Assignment 2: MDP Planning

Part 1:

In this section, I implement the algorithms by simply putting the equations into code. I made an MDP class for easy implementation, which is initialised using the filepath.

```python
class MDP():
    def __init__(self, path):
        # get MDP from filepath

        self.lines= []
        with open(path) as f:
            self.lines = f.readlines()
            self.lines = [line.strip() for line in self.lines]
            self.lines = [line.split() for line in self.lines]

        S = int(self.lines[0][1])
        A = int(self.lines[1][1])
        T = np.zeros((S,A,S))
        R = np.zeros((S,A,S))
        gamma = float(self.lines[-1][1])
        if self.lines[-2][1] == "continuing":
            mdptype = "continuing"
            end = []
        else:
            mdptype = "episodic"
            end = [int(x) for x in self.lines[2][1:]]

        for line in self.lines[3:-2]:
            s = int(line[1])
            a = int(line[2])
            s1 = int(line[3])
            r = float(line[4])
            t = float(line[5])
            T[s][a][s1] = t
            R[s][a][s1] = r

        self.S = S
        self.T = T
        self.A = A
        self.R = R
        self.gamma = gamma
        self.mdptype = mdptype
        self.end = end
```

I also made a Policy class which reads the policy from the filepath.

```python
class Policy():
    def __init__(self, path=None, mdp=None, actions=None):
        if path != None:
            with open(path) as f:
                self.lines = f.readlines()
                self.lines = [line.strip() for line in self.lines]

            self.S = len(self.lines)
            self.A = [int(i) for i in self.lines]
        else:
            self.S = mdp.S
            self.A = actions
```

1. Value Iteration

Given the Bellman optimality equation,

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s') \{R(s, a, s') + \gamma V^*(s')\}.$$

Initialising the values and the policy such that each value and the action for each state is 0.

```python
values = np.zeros(mdp.S)
optimal_actions = np.zeros(mdp.S, dtype=int)
```

As a first step, we compute the values given the initial policy of using the 0th action for all states. Given the new set of values, I find the best set of actions from each state.
I then continue this process until $||V(s) - V'(s)||_\infty < \Delta$, where $V'(s)$ is the set of values at the t-1 timestep, $V(s)$ is the set of values at the t timestep and Δ is some threshold value.

```python
new_values = np.zeros(mdp.S)
for s in range(mdp.S):
    optimal_actions[s] = np.argmax([sum([mdp.T[s][a][s1]*(
        mdp.R[s][a][s1]+mdp.gamma*values[s1]
        ) for s1 in range(mdp.S)]
        ) for a in range(mdp.A)])
    new_values[s] = sum([mdp.T[s][optimal_actions[s]][s1]*(
        mdp.R[s][optimal_actions[s]][s1]+mdp.gamma*values[s1]
        ) for s1 in range(mdp.S)])
infnorm_delta = np.max(np.abs(new_values-values))
values = new_values
```

If the mdp is episodic, I add a skipping clause.

```python
if s in mdp.end:
    continue
```

2. Howard's Policy Iteration

In this part, I iterate over all states and actions, so that if I have found an improving action for a given state, I will change my policy such that the action for the current state is changed to the improving state, while the rest remains the same.

```
def action_value(mdp, s, a, values):
    return sum([mdp.T[s][a][s1]*(
        mdp.R[s][a][s1]+mdp.gamma*values[s1]
        ) for s1 in range(mdp.S)])

def Howards_Policy_Iteration(mdp):
    optimal_actions = np.zeros(mdp.S, dtype=int)
    values = np.zeros(mdp.S)

    improvable_states = 1
    if mdp.mdptype == "continuing":
        while improvable_states > 0:
            improvable_states = 0
            for s in range(mdp.S):
                for a in range(mdp.A):
                    new_val = action_value(mdp, s, a, values)
                    if new_val > values[s]:
                        optimal_actions[s] = a
                        values[s] = new_val
                        improvable_states += 1
                        break
```

Again, I add a skipping clause for when the mdp is episodic.

```
if s in mdp.end:
    continue
```

3. Linear Programming

For all states s, and all actions a, I set up all the Bellman optimality equations as inequalities, and set my objective function as the sum of all the values. The LP problem is set up so that the program tries to minimize the objective function.

```
def Linear_Programming(mdp):
    # PROBLEM
    prob = pulp.LpProblem("MDP", pulp.LpMinimize)
    values = [pulp.LpVariable("v{s}".format(s=s)) for s in range(mdp.S)]
    # actions = [pulp.LpVariable("a{}".format(s), lowBound=0, upBound=mdp.A-1, cat="Integer") for s in range(mdp.S)]

    # OBJECTIVE FXN
    prob += pulp.lpSum(values)

    # CONSTRAINTS
    for s in range(mdp.S):
        for a in range(mdp.A):
            prob += values[s] - sum([mdp.T[s][a][s1]*(mdp.R[s][a][s1] + mdp.gamma*values[s1]) for s1 in range(mdp.S)]) >= 0

    # SOLVE
    prob.solve(pulp.PULP_CBC_CMD(msg=False))

    # GET RESULTS
    final_values = [v.varValue for v in values]
    optimal_actions = [np.argmax([action_value(mdp, s, a, final_values) for a in range(mdp.A)]) for s in range(mdp.S)]
    return final_values, optimal_actions
```