/anaconda3/lib/python3.8/site-packages/seaborn
istributions.py:2619: FutureWarning: `distplot` is a deprecated functio
n and will be removed in a future version. Please adapt your code to us
e either `displot` (a figure-level function with similar flexibility) o
r `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



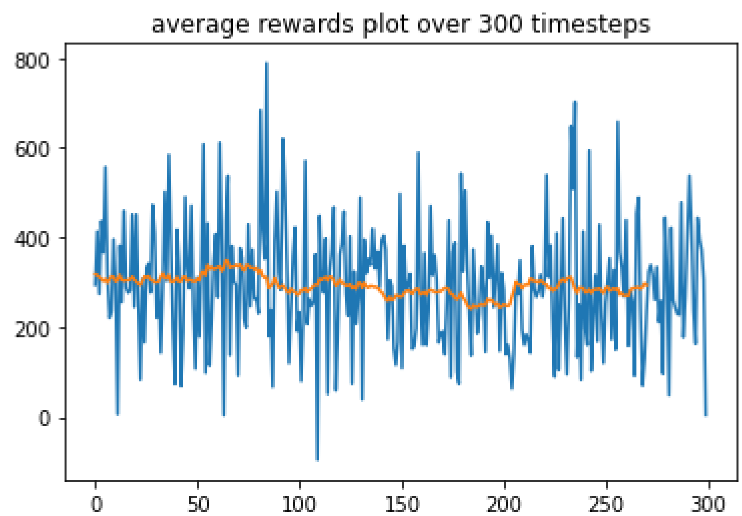
In [141]:
```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    avg_reward_list.append(sum_reward/(i+1))
plt.plot(avg_reward_list)
plt.title("average rewards plot over timesteps")
```

Out[141]:
```
Text(0.5, 1.0, 'average rewards plot over timesteps')
```



In [142]:

```
# plotting average rewards plot over time
avg_reward_list = []
sum_reward = 0
for i in range(len(reward_history)):
    sum_reward += reward_history[i]
    if (i+1) % n_runs == 0:
        avg_reward_list.append(sum_reward/n_runs)
        sum_reward = 0
def running_mean(x, N):
    #x == an array of data. N == number of samples per average
    cumsum = np.cumsum(np.insert(x, 0, 0))
    return (cumsum[N:] - cumsum[:-N]) / float(N)
moving_avg = running_mean(avg_reward_list, 30)
plt.plot(avg_reward_list)
plt.plot(moving_avg)
plt.title("average rewards plot over 300 timesteps")
plt.show()
```



Problem 7 (15 points)

For this problem, we return to the grid world task we considered in Part A. Specifically, you should combine the ideas about explore-exploit and incremental learning of q-values to implement a temporal-difference solution the to grid world problem you explored in Part A of the homework. You can reuse the code from that notebook in building your solution. This solution should also obtain similar results to the policy-iteration and monte-carlo versions you explored, but is learned more incrementally and online. The basic setup of the GridWorld environment is provided again for you below. Your solution to this problem should involve modifications to the solution to the Monte-Carlo problems in Part A. In particular, instead of waiting until a particular episode ends to update the values of the Q-values, use the Q-learning equation to incrementally updates these values as an episode unfolds. To balance exploration and exploitation try any of the methods you developed in the earlier parts of this assignment. As a reminder the question for updating the Q values in Q-learning is as follows: $Q(s,a) = Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$ The pseudo code for this algorithm is: ``` Initialize, for all $s \in S$, $a \in A(s)$: $Q(s,a)$ <- arbitrary Repeat many times (for each episode): a) Initialize s at the start state b) Repeat 1. Choose action a from s using policy dervived from Q values in that state (e.g., SoftMax) 2. Take action a, observe r, s' 3. Update $Q(s,a)$ Find max Q(s',a') over all action a' in state s' $Q(s,a) = Q(s,a) + \alpha[r + \gamma max Q(s',a') - Q(s,a)]$ s <- s' Until s is the goal state ```

In [65]:
```
# 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
```

In [66]:

```
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)

The following code sets up the major things you need to track. Note that unlike in the Monte Carlo solution you do not need a separate accounting of the returns as you are updating those to the Q-values directly. Also you shouldn't need to update the policy table until you have run many episodes through the maze. The final update to the `policy_table` should just to be to print out your final greedy solution and compare it to the solutions you obtained in Part A.

In [71]:

```
starting_state = 45
goal_state = 8 # terminate the MC roll out when you get to this state
GAMMA=0.9
EPSILON = 0.2
ALPHA = 0.5


# set up initial data strucutres that might be useful for you
# q(s,a) - the q-values for each action in each state
def zero_q_values():
    qvals = {"up": 0.0, "right": 0.0, "down": 0.0, "left": 0.0}
    return qvals
q_value_table = [[zero_q_values() for i in range(mygrid.ncols)] for j in range(mygrid.nrows)]

# pi - the policy table
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)
```

Initial (randomized) policy

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

In [72]:

```
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
```

50000 Iterations

In [73]:

```
ITERATIONS=50000

from tqdm import tqdm
for i in tqdm(range(ITERATIONS)):  # you probably need to take many, man
y steps 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
    curr_state = starting_state  # select and exploring start state


    while curr_state != goal_state:
        # initialize all global variables
        best_action = ''
        best_value = -999999999
        sx, sy = mygrid.index_to_coord(curr_state)
        next_state = 0

        # choose a based on Q(s,a) with epsilon-greedy algorithm
        if random.random() < EPSILON:
            # choosing random action a
            best_action = random.choice(['right', 'left', 'up', 'down'])
        else:

            for a in ['right', 'left', 'up', 'down']:
                if q_value_table[sx][sy][a] > best_value:
                    best_value = q_value_table[sx][sy][a]
                    best_action = a

        # finding next_state (integer)
        next_state = mygrid.transitions[curr_state][best_action]

        # calculate reward for curr_state
        reward = rewards[str([curr_state, best_action, next_state])]

        # calculate max_a_Q_s'_a'
        best_action2 = ''
        best_value2 = -99999999

        # next_state coordinates
        sx2, sy2 = mygrid.index_to_coord(next_state)

        # finding best_action2 for next state
        for a in ['right', 'left', 'up', 'down']:
            if q_value_table[sx2][sy2][a] > best_value2:
                best_value2 = q_value_table[sx][sy][a]
                best_action2 = action
        max_a_Q_s_prime_a_prime = q_value_table[sx2][sy2][best_action2]

        # **** update q value table *****
        q_value_table[sx][sy][best_action] += ALPHA * (reward + GAMMA *
max_a_Q_s_prime_a_prime - q_value_table[sx][sy][best_action])

        # update the current state to next state
        curr_state = next_state
```

```python
# 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)
```

```
100%|██████████| 50000/50000 [30:49<00:00, 27.04it/s]
```

Improved policy

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

200000 iterations

In [74]:

```
starting_state = 45
goal_state = 8 # terminate the MC roll out when you get to this state
GAMMA=0.9
EPSILON = 0.2
ALPHA = 0.5


# set up initial data strucutres that might be useful for you
# q(s,a) - the q-values for each action in each state
def zero_q_values():
    qvals = {"up": 0.0, "right": 0.0, "down": 0.0, "left": 0.0}
    return qvals
q_value_table = [[zero_q_values() for i in range(mygrid.ncols)] for j in
range(mygrid.nrows)]

# pi - the policy table
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)
```

Initial (randomized) policy

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

In [76]:
```
ITERATIONS=200000

from tqdm import tqdm
for i in tqdm(range(ITERATIONS)):  # you probably need to take many, man
y steps 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
    curr_state = starting_state  # select and exploring start state


    while curr_state != goal_state:
        # initialize all global variables
        best_action = ''
        best_value = -99999999
        sx, sy = mygrid.index_to_coord(curr_state)
        next_state = 0

        # choose a based on Q(s,a) with epsilon-greedy algorithm
        if random.random() < EPSILON:
            # choosing random action a
            best_action = random.choice(['right', 'left', 'up', 'down'])
        else:

#             for a in ['right', 'left', 'up', 'down']:
#                 if q_value_table[sx][sy][a] > best_value:
#                     best_value = q_value_table[sx][sy][a]
#                     best_action = a
            best_action = max(q_value_table[sx][sy], key = q_value_table
[sx][sy].get)
        next_state = mygrid.transitions[curr_state][best_action]

        # calculate reward for curr_state
        reward = rewards[str([curr_state, best_action, next_state])]

        # calculate max_a_Q_s'_a' for next state
        best_action2 = ''
        best_value2 = -99999999
        sx2, sy2 = mygrid.index_to_coord(next_state)
        # for a in ['right', 'left', 'up', 'down']:
        #     if q_value_table[sx2][sy2][a] > best_value2:
        #         best_value2 = q_value_table[sx][sy][a]
        #         best_action2 = action
        best_action2 = max(q_value_table[sx2][sy2], key = q_value_table[
sx2][sy2].get)
        max_a_Q_s_prime_a_prime = q_value_table[sx2][sy2][best_action2]

        # **** update q value table *****
        q_value_table[sx][sy][best_action] += ALPHA * (reward + GAMMA *
max_a_Q_s_prime_a_prime - q_value_table[sx][sy][best_action])

        # update the current state to next state
        curr_state = next_state

# improve policy
```

```python
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)
```

```
100%|██████████| 200000/200000 [00:21<00:00, 9320.92it/s]
```

Improved policy

```
→ → → → → → ↓ ■ ↑
↑ ■ ■ ↑ ■ ■ ↓ ■ ↑
↑ ■ ■ ↑ ■ ■ ↓ ■ ↑
↓ ■ ■ ↓ ■ ■ → → ↑
↓ ■ ■ ↓ ■ ■ ■ → ↑
→ → → → → → → ↑ ■
```

Compared to Part A, the improved policy table looks very similar with temporal method at 200K iterations. We can see the policy table converged to a similar solution.

```
In [ ]:
```