Project 2: Reinforcement Learning

Anthony Weng ad2weng@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

For the `small.csv` problem environment, I utilized **one-shot, vanilla Q-learning**; i.e., the provided episodes ((s, a, r, sp) tuples) were iterated over *once* and, for each episode, the $Q(s, a)$ matrix (4 by 10 by 10 to represent the 4 actions for each of the 100 possible `GridWorld` locations) was updated according to the following rule:

$$\hat{Q}_{opt}(s, a) \leftarrow (1 - \eta)\hat{Q}_{opt}(s, a) + \eta(r + \gamma\hat{V}_{opt}(sp)) \tag{1}$$

where $\hat{V}_{opt}(sp) = \max_{a' \in Actions(sp)} \hat{Q}_{opt}(sp, a')$.

Key parameters include: a *constant* learning rate of $\eta = 0.05$; a discount rate of $\gamma = 0.95$; and a default state-action value of 0.

The extracted policy consisted of executing the action which corresponded to the maximum state-action value achievable in a given state, i.e. $\pi(s) = \arg\max_a Q(s, a)$.

- For *this environment only*, a minimum positive state-action value $k = 1$ was also enforced; i.e., if $\max_a Q(s, a) < k$, then state s was assigned the policy of executing whichever action progressed (in theory, if the environment involved deterministic transitions) the agent toward the maximum Q-value state instead of $\arg\max_a Q(s, a)$.

Performance characteristics include runtimes of: Q-learning: 0.328 seconds; policy extraction: < 1 second; total program: 0.343 seconds.

1.2 Medium Data Set

For the `medium.csv` problem environment, I utilized **multi-shot, vanilla Q-learning**; i.e., the provided episodes ((s, a, r, sp) tuples) were iterated over `num_replays` times (with `num_replays` being a tunable hyperparameter) and, for each episode, the $Q(s, a)$ matrix (7 by 100 by 500 to represent the 7 actions for each of the possible combinations of velocity and position values) was updated according to Equation (1) described in **Section 1.1**.

Also, *reward shaping* was employed for these Q-updates: a penalty of $F(s)$ = (500 - inferred position value of state s) was subtracted from episode rewards prior to the Q-update to encourage actions which directed the agent toward position 500 (assumed to be proximal to the goal state). Note that use of this reward shaping function does not guarantee convergence on the optimal policy's Q-values; instead of a optimality-bound construct, it is more of a expert knowledge heuristic.

Key parameters include: a training data loop count of `num_replays` = 3; a *constant* learning rate of $\eta = 0.05$ for all replays; a discount rate of $\gamma = 1$; and a default state-action value of

-300. A negative state-action value was selected given the majority of non-positive rewards associated with this problem.

The extracted policy consisted of executing the action which corresponded to the maximum state-action value achievable in a given state, i.e. $\pi(s) = \arg\max_a Q(s,a)$.

- For *this environment only*, if no meaningful learning occurred (i.e., if $\max_a Q(s,a) =$ default state-action value), then state s was assigned the policy of executing a random action.

Performance characteristics include runtimes of: Q-learning: 26.826 seconds; policy extraction: <1 second; total program: 27.058 seconds.

1.3 Large Data Set

For the `large.csv` problem environment, I utilized **multi-shot, vanilla Q-learning with a replay buffer**; i.e., the provided episodes ((s, a, r, sp) tuples) were iterated over `num_replays` times (with `num_replays` being a hyperparameter) and each episode's information was added to a buffer. Once the buffer length exceeded a tunable `batch_size` hyperparameter, `batch_size` episodes were randomly sampled from the buffer and used to sequentially update the $Q(s,a)$ matrix (9 by 312020 to represent the 9 actions for each of the possible states) according to Equation (1) described in **Section 1.1**.

The replay buffer was implemented to limit the degree to which training episode ordering would affect estimated Q-values (a concern rising from both an incomplete representation of our problem action, state, and reward spaces in the training data as well as self-enforced limits on program runtime). For each training data replay, the replay buffer was re-initialized to be empty.

For this problem, I also implemented a deep Q-learning and a value iteration approach, both of which I ultimately discarded due to: (1) poor performance likely owing to inadequate feature engineering and selection; and (2) intractable runtimes to achieve convergence, respectively. The code for both alternate approaches is included below.

Key parameters include: a training data loop count of `num_replays` $= 5$; a *decaying* learning rate of $\eta = 0.25 - 0.05 \times$ `REPLAY_IDX` across replays (starting at 0.25, ending with 0.05); a `batch_size` $= 32$ for replay buffer sampling; a discount rate of $\gamma = 0.95$; and a default state-action value of 0.

The extracted policy consisted of executing the action which corresponded to the maximum state-action value achievable in a given state, i.e. $\pi(s) = \arg\max_a Q(s,a)$.

- For *this environment, as in the* `medium.csv` *problem*, if no meaningful learning occurred (i.e., if $\max_a Q(s,a) =$ default state-action value), then state s was assigned the policy of executing a random action.

Performance characteristics include runtimes of: Q-learning: 794.009 seconds; policy extraction: ≈ 1 second; total program: 795.574 seconds.

2. Code

2.1 Small Data Set

```
## imports:
import time
import numpy as np
import pandas as pd

## environment specs:
# grid world dims:
NUM_GRID_ROWS = 10
NUM_GRID_COLS = 10
# actions - 1: left, 2: right, 3: up, 4: down.
ACTIONS = [1, 2, 3, 4]
# discount rate:
gamma = 0.95

## hyperparameters:
# interpolation rate for q-learning:
eta = 0.05
# minimum val. s.t. the extracted policy will prescribe taking the action
    which
# produces this value instead of just moving toward the maximum value state:
min_state_val = 1

def get_state(grid_pos):
    s = grid_pos - 1
    row_num = s // NUM_GRID_ROWS
    col_num = s - (NUM_GRID_ROWS * row_num)
    return (row_num, col_num)


def q_learning(data):
    # init. state-action value and state value matrices:
    q_opt = np.zeros(shape=(len(ACTIONS), NUM_GRID_ROWS, NUM_GRID_COLS))
    v_opt = np.zeros(shape=(NUM_GRID_ROWS, NUM_GRID_COLS))

    # iterate through episodes and conduct q-learning!
    for episode in data.values:
        # extract episode information:
        s, a, r, sp = get_state(episode[0]), episode[1], episode[2],
    get_state(episode[3])

        # apply q-learning update (i.e., updating the state-action value
    matrix):
        # equ: q_opt(s,a) <- (1 - eta) * q_opt(s,a) + eta * (r + gamma *
    v_opt(sp))
        q_opt[a-1][s[0]][s[1]] = (1 - eta) * q_opt[a-1][s[0]][s[1]] + eta * (
    r + gamma * v_opt[sp[0]][sp[1]])
```

```
        # update the state value matrix:
        v_opt = q_opt.max(axis=0)

    return q_opt, v_opt


def get_direction(i, j, max_val_state):
    # uncouple row and column coordinates of max value state
    row_max_state, col_max_state = max_val_state[0], max_val_state[1]

    # if current pos (i, j) is closer to the max_val_state row-wise, move
    along rows:
    if abs(i - row_max_state) <= abs(j - col_max_state):
        # if in same row, move along columns:
        if i - row_max_state == 0:
            # if to right of max state:
            if j - col_max_state >= 0:
                return 1 # go left
            # if to left of max state:
            else:
                return 2 # go right
        # if in higher row:
        elif i - row_max_state < 0:
            return 4 # go down
        # if in lower row:
        elif i - row_max_state > 0:
            return 3 # go up
    # if closer column-wise:
    else:
        # if in same column, move along rows:
        if j - col_max_state == 0:
            # if in lower row:
            if i - row_max_state >= 0:
                return 3 # go up
            # if in higher row:
            else:
                return 4 # go down
        # if to left of max state:
        elif j - col_max_state < 0:
            return 2 # go right
        # if to right of max state
        elif j - col_max_state > 0:
            return 1 # go left


def extract_policy(q_opt, v_opt, filename):
    # determine the maximum state value and the corresponding state:
    v_max = v_opt.max()
    array = np.asarray(v_opt)
    idx   = (np.abs(array - v_max)).argmin()
    max_val_state = get_state(idx + 1)

    # create a file writer:
    with open(filename, 'w') as f:
        # iterate through v_opt matrix:
        for i in range(NUM_GRID_ROWS):
            for j in range(NUM_GRID_COLS):
                # initialize the action for this state:
                action_num = 0

                # if there is a positive maximum value associated with this
    state:
                if v_opt[i][j] > 0 and v_opt[i][j] > min_state_val:
                    # determine which action generates this value:
                    action_num = list(q_opt[:,i,j]).index(v_opt[i][j]) + 1 #
    add 1 b/c Python is 0-index'd
                # if not, just determine which action (when taken) will
    advance us toward the max value state:
                else:
                    action_num = get_direction(i, j, max_val_state)

                # write that action to our policy for this state:
                f.write(str(action_num)+'\n')


def main():
    # program time bookkeeping:
    program_start_time = time.time()

    # intake data:
    data = pd.read_csv('./data/small.csv', header=0)

    # conduct vanilla q_learning with the data (report q-learning runtime):
    q_learning_start_time = time.time()
    q_opt, v_opt = q_learning(data)
    print("Q-learning took: --- %s seconds ---"
          % round((time.time() - q_learning_start_time), 3))

    # extract and write policy to file:
    extract_policy(q_opt, v_opt, './policy_files/small.policy')

    # report program runtime:
    print("Overall program took: --- %s seconds ---"
          % round((time.time() - program_start_time), 3))


if __name__ == "__main__":
    main()
```

2.2 Medium Data Set

```
## imports:
import time
import numpy as np
import pandas as pd
import random as rd

## environment specs:
# num vals for vel, pos:
NUM_VEL_VALS = 100
NUM_POS_VALS = 500
# actions - indicating amounts of acceleration.
ACTIONS = [1, 2, 3, 4, 5, 6, 7]
# discount rate:
gamma = 1

## hyperparameters:
# default value for states:
st_val_dft = -300
# interpolation rate for q-learning
eta = 0.05
# number of times to loop through training data:
num_replays = 3

## other constants:
# random seed:
rd.seed(238)


def q_learning(data):
    # init. state-action value and state value matrices:
    q_opt = np.full(shape=(len(ACTIONS), NUM_VEL_VALS, NUM_POS_VALS),
    fill_value=st_val_dft)
    v_opt = np.full(shape=(NUM_VEL_VALS, NUM_POS_VALS), fill_value=st_val_dft
    )

    for _ in range(num_replays):
        # iterate through episodes and conduct q-learning!
        for episode in data.values:
            # extract episode information:
            s, a, r, sp = episode[0], episode[1], episode[2], episode[3]

            # convert s, sp to indices into state-value & value matrices
            (s_r, s_c)   = np.unravel_index(s,  shape=(NUM_VEL_VALS,
    NUM_POS_VALS))
            (sp_r, sp_c) = np.unravel_index(sp, shape=(NUM_VEL_VALS,
    NUM_POS_VALS))

            # *manual* reward shaping - subtract from 'r' distance to goal:
```

```
            r -= (NUM_POS_VALS - s_c)

            # apply q-learning update (i.e., updating the state-action value
    matrix):
            # equ: q_opt(s,a) <- (1 - eta) * q_opt(s,a) + eta * (r + gamma *
    v_opt(sp))
            q_opt[a-1][s_r][s_c] = (1 - eta) * q_opt[a-1][s_r][s_c] + eta * (
    r + gamma * v_opt[sp_r][sp_c])

            # update the state value matrix:
            v_opt = q_opt.max(axis=0)

    return q_opt


def extract_policy(q_opt, filename):
    # create a file writer:
    with open(filename, 'w') as f:
        # iterate through states:
        for i in range(NUM_VEL_VALS):
            for j in range(NUM_POS_VALS):
                # initialize the policy action for this state:
                action_num = 0

                # retrieve the state-action values for this corresponding
    state:
                sa_vals = q_opt[:,i,j]
                # determine the maximum action value for this state:
                max_action_val = max(list(sa_vals))

                # if the max action value is the same as the default, no
    meaningful learning occurred.
                if max_action_val == st_val_dft:
                    # so, just execute a random action as the policy:
                    action_num = rd.randint(1, len(ACTIONS))
                # otherwise, if the max action value exceeds the default
    value:
                else:
                    # select the action which achieves the max action value
    as the policy:
                    action_num = list(sa_vals).index(max_action_val) + 1 #
    plus 1 because Python is 0-indexed

                # write the extracted action for this state to our policy
    file:
                f.write(str(action_num)+'\n')


def main():
    # program time bookkeeping:
```

```
    program_start_time = time.time()

    # intake data:
    data = pd.read_csv('./data/medium.csv', header=0)

    # conduct vanilla q_learning with the data (report q-learning runtime):
    q_learning_start_time = time.time()
    q_opt = q_learning(data)
    print("Q-learning took: --- %s seconds ---"
          % round((time.time() - q_learning_start_time), 3))

    # extract and write policy to file:
    extract_policy(q_opt, './policy_files/medium.policy')

    # report program runtime:
    print("Overall program took: --- %s seconds ---"
          % round((time.time() - program_start_time), 3))


if __name__ == "__main__":
    main()
```

2.3 Large Data Set

2.4 Q-Learning: Policy Submitted for Scoring

```
    ## imports:
import time
import numpy as np
import pandas as pd
import random as rd

## environment specs:
# number of states:
NUM_STATES = 312020
# actions: mysteries!
ACTIONS = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# discount rate:
lmb = 0.95

## hyperparameters:
# default value for states:
st_val_dft = 0
# number of times to loop through training data:
num_replays = 5
# interpolation rate for q-learning
eta = (num_replays + 1) * 0.05

## other constants:
# random seed:
rd.seed(238)
# replay buffer size:
batch_size = 32


def q_learning(data):
    # init. state-action value and state value matrices:
    q_opt = np.full(shape=(len(ACTIONS), NUM_STATES), fill_value=st_val_dft)
    v_opt = np.full(shape=(NUM_STATES), fill_value=st_val_dft)

    for rn in range(num_replays):
        # report replay progress:
        print("Working on replay number: {rn}".format(rn=rn))
        # decrement and report learning rate:
        global eta
        eta -= 0.05
        print("Learning rate for this replay: {eta}".format(eta=eta))
        # re-init. replay buffer:
        buffer = []

        # iterate through episodes and conduct q-learning!
        for idx, episode in enumerate(data.values):
            # report episode progress:
            if (idx % 10000 == 0): print("Now working on episode: {idx}".
    format(idx=idx))
            # extract episode information:
            s, a, r, sp = episode[0], episode[1], episode[2], episode[3]
            # add episode to buffer:
            buffer.append((s, a, r, sp))
            # if buffer has enough episodes:
            if len(buffer) > batch_size:
                # sample a batch:
                samples = rd.sample(buffer, batch_size)
                # conduct q-learning for each sample in batch:
                for (s, a, r, sp) in samples:
                    # apply q-learning update (i.e., updating the state-
    action value matrix):
                    # equ: q_opt(s,a) <- (1 - eta) * q_opt(s,a) + eta * (r +
    lmb * v_opt(sp))
                    q_opt[a-1][s] = (1 - eta) * q_opt[a-1][s] + eta * (r +
    lmb * v_opt[sp])
                # update the state value matrix after all Q-updates for this
    batch:
                v_opt = q_opt.max(axis=0)

    return q_opt


def extract_policy(q_opt, filename):
    # create a file writer:
    with open(filename, 'w') as f:
        # iterate through states:
        for i in range(NUM_STATES):
            # initialize the policy action for this state:
            action_num = 0
            # retrieve the state-action values for this corresponding state:
            sa_vals = q_opt[:,i]
            # determine the maximum action value for this state:
            max_action_val = max(list(sa_vals))
            # if the max action value is the same as the default, no
    meaningful learning occurred.
            if max_action_val == st_val_dft:
                # so, just execute a random action as the policy:
                action_num = rd.randint(1, len(ACTIONS))
            # otherwise, if the max action value exceeds the default value:
            else:
                # select the action which achieves the max action value as
    the policy:
                action_num = list(sa_vals).index(max_action_val) + 1 # plus 1
     because Python is 0-indexed

            # write the extracted action for this state to our policy file:
```

```
            f.write(str(action_num)+'\n')


def main():
    # program time bookkeeping:
    program_start_time = time.time()

    # intake data:
    data = pd.read_csv('./data/large.csv', header=0)

    # conduct vanilla q_learning with the data (report q-learning runtime):
    q_learning_start_time = time.time()
    q_opt = q_learning(data)
    print("Q-learning took: --- %s seconds ---"
          % round((time.time() - q_learning_start_time), 3))

    # extract and write policy to file:
    extract_policy(q_opt, './policy_files/large.policy')

    # report program runtime:
    print("Overall program took: --- %s seconds ---"
          % round((time.time() - program_start_time), 3))


if __name__ == "__main__":
    main()
```

2.5 Deep Q-Learning: Policy Not Submitted for Scoring

```
## imports:
import time
import numpy  as np
import pandas as pd
import random as rd

import torch
import torch.nn as nn
import torch.nn.functional as F

## environment specs:
# number of states:
NUM_STATES = 312020
# number of digits in max index state:
MAX_STATES_DIGITS = 6
# actions: mysteries!
ACTIONS = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# discount rate:
lmb = 0.95

## hyperparameters:
# number of nodes in hidden layer:
NUM_HIDDEN_NODES = 12
# learning rate:
lr = 6e-4
# number of times to loop through training data:
num_replays = 20
# number of episodes to process before transferring
# training NN weights to target NN:
num_episodes_per_copy = 1000

## other constants:
NUM_INPUTS = 12
KEY_STATES = [150000, 230000, 270000, 290000, 300000]
rd.seed(238)


class DeepQNN(nn.Module):
    """ A simple 4-layer neural network for deep Q-learning."""
    def __init__(self):
        super().__init__()
        # create layers:
        self.ll1 = nn.Linear(NUM_INPUTS, NUM_HIDDEN_NODES)
        self.ll2 = nn.Linear(NUM_HIDDEN_NODES, NUM_HIDDEN_NODES)
        self.ll3 = nn.Linear(NUM_HIDDEN_NODES, len(ACTIONS))
        # initialize layers:
        nn.init.xavier_uniform_(self.ll1.weight)
        nn.init.xavier_uniform_(self.ll2.weight)
        nn.init.xavier_uniform_(self.ll3.weight)

    def forward(self, episode_input):
        x1  = F.relu(self.ll1(episode_input))
        x2  = F.relu(self.ll2(x1))
        out = self.ll3(x2)
        return out


def construct_state_nn_input_tensor(s):
    # nn input vector consists of state idx:
    ret = [s]
    # distance to key states with high/low rewards:
    for state_val in KEY_STATES:
        ret.append(s - state_val)
    digits = [int(d) for d in str(s)]
    # digits of state index as individual features:
    if len(digits) < MAX_STATES_DIGITS:
        # pad with 0's if necessary
        ret = ret + ([0] * (MAX_STATES_DIGITS - len(digits))) + digits
    else:
        ret = ret + digits
    return torch.Tensor(ret)


def main():
    ## initializations and data:
    # program time bookkeeping:
    program_start_time = time.time()

    # take over any available gpus:
    device = torch.cuda.current_device() if torch.cuda.is_available() else '
    cpu'

    # instantiate trainer & target deep q-learning networks:
    train_nn, target_nn = DeepQNN().to(device), DeepQNN().to(device)

    # instantiate optimizer and loss function:
    optimizer = torch.optim.Adam(train_nn.parameters(), lr = lr)
    loss_fn   = nn.MSELoss()

    # intake training episodes:
    episodes = pd.read_csv('./data/large.csv', header=0)

    # log for recent episode training losses:
    ep_losses = []

    # training time bookkeeping:
    training_start_time = time.time()
```

```
## kick off training!
# for each training data loop:
for replay_idx in range(num_replays):
    # bookkeeping: report which training data replay loop we're on.
    print("\nI'm chugging on episode set replay number: {replay_idx}\n".
format(replay_idx = replay_idx))
    # for each training episode:
    for idx, episode in enumerate(episodes.values):
        # if sufficient episodes have elapsed, copy training weights to
target network:
        if idx % num_episodes_per_copy == 0:
            target_nn.load_state_dict(train_nn.state_dict())

        # extract episode information:
        s, a, r, sp = episode[0], episode[1], episode[2], episode[3]

        # construct the input tensors for train_nn and target_nn:
        s_nn_rep  = construct_state_nn_input_tensor(s).to(device)
        sp_nn_rep = construct_state_nn_input_tensor(sp).to(device)

        # compute the estimated q-value of the given episode (q_opt(s,a))
:
        q_val = train_nn(s_nn_rep)[a-1]
        # compute the optimal (action) state-action value for sp:
        v_opt_sp = target_nn(sp_nn_rep).max(-1).values
        # compute the deep q-learning target (r + lmb & v_opt(sp)):
        target = r + lmb * v_opt_sp

        # learn from this episode:
        loss = loss_fn(q_val, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # record episode loss:
        ep_losses.append(loss.item())
        # print average loss every 5000 episodes:
        if len(ep_losses) % 5000 == 0:
            print("Average loss in last 5000 episodes: {avg_loss}".format
(avg_loss = np.mean(ep_losses)))
            # bookkeeping: reset loss log:
            ep_losses = []

# report network training time:
print("Deep Q-learning took: --- %s seconds ---"
      % round((time.time() - training_start_time)))

# save training and target networks weights:
torch.save(train_nn.state_dict(),  "./large_model_weights/train_nn.params
")
```

```
    torch.save(target_nn.state_dict(), "./large_model_weights/target_nn.
    params")

    # policy extraction/eval. time bookkeeping:
    policy_extraction_start_time = time.time()

    ## extract and write policy to file:
    # create a file writer:
    with open('./policy_files/large.policy', 'w') as f:
        # iterate through states:
        for state_num in range(NUM_STATES):
            # increment state number (since Python is 0-index'd):
            state_num += 1
            # construct state vector for this state:
            s_nn_rep = construct_state_nn_input_tensor(state_num).to(device)
            # compute the state-action values for this state:
            q_vals = target_nn(s_nn_rep)
            # init. the max Q-val idx:
            max_action_idx = 0
            try:
                # extract the index of the action with the maximum q_val:
                max_action_idx = q_vals.detach().cpu().numpy().argmax() + 1 #
     increment by 1 b/c Python is 0-index'd
            except:
                # in case anything goes wrong, just gen. a random action:
                max_action_idx = rd.choice(ACTIONS)
            # write extracted action to policy file:
            f.write(str(max_action_idx)+'\n')

    ## program time bookkeeping:
    # report policy evaluation time:
    print("Policy extraction took: --- %s seconds ---"
          % round(time.time() - policy_extraction_start_time))

    # report program runtime:
    print("Overall program took: --- %s seconds ---"
          % round(time.time() - program_start_time))

if __name__ == "__main__":
    main()
```

2.6 Value Iteration: Policy Not Submitted for Scoring

```
## imports:
import time
import numpy as np
import pandas as pd
import random as rd

## environment specs:
# number of states:
NUM_STATES = 312020
# actions: mysteries!
ACTIONS = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# discount rate:
lmb = 0.95

## hyperparameters:
# default value for states:
st_val_dft = 0
# max iterations for value iteration:
MAX_ITERS = 1000000

## other constants:
# random seed:
rd.seed(238)

def normalize(d, target=1.0):
   raw = sum(d.values())
   factor = target/raw
   return {key:round(value*factor,6) for key,value in d.items()}

def main():
    # program time bookkeeping:
    program_start_time = time.time()

    # intake data:
    data = pd.read_csv('./data/large.csv', header=0)

    # init. transition prob. dictionaries:
    action_prob_dict = {
        1 : {}, 4: {}, 7: {},
        2 : {}, 5: {}, 8: {},
        3 : {}, 6: {}, 9: {}
    }
    # populate transition prob. dictionaries:
    for idx, episode in enumerate(data.values):
        s, a, r, sp = episode[0], episode[1], episode[2], episode[3]
        if sp - s in action_prob_dict[a]:
            action_prob_dict[a][sp-s] += 1
        else:
            action_prob_dict[a][sp-s] = 1

    # init. state values:
    state_vals = np.full(shape=(NUM_STATES, 1), fill_value=st_val_dft)

    # conduct asynchronous value iteration:
    vi_start_time = time.time()
    for idx in range(MAX_ITERS):
        if idx % 10000 == 0: print("Working on iteration {idx}".format(idx=
    idx))

        state_to_update = idx % NUM_STATES
        action_values = []

        for a in ACTIONS:
            action_value = 0

            for state_chg in action_prob_dict[a].keys():
                r = 0
                sp = min(state_to_update + state_chg, 0)
                if sp == 151211 and state_to_update != 151211:
                    r = 100
                elif sp == 151312 and state_to_update != 151312:
                    r = 100
                action_value += lmb * state_vals[sp] * action_prob_dict[a][
    state_chg]
                action_value += r

            action_values.append(action_value)

        state_vals[state_to_update] = max(action_values)

    print("Asynch VI took: --- %s seconds ---"
        % round((time.time() - vi_start_time), 3))

    # extract policy:
    pe_start_time = time.time()
    with open('./policy_files/large_vi.policy', 'w') as f:
        # iterate through states:
        for i in range(NUM_STATES):
            # initialize the policy action for this state:
            action_num = 0

            action_values = []

            for a in ACTIONS:
                action_value = 0

                for state_chg in action_prob_dict[a].keys():
                    r = 0
```

```
                        sp = min(state_to_update + state_chg, 0)
                        if sp == 151211 and state_to_update != 151211:
                            r = 100
                        elif sp == 151312 and state_to_update != 151312:
                            r = 100
                        action_value += lmb * state_vals[sp] * action_prob_dict[a
    ][state_chg]
                        action_value += r

                    action_values.append(action_value)
                    opt_val = max(action_values)
                    action_idx = action_values.index(opt_val) + 1

                f.write(str(action_num)+'\n')

    print("PE took: --- %s seconds ---"
        % round((time.time() - pe_start_time), 3))
    print("Overall program took: --- %s seconds ---"
        % round((time.time() - program_start_time), 3))

    return 0

if __name__ == "__main__":
    main()
```