In this project, we implemented Value Iteration and Q-learning, and tested our agents first on GridWorld and later to a simulated robot controller (Crawler) Pacman. Reinforcement learning is an agent that takes actions on an environment, and the output state and reward from the action are passed back to the agent that took the action. It is a Markov decision process, non-determininsitic. You have a set of states with a set of associated actions, and a transition function, a probability of one of the successor outcomes, a model of the dynamics, along with the reward of the associated transition. Markov means generally that given the present state, the future and past are independent. For decision processes, Markov means action outcomes depend only on the current state. This is just like search, where the successor function could only depend on the current state and not its history. In deterministic single agent search problems, we wanted an optimal plan, or sequence of actions, from start to goal. For Markov Decision processes, we want an optimal policy for each state. Expectimax computed action for a single state only, which means you can redo a lot of work if you end up in the same state, but is good if you have so many states, you could not write down an explicit policy. An optimal policy is one that maximizes expected utility if followed, and gives an action for each state. In Grid World, actions do not always go as planned. 80% of the time, the action North takes the agent North, but 10% of the time, North takes the agent West, and 10% East. If there is a wall in the direction of the agent would have taken, the agent stays put. The agent receives rewards each time step, a small living reward, with big rewards at the end. The goal is to maximize the sum of discounted rewards. Value (V) = utility of a state s = expected value of starting in a state and acting optimally. The value (Q) = utility of state q = expected utility starting out having taken action a from state s and thereafter, acting optimally. The optimal policy = pi = optimal action from state s. The fundamental operation: compute the expectimax value of each state recursively - the expected utility under optimal action, the average sum of discounted rewards, which is what expectimax computes for each state. The recursive definition builds from the bottom of the tree of game states and their successors. It defines an expectimax probabilistic value after the game ends after a certain number of steps, since sometimes trees can go on forever. It is a weighted average Bellman Ford recursive algorithm that is non-linear, rewards go to the next time step and is also discounted each step, with each state having 4 q-values - a maximum chance over the successor nodes to go to, and a probability of starting in that state and acting optimally. The recursive algorithm continues for a certain number of steps and is repeated until it converges. Note that rewards are instantaneous each time step, but values are cumulative. With Bellman equations, you take the correct first action, and keep being optimal. Policies (actions from states) simplify this since there is only one action per state. The values depend on the policy, good or bad, but you don't have to maximize it, so it is easier. So there is an alternative approach for optimal values. Calculate utilities for some fixed policy (not optimal utilities) until convergence. Then improve the policy using a one step look ahead with utility results converged (but not optimal) as future values. This policy iteration can converge fast where there are a large number of actions, but only where maximizing evaluation does not change much. In value iteration, every iteration updates both the values and implicitly the policy. We do not track the policy, but taking the max over actions implicitly recomputes it. In policy iteration, it is complex but faster. We do several passes that update utilities with fixed policy (consider just one action). After the policy is evaluated, a new policy is chosen slowly, like a value iteration pass. The new policy will be better or we are done. Value and Policy iteration are dynamic programs for Markov Decision processes. So to compute optimal values: use value iteration or policy iteration. Compute values for a particular policy using policy evaluation. Values are turned into a policy using policy extraction (one-step-look-ahead). They are all variations of Bellman updates using one-step-look-ahead expectimax (probabilities) fragments. They differ in whether we plug in a fixed policy or a max over actions. So the basic idea of this reinforcement learning assignment was to: receive feedback in the form of rewards, an agent's utility is defined by the reward function, we must learn to act so as to maximize expected rewards, and all learning is based on observed samples of outcomes. The difference with reinforcement learning to the above described Markov Decision processes were that you do not know which states are good or what actions to do. We must actually try actions and states out to learn from them. We first wrote a value iteration agent that was an offline planner rather than a reinforcement agent, so the relevant training option is the number of iterations of value iteration it should run in its initial planning phase. It takes a Markov Decision process and runs value iteration for the specified number of steps. It computes the best action from a given state according to the value function. It also returns the q-value of the (state, action) pair given by the value function. We used a value iteration where each vector V(k) is computed from a fixed vector V(k-1) and not a value iteration where each single weight vector is updated in place. This meant that when a state's value was updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration (k-1). A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next (k+1) rewards (returning policy k+1). Similarly the q-values for a game state (possible successor game states) will also reflect one more reward than the values (returning Q(k+1)). The next part of this looked at a Bridge Grid, a grid world map with the low reward terminal state and a high reward terminal state separated by a narrow bridge on either side of which is a chasm of high negative reward. The agent starts near the low reward state with a certain discount and noise value and the optimal policy does not cross the bridge. We then changed one of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Noise refers to how often an agent ends up in an unintended successor state when they perform an action. We then considered a DiscountGrid layout which had 2 terminal states with a positive payoff, a close exit with a payoff of 1, and a distant exit with a payoff of 10. The bottom row of the grid consisted of terminal states with negative payoff -10. Some paths risk the cliff and travel near the bottom row of the grid. These paths are shorter but risk earning a large negative payoff. Paths that avoid the cliff and travel along the top edge of the grid are longer but less likely to incur huge negative payoffs. In this part of the project, we chose settings of the discount, noise and living reward parameters for this Markov Decision Process to produce optimal policies of several different types. The setting of the parameter value for each part should have the property that if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. If a particular behavior is not achieved for any setting of the parameters, we asserted that the policy is impossible by returning the "not possible" string. Grading was by checking that the desired policy was returned in each test case. Note that the Value iteration agent did not learn from experience, but it pondered its Markov Decision Process model to arrive at a complete policy before ever interacting with the real environment. When it does interact with the environment, it simply follows the precomputed policy (becomes a reflex agent). This distinction may be subtle in a simulated environment like GridWorld, but it's very important in the real world, where the real Markov Decision Process is not available. We then wrote a Q-agent, which did very little on construction, but instead learns by trial and error from interactions with its environment through its update (state, action, nextState, reward) method. Ties were broken randomly for better behavior. In a particular state, actions your agent has not seen before but still have a q-value, specifically a q-value of zero, and if all actions your agent has seen before have a negative q-value, then an unseen action may be optimal. We watched how our agents learned about the state it was just in, not the one it moves to, and leaves learning in its wake. Grading was based on running your q-agent and checking that it learns the same q-values and policy as the reference implementation when each is presented with the same set of examples. The next part completed the q-learning agent by implementing epsilon-greedy-action selection, meaning it chooses random actions an epsilon fraction of the time, and follows its current best q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action. The final q-values should resemble those of the Value Iteration Agent, especially along the well-traveled paths. However, the average returns will be lower than the q-values

predict because of the random actions and the initial learning phase. A choice from a list is made uniformly at random. A binary variable with probability p of success is simulated by using a flip-coin function, which returns True with probability p and False with probability (1-p). If code is written too specifically to the GridWorld, it would not work, since it is supposed to be written in general for all Markov Decision Processes. It was graded by invoking a Crawler Robot they provided using your q-learner agent. We had to play around with various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, where as the learning rate and epsilon are parameters of the learning algorithm, and the discount factor is a property of the environment. We then had to train a completely random q-learner agent with the default learning rate on the noiseless BridgeGrid, for 50 episodes and observe whether it finds the optimal policy. We then tried the experiment with an epsilon of 0. We had to determine whether there was an epsilon and learning rate for which it is highly likely that the optimal policy will be learned after 50 iterations. This part of the project returned a tuple of either (epsilon, learning rate) or the string "not possible". The response should not depend on the exact tie-breaking mechanism used to choose actions. This meant our answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees. Then we moved from the Grid World to the Pacman World. In the first phase, training, Pacman began to learn about the values of positions and actions. Because it takes a very long time to learn accurate q-values even for tiny grids, Pacman's training games run in quiet mode by default (no noise). Once Pacman's training is complete, he will enter testing mode, and his epsilon and alpha will be set to zero, stopping the q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Pacman Agent was already defined by the Q-learning-Agent. His only difference was that it has default learning parameters that are more effective for the Pacman problem (epsilon, alpha, gamma). Grading was based on winning 80% of the time. During training, we output statistics on how Pacman was doing. Epsilon was positive during training, so Pacman played poorly even after having learned a good policy. This was because he occasionally makes a random exploratory move into a ghost. As a bench mark, it took 1000-1400 games before Pacman's rewards for a 100 episode segment becomes positive, reflecting that he's started winning more than losing. By the end of the training, rewards should remain positive and be fairly high. To understand what is happening here: the Markov Decision Process state is the exact board configuration facing Pacman, with the now complex transition describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are not Markov Decision Process states, but are bundled in to the transitions. Once Pacman was done training, he should win very reliably in test games, since now he is exploiting his learned policy. This was true for a small Pacman grid, but not for a medium sized Pacman grid. On the medium grid, Pacman's average training rewards remain negative, and at test time, he plays badly, probably losing all of his test games. Training also takes a long time on the medium grid, despite its ineffectiveness. Pacman fails to learn on larger layouts because each board configuration is a separate state with separate q-values. He has no way to generalize that running into a ghost is bad for all positions. This approach does not scale for large Pacman grids. The last part of project 3 was to implement a q-learning agent that learns weights for features of states, where many states might share the same features. It is called an Approximate-Q-Agent. Approximate q-learning assumes the existence of a feature function over state and action pairs, which yields a vector F(1)(s,a), F(2)(s,a) … F(n)(s,a) of feature values. Feature vectors are dictionary objects containing the non-zero pairs of features and values, and all omitted features have value zero. The approximate q-learning function is the weight associated with a particular feature f(i)(s,a). The code implemented the weight vector as a dictionary mapping feature to weight values. Weight vectors were updated similarly to how you update q-values. Approximate-q-agent uses the an extractor to assign a single feature to every (state,action) pair. With this feature extractor, the approximate q-learning agent should work identically to Pacman-q-agent. Once we were confident that your approximate learner works correctly with the identity features, we had to run our approximate-q-learning agent with our custom feature extractor, which can learn to win with ease. Even much larger layouts should be no problem for approximate-q-learning agent. It was graded such that you had no errors if your approximate-q-learning agent should win almost every time with simple features, with 50 training games. They ran the approximate-q-learning agent and checked that it learned the same q-values and feature weights as a reference implementation when each is presented with the same set of examples. We then had a learning Pacman agent. Although I learned from doing this project, I got a correctly working solution when completed from a class mate.