

Hosam Farouk

Policy Iteration Agent VS Random Agent:

```
import static org.junit.Assert.assertEquals;

public class TestPolicyIterationAgent {

    /**
     * @param a1
     * @param a2
     * @param howmanyTimes
     * @return list of results: [xWon, oWon, draw]
     */
    public static int[] playAgainstEachOther(Agent a1, Agent a2, int howmanyTimes) {
        // ...
    }
}
```

Finished after 1.513 seconds
Runs: 3/3 Errors: 0 Failures: 0

> TestPolicyIterationAgent [Runner: JUnit]

<terminated> TestPolicyIterationAgent [J...]
Playing move: X(2,0)

```
|X|O| |
|X| |O|
|X| | |
```

X won!
Wins: 49 Losses: 0 Draws: 1
Against Defensive Agent:
Playing move: X(0,0)

Wins: 49 | Losses: 0 | Draws: 1

Policy Iteration Agent VS Defensive Agent:

```
import static org.junit.Assert.assertEquals;

public class TestPolicyIterationAgent {

    /**
     * @param a1
     * @param a2
     * @param howmanyTimes
     * @return list of results: [xWon, oWon, draw]
     */
    public static int[] playAgainstEachOther(Agent a1, Agent a2, int howmanyTimes) {
        // ...
    }
}
```

Finished after 1.513 seconds
Runs: 3/3 Errors: 0 Failures: 0

> TestPolicyIterationAgent [Runner: JUnit]

<terminated> TestPolicyIterationAgent [J...]
Playing move: X(1,0)

```
|X| |O|
|X|X| |
|X|O|O|
```

X won!
Wins: 45 Losses: 0 Draws: 5
Against Aggressive Agent:
Playing move: X(0,0)

Wins: 45 | Losses: 0 | Draws: 5

Policy Iteration Agent VS Aggressive Agent:

```
import static org.junit.Assert.assertEquals;

public class TestPolicyIterationAgent {

    /**
     * @param a1
     * @param a2
     * @param howmanyTimes
     * @return list of results: [xWon, oWon, draw]
     */
    public static int[] playAgainstEachOther(Agent a1, Agent a2, int howmanyTimes) {
        Game gn;
        //int[] results=new int[3];

        int xWon = 0;
        int oWon = 0;
        int dr = 0;

        for (int i=0;i<howmanyTimes;i++) {
            gn = new Game(a1, a2, a1);
            try {
                gn.playOut();
            } catch (NullPointerException e) {
                // ...
            }
        }
    }
}
```

Finished after 1.513 seconds
Runs: 3/3 Errors: 0 Failures: 0

> TestPolicyIterationAgent [Runner: JUnit]

<terminated> TestPolicyIterationAgent [J...]
Playing move: X(2,0)

```
| | |O|
| | | |
| | | |
```

Playing move: O(2,1)

```
|X| | |
|X| | |
|X|O|O|
```

Playing move: X(1,0)

```
|X| | |
|X| | |
|X|O|O|
```

X won!
Wins: 50 Losses: 0 Draws: 0

Wins: 50 | Losses: 0 | Draws: 0

Function Implementations in 2-3 sentences:

initRandomPolicy():

- This method initializes a random policy by assigning a random valid move for each state and the terminal states are skipped.
- It assigns a random move from the list of possible moves. This provides a starting point for the process.
- And lastly it updates the curPolicy hashmap with the policy for each state.

evaluatePolicy():

- First it loops through states and skips terminal states, and for non-terminal states it calculates the Q value using the current policy.
- It updates the state's value in the policyValues hashmap and checks for convergence by comparing the difference of the new value to the previous one in relation with the delta factor.
- And lastly the previous value is set to the current value to continue with the iteration.

improvePolicy():

- First initialized a new Policy Object to store a copy of the current policy for comparison after update.
- Used the Iterator to loop through the states in the curPolicy HashMap to see and update the optimal move.
- For each state the Q value is calculated for all possible moves using the calculateQvalue helper method and then updates the best move and value if a higher q value was found and it updates the policy for that state in the curPolicy HashMap and returns if the policy has changed.

train():

- First called the initRandomPolicy() to initialize a random policy.
- It iteratively evaluates a policy and checks for improvement until convergence.
- When the policy stops changing i.e is converged the final policy is assigned to the agent.

calculateQValue:

- This is a helper method that I implemented to compute the Q value for a given state and move.
- It first generates all possible transitions.

- Then it calculates the Q value using Bellmans Equation with a loop to account for all the transitions.