Michael Naper
CS 4804
2012.11.29
Problem Set 5

Problem 1 A weighted coin with probability p of coming up heads is flipped repeatedly. You have some whole dollar amount of money that is less than a hundred dollars. At each time, you can choose to bet any whole dollar amount that is less than or equal to the amount you possess. If the coin comes up heads you double the amount you bet, and if it comes up tails you lose the entire amount you bet. Your goal is to accumulate \$100 – after that you retire. So there are two eventual outcomes – accumulate \$100 (you win), or go bust (end up with \$0 and lose).

The state is your capital $s \in \{0, 1, 2, \dots, 99, 100\}$. The actions are the amounts you may choose to bet. In state s, $a \in \{1, 2, \dots, s\}$. The states 0 and 100 represent terminal states, with rewards 0 and 1 respectively. The utility function is undiscounted, and non-terminal states have 0 utility.

You will solve the problem of how much to bet at any state using value iteration. Before you start, though, think about and answer one question: what does the value function for a state represent? Please make sure you know the answer to this question before you proceed!

Implement value iteration and solve for the optimal policy for $p = 0.25$, $p = 0.4$, and $p = 0.55$. Present your results for each case as two graphs, one showing the final value estimates as a function of the state and one showing the optimal policy (how much you should bet as a function of the state). Write down your interpretation of the forms of the optimal policies in these cases and explain the differences.

Solution:

The proposed game involves situations where outcomes are partly random and partly under the control of the decision maker, and thus can be modeled as a Markov decision process. The problem of how much to bet for a given capital was addressed by applying value iteration, in which we obtain the optimal policy for the gambler. That is, a function π was found that specifies the action $\pi(s)$ that the gambler should choose when in state s such that the probability of accumulating the goal amount is maximized. This policy can be described by

$$\pi^*(s) = \arg\max_{a \in A(s)} \left\{ \sum_{s'} P(s' | s, a) U(s') \right\}$$

where s is the current state, $A(s)$ is the set of actions available in state s, s' is a subsequent state reachable from state s, $P(s' | s, a)$ is a transition model that gives the probability that action $a \in A(s)$ in state s will lead to state s', and $U(s')$ is the utility obtained by executing π^* starting in state s'.

The utility function $U(s)$ is given by

$$U(s) = R(s) + \gamma \sum_{s'} P\big(s' | s, \pi(s)\big) U(s')$$

where $R(s)$ is the reward for being in state s and γ is the discount factor. For the given situation, $\gamma = 1$ and thus the utility function expresses the probability of winning from state s.

By substituting the calculation of $\pi(s)$ into the calculation of $U(s)$, we obtain the combined step

$$U(s) = R(s) + \max_{a \in A(s)} \left\{ \sum_{s'} P(s' | s, a) U(s') \right\}$$

We can now proceed to implement value iteration. Code is provided in Appendix A. We begin by assigning a value of $U(100) = 1$ to the goal state and $U(s) = 0$ for all states $s \in \{0, 1, \ldots, 99\}$. The update rule $U(s)$ is iterated for all states until it converges with the left-hand side equal to the right-hand side. That is,

$$U_{i+1}(s) = R(s) + \max_{a \in A(s)} \left\{ \sum_{s'} P(s' | s, a) U_i(s') \right\}$$

where the update is applied to all states at each iteration until equilibrium is reached.

We first present the probability of winning the game as a function of state for $p = 0.25$, $p = 0.4$, and $p = 0.55$, where p is the probability of the coin coming up heads for a flip, in Figure 1.

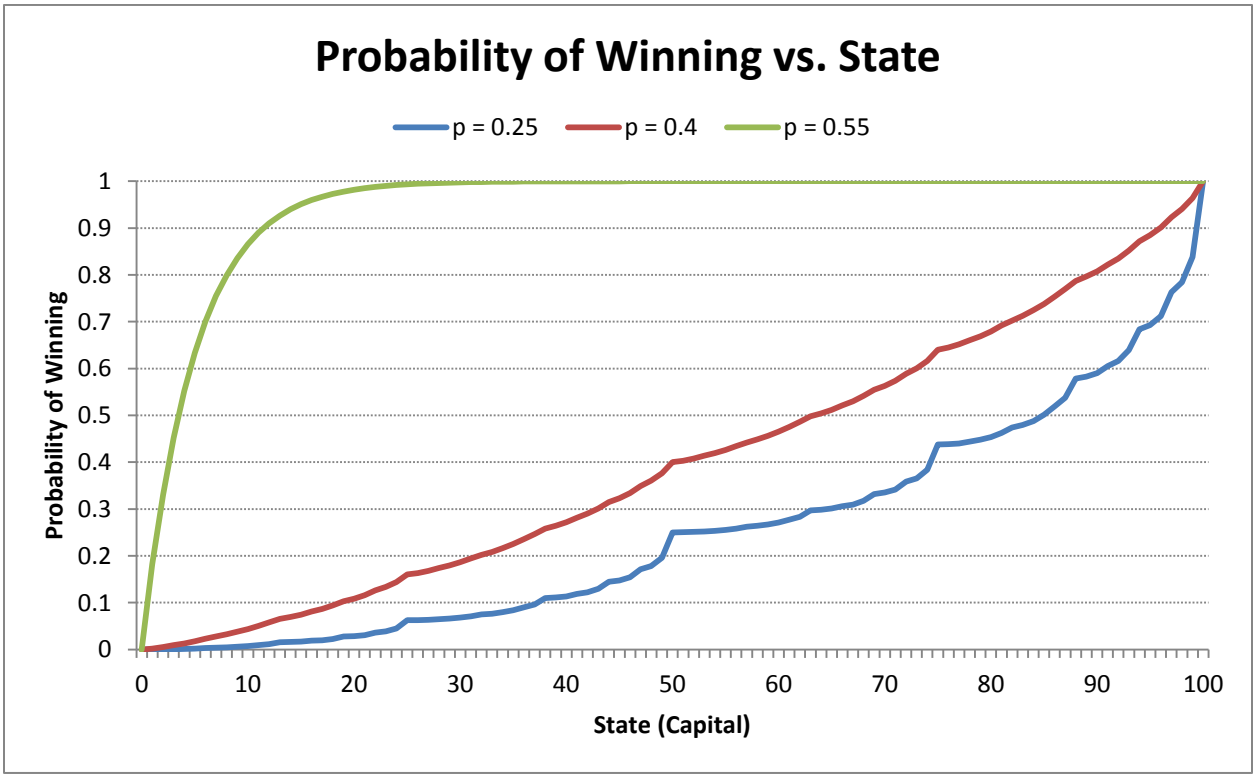


Figure 1. Final value estimates for $p = 0.25$, $p = 0.4$, and $p = 0.55$. The value function for a state corresponds to the probability of winning with a given capital when the gambler always chooses the optimal action.

As expected, the probability of winning increases as the gambler obtains a higher capital. The rate at which this probability increases is solely dependent on the value of p, increasing faster as p grows. For a probability of heads coming up being $p = 0.4$, the probability of winning takes a relatively linear form. A probability of $p = 0.25$ results in a convex probability function, slightly below the trend for $p = 0.4$. For a probability of $p = 0.55$, the resulting probability function is concave and rapidly approaches a guaranteed chance of winning, taking values near 1 as soon as a capital of $\$25$ is obtained.

Next we present the optimal policy as a function of state for the same values of p in Figure 2. The optimal policy identifies the bets the gambler should make so as to maximize his chances of winning the game.

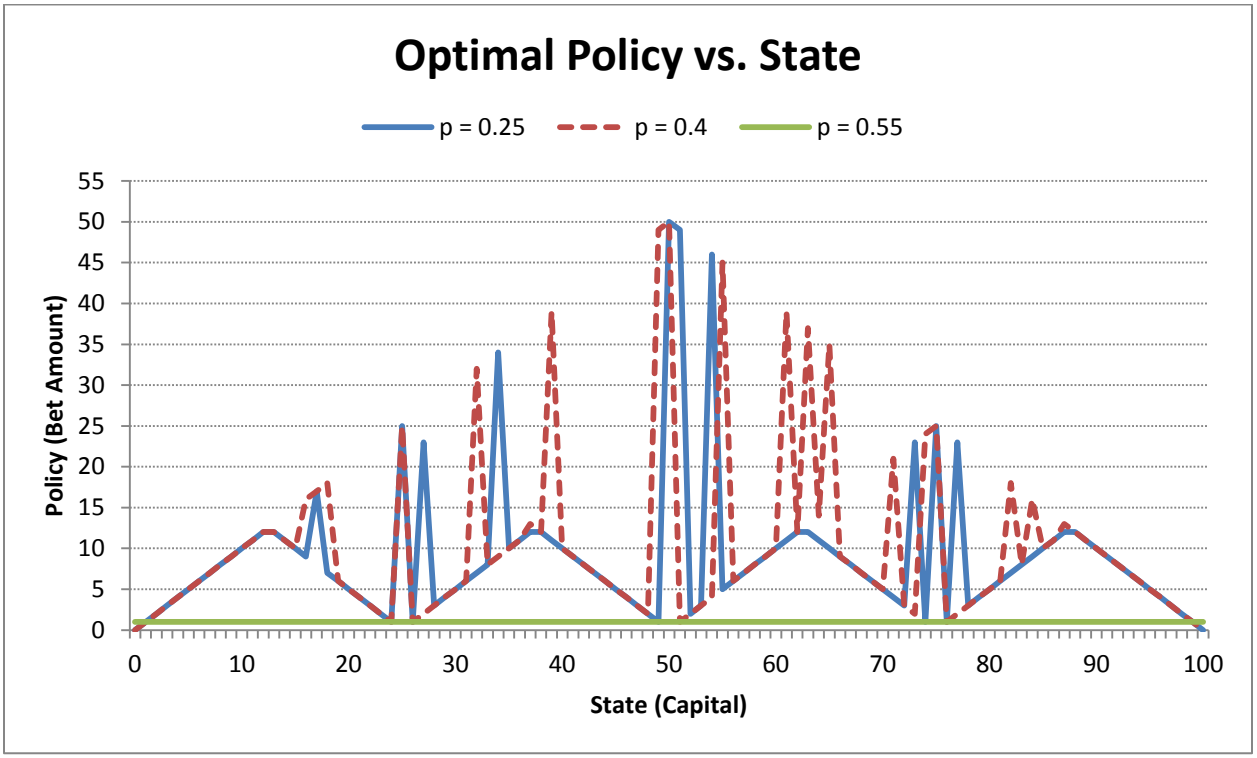


Figure 2. Optimal policy for $p = 0.25$, $p = 0.4$, and $p = 0.55$. The optimal policy for a state corresponds to the amount the gambler should bet with a given capital to maximize his probability of winning.

The case of $p = 0.55$ produces a trivial policy, one that always specifies a bet amount of $\$1$. When the chances of the coin landing in the gambler's favor are greater than not, the gambler can expect to increase his capital in the long run as he continues to flip the coin. Thus he should take advantage of this expected value by maximizing the number of coin flips.

For $p = 0.4$, the policy is a bit more intricate. The general trend is similar to a triangle wave with occasional spikes. In this case, the coin is biased against the gambler. As such, making many small bets is likely to result in losing. He will thus want to minimize the number of times he flips the coin. With a capital of $\$50$, the gambler can bet it all and have a 40%

chance of winning. With a capital of $\$51$, however, he has some additional room for misfortune. By betting $\$1$, the gambler can lose, placing him back at a capital of $\$50$ and still giving him a 40% chance of winning. If he wins, he will have obtained a capital of $\$52$, and thus can bet $\$2$, resulting in a capital of $\$54$ at which point he can continue this process, or a capital of $\$50$ at which point he has a 40% chance of winning. If the gambler is fortunate enough to reach a capital of $\$75$, he can now safely bet $\$25$ with the possibility of winning the game. This logic is what ultimately describes the relatively consistent policy with occasional spikes.

The case of $p = 0.25$follows similar to the case of $p = 0.4$, producing a triangular wave with occasional spikes. Again, the coin is not in the gambler's favor, and thus it is in his best interest to flip the coin as least times as possible. It should be noted that the number of spikes in the graph for $p = 0.25$ is less than that of the graph for $p = 0.4$.

Problem 2 [Based on AIMA 16.3] The St. Petersburg paradox goes as follows. A fair coin is tossed repeatedly until it comes up heads. If the first heads appears on the n^{th} toss, you win $\$2^n$. First, show that the expected monetary value of this game is infinite (the paradox is that no one would actually pay a huge amount to play this game). Second, consider a possible resolution of the paradox: suppose your utility for money is given by $a \log_2 x + b$ where x is the number of dollars you have. Suppose you start with 0 dollars, what is the expected utility of this game?

Solution:

We first show that the expected monetary value of this game is infinite. The probability that a fair coin lands heads up is $\frac{1}{2}$. Each coin toss is an independent event, thus the probability that the first heads appears on the first toss is $\frac{1}{2}$, on the second toss is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, on the third toss is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$, and so forth. The probability that the first heads appears on the nth toss is given by $\frac{1}{2^n}$. Now let us consider the payoffs. If the coin lands heads up for the first time on the first toss, we win $\$2$; on the second toss, we win $\$4$; on the third toss, we win $\$8$. More generally, if the coin lands heads up for the first time on the nth toss, we win $\$2^n$. The expected value is thus

$$\frac{1}{2}(\$2) + \frac{1}{4}(\$4) + \frac{1}{8}(\$8) + \cdots = \sum_{k=1}^{\infty} \frac{1}{2^n}(\$2^n) = \sum_{k=1}^{\infty} \$1 = \infty$$

Next, we consider a possible resolution of the paradox. Suppose the utility for money is given by $a \log_2 x + b$ where x is the number of dollars the player has. If the player starts with $\$0$, the expected utility of the game is given by

$$\sum_{k=1}^{\infty} \frac{1}{2^n}\big(\$(a \log_2 2^n + b)\big) = \sum_{k=1}^{\infty} \frac{1}{2^n}\big(\$(an + b)\big) = \$a \sum_{k=1}^{\infty} \frac{n}{2^n} + \$b \sum_{k=1}^{\infty} \frac{1}{2^n} = \$2a + \$b = \$(2a + b)$$

Michael Naper
CS 4804
2012.11.29
Problem Set 5

Problem 3 [Based on AIMA 16.17] You are considering buying a used car, which is being offered for $1500. It is worth $2000 to you if it is in good shape, and $1300 if it is in bad shape. There is a 70% chance that it is in good shape. Your decision is simple: buy the car, or don't. But, as you're examining it, a mechanic you know and trust drives by and sees you looking at the car. He tells you that he has a quick test he can do: if the car is in good shape, there is an 80% chance that it will pass the test, and if the car is in bad shape, there is only a 35% chance that it will pass the test. How much would you be willing to pay the mechanic to undertake the test? Give an explanation in layman's terms of why the value of information in this case comes out to what it does.

Solution:

We first calculate the expected net gain from buying the car given no test:

$$P(\text{good})(\$2000 - \$1500) + P(\text{bad})(\$1300 - \$1500) = 0.7(\$500) + 0.3(-\$200) = \$290$$

Next, we calculate the probability that the car passes or fails the test:

$$P(\text{passes}) = P(\text{passes}|\text{good})P(\text{good}) + P(\text{passes}|\text{bad})P(\text{bad}) = (0.8)(0.7) + (0.35)(0.3) = 0.665$$

$$P(\text{fails}) = 1 - P(\text{passes}) = 1 - 0.665 = 0.335$$

Using Bayes' theorem, we can calculate the probability that the car is in good or in bad shape based on whether or not it passes the test:

$$P(\text{good}|\text{passes}) = \frac{P(\text{passes}|\text{good})P(\text{good})}{P(\text{passes})} = \frac{(0.8)(0.7)}{(0.665)} = 0.842105$$

$$P(\text{bad}|\text{passes}) = \frac{P(\text{passes}|\text{bad})P(\text{bad})}{P(\text{passes})} = \frac{(0.35)(0.3)}{(0.665)} = 0.157895$$

$$P(\text{good}|\text{fails}) = \frac{P(\text{fails}|\text{good})P(\text{good})}{P(\text{fails})} = \frac{(0.2)(0.7)}{(0.335)} = 0.41791$$

$$P(\text{bad}|\text{fails}) = \frac{P(\text{fails}|\text{bad})P(\text{bad})}{P(\text{fails})} = \frac{(0.65)(0.3)}{(0.335)} = 0.58209$$

Now we can calculate the expected net gain from buying the car if it passes the test:

$$P(\text{good}|\text{passes})(\$2000 - \$1500) + P(\text{bad}|\text{passes})(\$1300 - \$1500) = (0.842105)(\$500) + (0.157895)(-\$200) = \$389.47$$

Thus you should buy the car. Similarly, we can calculate the expected net gain from buying the car if it fails the test:

$$P(\text{good}|\text{fails})(\$2000 - \$1500) + P(\text{bad}|\text{fails})(\$1300 - \$1500) = (0.41791)(\$500) + (0.58209)(-\$200) \\ = \$92.54$$

Once again, you should buy the car. Thus we see that you should buy the car no matter the result of the test. Consequently, the test is worthless, and you should not pay the mechanic any amount to undertake the test.

In order for the test to be worthwhile, either the test would need to be more reliable (that is, decrease the probability that the car will pass the test given that the car is in bad shape) or the car would need to be worth less at the same cost.

Problem 4 [Based on AIMA 17.10] Consider an undiscounted MDP with three states, $(1, 2, 3)$ with rewards $-1, -2, 0$ respectively, and state 3 as a terminal state. In states 1 and 2, there are two possible actions, a and b. The transition model is as follows:

- In state 1, a transitions to state 2 with probability 0.8 and stays in state 1 with probability 0.2.
- In state 2, a transitions to state 1 with probability 0.8 and stays in state 2 with probability 0.2.
- In either state 1 or state 2, b transitions to state 3 with probability 0.1 and stays in the original state with probability 0.9.

Answer the following questions:

(a) Before doing any math, what can you say qualitatively about the optimal policy in states 1 and 2?
(b) Apply policy iteration, starting with the policy that takes action b in both states, to determine the optimal policy and the values of states 1 and 2. Show each step in full.
(c) Repeat part (b), but starting with the policy that takes action a in both states. What is the problem here? Does discounting help? How does the discount factor affect the optimal policy?

Solution:

(a) The optimal policy will attempt to maximize the reward, and so the agent will want to get into state 3 as quickly as possible. However, since the probability of transitioning to state 3 is low, the agent will want to minimize the cost incurred as it continuously attempts to transition to state 3. Thus, if the agent is in state 1, it should continue to choose action b in the hopes of transitioning to state 3. Choosing action a would never yield a higher reward. On the other hand, if the agent is in state 2, failing to transition to state 3 will result in a higher penalty than failing to transition when in state 1. Since the probability of transitioning to state 1 is greater than the probability of transitioning to state 3, the agent should first attempt to transition to state 1 by choosing action a. Once it is in state 1, it would then attempt to transition to state 3 by choosing action b.

(b) Initialization:

$U_0 = \langle 0, 0, 0 \rangle$
$\pi_1 = \langle b, b, \text{NULL} \rangle$

Iteration 1:

Policy evaluation:

$$\begin{cases} U_1(1) = -1 + 0.9U_1(1) + 0.1U_1(3) \\ U_1(2) = -2 + 0.9U_1(2) + 0.1U_1(3) \\ U_1(3) = 0 \end{cases} \Rightarrow \begin{cases} U_1(1) = -10 \\ U_1(2) = -20 \\ U_1(3) = 0 \end{cases}$$

Policy update:

State 1:

$\sum_{s'} P(s' | 1, a) U_1(s') = (0.2)(-10) + (0.8)(-20) = -18$
$\sum_{s'} P(s' | 1, b) U_1(s') = (0.9)(-10) + (0.1)(0) = -9$

$\therefore$ Action b remains optimal

State 2:

$\sum_{s'} P(s' | 2, a) U_1(s') = (0.2)(-20) + (0.8)(-10) = -12$
$\sum_{s'} P(s' | 2, b) U_1(s') = (0.9)(-20) + (0.1)(0) = -18$

$\therefore$ Action a becomes optimal

$\pi_2 = \langle b, a, \text{NULL} \rangle$

Iteration 2:

Policy evaluation:

$$\begin{cases} U_2(1) = -1 + 0.9U_2(1) + 0.1U_2(3) \\ U_2(2) = -2 + 0.8U_2(1) + 0.2U_2(2) \\ U_2(3) = 0 \end{cases} \Rightarrow \begin{cases} U_2(1) = -10 \\ U_2(2) = -12.5 \\ U_2(3) = 0 \end{cases}$$

Policy update:

State 1:

$\sum_{s'} P(s'|\, 1, a) U_1(s') = (0.2)(-10) + (0.8)(-12.5) = -12$
$\sum_{s'} P(s'|\, 1, b) U_1(s') = (0.9)(-10) + (0.1)(0) = -9$

$\therefore$ Action b remains optimal

State 2:

$\sum_{s'} P(s'|\, 2, a) U_1(s') = (0.2)(-12.5) + (0.8)(-10) = -10.5$
$\sum_{s'} P(s'|\, 2, b) U_1(s') = (0.9)(-12.5) + (0.1)(0) = -11.25$

$\therefore$ Action a remains optimal

$\pi_3 = \langle b, a, \text{NULL} \rangle$

Therefore, optimal policy is $\pi^* = \langle b, a, \text{NULL} \rangle$.

(c) Initialization:

$U_0 = \langle 0, 0, 0 \rangle$
$\pi_1 = \langle a, a, \text{NULL} \rangle$

Iteration 1:

Policy evaluation:

$$\begin{cases} U_1(1) = -1 + 0.2U_1(1) + 0.8U_1(2) \\ U_1(2) = -2 + 0.8U_1(1) + 0.2U_1(2) \\ U_1(3) = 0 \end{cases}$$

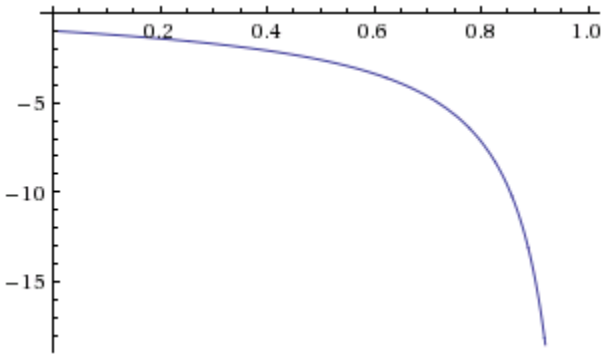
Attempting to perform the policy evaluation on the first iteration leads to an inconsistent system of linear equations. Thus the system has no solution, and we cannot solve for the utilities of states 1 and 2 when following policy π_1.

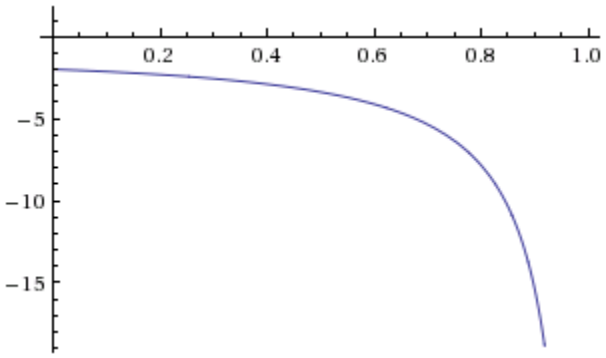
Introducing discounting, however, will allow for this system to become consistent:

$$\begin{cases} U_1(1) = -1 + \gamma[0.2U_1(1) + 0.8U_1(2)] \\ U_1(2) = -2 + \gamma[0.8U_1(1) + 0.2U_1(2)] \\ U_1(3) = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} U_1(1) = \dfrac{7\gamma + 5}{(\gamma - 1)(3\gamma + 5)} \\ U_1(2) = \dfrac{2(\gamma + 5)}{(\gamma - 1)(3\gamma + 5)} \\ U_1(3) = 0 \end{cases}$$

Note that when the discount factor $\gamma = 1$, the utilities of states 1 and 2 become infinite, reducing to our original problem of no solution.

Plotting $U_1(1)$ and $U_1(2)$ shows the behavior of the utility estimates as a function of γ:





Our choice of γ, however, will affect the importance of the distant future in our computation. For small values of γ, we rely more heavily on the immediate future. For this scenario, such an approach might lead to a policy that insists on the agent choosing action b from state 2. If the benefit of moving to state 3 outweighs the discounted penalty of remaining in state 2, policy iteration will result in a fixed point of $\pi = \langle b, b, \text{NULL} \rangle$ despite this being not the optimal policy.

Michael Naper
CS 4804
2012.11.29
Problem Set 5

Problem 5 Write out the payoff matrix for the game "Rock, Paper, Scissors", in which both players simultaneously call out (or indicate with their hands) one of these three options. Scissors beats Paper, Paper beats Rock, and Rock beats Scissors (and the same option selected by both players results in a tie). What is a mixed-strategy equilibrium for this game?

Solution:

PAYOFF MATRIX

	O: *rock*	O: *scissors*	O: *paper*
E: *rock*	$E = 0, O = 0$	$E = +1, O = -1$	$E = -1, O = +1$
E: *scissors*	$E = -1, O = +1$	$E = 0, O = 0$	$E = +1, O = -1$
E: *paper*	$E = +1, O = -1$	$E = -1, O = +1$	$E = 0, O = 0$

Since the game is symmetric, it suffices to find a mixed-strategy for one player. Let $P(\mathrm{r})$ be the probability that Player E chooses rock, $P(\mathrm{s})$ be the probability that Player E chooses scissors, and $P(\mathrm{p}) = 1 - P(\mathrm{r}) - P(\mathrm{s})$ be the probability that Player E chooses paper. Then the expected value for Player E for each choice that Player O can make is:

$$\begin{aligned} E[O\text{: } rock] &= P(\mathrm{r}) \cdot (0) + P(\mathrm{s}) \cdot (-1) + P(\mathrm{p}) \cdot (+1) = P(\mathrm{p}) - P(\mathrm{s}) = -P(\mathrm{r}) - 2P(\mathrm{s}) + 1 \\ E[O\text{: } scissors] &= P(\mathrm{r}) \cdot (+1) + P(\mathrm{s}) \cdot (0) + P(\mathrm{p}) \cdot (-1) = P(\mathrm{r}) - P(\mathrm{p}) = 2P(\mathrm{r}) + P(\mathrm{s}) - 1 \\ E[O\text{: } paper] &= P(\mathrm{r}) \cdot (-1) + P(\mathrm{s}) \cdot (+1) + P(\mathrm{p}) \cdot (0) = -P(\mathrm{r}) + P(\mathrm{s}) \end{aligned}$$

A mixed-strategy equilibrium for this game are values of $P(\mathrm{r})$, $P(\mathrm{s})$, and $P(\mathrm{p})$ such that $E[O\text{: } rock] = E[O\text{: } scissors] = E[O\text{: } paper]$.

$$\begin{cases} E[O\text{: } rock] = E[O\text{: } paper] \\ E[O\text{: } scissors] = E[O\text{: } paper] \end{cases} \rightarrow \begin{cases} -P(\mathrm{r}) - 2P(\mathrm{s}) + 1 = -P(\mathrm{r}) + P(\mathrm{s}) \\ 2P(\mathrm{r}) + P(\mathrm{s}) - 1 = -P(\mathrm{r}) + P(\mathrm{s}) \end{cases} \rightarrow \begin{cases} P(\mathrm{s}) = 1/3 \\ P(\mathrm{r}) = 1/3 \end{cases}$$

Thus a mixed-strategy equilibrium is one such that all three options are selected with equal probability.

Appendix A: Code for MDP Value Iteration

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MDPValueIteration {

  private static final int GOAL_CAPITAL = 100;
  private static final int FAIL_CAPITAL = 0;
  private static final double TOLERANCE = 0;

  private final MDP mdp;

  public static void main(String[] args) throws IOException {
    createCSVFile(new MDPValueIteration(0.25).valueIteration(), "p25.txt");
    createCSVFile(new MDPValueIteration(0.40).valueIteration(), "p40.txt");
    createCSVFile(new MDPValueIteration(0.55).valueIteration(), "p55.txt");
  }

  private static void createCSVFile(ValueIterationRetVal retVal, String fileName)
      throws IOException {
    List<State> states = retVal.states;
    Map<State, Action> policy = retVal.policy;
    Map<State, Double> utilities = retVal.utilities;
    PrintWriter printWriter = new PrintWriter(fileName);
    printWriter.println("State,Utility,Policy");
    for (State state : states) {
      printWriter.println(state + "," + utilities.get(state) + ","
          + policy.get(state));
    }
    printWriter.flush();
    printWriter.close();
  }

  public MDPValueIteration(double p) {
    mdp = new MDP(p);
  }

  public ValueIterationRetVal valueIteration() {
    Map<State, Double> currUtilities, nextUtilities;
    currUtilities = new HashMap<State, Double>();
    for (State state : mdp.states) {
      currUtilities.put(state, 0d);
    }
    currUtilities.put(MDP.GOAL_STATE, 1d);
    Map<State, Action> policy = new HashMap<State, Action>();
    double delta;
    do {
      nextUtilities = new HashMap<State, Double>();
      delta = 0;
      for (State state : mdp.states) {
        if (state.equals(MDP.GOAL_STATE) || state.equals(MDP.FAIL_STATE)) {
```

```
        nextUtilities.put(state, currUtilities.get(state));
        continue;
      }
      Action optimalAction = optimalAction(state, currUtilities);
      policy.put(state, optimalAction);
      nextUtilities.put(state, mdp.reward(state) + mdp.discountFactor
          * transitionUtility(state, optimalAction, currUtilities));
      double stateDelta = Math.abs(nextUtilities.get(state)
          - currUtilities.get(state));
      if (stateDelta > delta) {
        delta = stateDelta;
      }
    }
    currUtilities = nextUtilities;
  } while (delta > TOLERANCE);
  return new ValueIterationRetVal(new ArrayList<State>(mdp.states), policy,
      currUtilities);
}

private Action optimalAction(State currState, Map<State, Double> utilities) {
  Action optimalAction = null;
  double optimalUtility = 0;
  for (Action action : mdp.actions(currState)) {
    double transitionUtility = transitionUtility(currState, action, utilities);
    if (transitionUtility > optimalUtility || optimalAction == null) {
      optimalAction = action;
      optimalUtility = transitionUtility;
    }
  }
  return optimalAction;
}

private double transitionUtility(State currState, Action action,
    Map<State, Double> utilities) {
  double transitionUtility = 0;
  for (State successor : currState.apply(action)) {
    transitionUtility += mdp.transitionModel(successor, currState, action)
        * utilities.get(successor);
  }
  return transitionUtility;
}

private static class MDP {

  static final State GOAL_STATE = new State(GOAL_CAPITAL);
  static final State FAIL_STATE = new State(FAIL_CAPITAL);

  final double p;
  final List<State> states;
  final double discountFactor;

  MDP(double p, double discountFactor) {
    this.p = p;
    states = new ArrayList<State>(GOAL_CAPITAL + 1);
    for (int capital = 0; capital <= GOAL_CAPITAL; capital++) {
      states.add(new State(capital));s
```

```
      }
      this.discountFactor = discountFactor;
    }

    MDP(double p) {
      this(p, 1);  // undiscounted
    }

    List<Action> actions(State state) {
      List<Action> actions = new ArrayList<Action>(state.capital);
      int maxStake = Math
          .min(state.capital, GOAL_STATE.capital - state.capital);
      for (int stake = 1; stake <= maxStake; stake++) {
        actions.add(new Action(stake));
      }
      return actions;
    }

    double transitionModel(State successor, State current, Action action) {
      if (successor.equals(new State(current.capital + action.stake))) {
        return p;
      } else if (successor.equals(new State(current.capital - action.stake))) {
        return 1 - p;
      } else {
        return 0;
      }
    }

    int reward(State state) {
      return state.equals(GOAL_STATE) ? 1 : 0;
    }
  }

  private static class State {

    final int capital;

    State(int capital) {
      this.capital = capital;
    }

    List<State> apply(Action action) {
      List<State> successors = new ArrayList<State>();
      successors.add(new State(capital + action.stake));
      successors.add(new State(capital - action.stake));
      return successors;
    }

    @Override
    public int hashCode() {
      return Integer.valueOf(capital).hashCode();
    }

    @Override
    public boolean equals(Object o) {
      if (o == this) {
```

```
        return true;
      }
      if (!(o instanceof State)) {
        return false;
      }
      return this.capital == ((State) o).capital;
    }

    @Override
    public String toString() {
      return Integer.toString(capital);
    }
  }

  private static class Action {

    final int stake;

    Action(int stake) {
      this.stake = stake;
    }

    @Override
    public String toString() {
      return Integer.toString(stake);
    }
  }

  private static class ValueIterationRetVal {

    final List<State> states;
    final Map<State, Action> policy;
    final Map<State, Double> utilities;

    ValueIterationRetVal(List<State> states, Map<State, Action> policy,
        Map<State, Double> utilities) {
      this.states = states;
      this.policy = policy;
      this.utilities = utilities;
    }
  }
}
```