Project 2: Reinforcement Learning

Yu-Wei Lin ywlin@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

Algorithm Overview This implementation uses Q-learning, a model-free reinforcement learning algorithm, to solve three different Markov Decision Process (MDP) problems of varying complexity. The core algorithm uses the standard Q-learning update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1}$$

where:

1. α (alpha) is the learning rate
2. γ (gamma) is the discount factor
3. r_t is the immediate reward received after taking action a_t in state s_t
4. s_t is the current state at time step t
5. a_t is the action taken at time step t
6. s_{t+1} is the next state reached after taking action a_t in state s_t
7. $\max_a Q(s_{t+1}, a)$ is the maximum Q-value over all possible actions in the next state s_{t+1}
8. $Q(s_t, a_t)$ is the current estimate of the Q-value for state s_t and action a_t

2. Implementation Details

2.1 Key Implementation Features

1. **Optimistic Initialization**
   - Q-values are initialized to 0.1 instead of 0 to encourage exploration
   - This approach helps prevent the algorithm from getting stuck in local optima early in training
2. **Training Process**
   - Random shuffling of transitions each episode
   - Fixed learning rate ($\alpha = 0.1$) and episode count (1000) for all problems
   - Configurable discount factors (γ) per problem specification

2.2 Problem-Specific Implementations

2.2.1 Small Grid World (small.csv)

- **State Space**: 10×10 grid (100 states)
- **Action Space**: 4 actions (left, right, up, down)
- **State Conversion**: Linear indexing with `grid_to_state` and `state_to_grid` utilities
- **Discount Factor**: $\gamma = 0.95$
- **Running Time**: 150.25 seconds

2.2.2 Mountain Car (medium.csv)

- **State Space**: 50,000 states (500 positions $\times$ 100 velocities)
- **Action Space**: 7 actions (different acceleration values)
- **State Conversion**: Custom encoding using $1 + pos + 500 * vel$
- **Discount Factor**: $\gamma = 1.0$ (undiscounted)
- **Running Time**: 318.22 seconds

2.2.3 Large-Scale MDP (large.csv)

- **State Space**: 302,020 states
- **Action Space**: 9 actions
- **Discount Factor**: $\gamma = 0.95$
- **Running Time**: 330.70 seconds

3. Code

```
# CODE GOES HERE (change the language option to your language of choice)
import numpy as np
import pandas as pd
import time
from abc import ABC, abstractmethod
from typing import List, Tuple, Dict
import os
from pathlib import Path

class StateConverter:
    @staticmethod
    def grid_to_state(x: int, y: int, grid_size: int = 10) -> int:
        """Convert grid coordinates to state number for small.csv"""
```

```
        return np.ravel_multi_index((x, y), (grid_size, grid_size)) + 1  # 1-
    based indexing

    @staticmethod
    def state_to_grid(state: int, grid_size: int = 10) -> Tuple[int, int]:
        """Convert state number to grid coordinates for small.csv"""
        state = state - 1  # Convert to 0-based indexing
        return np.unravel_index(state, (grid_size, grid_size))

    @staticmethod
    def mountain_car_to_state(pos: int, vel: int) -> int:
        """Convert position and velocity to state number for medium.csv"""
        return 1 + pos + 500 * vel

    @staticmethod
    def state_to_mountain_car(state: int) -> Tuple[int, int]:
        """Convert state number to position and velocity for medium.csv"""
        state = state - 1  # Adjust for 1-based indexing
        vel = state // 500
        pos = state % 500
        return pos, vel

class RLMDP(ABC):
    def __init__(self, A: list[int], gamma: float):
        self.A = A          # action space (assumes 1:nactions)
        self.gamma = gamma  # discount factor

    @abstractmethod
    def lookahead(self, s: int, a: int) -> float:
        pass

    @abstractmethod
    def update(self, s: int, a: int, r: float, s_prime: int):
        pass

class QLearning(RLMDP):
    def __init__(self, S: list[int], A: list[int], gamma: float, alpha: float
    ):
        super().__init__(A, gamma)
        self.S = S
        self.alpha = alpha
        # self.Q = np.zeros((len(S) + 1, len(A) + 1))  # +1 for 1-based
    indexing

        self.Q = np.ones((len(S) + 1, len(A) + 1)) * 0.1 # Initialize Q-
    values optimistically to encourage exploration

    def lookahead(self, s: int, a: int) -> float:
        return self.Q[s, a]
```

```
    def update(self, s: int, a: int, r: float, s_prime: int):
        self.Q[s, a] += self.alpha * (r + self.gamma * np.max(self.Q[s_prime
    ]) - self.Q[s, a])

    def get_policy(self) -> List[int]:
        # Use Q values only for valid actions (columns 1 to n_actions)
        # Add 1 to convert from 0-based to 1-based indexing
        return [np.argmax(self.Q[s, 1:]) + 1 for s in self.S]



class MDPSolver:
    def __init__(self, data_dir: str = "./data"):
        self.data_dir = Path(data_dir)
        # Define problem configurations
        self.problems = {
            'small': {
                'states': 100,
                'actions': 4,
                'gamma': 0.95,
                'grid_size': 10,
                'validate_state': self.validate_small_state
            },
            'medium': {
                'states': 50000,
                'actions': 7,
                'gamma': 1.0,
                'pos_range': 500,
                'vel_range': 100,
                'validate_state': self.validate_medium_state
            },
            'large': {
                'states': 302020,
                'actions': 9,
                'gamma': 0.95,
                'validate_state': lambda x: 1 <= x <= 302020
            }
        }
        self.converter = StateConverter()

    def validate_small_state(self, state: int) -> bool:
        """Validate state for small problem"""
        if not (1 <= state <= 100):
            return False
        x, y = self.converter.state_to_grid(state)
        return 0 <= x < 10 and 0 <= y < 10

    def validate_medium_state(self, state: int) -> bool:
        """Validate state for medium problem"""
        if not (1 <= state <= 50000):
```

```
            return False
        pos, vel = self.converter.state_to_mountain_car(state)
        return 0 <= pos < 500 and 0 <= vel < 100

    def process_csv(self, filename: str) -> Tuple[List[Tuple[int, int, float,
     int]], Dict]:
        # Read and process the CSV file
        filepath = self.data_dir / filename
        df = pd.read_csv(filepath, header=None, names=['s', 'a', 'r', 'sp'],
    skiprows=1)


        # Get problem configuration
        problem_name = os.path.splitext(os.path.basename(filename))[0]
        config = self.problems[problem_name]

        # Validate states in the transitions
        valid_transitions = []
        for _, row in df.iterrows():
            # s, a, r, sp = row['s'], row['a'], row['r'], row['sp']
            s, a, r, sp = int(row['s']), int(row['a']), row['r'], int(row['sp
    '])
            if config['validate_state'](s) and config['validate_state'](sp):
                valid_transitions.append((s, a, r, sp))

        return valid_transitions, config

    def train_qlearning(self,
                       transitions: List[Tuple[int, int, float, int]],
                       config: Dict,
                       episodes: int = 1000,
                       alpha: float = 0.1) -> List[int]:
        # Initialize Q-learning
        states = list(range(1, config['states'] + 1))
        actions = list(range(1, config['actions'] + 1))
        ql = QLearning(states, actions, config['gamma'], alpha)

        # Training loop
        for episode in range(episodes):
            np.random.shuffle(transitions)
            for s, a, r, sp in transitions:
                ql.update(s, a, r, sp)

            # # Decay learning rate
            # if episode > episodes // 2:
            #     alpha *= 0.995

        policy = ql.get_policy()
```

```
        # Ensure policy has an action for every state and is within the
    action range
        if len(policy) != config['states']:
            raise ValueError(f"Policy length {len(policy)} does not match
    expected state count {config['states']}")
        for action in policy:
            if action < 1 or action > config['actions']:
                raise ValueError(f"Invalid action {action} in policy. Must be
     between 1 and {config['actions']}")

        return policy

    def save_policy(self, filename: str, policy: List[int]):
        # Save to the same directory as the input file
        policy_filename = self.data_dir / f"{os.path.splitext(os.path.
    basename(filename))[0]}.policy"
        with open(policy_filename, 'w') as f:
            for action in policy:
                f.write(f"{action}\n")

def main():
    # Create data directory if it doesn't exist
    data_dir = Path("./data")
    if not data_dir.exists():
        print(f"Creating data directory at {data_dir}")
        data_dir.mkdir(parents=True)

    solver = MDPSolver(data_dir=data_dir)

    # Process all CSV files in the data directory
    for filename in ['small.csv', 'medium.csv', 'large.csv']:
        filepath = data_dir / filename
        start_time = time.time()
        if filepath.exists():
            print(f"Processing {filename}...")

            # Read and process the CSV file
            transitions, config = solver.process_csv(filename)

            # Adjust training parameters based on problem size
            episodes = {
                'small': 1000,
                'medium': 1000,
                'large': 1000
            }[os.path.splitext(filename)[0]]

            alpha = {
                'small': 0.1,
                'medium': 0.1,
                'large': 0.1
```

```
            }[os.path.splitext(filename)[0]]

            # Train Q-learning model
            policy = solver.train_qlearning(transitions, config,
                                            episodes=episodes,
                                            alpha=alpha)

            # Save the policy
            solver.save_policy(filename, policy)
            print(f"Completed {filename}. Policy saved in {data_dir}")
        else:
            print(f"Warning: {filename} not found in {data_dir}")

        total_time = time.time() - start_time
        print(f"Total time for {filename}: {total_time:.2f} seconds")

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