Project 3: Reinforcement Learning

Submit Assignment

Due Thursday by 11:59pm **Points** 23 **Submitting** a file upload **File Types** py
Available until Nov 8 at 11:59pm

Introduction

Download project code: p3_reinforcement.zip

In this project, you will implement value iteration and Q-learning. You will test your agents first on Gridworld (from class), then apply them to a simulated robot controller (Crawler) and Pacman.

An autograder is included in the code. You can run the autograder from the command line with the following prompt.

```
python3 autograder.py
```

The points you receive from the autograder reflect what you will receive for each question in this project. Keep in mind that not all questions in the autograder may be required. Be sure to consult the rubric for a breakdown of what is needed to complete the project.

You can also autograde a single question using the `-q` option with an argument of the form `q#`. See the example below, which only autogrades question 1.

```
python3 autograder.py -q q1
```

By default, the autograder displays graphics with the `-t` option, but doesn't with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.

Files that you will edit and submit

- `valueIterationAgents.py` - A value iteration agent for solving known MDPs
- `qLearningAgents.py` - Q-learning agents for Gridworld, Crawler and Pacman
- `analysis.py` - A file to put your answers to questions given in the project

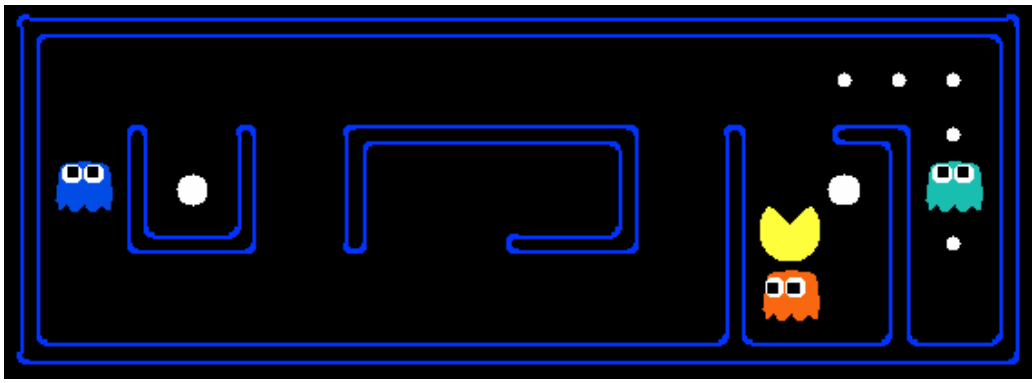
Files that you may want to look at

- `mdp.py` - Defines methods on general MDPs
- `learningAgents.py` - Defines the base classes `ValueEstimationAgent` and `QLearningAgent`, which your agents will extend
- `util.py` - Utilities, including `util.Counter`, which is particularly useful for Q-learners
- `gridworld.py` - The Gridworld implementation

- `featureExtractors.py` - Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in `qlearningAgents.py`)

You are welcome to look at the other files included in the project, but you should only need to read through the files specified above in order to complete the project.

Do not edit any files other than the ones that you will submit. Your code will be graded against the provided project code, so if your code is dependent on changes you've made to files that are not required of you to submit, you will end up losing points.



MDPs

To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
python3 gridworld.py -m
```

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press *up*, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

```
python3 gridworld.py -h
```

The default agent moves randomly

```
python3 gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by `(x,y)` Cartesian coordinates and any arrays are indexed by `[x][y]`, with `'north'` being the direction of increasing `y`, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

Question 1 (4 points): Value Iteration

Recall the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, so the relevant training option is the number of iterations of value iteration it should run (option `-i` in the command prompts) in its initial planning phase. `ValueIterationAgent` takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for `ValueIterationAgent` using V_k.

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

Important: Use the "batch" version of value iteration where each vector V_k is computed from a fixed vector V_{k-1} (like in lecture), not the "online" version where one single weight vector is updated in place. This means that when a state's value is 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 (even if some of the successor states had already been updated in iteration k).

Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next k+1 rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}).

You should return the synthesized policy π_{k+1}.

Hint: You may optionally use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. Be careful with `argMax`: the actual argmax you want may be a key not in the counter!

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder:

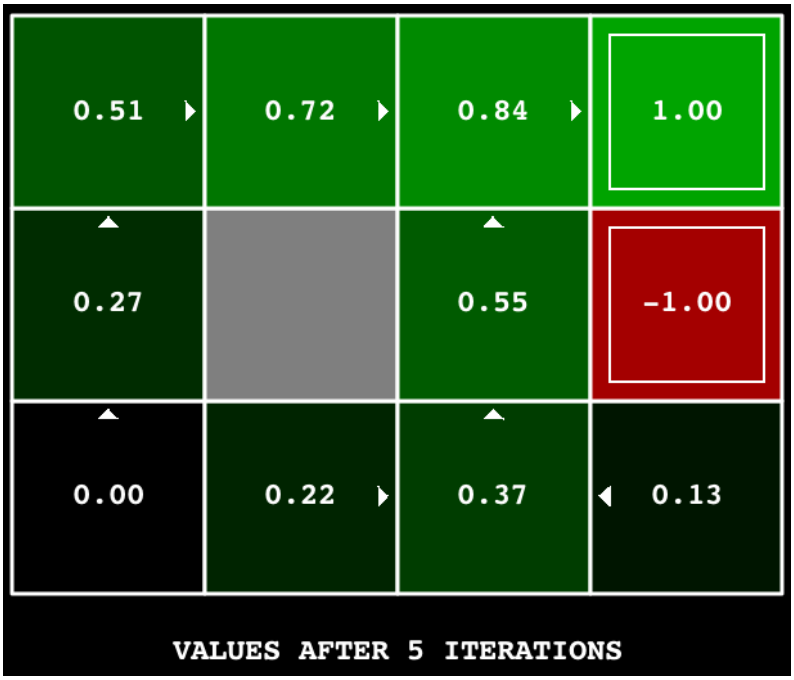
```
python3 autograder.py -q q1
```

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python3 gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, running value iteration for 5 iterations should give you this output:

```
python3 gridworld.py -a value -i 5
```



Grading: Your value iteration agent will be graded on a new grid. The autograder will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

Question 2 (1 point): Bridge-Crossing Analysis

`BridgeGrid` is a grid world map with the a 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 the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters (but keep the value non-zero and less than 1) so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. (Noise refers to how often an agent ends up in an unintended successor state when they perform an action.)

In addition to changing only one of the two parameters, include a comment specifying your reasoning as to why increasing/decreasing the chosen parameter causes the agent to cross the bridge.

The default corresponds to:

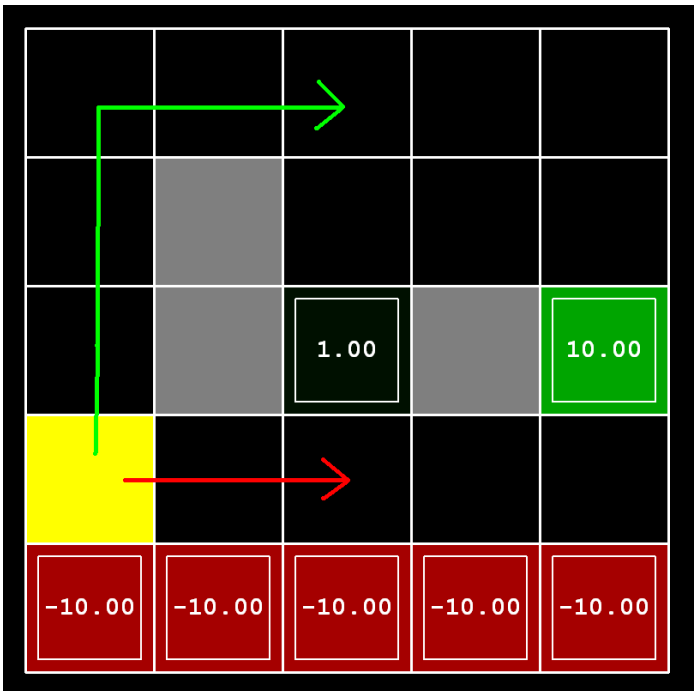
```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

Grading: We will check that you only changed one of the given parameters, and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:

```
python autograder.py -q q2
```

Question 3 (5 points): Policies

Consider the `DiscountGrid` layout, shown below. This grid has two terminal states with positive payoff (in the middle row), a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure below.



In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. **Your setting of the parameter values 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, assert that the policy is impossible by returning the string `'NOT POSSIBLE'`.

As in question 2, each of your functions `question3a()` through `question3e()` should have comments included explaining your intuition as to why the values you chose achieve the specified behavior.

Here are the optimal policy types you should attempt to produce:

a. Prefer the close exit (+1), risking the cliff (-10)
b. Prefer the close exit (+1), but avoiding the cliff (-10)
c. Prefer the distant exit (+10), risking the cliff (-10)
d. Prefer the distant exit (+10), avoiding the cliff (-10)
e. Avoid both exits and the cliff (so an episode should never terminate)

To check your answers, run the autograder:

```
python autograder.py -q q3
```

`question3a()` through `question3e()` should each return a 3-item tuple of (discount, noise, living reward) in `analysis.py`.

Note: You can check your policies in the GUI. For example, using a correct answer to 3(a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and the arrow in (2,1) should point north.

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValue display, and mentally calculate the policy by taking the arg max of the available qValues for each state.

Question 4 (1 point): Asynchronous Value Iteration

Write a value iteration agent in `AsynchronousValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option `-i`) in its initial planning phase. `AsynchronousValueIterationAgent` takes an MDP on construction and runs *cyclic* value iteration (described in the next paragraph) for the specified number of iterations before the constructor returns. Note that all this value iteration code should be placed inside the constructor (`__init__` method).

The reason this class is called `AsynchronousValueIterationAgent` is because we will update only **one** state in each iteration, as opposed to doing a batch-style update. Here is how cyclic value iteration works. In the first iteration, only update the value of the first state in the states list. In the second iteration, only update the value of the second. Keep going until you have updated the value of each state once, then start back at the first state for the subsequent iteration. **If the state picked for updating is terminal, nothing happens in that iteration**. You can implement it as indexing into the states variable defined in the code skeleton.

As a reminder, here's the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

`AsynchronousValueIterationAgent` inherits from `ValueIterationAgent` from Q1, so the only method you need to implement is `runValueIteration`. Since the superclass constructor calls `runValueIteration`, overriding it is sufficient to change the agent's behavior as desired.

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder. It should take less than a second to run. **If it takes much longer, you may run into issues later in the project, so make your implementation more efficient now.**

```
python autograder.py -q q4
```

The following command loads your `AsynchronousValueIterationAgent` in the Gridworld, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a asynchvalue -i 1000 -k 10
```

Question 5 is not required for this project.

Question 6 (4 points): Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridworld, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option `'-a q'` in the command prompt. For this question, you must implement the `update`, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

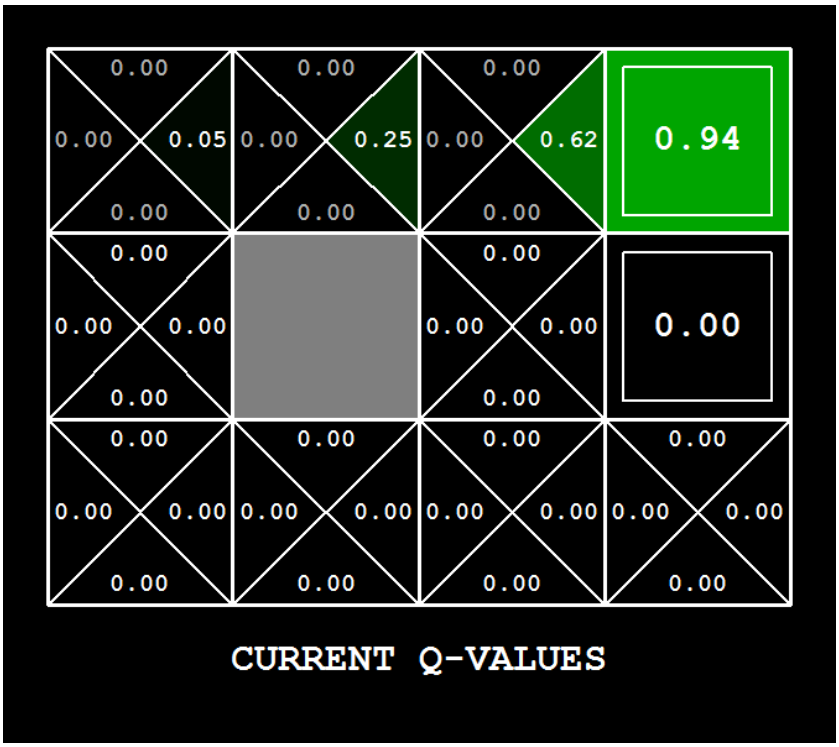
Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function can help. In a particular state, actions that your agent *hasn't* seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent *has* seen before have a negative Q-value, an unseen action may be optimal.

Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue` . This abstraction will be useful for question 10 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

```
python gridworld.py -a q -k 5 -m
```

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `--noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the following Q-values:



To grade your implementation, run the autograder:

```
python autograder.py -q q6
```

Question 7 (2 points): Epsilon Greedy

Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`, 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.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability `p` of success by using `util.flipCoin(p)`, which returns `True` with probability `p` and `False` with probability `1-p`.

After implementing the `getAction` method, observe the following behavior of the agent in gridworld (with epsilon = 0.3).

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect?

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

To test your implementation, run the autograder:

```
python autograder.py -q q7
```

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python crawler.py
```

If this doesn't work, you've probably written some code too specific to the `GridWorld` problem and you should make it more general to all MDPs.

This will invoke the crawling robot from class using your Q-learner. Play around with the 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, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

Questions 8 and 9 are not required for this project.

Question 10 (3 points): Approximate Q-Learning

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note: Approximate Q-learning assumes the existence of a feature function f(s,a) over state and action pairs, which yields a vector $f_1(s,a)$ .. $f_i(s,a)$ .. $f_n(s,a)$ of feature values. We provide feature functions for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate Q-function takes the following form:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

where each weight w_i is associated with a particular feature $f_i(s,a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Note that the $difference$ term is the same as in normal Q-learning, and r is the experienced reward.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every `(state,action)` pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Important: `ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

Even much larger layouts should be no problem for your `ApproximateQAgent`. (*warning*: this may take a few minutes to train)

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

Grading: We will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q10
```

Congratulations! You have a learning Pacman agent!

Submission

When you are finished, submit the following completed files to Canvas:

- `analysis.py`
- `qLearningAgents.py`
- `valueIterationAgents.py`

Project 3

Criteria	Ratings		Pts
Coding Style Code is cleanly written with comments providing non-redundant supplementary explanation	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Question 1 Completed as described	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
Question 2 Completed as described	**1.0 pts** **Full Marks**	**0.0 pts** **No Marks**	1.0 pts
Question 3 Completed as described	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Question 4 Completed as described	**1.0 pts** **Full Marks**	**0.0 pts** **No Marks**	1.0 pts
Question 6 Completed as described	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
Question 7 Completed as described	**2.0 pts** **Full Marks**	**0.0 pts** **No Marks**	2.0 pts
Question 10 Completed as described	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
			Total Points: 23.0