AI3000: REINFORCEMENT LEARNING

VELMA DHATRI REDDY
AI24BTECH11030

1) (a) According to Monte-Carlo method:

$$V(A) = \frac{14+15+17+16+15}{5} = 15.4$$

$$V(B) = \frac{13+14+16+15+14}{5} = 14.4$$

$$V(C) = \frac{12+13+15+14+13}{5} = 13.4$$

$$V(D) = \frac{12+12+0+12+11}{5} = 9.4$$

$$V(E) = \frac{11+11+11+10+9}{5} = 10.4$$

$$V(F) = \frac{10+10+10+10+9}{5} = 9.8$$

(b) B, E, F. These states will have different value estimates as the above states occur multiple times in the same markov chain trajectory.

(c) Q-learning update equation

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

$\alpha = 0.7$ and initial Q-values are -10.

$$Q(C, jump) = -10 + 0.7(4 + (-10) - (-10))$$
$$= -7.2$$

$$Q(E, right) = -10 + 0.7(1 + (-10) - (-10))$$
$$= -9.3$$

$$Q(F, left) = -10 + 0.7(-2 + (-9.3) - (-10))$$
$$= -10 + 0.7(-1.3) = -10.91$$

$$Q(\cancel{E}, right) = \cancel{-10} + 0.7(1 + \cancel{(-10)} - (-\cancel{10}))$$

$$Q(E, right) = -9.3 + 0.7(1 + (-10) - (-9.3))$$
$$= -9.3 + 0.7(0.3)$$
$$= -9.09$$

	$Q(C, \text{left})$	$Q(C, \text{jump})$	$Q(E, \text{left})$	$Q(E, \text{right})$	$Q(F, \text{left})$	$Q(F, \text{right})$
Initial	-10	-10	-10	-10	-10	-10
Transition 1		-7.2				
Transition 2				-9.3		
Transition 3					-10.91	
Transition 4				-9.09		

1(d) Greedy policy -

For state C: Jump

For state E: Right

For state F: Right

(as these have the maximum Q-value for the action mentioned above for the given state)

1(e) (i) $\alpha_t = 1/t$

$\sum_{t=1}^{\infty} \alpha_t = \sum_{t=1}^{\infty} 1/t = \infty$ (It is a divergent series).

Hence, $\sum_{t=1}^{\infty} \alpha_t = \infty$ is satisfied

$\sum_{t=1}^{\infty} \alpha_t^2 = \sum_{t=1}^{\infty} 1/t^2 < \infty$ (This is a p-series with $p=2$) (convergent)

$\alpha_t = 1/t$ satisfies Robbins Monroe conditions

(ii) $\alpha_t = 1/t^2$

$\sum_{t=1}^{\infty} \alpha_t = \sum_{t=1}^{\infty} 1/t^2 < \infty$ ($\neq \infty$) (It's a convergent series)

Hence, $\sum_{t=1}^{\infty} \alpha_t = \infty$ is not satisfied

$\sum_{t=1}^{\infty} \alpha_t^2 = \sum_{t=1}^{\infty} 1/t^4 < \infty$

As it is a p-series with $p=4$ it is a convergent series.

However, as the first condition is not satisfied. Robbins Monroe condition is not satisfied by $\alpha_t = 1/t^2$

f) (i) Yes, it converges to optimal Q function. In this case, it is following action with probability 0.5 for the policy and 0.5 for random action. Exploration helps in updating Q values based on reward and Q value of next state.

(ii) In this case, it converges to a Q function that represents mixed strategy of following policy and random actions. For it to converge to optimal Q value it is often neccesary to reduce randomness over time. But, here the exploration remains the same.

It maynot converge to optimal value unless the initial policy π is optimal.

In [23]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

In [24]:
```
num_games = 10000
```

Problem 2(a)

In [25]:
```
class TicTacToe():
  def __init__(self, n):
    self.size = n
    self.board = np.empty((n, n), dtype='U1')
    self.board[:] = ''
    self.player = random.choice(['X', 'O'])
    self.end = False

  def act(self, move):
    if self.board[move] == '':
      self.board[move] = self.player
      if self.player == 'X':
        self.player = 'O'
      else:
        self.player = 'X'
    else:
      print("Move is invalid")

  def print_board(self):
    print(self.board)

  def available_positions(self):
    positions = []
    for i in range(self.size):
      for j in range(self.size):
        if self.board[i, j] == '':
          positions.append((i, j))
    return positions

  def winner(self):

    #Check along rows
    for i in range(self.size):
      if sum(np.char.count(self.board[i, :], 'X')) == self.size:
        self.end = True
        return 'X'
      if sum(np.char.count(self.board[i, :], 'O')) == self.size:
        self.end = True
        return 'O'

    #Check along columns
    for i in range(self.size):
      if sum(np.char.count(self.board[:, i], 'X')) == self.size:
        self.end = True
        return 'X'
      if sum(np.char.count(self.board[:, i], 'O')) == self.size:
        self.end = True
        return 'O'
```

```
    #Check along diagonals
    if sum(np.char.count(self.board.diagonal(), 'X')) == self.size:
      self.end = True
      return 'X'
    if sum(np.char.count(self.board.diagonal(), 'O')) == self.size:
      self.end = True
      return 'O'
    if np.sum(np.char.count((self.board[::-1]).diagonal(), 'X')) == self.
      self.end = True
      return 'X'
    if np.sum(np.char.count((self.board[::-1]).diagonal(), 'O')) == self.
      self.end = True
      return 'O'

    #Tie
    if len(self.available_positions()) == 0:
      self.end = True
      return 0

    self.end = False
    return None

  def reward(self):
    if self.winner() == 'X':
      return 1
    if self.winner() == 'O':
      return -1
    if self.winner() == 0:
      return 0.5
    return 0

  def state(self):
    return str(self.board.reshape(self.size*self.size))
```

Problem 2(b)

In [26]:
```
class random_agent():

  def __init__(self):
    pass

  def policy(self, env):
    actions = env.available_positions()
    return random.choice(actions)
```

In [27]:

```
class safe_agent():

  def __init__(self):
    pass

  def policy(self, env):
    actions = env.available_positions()

    #check for win along row
    for i in range(env.size):
      if sum(np.char.count(env.board[i, :], 'X')) == env.size - 1 and sum
        for j in range(env.size):
          if env.board[i, j] == '':
            return (i, j)

    #check for win along column
    for i in range(env.size):
      if sum(np.char.count(env.board[:, i], 'X')) == env.size - 1 and sum
        for j in range(env.size):
          if env.board[j, i] == '':
            return (j, i)

    #check for win along diagonal
    if sum(np.char.count(env.board.diagonal(), 'X')) == env.size - 1 and
      for i in range(env.size):
        if env.board[i, i] == '':
          return (i, i)

    if sum(np.char.count((env.board[::-1]).diagonal(), 'X')) == env.size
      for i in range(env.size):
        if env.board[i, env.size - 1 - i] == '':
          return (i, env.size - 1 - i)

    #check for block along row
    for i in range(env.size):
      if sum(np.char.count(env.board[i, :], 'O')) == env.size - 1 and sum
        for j in range(env.size):
          if env.board[i, j] == '':
            return (i, j)

    #check for block along column
    for i in range(env.size):
      if sum(np.char.count(env.board[:, i], 'O')) == env.size - 1 and sum
        for j in range(env.size):
          if env.board[j, i] == '':
            return (j, i)

    #check for block along diagonal
    if sum(np.char.count(env.board.diagonal(), 'O')) == env.size - 1 and
      for i in range(env.size):
        if env.board[i, i] == '':
          return (i, i)

    if sum(np.char.count((env.board[::-1]).diagonal(), 'O')) == env.size
      for i in range(env.size):
        if env.board[i, env.size - 1 - i] == '':
          return (i, env.size - 1 - i)

    #If there are no winning or blocking moves, it picks a random action
    return random.choice(actions)
```

In [28]:

```
class QlearningAgent():

  def __init__(self):
    self.alpha = 0.01
    self.gamma = 0.9
    self.epsilon = 0.05
    self.Q_table = {}

  def policy(self, env):
    s = env.state()
    actions = env.available_positions()
    max_value = max([self.Q(s, a) for a in actions])
    max_actions = [a for a in actions if self.Q(s,a) == max_value]
    return random.choice(max_actions)

  def choose_action(self, env):
    p = random.random()
    actions = env.available_positions()
    if p <= self.epsilon:
      return random.choice(actions)
    return self.policy(env, actions)

  def Q(self, s, a):
    if (s, a) in self.Q_table:
      return self.Q_table[(s, a)]
    else:
      self.Q_table[(s, a)] = 0
    return self.Q_table[(s, a)]

  def update_Q(self, env, s1, a1, s2):

    actions = env.available_positions()
    reward = env.reward()

    if len(actions) == 0:
      max_value = 0
    else:
      max_val = max([self.Q(s2, a) for a in actions])

    self.Q_table[(s1, a1)] = self.Q_table[(s1, a1)] + self.alpha*(reward
```

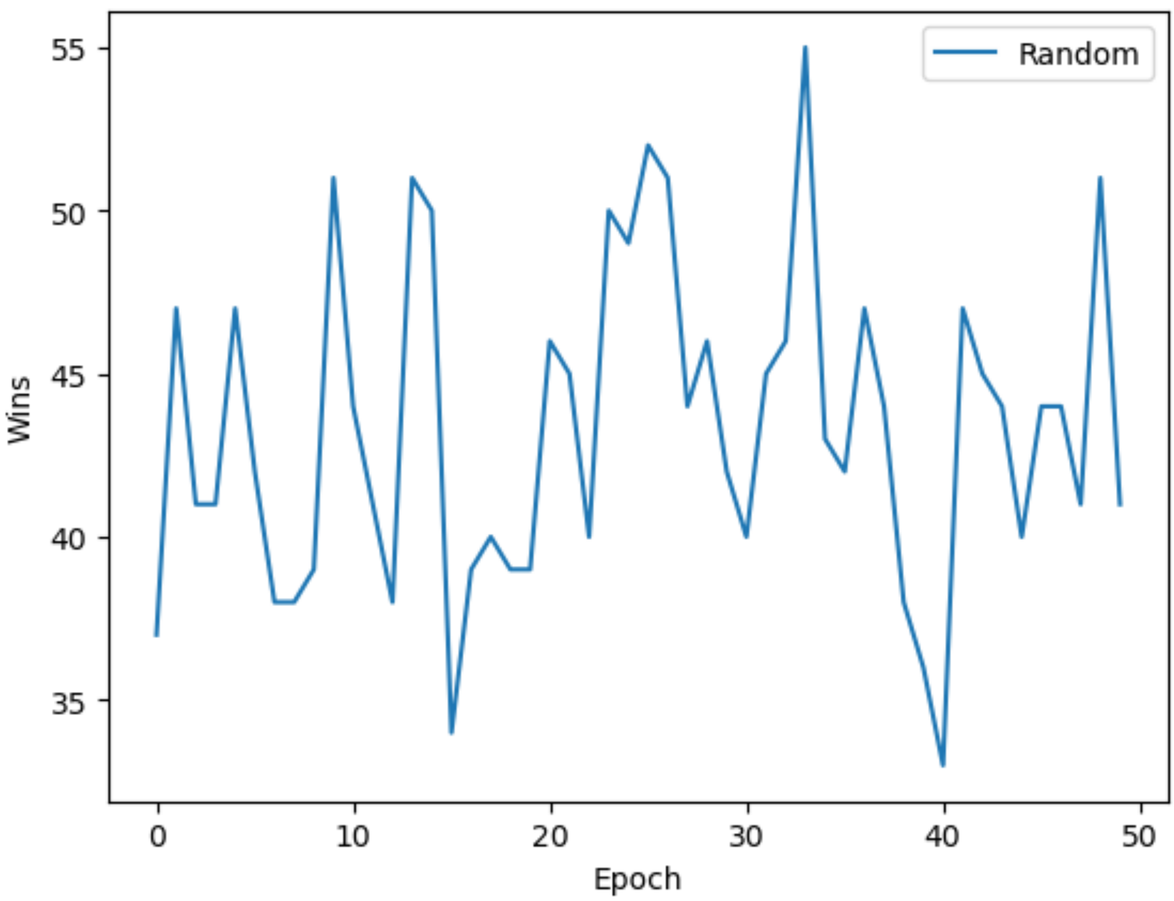
In [29]:

```
def train(agent, opponents):
  opponent = random.choice(opponents)
  num_train_wins = []

  for i in range(num_games):
    env = TicTacToe(3)
    if(env.player == 'X'):
      player1 = agent
      player2 = opponent
    else:
      player1 = opponent
      player2 = agent

    while not env.end:
      s1 = env.state()
      a_p1 = player1.policy(env)
      env.act(a_p1)

      winner = env.winner()
      if winner == None:
        a_p2 = player2.policy(env)
        env.act(a_p2)
        if (player1 == agent):
          a = a_p1
        else:
          a = a_p2
        s2 = env.state()
        agent.update_Q(env, s1, a, s2)
        winner = env.winner()

    if(i % 200 == 0):
      win_count = test(agent, opponent, 100)
      print(f'Epoch:{i/200} - Wins:{win_count}')
      num_train_wins.append(win_count)

  return agent, num_train_wins
```

In [30]:
```
def test(agent, opponent, num_games):
    num_wins = 0
    for i in range(num_games):
        env = TicTacToe(3)
        agent = QlearningAgent()
        agent.epsilon = 0
        agent.Q_table = agent.Q_table

        if(env.player == 'X'):
            player1 = agent
            player2 = opponent
        else:
            player1 = opponent
            player2 = agent

        while not env.end:
            a_p1 = player1.policy(env)
            env.act(a_p1)

            winner = env.winner()
            if winner == None:
                a_p2 = player2.policy(env)
                env.act(a_p2)
                winner = env.winner()

        if(env.winner() == 'X'):
          num_wins += 1

    return num_wins
```

In [31]:
```
env = TicTacToe(3)

opponent = random_agent()
agent = QlearningAgent()
agent, wins = train(agent, [opponent])

epochs = np.arange(0,50)
plt.plot(epochs, wins, label = "Random")
plt.xlabel("Epoch")
plt.ylabel("Wins")
plt.legend()
plt.show()

test1 = random_agent()
count = 0
for i in range(1000):
    count += test(agent, test1, 1)
print('Wins against random agent: ', count)


test2 = safe_agent()
count = 0
for i in range(1000):
    count += test(agent, test2, 1)
print('Wins against safe agent: ', count)
```

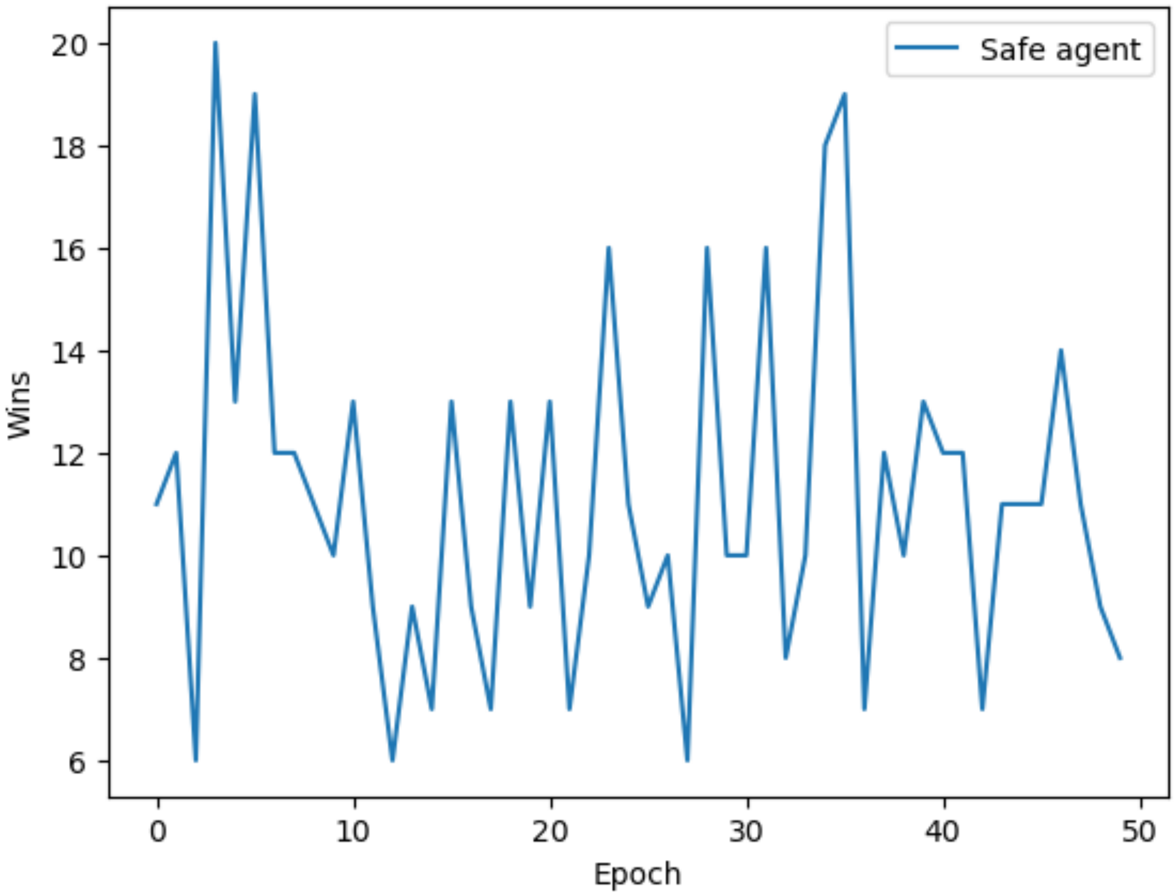
```
Epoch:0.0 - Wins:37
Epoch:1.0 - Wins:47
Epoch:2.0 - Wins:41
Epoch:3.0 - Wins:41
Epoch:4.0 - Wins:47
Epoch:5.0 - Wins:42
Epoch:6.0 - Wins:38
Epoch:7.0 - Wins:38
Epoch:8.0 - Wins:39
Epoch:9.0 - Wins:51
Epoch:10.0 - Wins:44
Epoch:11.0 - Wins:41
Epoch:12.0 - Wins:38
Epoch:13.0 - Wins:51
Epoch:14.0 - Wins:50
Epoch:15.0 - Wins:34
Epoch:16.0 - Wins:39
Epoch:17.0 - Wins:40
Epoch:18.0 - Wins:39
Epoch:19.0 - Wins:39
Epoch:20.0 - Wins:46
Epoch:21.0 - Wins:45
Epoch:22.0 - Wins:40
Epoch:23.0 - Wins:50
Epoch:24.0 - Wins:49
Epoch:25.0 - Wins:52
Epoch:26.0 - Wins:51
Epoch:27.0 - Wins:44
Epoch:28.0 - Wins:46
Epoch:29.0 - Wins:42
Epoch:30.0 - Wins:40
Epoch:31.0 - Wins:45
Epoch:32.0 - Wins:46
Epoch:33.0 - Wins:55
Epoch:34.0 - Wins:43
Epoch:35.0 - Wins:42
Epoch:36.0 - Wins:47
Epoch:37.0 - Wins:44
Epoch:38.0 - Wins:38
Epoch:39.0 - Wins:36
Epoch:40.0 - Wins:33
Epoch:41.0 - Wins:47
Epoch:42.0 - Wins:45
Epoch:43.0 - Wins:44
Epoch:44.0 - Wins:40
Epoch:45.0 - Wins:44
Epoch:46.0 - Wins:44
Epoch:47.0 - Wins:41
Epoch:48.0 - Wins:51
Epoch:49.0 - Wins:41
```




```
Wins against random agent:  415
Wins against safe agent:  108
```

In [32]:
```
env = TicTacToe(3)

opponent = safe_agent()
agent = QlearningAgent()
agent, wins = train(agent, [opponent])

plt.plot(epochs, wins, label = "Safe agent")
plt.xlabel("Epoch")
plt.ylabel("Wins")
plt.legend()
plt.show()

test1 = random_agent()
count = 0
for i in range(1000):
    count += test(agent, test1, 1)
print('Wins against Random agent: ', count)

test2 = safe_agent()
count = 0
for i in range(1000):
    count += test(agent, test2, 1)
print('Wins against Safe agent: ', count)
```

```
Epoch:0.0 - Wins:11
Epoch:1.0 - Wins:12
Epoch:2.0 - Wins:6
Epoch:3.0 - Wins:20
Epoch:4.0 - Wins:13
Epoch:5.0 - Wins:19
Epoch:6.0 - Wins:12
Epoch:7.0 - Wins:12
Epoch:8.0 - Wins:11
Epoch:9.0 - Wins:10
Epoch:10.0 - Wins:13
Epoch:11.0 - Wins:9
Epoch:12.0 - Wins:6
Epoch:13.0 - Wins:9
Epoch:14.0 - Wins:7
Epoch:15.0 - Wins:13
Epoch:16.0 - Wins:9
Epoch:17.0 - Wins:7
Epoch:18.0 - Wins:13
Epoch:19.0 - Wins:9
Epoch:20.0 - Wins:13
Epoch:21.0 - Wins:7
Epoch:22.0 - Wins:10
Epoch:23.0 - Wins:16
Epoch:24.0 - Wins:11
Epoch:25.0 - Wins:9
Epoch:26.0 - Wins:10
Epoch:27.0 - Wins:6
Epoch:28.0 - Wins:16
Epoch:29.0 - Wins:10
Epoch:30.0 - Wins:10
Epoch:31.0 - Wins:16
Epoch:32.0 - Wins:8
Epoch:33.0 - Wins:10
Epoch:34.0 - Wins:18
Epoch:35.0 - Wins:19
Epoch:36.0 - Wins:7
Epoch:37.0 - Wins:12
Epoch:38.0 - Wins:10
Epoch:39.0 - Wins:13
Epoch:40.0 - Wins:12
Epoch:41.0 - Wins:12
Epoch:42.0 - Wins:7
Epoch:43.0 - Wins:11
Epoch:44.0 - Wins:11
Epoch:45.0 - Wins:11
Epoch:46.0 - Wins:14
Epoch:47.0 - Wins:11
Epoch:48.0 - Wins:9
Epoch:49.0 - Wins:8
```

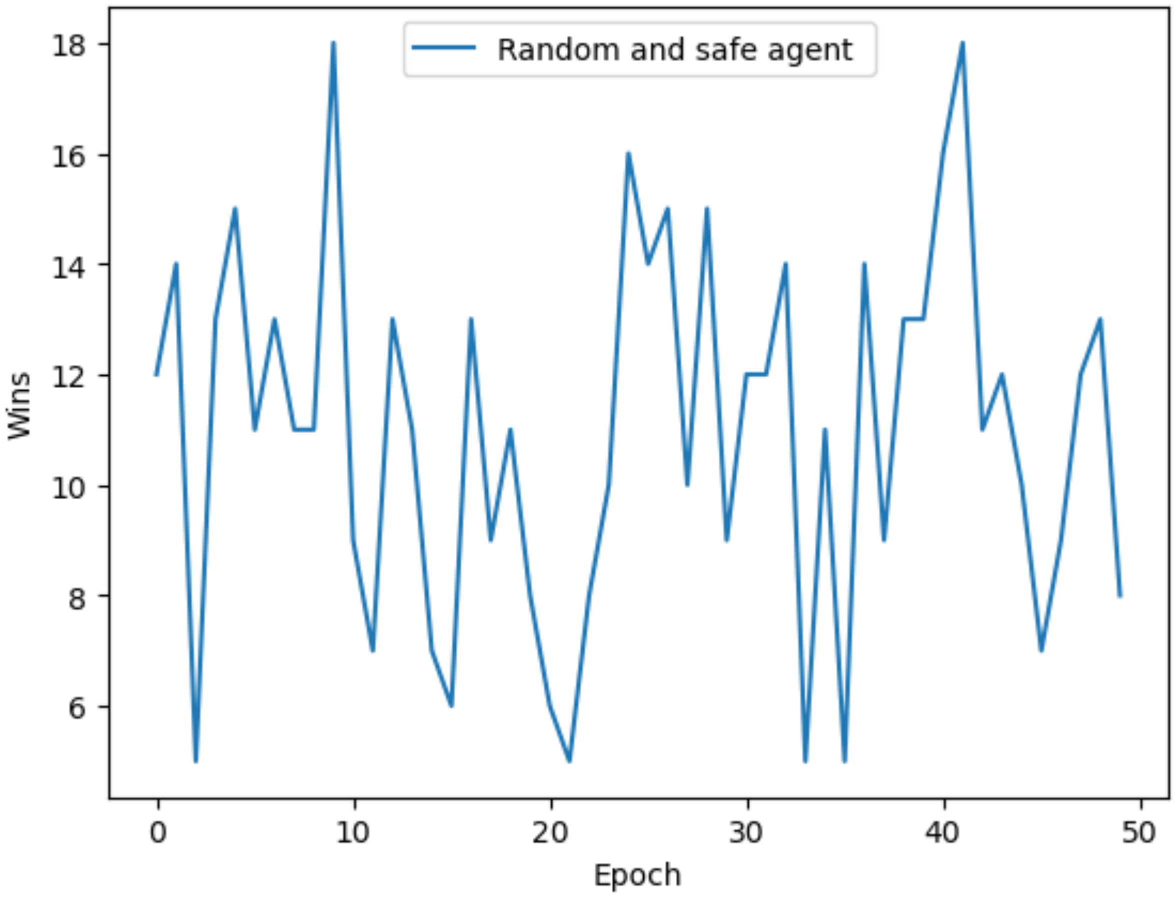



```
Wins against Random agent:  432
Wins against Safe agent:  113
```

In [33]:

```python
env = TicTacToe(3)

opponent1 = random_agent()
opponent2 = safe_agent()
agent = QlearningAgent()
agent, wins = train(agent, [opponent1, opponent2])

epochs = np.arange(0,50)
plt.plot(epochs, wins, label = "Random and safe agent ")
plt.xlabel("Epoch")
plt.ylabel("Wins")
plt.legend()
plt.show()

random_agent_test = random_agent()
count = 0
for i in range(1000):
    count += test(agent, random_agent_test, 1)
print('Wins against Random agent: ', count)


safe_agent_test = safe_agent()
count = 0
for i in range(1000):
    count += test(agent, safe_agent_test, 1)
print('Wins against Safe agent: ', count)
```

```
Epoch:0.0 - Wins:12
Epoch:1.0 - Wins:14
Epoch:2.0 - Wins:5
Epoch:3.0 - Wins:13
Epoch:4.0 - Wins:15
Epoch:5.0 - Wins:11
Epoch:6.0 - Wins:13
Epoch:7.0 - Wins:11
Epoch:8.0 - Wins:11
Epoch:9.0 - Wins:18
Epoch:10.0 - Wins:9
Epoch:11.0 - Wins:7
Epoch:12.0 - Wins:13
Epoch:13.0 - Wins:11
Epoch:14.0 - Wins:7
Epoch:15.0 - Wins:6
Epoch:16.0 - Wins:13
Epoch:17.0 - Wins:9
Epoch:18.0 - Wins:11
Epoch:19.0 - Wins:8
Epoch:20.0 - Wins:6
Epoch:21.0 - Wins:5
Epoch:22.0 - Wins:8
Epoch:23.0 - Wins:10
Epoch:24.0 - Wins:16
Epoch:25.0 - Wins:14
Epoch:26.0 - Wins:15
Epoch:27.0 - Wins:10
Epoch:28.0 - Wins:15
Epoch:29.0 - Wins:9
Epoch:30.0 - Wins:12
Epoch:31.0 - Wins:12
Epoch:32.0 - Wins:14
Epoch:33.0 - Wins:5
Epoch:34.0 - Wins:11
Epoch:35.0 - Wins:5
Epoch:36.0 - Wins:14
Epoch:37.0 - Wins:9
Epoch:38.0 - Wins:13
Epoch:39.0 - Wins:13
Epoch:40.0 - Wins:16
Epoch:41.0 - Wins:18
Epoch:42.0 - Wins:11
Epoch:43.0 - Wins:12
Epoch:44.0 - Wins:10
Epoch:45.0 - Wins:7
Epoch:46.0 - Wins:9
Epoch:47.0 - Wins:12
Epoch:48.0 - Wins:13
Epoch:49.0 - Wins:8
```




```
Wins against Random agent:  420
Wins against Safe agent:  104
```

(4) Among the above three developed agents, the third agent which is trained by both random and safe agent is the best (which obviously is better that the agents which are trained by either one of them) This is beacause it is trained against all possible random and safe moves making it more robust.

(5) The Q-learning agent developed isn't unbeatable as we can see from the non-zero number of losses in the games.But, the agent can be improved by

- increasing the number of training epochs
- optimizing the hyperparameters.