Artificial Intelligence	**Assignment 3**
comp440/comp557 Fall 2018	**due 1 October 2018**

This assignment is due October 1 at 8 pm on Canvas. Download `assignment3.zip` *from Canvas. There are four problems worth a total of 120 points + 20 extra credit points for comp440 and a total of 140 points for comp557 students. Problems 1, 2 and 4 require written work only, Problem 3 requires Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. Problem 4 part (a) is best solved with a small Python program which should be turned in as* `prob4.py`*. All code for Problem 3 should be included in* `submission.py` *at the labeled points. For Problem 3, please run the auto grader using the command line* `python grader.py` *and report the results in your writeup. Submit* `writeup.pdf`*,* `submission.py` *and* `prob4.py` *(if applicable) as separate files on Canvas by the due date/time. One submission per group, please.*

1 Nash equilibria (20 points)

- (10 points) The following payoff matrix shows a game between politicians and the Federal Reserve.

	Fed: contract	Fed: do nothing	Fed: expand
Pol: contract	F = 7, P = 1	F = 9, P = 4	F = 6, P = 6
Pol: do nothing	F = 8, P = 2	F = 5, P = 5	F = 4, P = 9
Pol: expand	F = 3, P = 3	F = 2, P = 7	F = 1, P = 8

  Politicians can expand or contract fiscal policy, while the Fed can expand or contract monetary policy. And of course, both sides can choose to do nothing. Each side has preferences for who should do what – neither side wants to look like the bad guys. The payoffs shown in the matrix are rank orderings: 9 for first choice through 1 for last choice.

  - Find the Nash equilibrium of the game in pure strategies.
  - Is this a Pareto-optimal solution? You might wish to analyze the policies of recent US administrations in this light

- (10 points) In the children's game of rock-paper-scissors each player reveals at the same time a choice of rock, paper or scissors. Paper wraps rock, rock blunts scissors, and scissors cut paper. In the extended version rock-paper-scissors-fire-water, fire beats rock, paper and scissors; rock, paper and scissors beat water; and water beats fire. Write out the payoff matrix for this game and find a mixed-strategy solution to the game.

2 Policy iteration (25 points)

Consider an MDP with three states 1, 2 and 3 with rewards r of -1, -2 and 0 respectively. State 3 is 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, action a moves the agent to state 2 with probability 0.8, and makes the agent stay in state 1 with probability 0.2.
- In state 2, action a moves the agent to state 1 with probability 0.8, and makes the agent stay in state 2 with probability 0.2.
- In states 1 and 2, the action b moves the agent to state 3 with probability 0.1 and makes the agent stay put with probability 0.9.

Answer the following questions about this MDP.

- (3 points) Without actually solving the MDP, what can you say about the optimal policy at states 1 and 2?
- (10 points) Assume the initial policy is b in states 1 and 2. Apply policy iteration to determine the optimal policy for states 1 and 2. Show your work in full, including the policy evaluation and policy update steps.
- (5 points) Assume the initial policy is a in states 1 and 2. Apply policy iteration as in the previous part. Can you solve for the optimal policy with this starting point? Why?
- (7 points) Does the inclusion of a discount factor $\gamma < 1$ allow policy iteration to work with an initial policy of a in states 1 and 2? Compute one round of policy iteration (policy evaluation + policy update) for this initial policy with discount factors of 0.9, and then 0.1. What are the policies at the end of the first round of computation for these two discount factors?

3 MDPs and peeking blackjack (75 points)

Markov decision processes (MDPs) can be used to formalize uncertain situations where the goal is to maximize some kind of reward. In this problem, you will implement the algorithms that can be used to automatically construct an optimal policy for such situations. You will then formalize a modified version of Blackjack as an MDP, and apply your algorithm to come up with an optimal policy.

3.1 Problem 1: Solving MDPs (55 points)

3.1.1 Computing Q from value function V (5 points)

As a warmup, we'll start by implementing the computation of Q from V, filling out the `computeQ()` function in `submission.py`. Recall that $V(s)$ is the value (expected utility) starting at state s, given some policy. Given a value function, we can define $Q(s, a)$, the expected utility received when performing action a in state s.

$$Q(s,a) = \sum_{s'} T(s,a,s')[reward(s,a,s') + \gamma V(s')]$$

In this equation, the transition probability $T(s, a, s')$ is the probability of ending up in state s' after performing action a in state s, $reward(s, a, s')$ is the reward when you end up in state s' after

performing action a in state s, and γ is the discount factor, which is a parameter indicating how much we devalue rewards from future states. Intuitively, $V(s)$ represents the value of a state, and $Q(s, a)$ represents how valuable it is to perform a particular action in a particular state. We will use the `computeQ` function in building the policy iteration algorithm.

3.1.2 Policy evaluation (15 points)

Policy iteration proceeds by alternating between (i) finding the value of all states given a particular policy (policy evaluation) and (ii) finding the optimal policy given a value function (policy improvement). We will first implement policy evaluation by filling out the function `policyEvaluation()` in `submission.py`. Given a policy π, we compute the value $V_\pi(s)$ of each state s in our MDP. We use the Bellman equation iteratively:

$$V_\pi^{(t)}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[reward(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]$$

where $V_\pi^{(t)}$ is the estimate of the value function of policy π after t iterations. We repeatedly apply this update until $V_\pi^{(t)}(s) \approx V_\pi^{(t-1)}(s)$, for every state s.

3.1.3 Extracting the optimal policy from value function V (5 points)

Next, we compute the optimal policy given a value function V, in the function `computeOptimalPolicy()` in `submission.py`. This policy simply selects the action that has maximal value for each state.

$$\pi(s) = argmax_{a \in Actions(s)} Q(s, a)$$

3.1.4 Policy iteration (10 points)

Once we know how to construct a value function given a policy, and how to find the optimal policy given a value function, we can perform policy iteration. Fill out the `solve()` function in class `PolicyIteration` in `submission.py`. Start with a value function that is for 0 all states, and then alternate between finding the optimal policy for your current value function, and finding the value function for your current policy. Stop when your optimal policy stops changing.

3.1.5 Value iteration (10 points)

As an alternative to performing a full policy evaluation in each iteration, as in policy iteration, we can replace it with a single step of policy evaluation. That is, we first find $\pi(s)$ with respect to $V^{(t-1)}$ for every non-terminal state s, and use the equation below **once** to find $V^{(t)}$.

$$V^{(t)}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[reward(s, \pi(s), s') + \gamma V^{(t-1)}(s')]$$

We alternate between finding the optimal policy for the current value function, and doing a *single step* of policy evaluation. Stop when the new value function $V^{(t)} \approx V^{(t-1)}$. This algorithm is called value iteration. Implement it in the `solve()` function in class `ValueIteration` in `submission.py`.

3.1.6 Noisy transition model (10 points)

If we add noise to the transitions of an MDP, does the optimal value get worse? Specifically, consider an MDP with reward function $reward(s, a, s')$, state space S, and transition function $T(s, a, s')$. We define a new MDP which is identical to the original, except for its transition function $T'(s, a, s')$ defined as

$$T'(s, a, s') = \frac{T(s, a, s') + \alpha}{\sum_{s' \in S}[T(s, a, s') + \alpha]}$$

for some $\alpha > 0$. Let V_1 be the optimal value function for the original MDP, and let V_2 be the optimal value function for the MDP with added uniform noise. Is it always the case that $V_1(s_0) \geq V_2(s_0)$ where s_0 is the start state? If so, prove it in `writeup.pdf` and put `return None` for each of the code blocks for this problem in `submission.py`. Otherwise, construct a counterexample by filling out `CounterexampleMDP` and `counterexampleAlpha()` in `submission.py`. This problem is manually graded – there is no test case for it in the auto grader.

3.2 Problem 2: Peeking blackjack (20 points)

Now that we have written general-purpose MDP algorithms, let us use them to play (a modified version of) Blackjack. For this problem, you will be creating an MDP to describe a modified version of Blackjack. For our version of Blackjack, the deck can contain an arbitrary collection of cards with different values, each with a given multiplicity. For example, a standard deck would have card values $\{1, 2, \ldots, 13\}$ and multiplicity 4. However, you could also have a deck with card values $\{1, 5, 20\}$, or any other set of numbers. The deck is shuffled (each permutation of the cards is equally likely). The game occurs in a sequence of rounds. Each round, the player either

- takes a card from the top of the deck (costing nothing)
- peeks at the top card (costing peekCost, in which case the next round, that card will be drawn)
- quits the game

Note that it is not possible to peek twice; if the player peeks twice in a row, then `succAndProbReward()` should return []. The game continues until one of the following conditions becomes true:

- The player quits, in which case her reward is the sum of the cards in her hand.
- The player takes a card, and this leaves her with a sum that is greater than the threshold, in which case her reward is 0.
- The deck runs out of cards, in which case it is as if she quits, and she gets a reward which is the sum of the cards in her hand.

As an example, assume the deck has card values $\{1, 5\}$, with multiplicity 2. Let us say the threshold is 10. Initially, the player has no cards, so her total is 0. At this point, she can peek, take, or quit. If she quits, the game is over and she receives a reward of 0. If she takes the card, a card will be selected from the deck uniformly at random. Assuming the card is a 5, then her total is 5, and the

deck would then contain two 1's and one 5. If she peeks, then the deck remains the same, and she still has no cards in her hand, but on the next round she is allowed to make her decision using her knowledge of the next card.

Let us assume she peeks and the card is a 5. Then her hand still contains no cards, and on the next round, she is faced with the same choice of peek, take or quit. If she peeks again, then the set of possible next states is empty. If she takes, then the card will be a 5, and the deck will be left with two 1's and one 5.

3.2.1 Implementing blackjack as an MDP (15 points)

Implement the game of Blackjack as an MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP` in `submission.py`. To help out out, we have already given you `startState()`.

Hint: For the implementation of `succAndProbReward` there are two special behaviors that the grader looks for: on most Quits, a single tuple for the next state should be returned, but if the action is Quit and `state[2]` is already (0,), then only an empty array should be returned. On most Takes, the reward is 0. But if a Take consumes the last card, then the reward actually needs to be the player's final score.

3.2.2 Engineering MDPs for specific policies (5 points)

Let's say you're running a casino, and you're trying to design a deck to make people peek a lot. Assuming a fixed threshold of 20, and a peek cost of 1, your job is to design a deck where for at least 10% of states, the optimal policy is to peek. Fill out the function `peekingMDP()` in `submission.py` to return an instance of `BlackjackMDP` where the optimal action is to peek in at least 10% of states.

4 MDPs and value iteration (20 points) (optional for comp440/required for comp557)

- (10 points) Consider the 3×3 world shown in Figure 17.14 (a) in your textbook (page 690). The transition model is the same as in the 4×3 world of Figure 17.1 shown in class: 80% of the time the agent goes in the direction it selects, the rest of the time it moves at right angles to the intended direction (equally likely in both right angle directions). Implement value iteration for this world with each value of r below. use discounted rewards with a discount factor of 0.99. Show the policy obtained in each case. Explain intuitively why the value of r leads to each policy.
  - $r = 100$
  - $r = -100$
  - $r = -3$
  - $r = 0$
  - $r = +3$

- (10 points) Consider the 101×3 world shown in Figure 17.14 (b) of your textbook (page 690). In the start state the agent has a choice of two deterministic actions, up or down; but in the other states the agent has one deterministic action: right. Assuming a discounted reward function, for what values of the discount γ should the agent choose up and for what values should the agent choose down? Compute the utility of each action as a function of γ. Note that this simple example reflects many real-world situations in which one must weigh the immediate action versus the potential continual long-term consequences, such as choosing to dump pollutants into a lake.