Practical Activity 1 (**PRA1**)

Evaluable Practical Exercise

General considerations:

- The proposed solution cannot use methods, functions or parameters declared ***deprecated*** in future versions.
- This activity must be carried out on a **strictly individual** basis. Any indication of copying will be penalized with a failure for all parties involved and the possible negative evaluation of the subject in its entirety.
- It is necessary for the student to indicate **all the sources** that she/he has used to carry out the PRA. If not, the student will be considered to have committed plagiarism, being penalized with a failure and the possible negative evaluation of the subject in its entirety.

Delivery format:

- Some exercises may require several minutes of execution, so the delivery must be done in **Notebook format** and in **HTML format**, where the code, results and comments of each exercise can be seen. You can export the notebook to HTML from the menu File $\rightarrow$ Download as $\rightarrow$ HTML.
- There is a special type of cell to hold text. This type of cell will be very useful to answer the different theoretical questions posed throughout the activity. To change the cell type to this type, in the menu: Cell $\rightarrow$ Cell Type $\rightarrow$ Markdown.

Name and surname: Martina Carretta (1673930)

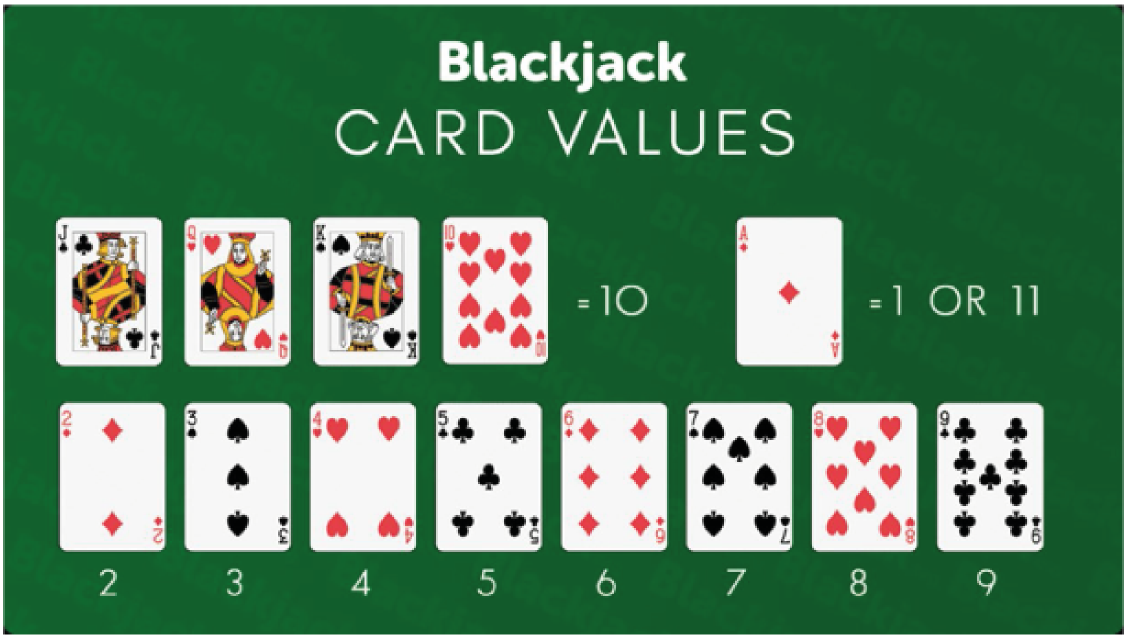
Introduction

Blackjack environment is part of the Gymnasium's Toy Text environments. Blackjack is a card game where the goal is to beat the dealer by obtaining cards that sum to closer to 21 (without going over 21) than the dealer's cards.

The game starts with the dealer having one face up and one face down card, while the player has two face up cards. All cards are drawn from an infinite deck (i.e. with replacement).

The card values are, as depicted in the following figure:

- Face cards (Jack, Queen, King) have a point value of **10**.
- Aces can either count as **11** (called a ``usable ace'') or **1**.
- Numerical cards (**2-9**) have a value equal to their number.



The player has the sum of cards held. The player can request additional cards (**hit**) until they decide to stop (**stick**) or exceed 21 (**bust**, immediate loss).

After the player sticks, the dealer reveals their face down card, and draws cards until their sum is 17 or greater. If the dealer goes bust, the player wins.

If neither the player nor the dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21.

Further information could be found at:

- Gymnasium Blackjack

In order to initialize the environment, we will use `natural=True` to give an additional reward for starting with a natural blackjack, i.e. starting with an ace and ten (sum is 21), as depicted in the following piece of code:

In [36]:
```
import gymnasium as gym

env = gym.make('Blackjack-v1', natural=True, sab=False)
```

In [37]:
```
print("Action space is {} ".format(env.action_space))
print("Observation space is {} ".format(env.observation_space))
print("Reward range is {} ".format(env.reward_range))
```

```
Action space is Discrete(2)
Observation space is Tuple(Discrete(32), Discrete(11), Discrete(2))
Reward range is (-inf, inf)
```

Part 1. Naïve Policy

Implement an agent that carries out the following deterministic policy:

- The agent will **stick** if it gets a score of 20 or 21.
- Otherwise, it will **hit**.

Questions (**1 point**):

1. Using this agent, simulate 100,000 games and calculate the agent's return (total accumulated reward).
2. Additionally, calculate the % of wins, natural wins, losses and draws.
3. Comment on the results.

In [38]:
```
import collections

class Agent:
    def __init__(self, env):
        self.env = env
        self.state = self.env.reset()[0]
        self.rewards = collections.defaultdict(float)
        self.transits = collections.defaultdict(collections.Counter)
        self.values = collections.defaultdict(float)


    def select_action(self, state) -> int:
        score = self.calculate_score(state)
        if score == 20 or score == 21:
            return 0
        else:
            return 1


    def calculate_score(self, state) -> int:
        player_sum, dealer_card, usable_ace = state
        return player_sum


    def play_episode(self, env) -> float:
        total_reward = 0.0
        state, _ = env.reset()

        while True:
            action = self.select_action(state)
            new_state, reward, terminated, truncated, _ = env.step(action)
            is_done = terminated or truncated
            if is_done:
                total_reward = reward # only compute once since the reward is only giv
                break
            state = new_state
        return total_reward

agent = Agent(env)
```

In [39]:
```
num_games = 100000

total_reward = 0.0

for _ in range(num_games):
    total_reward += agent.play_episode(env)

print(f"Total accumulated reward: {total_reward}")
```

```
Total accumulated reward: -33752.5
```

In [40]:
```
# Calculate statistics
num_games = 100000
wins = 0
natural_wins = 0
losses = 0
draws = 0
```

```
for _ in range(num_games):
    state, _ = agent.env.reset()
    episode_reward = 0.0
    while True:
        action = agent.select_action(state)
        new_state, reward, terminated, truncated, _ = agent.env.step(action)
        is_done = terminated or truncated
        episode_reward += reward
        if is_done:
            if reward == 1.0 or reward == 1.5: # Normal and natural wins
                wins += 1
                if reward == 1.5:  # Natural win is a counted both as a win and a natu
                    natural_wins += 1
            elif reward == -1.0:
                losses += 1
            else:
                draws += 1
            break
        state = new_state

win_percentage = (wins / num_games) * 100
natural_win_percentage = (natural_wins / num_games) * 100
loss_percentage = (losses / num_games) * 100
draw_percentage = (draws / num_games) * 100

print(f"Win percentage (counting the natural wins): {win_percentage}%")
print(f"Natural win percentage: {natural_win_percentage}%")
print(f"Loss percentage: {loss_percentage}%")
print(f"Draw percentage: {draw_percentage}%")
```

```
Win percentage (counting the natural wins): 29.576%
Natural win percentage: 4.048%
Loss percentage: 64.671%
Draw percentage: 5.753%
```

Answer: For the moment it has a low win rate, an even lower draw rate and most of the games are losses. there are only 5000 out of 100000 games were won with a natural win (meaning that the ace was used as an 11 value card). this is quite normal taking into account that it means the player has been handed an Ace (only 4 in the whole car deck making it a 0'07%) and a 10, Jack, Queen, or King (16). Being handed a card with a value of 10 has about a 30% porbability. This means that being handed an ace AND a 10-value card is quite improbable, so it's normal to have only a few natural wins

Part 2. Monte Carlo method

The objective of this section is to estimate the optimal policy using Monte Carlo methods. Specifically, you can choose and implement one of the algorithms related to *Control using MC methods* (with "exploring starts" or without "exploring starts", both on-policy or off-policy).

Questions (**2.5 points**):

1. Implement the selected algorithm and justify your choice.
2. Comment and justify all the parameters, such as:

- Number of episodes
- Discount factor
- Etc.

3. Implement a function that prints on the screen the optimal policy found for each state (similar to the figure in Section 3.1).
4. Using the trained agent, simulate 100,000 games and calculate the agent's return (total accumulated reward).
5. Additionally, calculate the % of wins, natural wins, losses and draws.

Answer: Exploring starts means that every possible state-action pair has a chance of being explored at least one. It could be beneficial to explore all possible starts. to do so, i will try to implement algorithms with and without exploring starts and check if it actually makes a difference.

MonteCarlo off-policy (whihc means that there are two policies, one to generate the episode and one that will be optimized) has a higher variance and the advantage it has is that it can add external knowledge. However, for the BlackJack game, it is not necessary since it will learn throught the policy creation phase what is a good action in each state

In [41]:
```
import numpy as np
from collections import defaultdict
import sys

def make_epsilon_greedy_policy(Q, epsilon, num_Actions):

    def policy_fn(observation): #policy needs to be the same inside an episode but we
        A = np.ones(num_Actions, dtype=float) * epsilon / num_Actions #array with numb
        best_action = np.argmax(Q[observation])
        A[best_action] += (1.0 - epsilon)
        return A

    return policy_fn

def mc_control_on_policy_epsilon_greedy(env, num_episodes, discount=1.0, epsilon=0.1,
    # We store the sum and number of returns for each state to calculate the average.
    # We could use an array to store all the returns, but it is inefficient in terms o
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The Q action value function.
    # A nested dictionary whose correspondence is state -> (action -> action-value).
    # Initially we initialize it to zero
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    for i_episode in range(1, num_episodes + 1):
        policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        episode = []
        state, _ = env.reset()
        done = False

        while not done:
            probs = policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs) # choose one acc
            next_state, reward, done, _, _ = env.step(action)
            episode.append((state, action, reward)) # add tuple with state, action and
            if done:
                break
```

```
            state = next_state

        G = 0
        # for each time step
        for state, action, reward in episode[::-1]:
            sa_pair = (state, action)

            # We add up all the rewards since the first appearance
            G = reward + discount * G

            # We calculate the average return for this state over all sampled episodes
            returns_sum[sa_pair] += G
            returns_count[sa_pair] += 1.0
            Q[state][action] = returns_sum[sa_pair] / returns_count[sa_pair]

        epsilon = max(epsilon_decay*epsilon, 0.01)


    return Q, policy
```

In [42]:
```
Q_every_visit, policy = mc_control_on_policy_epsilon_greedy(env, num_episodes=500000,
```

```
Episode 100/500000
Episode 500000/500000
```

In [43]:
```
import pandas as pd

data = []
for state in Q_every_visit.keys():
    action_probabilities = policy(state)
    data.append({'State': state, 'Policy': action_probabilities})

df = pd.DataFrame(data) # I put it in a dataframe to make it easier to display

# Display only the first 10 rows
df.head(10)
```

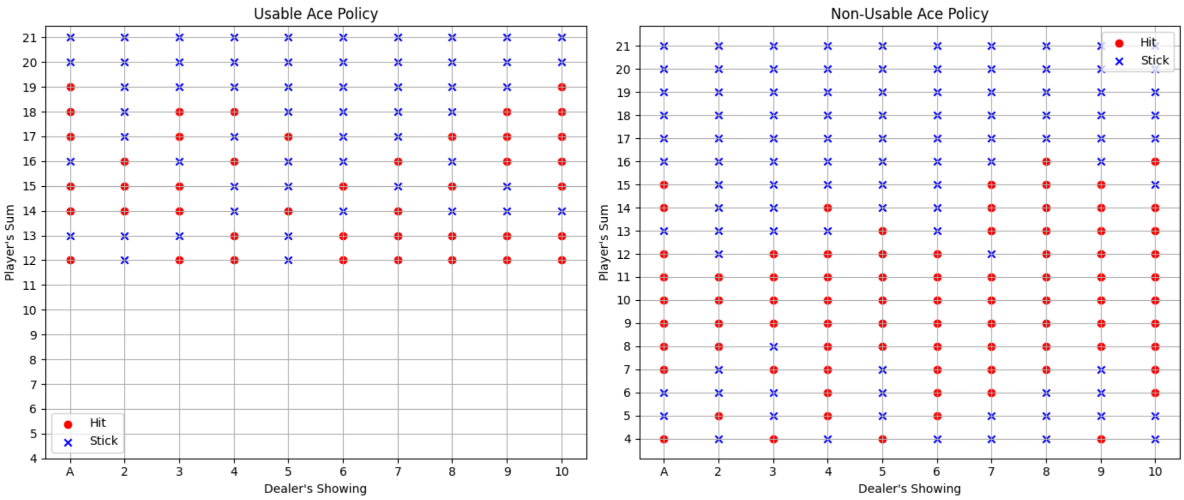
Out[43]:

	State	Policy
0	(15, 10, 0)	[0.995, 0.005]
1	(9, 8, 0)	[0.005, 0.995]
2	(16, 8, 0)	[0.005, 0.995]
3	(12, 10, 0)	[0.005, 0.995]
4	(13, 6, 1)	[0.005, 0.995]
5	(15, 6, 1)	[0.005, 0.995]
6	(18, 10, 0)	[0.995, 0.005]
7	(20, 10, 0)	[0.995, 0.005]
8	(21, 10, 0)	[0.995, 0.005]
9	(21, 1, 1)	[0.995, 0.005]

In [44]:
```
import matplotlib.pyplot as plt

def plot_policy(Q, policy):
    # Separate lists for usable and non-usable ace cases
    hit_player_sums_usable = []
    hit_dealer_showings_usable = []
```

```
    stick_player_sums_usable = []
    stick_dealer_showings_usable = []

    hit_player_sums_non_usable = []
    hit_dealer_showings_non_usable = []
    stick_player_sums_non_usable = []
    stick_dealer_showings_non_usable = []

    # Separate the states into hit and stick regions for usable and non-usable ace cas
    for state, policy in Q.items():
        player_sum, dealer_showing, usable_ace = state
        if usable_ace:  # Usable ace case
            if policy[0] > policy[1]:  # Stick if stick probability is higher
                stick_player_sums_usable.append(player_sum)
                stick_dealer_showings_usable.append(dealer_showing)
            else:  # Hit if hit probability is higher
                hit_player_sums_usable.append(player_sum)
                hit_dealer_showings_usable.append(dealer_showing)
        else:  # Non-usable ace case
            if policy[0] > policy[1]:  # Stick if stick probability is higher
                stick_player_sums_non_usable.append(player_sum)
                stick_dealer_showings_non_usable.append(dealer_showing)
            else:  # Hit if hit probability is higher
                hit_player_sums_non_usable.append(player_sum)
                hit_dealer_showings_non_usable.append(dealer_showing)

    # Create two subplots: one for usable ace and one for non-usable ace
    fig, axs = plt.subplots(1, 2, figsize=(14, 6))

    # Plot for usable ace
    axs[0].scatter(hit_dealer_showings_usable, hit_player_sums_usable, color='red', la
    axs[0].scatter(stick_dealer_showings_usable, stick_player_sums_usable, color='blue
    axs[0].set_title("Usable Ace Policy")
    axs[0].set_xlabel("Dealer's Showing")
    axs[0].set_ylabel("Player's Sum")
    axs[0].set_xticks(np.arange(1, 11))
    axs[0].set_xticklabels(['A', '2', '3', '4', '5', '6', '7', '8', '9', '10'])  # Dea
    axs[0].set_yticks(np.arange(4, 22))  # Player sums from 4 to 21
    axs[0].grid(True)
    axs[0].legend()

    # Plot for non-usable ace
    axs[1].scatter(hit_dealer_showings_non_usable, hit_player_sums_non_usable, color='
    axs[1].scatter(stick_dealer_showings_non_usable, stick_player_sums_non_usable, col
    axs[1].set_title("Non-Usable Ace Policy")
    axs[1].set_xlabel("Dealer's Showing")
    axs[1].set_ylabel("Player's Sum")
    axs[1].set_xticks(np.arange(1, 11))
    axs[1].set_xticklabels(['A', '2', '3', '4', '5', '6', '7', '8', '9', '10'])  # Dea
    axs[1].set_yticks(np.arange(4, 22))  # Player sums from 4 to 21
    axs[1].grid(True)
    axs[1].legend()

    # Display the plot
    plt.tight_layout()
    plt.show()

print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EVERY VISIT")
plot_policy(Q_every_visit, policy)
```

```
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EVERY VISIT
```



I have used chatGPT to help me construct the image to show the policy

The following function will serve for every new method implemented to run some episodes after it's trained and get the statistics

In [45]:
```
def run_episodes_and_get_stats (num_games, q, env, debug):
    wins = 0
    natural_wins = 0
    losses = 0
    draws = 0

    total_return = 0.0

    for i in range(num_games):
        obs, _ = agent.env.reset()
        if debug:
            if i % 1000 == 0:
                print("Obs initial: {}".format(obs))

        episode_reward = 0.0
        while True:
            # Choose an action following the optimal policy (no epsilon greedy)
            arr = np.array(q[obs])
            action = arr.argmax()

            # execute the action and wait for the response from the environment
            new_obs, reward, terminated, truncated, _ = env.step(action)
            if debug:
                if i % 1000 == 0:
                    print("Action: {} -> Obs:{} and reward: {}".format(action, new_obs

            done = terminated or truncated

            obs = new_obs
            episode_reward += reward

            if done:
                if reward == 1.0 or reward == 1.5: # Normal and natural wins
                    wins += 1
                    if reward == 1.5:  # Natural win is a counted both as a win and a
                        natural_wins += 1
                elif reward == -1.0:
                    losses += 1
                else:
                    draws += 1
```

```
                break

        total_return += episode_reward

    win_percentage = (wins / num_games) * 100
    natural_win_percentage = (natural_wins / num_games) * 100
    loss_percentage = (losses / num_games) * 100
    draw_percentage = (draws / num_games) * 100

    if num_games > 1: # For the trial i don't need the stats. I add this if so that it
        print()
        print(f"Statistics for {num_games} games")
        print(f"   Win percentage (counting the natural wins): {win_percentage}%")
        print(f"   Natural win percentage: {natural_win_percentage}%")
        print(f"   Loss percentage: {loss_percentage}%")
        print(f"   Draw percentage: {draw_percentage}%")
        print(f"   Average return: {total_return/num_games}")
```

In [46]:
```
# Trial:
run_episodes_and_get_stats(num_games=1, q=Q_every_visit, env=env, debug=True)
```

```
Obs initial: (6, 10, 0)
Action: 1 -> Obs:(15, 10, 0) and reward: 0.0
Action: 0 -> Obs:(15, 10, 0) and reward: -1.0
```

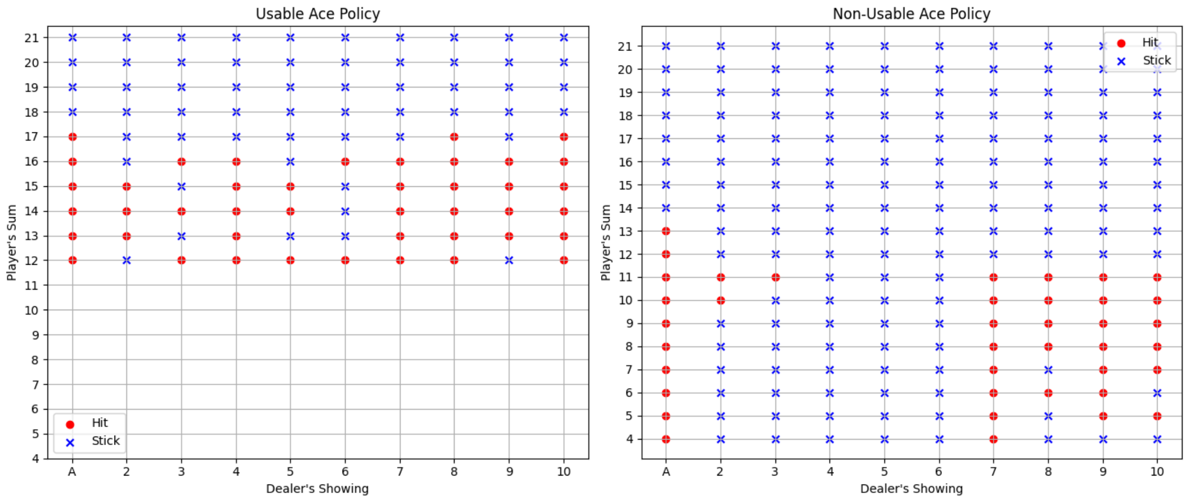
In [47]:
```
run_episodes_and_get_stats (num_games=100000, q=Q_every_visit, env=env, debug=False)

Q_every_visit2, policy = mc_control_on_policy_epsilon_greedy(env, num_episodes=100000,
run_episodes_and_get_stats (num_games=100000, q=Q_every_visit2, env=env, debug=False)
```

```
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.811%
   Natural win percentage: 4.18%
   Loss percentage: 49.586999999999996%
   Draw percentage: 8.602%
   Average return: -0.05686
Episode 100000/100000
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.778999999999996%
   Natural win percentage: 4.083%
   Loss percentage: 51.49%
   Draw percentage: 7.731000000000001%
   Average return: -0.086695
```

In [48]:
```
def mc_control_on_policy_epsilon_greedy_first_visit(env, num_episodes, discount=1.0, e
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The Q action value function.
    # A nested dictionary where state -> (action -> action-value).
    # Initially set to zero
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Rewards
    y = np.zeros(num_episodes, dtype=np.float16)

    for i_episode in range(num_episodes):
        # The policy we are following
        policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

        # Generate an episode and store it
        episode = []
        state, _ = env.reset()
        done = False
```

```
        total_reward = 0
        while not done:
            probs = policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode.append((state, action, reward))
            total_reward += reward
            if done:
                break
            state = next_state

        y[i_episode] = total_reward

        # Track first-visit state-action pairs
        visited_sa_pairs = set()

        # Calculate returns for each time step in reverse
        G = 0
        for i in range(len(episode) - 1, -1, -1):
            state, action, reward = episode[i]
            sa_pair = (state, action)
            # Calculate return G
            G = reward + discount * G

            # Update only if it's the first occurrence in the episode
            if sa_pair not in visited_sa_pairs:
                visited_sa_pairs.add(sa_pair)
                returns_sum[sa_pair] += G
                returns_count[sa_pair] += 1.0
                Q[state][action] = returns_sum[sa_pair] / returns_count[sa_pair]

        # Print progress every 100 episodes
        if i_episode % 100 == 0 and i_episode > 0:
            print(f"\rEpisode {i_episode}/{num_episodes} - Average reward {np.average(
            sys.stdout.flush()

        # Update epsilon
        epsilon = max(epsilon * epsilon_decay, 0.01)

    return Q, policy


Q_first_visit, policy = mc_control_on_policy_epsilon_greedy_first_visit(env, num_episo
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, NO DECAY")
plot_policy(Q_first_visit, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_first_visit, env=env, debug=False)
```

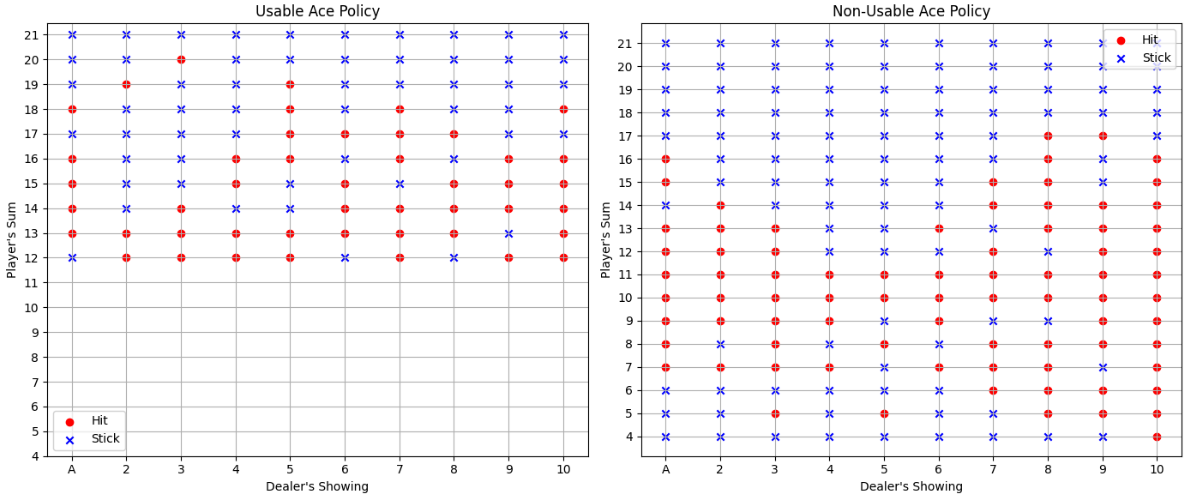
```
Episode 200/500000 - Average reward -0.1500244140625
Episode 499900/500000 - Average reward -0.554892578125555
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, NO DECAY
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.859%
   Natural win percentage: 4.112%
   Loss percentage: 52.44799999999999%
   Draw percentage: 6.6930000000000005%
   Average return: -0.09533
```

In [49]:
```
Q_first_visit_decay, policy = mc_control_on_policy_epsilon_greedy_first_visit(env, num
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, DECAY")
plot_policy(Q_first_visit_decay, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_decay, env=env, debug=Fa
```

```
Episode 200/500000 - Average reward -0.465087890625
Episode 499900/500000 - Average reward -0.194946289062575575
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, DECAY
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.945%
   Natural win percentage: 4.256%
   Loss percentage: 49.33%
   Draw percentage: 8.725%
   Average return: -0.05257
```

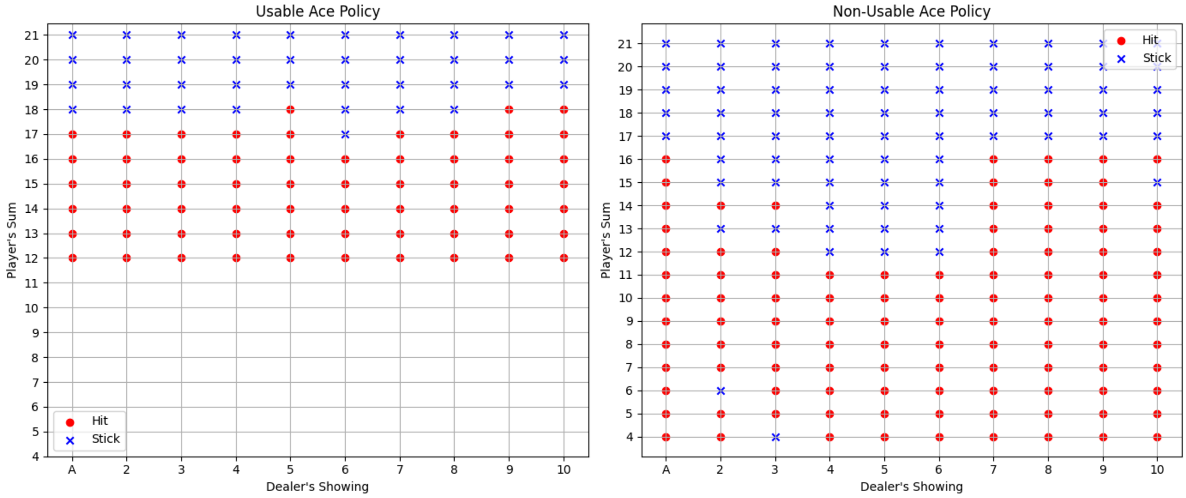
I will now try to work with weighted average.

In [61]:
```
def mc_control_on_policy_epsilon_greedy_first_visit(env, num_episodes, discount=1.0, e
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The Q action value function.
    # A nested dictionary where state -> (action -> action-value).
```

```
    # Initially set to zero
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Rewards
    y = np.zeros(num_episodes, dtype=np.float16)

    for i_episode in range(num_episodes):
        # The policy we are following
        policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

        # Update epsilon
        epsilon = max(epsilon * epsilon_decay, 0.01)

        # Generate an episode and store it
        episode = []
        state, _ = env.reset()
        done = False
        total_reward = 0
        while not done:
            probs = policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode.append((state, action, reward))
            total_reward += reward
            if done:
                break
            state = next_state

        y[i_episode] = total_reward

        # Track first-visit state-action pairs
        visited_sa_pairs = set()

        # Calculate returns for each time step in reverse
        G = 0
        for i in range(len(episode) - 1, -1, -1):
            state, action, reward = episode[i]
            sa_pair = (state, action)
            # Calculate return G
            G = reward + discount * G

            # Update only if it's the first occurrence in the episode
            if sa_pair not in visited_sa_pairs:
                visited_sa_pairs.add(sa_pair)
                returns_sum[sa_pair] += G
                returns_count[sa_pair] += 1.0
                Q[state][action] = Q[state][action] + alpha * (G - Q[state][action])

        # Print progress every 100 episodes
        if i_episode % 100 == 0 and i_episode > 0:
            print(f"\rEpisode {i_episode}/{num_episodes} - Average reward {np.average(
            sys.stdout.flush()

    return Q, policy

Q_first_visit_decay_weighted_avg, policy_optimalMC = mc_control_on_policy_epsilon_gree
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE,
plot_policy(Q_first_visit_decay_weighted_avg, policy_optimalMC)
run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_decay_weighted_avg, env=
```

```
Episode 100/500000 - Average reward -0.25
```

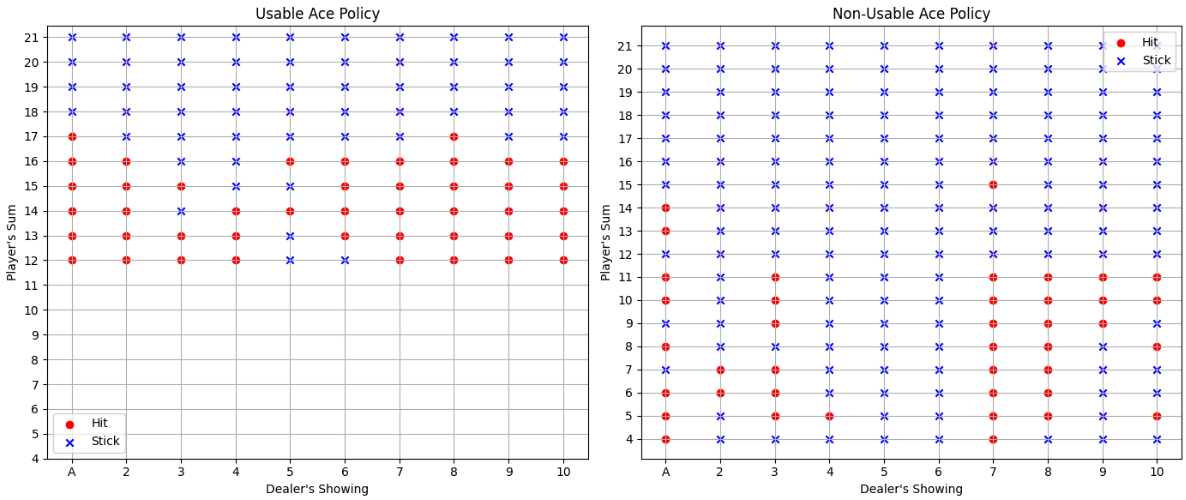
```
Episode 499900/500000 - Average reward 0.04998779296875755 75
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE, DECAY
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 42.595%
   Natural win percentage: 4.165%
   Loss percentage: 47.989%
   Draw percentage: 9.415999999999999%
   Average return: -0.033115
```

In [52]:
```
Q_first_visit_weighted_avg, policy = mc_control_on_policy_epsilon_greedy_first_visit(e
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE,
plot_policy(Q_first_visit_weighted_avg, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_weighted_avg, env=env, d
```

```
Episode 100/500000 - Average reward -0.60009765625
Episode 499900/500000 - Average reward -0.185058593755555
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE, NO DECA
Y
```



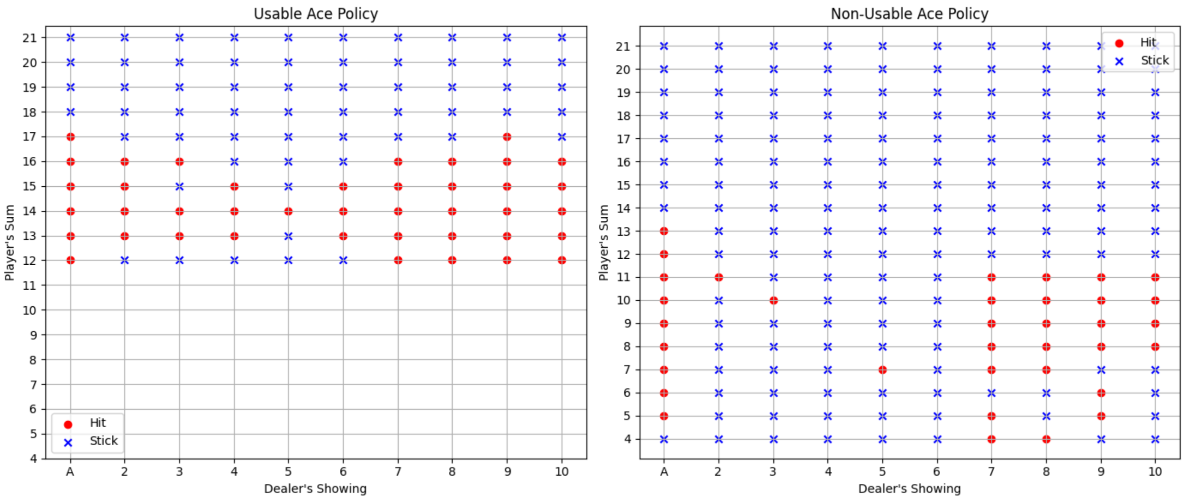

```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.778%
   Natural win percentage: 4.227%
   Loss percentage: 52.664%
   Draw percentage: 6.558%
   Average return: -0.097725
```

I'm going to try Monte Carlo with exploring starts. First and every visit.

In [53]:
```
def mc_control_exploring_starts(env, num_episodes, discount=1.0, epsilon=0.1):

    # We store the sum and number of returns for each state to calculate the average.
```

```
    # We could use an array to store all the returns, but it is inefficient in terms o
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The Q action value function.
    # A nested dictionary whose correspondence is state -> (action -> action-value).
    # Initially we initialize it to zero
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    for i_episode in range(1, num_episodes + 1):
        # We print which episode we are in, useful for debugging.
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        # Generate an episode
        state, _ = env.reset()
        action = np.random.choice(np.arange(env.action_space.n)) # Start with random a

        episode = []
        done = False

        while not done:
            next_state, reward, done, _, _ = env.step(action)
            episode.append((state, action, reward))

            if not done:
                if np.random.rand() < epsilon:
                    action = np.random.choice(np.arange(env.action_space.n))
                else:
                    action = np.argmax(Q[next_state]) # Choose the best action accordi
            state = next_state

        G = 0
        for state, action, reward in reversed(episode):
            sa_pair = (state, action)

            G = reward + discount * G

            # Update the returns_sum and returns_count
            returns_sum[sa_pair] += G
            returns_count[sa_pair] += 1.0
            Q[state][action] = returns_sum[sa_pair] / returns_count[sa_pair]

    # The greedy policy is implicitly improved by using the updated Q values
    def greedy_policy(state):
        """Returns the greedy action for a given state based on Q values."""
        return np.argmax(Q[state])

    return Q, greedy_policy

Q_exploring_every_visit, policy = mc_control_exploring_starts(env, 500000, discount=1,
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND EVERY VI
plot_policy(Q_exploring_every_visit, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_exploring_every_visit, env=env, debu
```

```
Episode 100/500000
Episode 500000/500000
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND EVERY VISIT, NO
DECAY
```



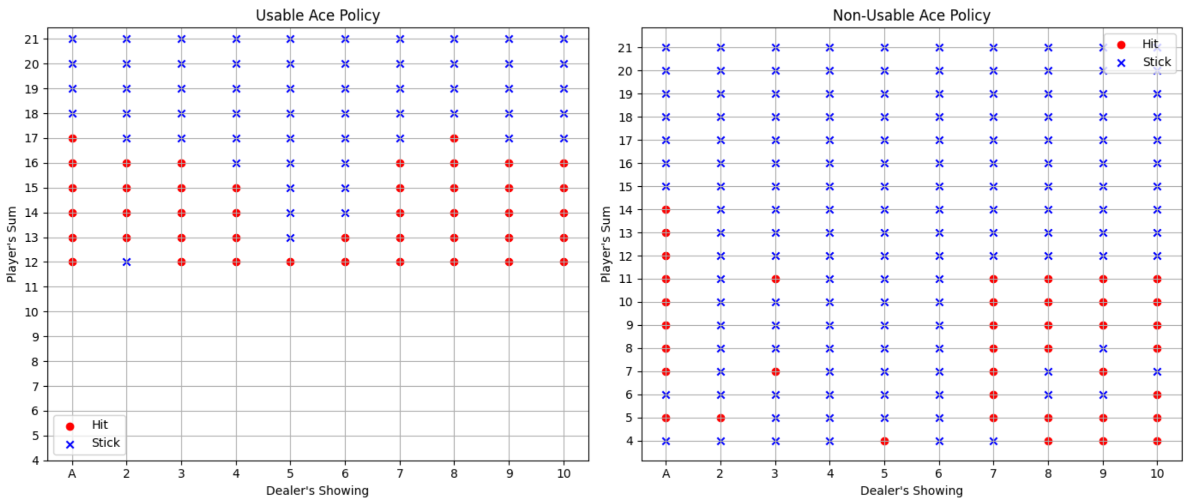

```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.858%
   Natural win percentage: 4.182%
   Loss percentage: 52.397000000000006%
   Draw percentage: 6.744999999999999%
   Average return: -0.09448
```

Now Monte Carlo with exploring starts for first visit only since it seems to perform slightly better

In [54]:
```
def mc_control_exploring_starts_first_visit(env, num_episodes, discount=1.0, epsilon=0

    # We store the sum and count of returns for each state-action pair
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # Initialize Q to zero for each state-action pair
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    for i_episode in range(1, num_episodes + 1):
        # Print episode number every 100 episodes for tracking progress
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        # Generate an episode with exploring starts
        state, _ = env.reset()
        action = np.random.choice(np.arange(env.action_space.n))  # Start with a rando

        episode = []
        done = False

        while not done:
            next_state, reward, done, _, _ = env.step(action)
            episode.append((state, action, reward))

            if not done:
                if np.random.rand() < epsilon:
                    action = np.random.choice(np.arange(env.action_space.n))
                else:
                    action = np.argmax(Q[next_state])  # Choose the best action accord
            state = next_state

        G = 0
        visited_sa_pairs = set()  # Track visited state-action pairs for first-visit

        for state, action, reward in reversed(episode):
            sa_pair = (state, action)
```

```
            G = reward + discount * G

            # Update only on the first visit to each state-action pair in the episode
            if sa_pair not in visited_sa_pairs:
                visited_sa_pairs.add(sa_pair)

                # Update the returns_sum and returns_count
                returns_sum[sa_pair] += G
                returns_count[sa_pair] += 1.0
                Q[state][action] = returns_sum[sa_pair] / returns_count[sa_pair]

    # Define the greedy policy based on Q values
    def greedy_policy(state):
        """Returns the greedy action for a given state based on Q values."""
        return np.argmax(Q[state])

    return Q, greedy_policy

# Run the first-visit MC control with exploring starts
Q_exploring_first_visit, policy = mc_control_exploring_starts_first_visit(env, 500000,
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND FIRST VI
plot_policy(Q_exploring_first_visit, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_exploring_first_visit, env=env, debu
```

```
Episode 200/500000
Episode 500000/500000
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND FIRST VISIT
```

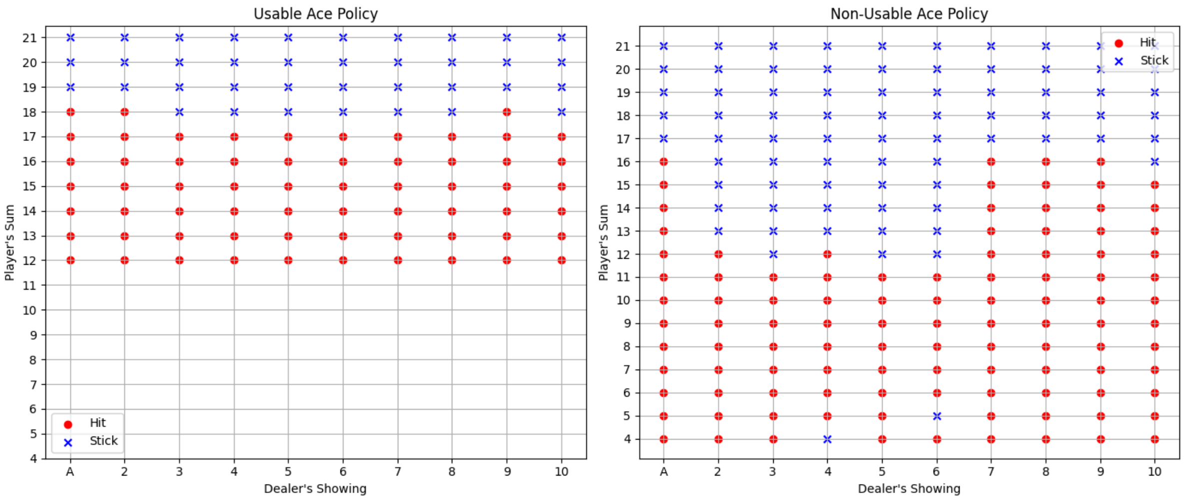



```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.86%
   Natural win percentage: 4.176%
   Loss percentage: 52.473000000000006%
   Draw percentage: 6.666999999999999%
   Average return: -0.09525
```

I'll try an epsilon of 0.1

In [78]:
```
# Run the first-visit MC control with exploring starts
Q_exploring_first_visit, policy = mc_control_exploring_starts_first_visit(env, 500000,
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND FIRST VI
plot_policy(Q_exploring_first_visit, policy)
run_episodes_and_get_stats (num_games=100000, q=Q_exploring_first_visit, env=env, debu
```

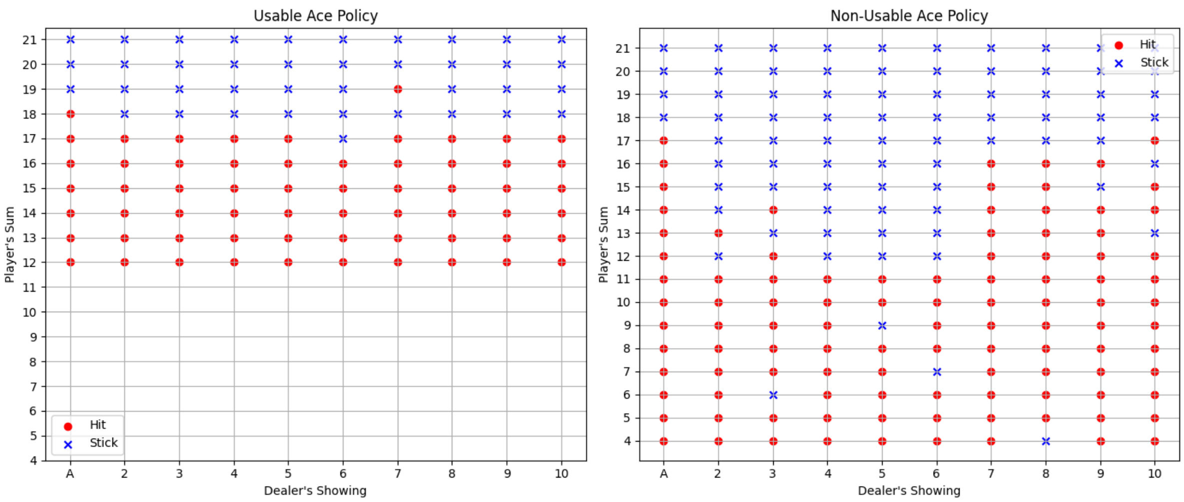
```
Episode 500000/500000
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, EXPLORING STARTS, AND FIRST VISIT
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 43.275999999999996%
   Natural win percentage: 4.112%
   Loss percentage: 47.589999999999996%
   Draw percentage: 9.134%
   Average return: -0.02258
```

In [79]:
```python
q, policy = mc_control_on_policy_epsilon_greedy_first_visit(env, num_episodes=500000,
print()
print("MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE,
plot_policy(q, policy)
run_episodes_and_get_stats (num_games=100000, q=q, env=env, debug=False)
```

```
Episode 499900/500000 - Average reward 0.0850585937556255755
MC CONTROL ON-POLICY WITH EPSILON GREEDY POLICY, FIRST VISIT, WEIGHTED AVERAGE, DECAY
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 42.657000000000004%
   Natural win percentage: 4.178%
   Loss percentage: 48.729%
   Draw percentage: 8.613999999999999%
   Average return: -0.03983
```

I have tried to move a little bit the parameters such as the number of episodes, the epsilon and the discount. 500000 epsiodes seem to be a bit better than 100000, that's why in some places, i'm training the agent with 500000 episodes. Nevertheless, discount to 1 seems to be a good fit to balance the optimality of the methods implemented.

ϵ is the probability of exploring (choosing a random action) rather than exploiting (choosing the best-known action based on current Q-values). If epsilon is 1, it will fully explore, each random is chosen randomly. Instead, epsilon to 0 means it's purely greedy. the agent chooses the action with the highest Q-value (argmax). Setting epsilon to one means that the agent explores actions uniformly at random ensuring to fully explore the action space. If epsilon is 1, it can try both hitting and sticking in situations where it might not be obvious what to do. However, if the epsilon is set to a lower number like 0.1, it works pretty well in the two examples i tried. So it should be important to keep a balance of both values and try to find a minima in th e landscape. I tried setting epsilon to 0.9 and it performed worse than with epsilon=1, so i probably hit a local minim either with epsilon = 1 or with epsilon = 0.1.

Setting the discount factor to 1 makes the agent maximize the reward. If we remember the formula of the return, each reward that is more far in advance is multiplied by the discount factor to the power of how far it is. So if it's set to 1, it will multiply all rewards by 1. Meaning that all rewards have the same importance. This is suitable for games where the last reward is what matters, like the case of blackjack.

Part 3. TD learning

The objective of this section is to estimate the optimal policy using TD learning methods. Specifically, you have to implement the SARSA algorithm.

Questions (**2.5 points**):

1. Implement the algorithm.
2. Comment and justify all the parameters.
3. Print on the screen the optimal policy found for each state.
4. Using the trained agent, simulate 100,000 games and calculate the agent's return (total accumulated reward).
5. Additionally, calculate the % of wins, natural wins, losses and draws.

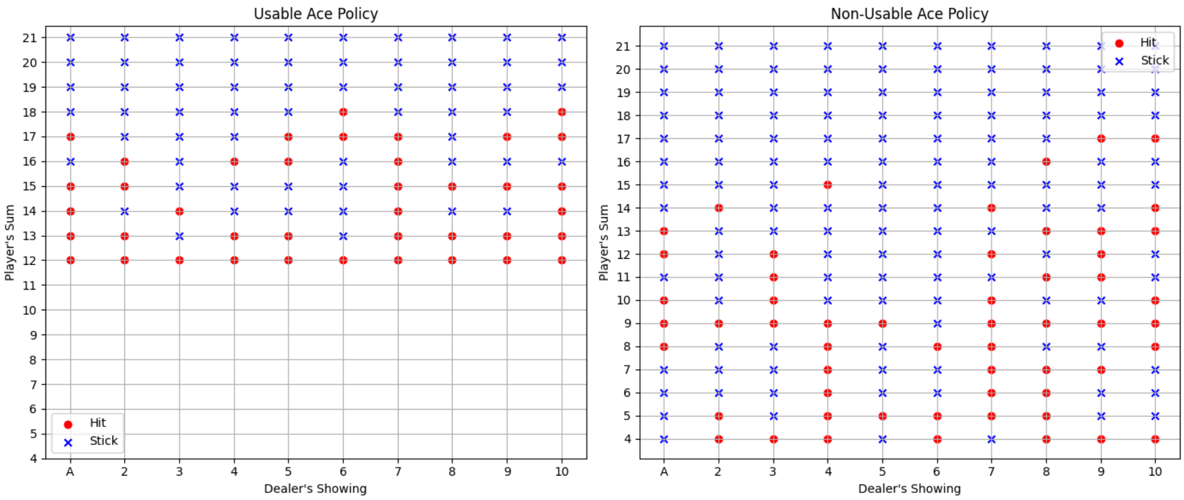
In [55]:
```
def epsilon_greedy_policy(Q, state, nA, epsilon):

    probs = np.ones(nA, dtype=float) * epsilon / nA
    best_action = np.argmax(Q[state])
    probs[best_action] += (1.0 - epsilon)

    return probs


def SARSA(env, episodes:int, learning_rate:float, discount:float, epsilon:float):
    # Link actions to states
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Rewards
    y = np.zeros(episodes) # rewards per episode
    a = defaultdict(lambda: 0)
    wins = 0

    for episode in range(episodes):
        state, _ = env.reset()
        done = False

        # Select and execute an action
        probs = epsilon_greedy_policy(Q, state, env.action_space.n, epsilon)
        action = np.random.choice(np.arange(len(probs)), p=probs)

        # train bucle for each episode
        step = 1
```

```
        total_reward = 0
        while not done:
            a[action] += 1

            # Execute action
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            if not done:
                # Select and execute action
                probs = epsilon_greedy_policy(Q, next_state, env.action_space.n, epsil
                next_action = np.random.choice(np.arange(len(probs)), p=probs)
            else:
                next_action = None

            # Update TD
            if not done:
                td_target = reward + discount * Q[next_state][next_action]
            else:
                td_target = reward

            td_error = td_target - Q[state][action]
            Q[state][action] += learning_rate * td_error

            total_reward += reward

            if done:
                y[episode] = total_reward
                if reward > 0:
                    wins += 1
                break

            state = next_state
            action = next_action
            step += 1

        # We print which episode we are in, useful for debugging.
        if episode % 100 == 0 and episode > 0:
            print("\rEpisode {:8d}/{:8d} - Average reward {:.2f}".format(episode, epis
            sys.stdout.flush()

    return y, Q

def extract_policy(Q, env):
    policy = {}
    for state in Q.keys():
        if state[0] <= 21:  # Only consider valid states (sum <= 21)
            best_action = np.argmax(Q[state])
            policy[state] = best_action
    return policy
```

In [62]:

```
y, q = SARSA(env, episodes=500000, learning_rate=0.3, discount=1, epsilon=1)
policy = extract_policy(q, env)
print("SARSA policy")
plot_policy(q, policy)
run_episodes_and_get_stats (num_games=100000, q=q, env=env, debug=False)
```

```
Episode   499900/  500000 - Average reward -0.12SARSA policy
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.386%
   Natural win percentage: 4.201%
   Loss percentage: 52.703%
   Draw percentage: 6.9110000000000005%
   Average return: -0.102165
```

I'll now try to implement SARSA with decay

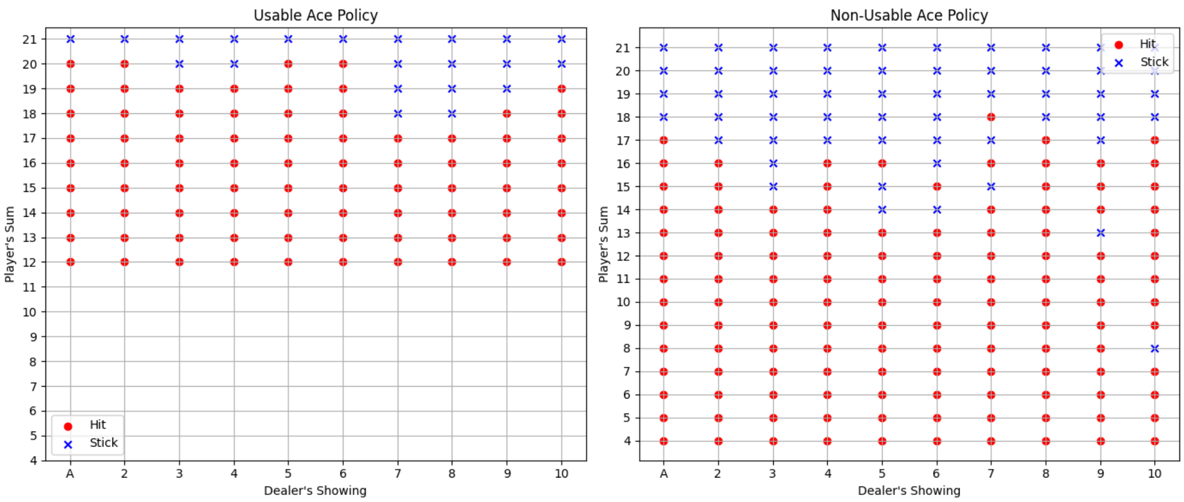
In [63]:
```python
def SARSA_decay(env, episodes:int, learning_rate:float, discount:float, epsilon:float,
    # Link actions to states
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Rewards
    y = np.zeros(episodes) # rewards per episode
    a = defaultdict(lambda: 0)
    wins = 0

    for episode in range(episodes):
        state, _ = env.reset()
        done = False

        # Select and execute an action
        probs = epsilon_greedy_policy(Q, state, env.action_space.n, epsilon)
        action = np.random.choice(np.arange(len(probs)), p=probs)

        # train bucle for each episode
        step = 1
        total_reward = 0
        while not done:
            a[action] += 1

            # Execute action
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            if not done:
                # Select and execute action
                probs = epsilon_greedy_policy(Q, next_state, env.action_space.n, epsil
                next_action = np.random.choice(np.arange(len(probs)), p=probs)
            else:
                next_action = None

            # Update TD
            if not done:
                td_target = reward + discount * Q[next_state][next_action]
            else:
```

```
                td_target = reward

            td_error = td_target - Q[state][action]
            Q[state][action] += learning_rate * td_error

            total_reward += reward

            if done:
                y[episode] = total_reward
                if reward > 0:
                    wins += 1
                break

            state = next_state
            action = next_action
            step += 1

        # We print which episode we are in, useful for debugging.
        if episode % 100 == 0 and episode > 0:
            print("\rEpisode {:8d}/{:8d} - Average reward {:.2f}".format(episode, epis
            sys.stdout.flush()

        epsilon = max(epsilon*decay, 0.01)

    return y, Q
```

In [64]:

```
_, q_sarsa = SARSA_decay(env, episodes=500000, learning_rate=0.3, discount=1, epsilon=
policy_optimal_sarsa = extract_policy(q_sarsa, env)
print("SARSA policy with decay")
plot_policy(q_sarsa, policy_optimal_sarsa)
run_episodes_and_get_stats (num_games=100000, q=q_sarsa, env=env, debug=False)
```

```
Episode   499900/   500000 - Average reward -0.20SARSA policy with decay
```




```
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.794000000000004%
   Natural win percentage: 4.062%
   Loss percentage: 49.762%
   Draw percentage: 9.443999999999999%
   Average return: -0.06937
```

With the sarsa i did notice a difference when the decay was 0.999. Decay makes the epsilon go to lower values as the iterations go by. so we start with a high epsilon like 1 to favor exploration and as we go through the episode, we exploit more than explore. this makes sense. however, we have to try and not make the decay go too slow since Blackjack doesn't have long epsiodes. If it were higher, we would be trying to converge faster than we should.

Part 4. Comparison of the algorithms

In this section, we will make a comparison among the algorithms.

We will compare the performance of the algorithms when changing the number of episodes, the discount factor and the *learning rate* values (in the case of the SARSA method).

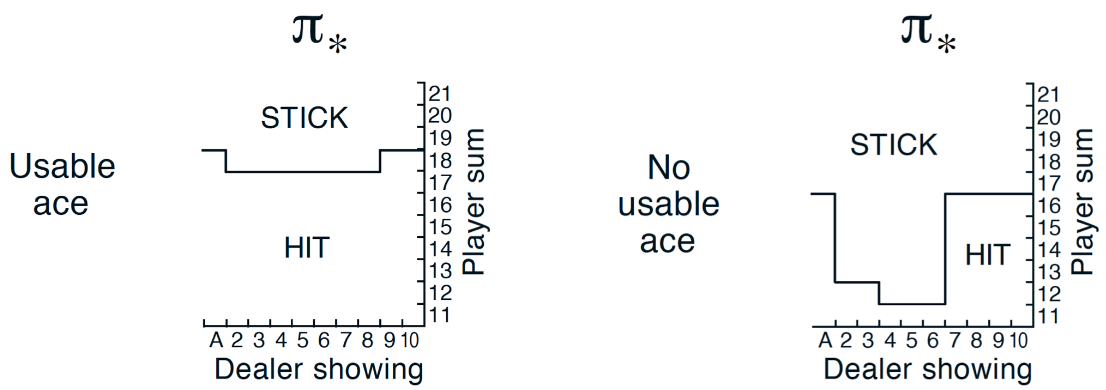
For each exercise, the results must be presented and justified.

Note:

- It is recommended to run the simulations multiple times for each exercise, as these are random, and to comment on the most frequent result or the average of these.

4.1. Comparison to the optimal policy

The optimal policy for this problem, described by Sutton & Barto is depicted in the following image:



Questions (**1 point**):

- Compare the *optimal* policies of the naïve, Monte Carlo and SARSA methods to the optimal one provided by Sutton & Barto.
- Comment on the results and justify your answer.

In [65]:
```
# Create Sutton & Barto's Blackjack optimal policy
s_and_b = {(4, 1, 0):1, (4, 2, 0):1, (4, 3, 0):1, (4, 4, 0):1, (4, 5, 0):1, (4, 6, 0):
           (5, 1, 0):1, (5, 2, 0):1, (5, 3, 0):1, (5, 4, 0):1, (5, 5, 0):1, (5, 6, 0):
           (6, 1, 0):1, (6, 2, 0):1, (6, 3, 0):1, (6, 4, 0):1, (6, 5, 0):1, (6, 6, 0):
           (7, 1, 0):1, (7, 2, 0):1, (7, 3, 0):1, (7, 4, 0):1, (7, 5, 0):1, (7, 6, 0):
           (8, 1, 0): 1, (8, 2, 0): 1, (8, 3, 0): 1, (8, 4, 0): 1, (8, 5, 0): 1, (8, 6
           (9, 1, 0): 1,  (9, 2, 0): 1, (9, 3, 0): 1, (9, 4, 0): 1, (9, 5, 0): 1, (9,
           (10, 1, 0): 1, (10, 2, 0): 1, (10, 3, 0): 1, (10, 4, 0): 1, (10, 5, 0): 1,
           (11, 1, 0): 1, (11, 2, 0): 1, (11, 3, 0): 1, (11, 4, 0): 1, (11, 5, 0): 1,

    (12, 1, 0): 1, (12, 1, 1): 1,(12, 2, 0): 1, (12, 2, 1): 1,(12, 3, 0): 1, (12, 3, 1
    (12, 6, 0): 0, (12, 6, 1): 1,(12, 7, 0): 1, (12, 7, 1): 1,(12, 8, 0): 1, (12, 8, 1

    (13, 1, 0): 1, (13, 1, 1): 1, (13, 2, 0): 0, (13, 2, 1): 1, (13, 3, 0): 0, (13, 3,
    (13, 6, 0): 0, (13, 6, 1): 1, (13, 7, 0): 1, (13, 7, 1): 1, (13, 8, 0): 1, (13, 8,

    (14, 1, 0): 1, (14, 1, 1): 1, (14, 2, 0): 0, (14, 2, 1): 1, (14, 3, 0): 0, (14, 3,
    (14, 6, 0): 0, (14, 6, 1): 1, (14, 7, 0): 1, (14, 7, 1): 1, (14, 8, 0): 1, (14, 8,
```

```
    (15, 1, 0): 1, (15, 1, 1): 1, (15, 2, 0): 0, (15, 2, 1): 1, (15, 3, 0): 0, (15, 3,
    (15, 6, 0): 0, (15, 6, 1): 1, (15, 7, 0): 1, (15, 7, 1): 1, (15, 8, 0): 1, (15, 8,

    (16, 1, 0): 1, (16, 1, 1): 1, (16, 2, 0): 0, (16, 2, 1): 1, (16, 3, 0): 0, (16, 3,
    (16, 6, 0): 0, (16, 6, 1): 1, (16, 7, 0): 1, (16, 7, 1): 1, (16, 8, 0): 1, (16, 8,

    (17, 1, 0): 0, (17, 1, 1): 1, (17, 2, 0): 0, (17, 2, 1): 1, (17, 3, 0): 0, (17, 3,
    (17, 6, 0): 0, (17, 6, 1): 1, (17, 7, 0): 0, (17, 7, 1): 1, (17, 8, 0): 0, (17, 8,

    (18, 1, 0): 0, (18, 1, 1): 1, (18, 2, 0): 0, (18, 2, 1): 0, (18, 3, 0): 0, (18, 3,
    (18, 6, 0): 0, (18, 6, 1): 0, (18, 7, 0): 0, (18, 7, 1): 0, (18, 8, 0): 0, (18, 8,

    (19, 1, 0): 0, (19, 1, 1): 0, (19, 2, 0): 0, (19, 2, 1): 0, (19, 3, 0): 0, (19, 3,
    (19, 6, 0): 0, (19, 6, 1): 0, (19, 7, 0): 0, (19, 7, 1): 0, (19, 8, 0): 0, (19, 8,

    (20, 1, 0): 0, (20, 1, 1): 0, (20, 2, 0): 0, (20, 2, 1): 0, (20, 3, 0): 0, (20, 3,
    (20, 6, 0): 0, (20, 6, 1): 0, (20, 7, 0): 0, (20, 7, 1): 0, (20, 8, 0): 0, (20, 8,

    (21, 1, 0): 0, (21, 1, 1): 0, (21, 2, 0): 0, (21, 2, 1): 0, (21, 3, 0): 0, (21, 3,
    (21, 6, 0): 0, (21, 6, 1): 0, (21, 7, 0): 0, (21, 7, 1): 0, (21, 8, 0): 0, (21, 8,
}
```

In [66]:

```python
# Naive policy

from itertools import islice

def simplified_policy(state):
    player_sum = state[0]
    return 0 if player_sum >= 20 else 1

naive_policy = {}
for key in s_and_b.keys(): # Same keys for both policies
    naive_policy[key] = simplified_policy(key)

# Print only the first five entries
for state, action in islice(naive_policy.items(), 5):
    print(f"State: {state}, Action: {action}")
```

```
State: (4, 1, 0), Action: 1
State: (4, 2, 0), Action: 1
State: (4, 3, 0), Action: 1
State: (4, 4, 0), Action: 1
State: (4, 5, 0), Action: 1
```

In [67]:

```python
policy_optimalMC_dict = {}
for state in Q_first_visit_decay_weighted_avg.keys():
    action_probabilities = policy_optimalMC(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict[state] = 0
    else:
        policy_optimalMC_dict[state] = 1

# Print only the first five entries
for state, action in islice(policy_optimalMC_dict.items(), 5):
    print(f"State: {state}, Action: {action}")
```

```
State: (16, 4, 0), Action: 0
State: (7, 6, 0), Action: 1
State: (18, 6, 1), Action: 0
State: (12, 2, 0), Action: 1
State: (15, 2, 0), Action: 0
```

In [68]:

```python
# Print only the first five entries
for state, action in islice(policy_optimal_sarsa.items(), 5):
```

```
    print(f"State: {state}, Action: {action}")
```

```
State: (13, 2, 0), Action: 1
State: (18, 4, 1), Action: 1
State: (9, 2, 0), Action: 1
State: (9, 1, 0), Action: 1
State: (13, 10, 0), Action: 1
```

In [69]:
```
print(f"{len(s_and_b)}, {len(naive_policy)}, {len(policy_optimalMC_dict)}, {len(policy
```

```
280, 280, 280, 280
```

In [70]:
```
policies = {
    "Naive": naive_policy,
    "Monte Carlo": policy_optimalMC_dict,
    "SARSA": policy_optimal_sarsa
}

for name, policy in policies.items():
    print(f"Comparison between Sutton & Barto's policy and the {name} policy")

    common_states = set(s_and_b.keys()) & set(policy.keys())
    agree = sum(s_and_b[state] == policy[state] for state in common_states)
    disagree = len(common_states) - agree

    print(f"Common states: {common_states}")
    print(f"States with agreement: {agree}")
    print(f"States with disagreement: {disagree}")
    print(f"Agreement percentage: {(agree / len(common_states)) * 100:.2f}%")
    print("=" * 50)
```

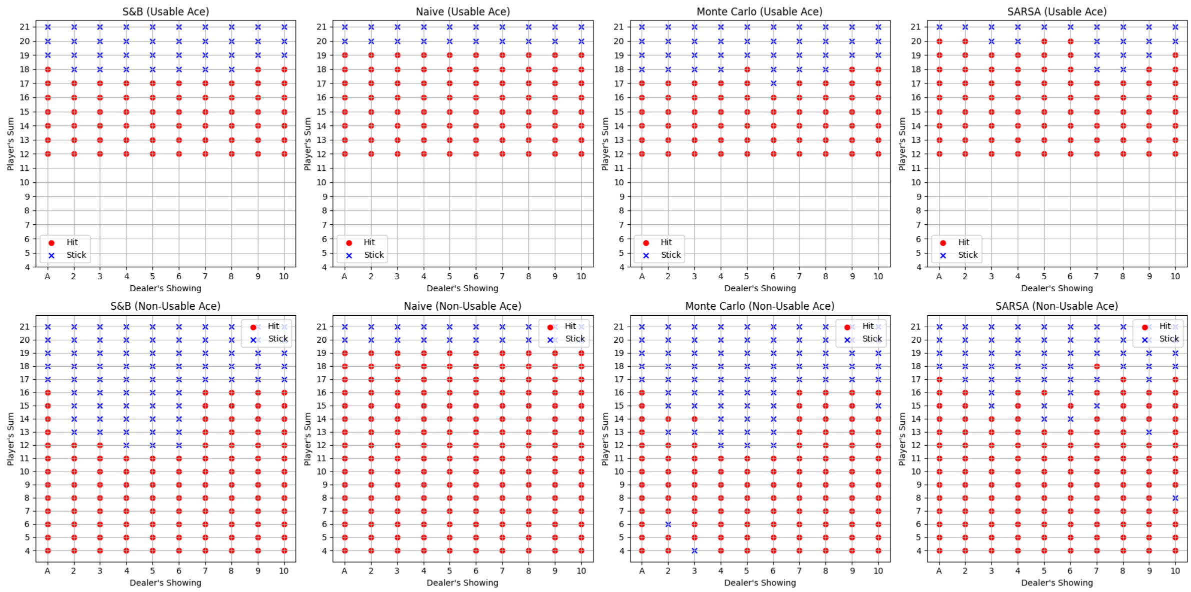
```
Comparison between Sutton & Barto's policy and the Naive policy
Common states: {(8, 6, 0), (12, 10, 0), (18, 10, 0), (15, 5, 0), (10, 4, 0), (7, 9, 0),
(21, 6, 0), (16, 5, 0), (17, 10, 1), (20, 6, 0), (10, 8, 0), (21, 4, 0), (11, 4, 0), (1
2, 9, 1), (5, 7, 0), (13, 4, 1), (6, 2, 0), (21, 8, 0), (14, 3, 0), (11, 8, 0), (18, 7,
0), (16, 6, 1), (19, 2, 0), (16, 7, 0), (20, 7, 1), (10, 10, 0), (9, 3, 0), (15, 6, 1),
(5, 9, 0), (20, 1, 0), (4, 2, 0), (21, 7, 1), (14, 5, 0), (11, 10, 0), (15, 10, 1), (1
7, 1, 0), (20, 9, 1), (4, 6, 0), (9, 5, 0), (13, 5, 1), (19, 5, 1), (19, 8, 0), (14, 4,
1), (13, 10, 0), (17, 3, 0), (7, 3, 0), (4, 8, 0), (18, 1, 1), (8, 9, 0), (12, 2, 0),
(18, 2, 0), (13, 7, 1), (19, 7, 1), (21, 9, 0), (14, 6, 1), (15, 1, 0), (12, 6, 0), (1
7, 2, 1), (7, 5, 0), (14, 10, 1), (16, 1, 0), (17, 6, 1), (20, 2, 0), (21, 10, 1), (12,
5, 1), (16, 9, 1), (5, 3, 0), (17, 4, 1), (6, 9, 0), (18, 3, 0), (19, 9, 0), (16, 3,
0), (20, 3, 1), (17, 8, 1), (20, 4, 0), (14, 7, 1), (15, 2, 1), (12, 7, 1), (5, 5, 0),
(20, 8, 0), (21, 3, 1), (14, 1, 0), (11, 6, 0), (15, 7, 0), (16, 2, 1), (20, 5, 1), (1
0, 6, 0), (9, 1, 0), (13, 1, 1), (15, 4, 1), (13, 2, 0), (19, 1, 1), (18, 4, 1), (15,
8, 1), (13, 6, 0), (17, 10, 0), (4, 4, 0), (18, 8, 1), (8, 5, 0), (12, 9, 0), (18, 9,
0), (13, 3, 1), (19, 3, 1), (13, 4, 0), (21, 5, 0), (14, 2, 1), (16, 4, 0), (13, 8, 0),
(17, 9, 1), (7, 1, 0), (18, 10, 1), (8, 7, 0), (5, 6, 0), (16, 8, 0), (21, 6, 1), (6,
1, 0), (21, 7, 0), (11, 7, 0), (12, 1, 1), (15, 10, 0), (16, 5, 1), (20, 6, 1), (6, 5,
0), (10, 9, 0), (19, 5, 0), (16, 10, 0), (20, 10, 1), (14, 3, 1), (14, 4, 0), (11, 9,
0), (15, 9, 1), (12, 3, 1), (5, 1, 0), (6, 7, 0), (14, 8, 0), (18, 1, 0), (15, 3, 0),
(19, 4, 1), (19, 7, 0), (20, 1, 1), (10, 2, 0), (14, 6, 0), (9, 8, 0), (13, 9, 0), (17,
2, 0), (4, 7, 0), (14, 10, 0), (19, 6, 1), (17, 6, 0), (8, 1, 0), (12, 5, 0), (9, 10,
0), (13, 10, 1), (18, 5, 0), (19, 10, 1), (21, 1, 0), (7, 4, 0), (4, 9, 0), (14, 9, 1),
(18, 2, 1), (17, 5, 1), (17, 8, 0), (10, 3, 0), (7, 8, 0), (21, 9, 1), (18, 6, 1), (8,
3, 0), (12, 4, 1), (5, 2, 0), (12, 7, 0), (21, 3, 0), (11, 3, 0), (12, 8, 1), (16, 1,
1), (16, 2, 0), (20, 2, 1), (17, 7, 1), (10, 5, 0), (7, 10, 0), (19, 1, 0), (5, 4, 0),
(16, 6, 0), (20, 7, 0), (21, 2, 1), (11, 5, 0), (15, 5, 1), (12, 10, 1), (15, 6, 0),
(5, 8, 0), (16, 3, 1), (20, 4, 1), (6, 3, 0), (18, 8, 0), (19, 3, 0), (20, 8, 1), (20,
9, 0), (21, 4, 1), (14, 2, 0), (9, 4, 0), (15, 7, 1), (5, 10, 0), (13, 5, 0), (17, 9,
0), (4, 3, 0), (21, 8, 1), (18, 7, 1), (8, 4, 0), (9, 2, 0), (13, 2, 1), (16, 7, 1), (1
9, 2, 1), (8, 8, 0), (12, 1, 0), (9, 6, 0), (13, 6, 1), (13, 7, 0), (4, 5, 0), (14, 5,
1), (18, 9, 1), (17, 1, 1), (20, 10, 0), (21, 5, 1), (8, 10, 0), (12, 3, 0), (15, 9,
0), (13, 8, 1), (16, 4, 1), (19, 8, 1), (6, 4, 0), (21, 10, 0), (7, 2, 0), (16, 8, 1),
(19, 4, 0), (16, 9, 0), (17, 3, 1), (6, 8, 0), (10, 1, 0), (7, 6, 0), (17, 4, 0), (12,
2, 1), (20, 3, 0), (6, 6, 0), (14, 7, 0), (11, 1, 0), (15, 1, 1), (12, 6, 1), (15, 2,
0), (16, 10, 1), (19, 6, 0), (6, 10, 0), (9, 7, 0), (19, 10, 0), (20, 5, 0), (14, 8,
1), (14, 9, 0), (15, 3, 1), (15, 4, 0), (13, 1, 0), (17, 5, 0), (4, 10, 0), (18, 3, 1),
(18, 6, 0), (12, 4, 0), (9, 9, 0), (13, 9, 1), (15, 8, 0), (18, 4, 0), (19, 9, 1), (10,
7, 0), (12, 8, 0), (13, 3, 0), (17, 7, 0), (7, 7, 0), (4, 1, 0), (14, 1, 1), (18, 5,
1), (8, 2, 0), (21, 1, 1), (21, 2, 0), (11, 2, 0)}
States with agreement: 210
States with disagreement: 70
Agreement percentage: 75.00%
==================================================
Comparison between Sutton & Barto's policy and the Monte Carlo policy
Common states: {(8, 6, 0), (18, 10, 0), (12, 10, 0), (15, 5, 0), (10, 4, 0), (7, 9, 0),
(21, 6, 0), (16, 5, 0), (17, 10, 1), (20, 6, 0), (10, 8, 0), (21, 4, 0), (11, 4, 0), (1
2, 9, 1), (5, 7, 0), (13, 4, 1), (6, 2, 0), (21, 8, 0), (14, 3, 0), (18, 7, 0), (11, 8,
0), (16, 6, 1), (19, 2, 0), (16, 7, 0), (20, 7, 1), (10, 10, 0), (9, 3, 0), (15, 6, 1),
(5, 9, 0), (20, 1, 0), (21, 7, 1), (4, 2, 0), (14, 5, 0), (11, 10, 0), (15, 10, 1), (1
7, 1, 0), (20, 9, 1), (4, 6, 0), (9, 5, 0), (19, 5, 1), (19, 8, 0), (13, 5, 1), (14, 4,
1), (13, 10, 0), (17, 3, 0), (7, 3, 0), (4, 8, 0), (18, 1, 1), (8, 9, 0), (12, 2, 0),
(18, 2, 0), (19, 7, 1), (13, 7, 1), (21, 9, 0), (14, 6, 1), (15, 1, 0), (12, 6, 0), (1
7, 2, 1), (7, 5, 0), (14, 10, 1), (16, 1, 0), (17, 6, 1), (20, 2, 0), (21, 10, 1), (12,
5, 1), (16, 9, 1), (5, 3, 0), (17, 4, 1), (6, 9, 0), (18, 3, 0), (19, 9, 0), (16, 3,
0), (20, 3, 1), (17, 8, 1), (20, 4, 0), (14, 7, 1), (15, 2, 1), (12, 7, 1), (5, 5, 0),
(20, 8, 0), (21, 3, 1), (14, 1, 0), (11, 6, 0), (15, 7, 0), (16, 2, 1), (20, 5, 1), (1
0, 6, 0), (9, 1, 0), (15, 4, 1), (19, 1, 1), (13, 2, 0), (13, 1, 1), (18, 4, 1), (15,
8, 1), (13, 6, 0), (17, 10, 0), (4, 4, 0), (18, 8, 1), (8, 5, 0), (18, 9, 0), (12, 9,
0), (19, 3, 1), (13, 3, 1), (13, 4, 0), (21, 5, 0), (14, 2, 1), (16, 4, 0), (13, 8, 0),
(17, 9, 1), (7, 1, 0), (18, 10, 1), (8, 7, 0), (16, 8, 0), (5, 6, 0), (21, 6, 1), (6,
1, 0), (21, 7, 0), (11, 7, 0), (15, 10, 0), (16, 5, 1), (12, 1, 1), (20, 6, 1), (6, 5,
0), (10, 9, 0), (19, 5, 0), (16, 10, 0), (20, 10, 1), (14, 3, 1), (14, 4, 0), (11, 9,
0), (15, 9, 1), (12, 3, 1), (5, 1, 0), (6, 7, 0), (14, 8, 0), (18, 1, 0), (15, 3, 0),
```

```
(19, 7, 0), (19, 4, 1), (20, 1, 1), (10, 2, 0), (14, 6, 0), (9, 8, 0), (13, 9, 0), (17,
2, 0), (4, 7, 0), (14, 10, 0), (19, 6, 1), (17, 6, 0), (8, 1, 0), (12, 5, 0), (9, 10,
0), (19, 10, 1), (13, 10, 1), (18, 5, 0), (21, 1, 0), (7, 4, 0), (4, 9, 0), (14, 9, 1),
(18, 2, 1), (17, 5, 1), (21, 9, 1), (10, 3, 0), (17, 8, 0), (7, 8, 0), (18, 6, 1), (8,
3, 0), (12, 7, 0), (5, 2, 0), (12, 4, 1), (21, 3, 0), (11, 3, 0), (12, 8, 1), (16, 1,
1), (16, 2, 0), (20, 2, 1), (17, 7, 1), (10, 5, 0), (7, 10, 0), (19, 1, 0), (16, 6, 0),
(5, 4, 0), (20, 7, 0), (21, 2, 1), (11, 5, 0), (15, 5, 1), (12, 10, 1), (15, 6, 0), (5,
8, 0), (16, 3, 1), (20, 4, 1), (6, 3, 0), (18, 8, 0), (19, 3, 0), (20, 8, 1), (20, 9,
0), (21, 4, 1), (14, 2, 0), (9, 4, 0), (15, 7, 1), (13, 5, 0), (17, 9, 0), (5, 10, 0),
(21, 8, 1), (4, 3, 0), (18, 7, 1), (8, 4, 0), (9, 2, 0), (19, 2, 1), (13, 2, 1), (16,
7, 1), (8, 8, 0), (12, 1, 0), (9, 6, 0), (13, 6, 1), (13, 7, 0), (4, 5, 0), (14, 5, 1),
(18, 9, 1), (20, 10, 0), (21, 5, 1), (17, 1, 1), (8, 10, 0), (16, 4, 1), (12, 3, 0), (1
5, 9, 0), (13, 8, 1), (19, 8, 1), (6, 4, 0), (21, 10, 0), (7, 2, 0), (16, 8, 1), (19,
4, 0), (16, 9, 0), (17, 3, 1), (6, 8, 0), (17, 4, 0), (7, 6, 0), (10, 1, 0), (12, 2,
1), (20, 3, 0), (6, 6, 0), (14, 7, 0), (11, 1, 0), (15, 1, 1), (12, 6, 1), (15, 2, 0),
(16, 10, 1), (19, 6, 0), (6, 10, 0), (9, 7, 0), (19, 10, 0), (20, 5, 0), (14, 8, 1), (1
4, 9, 0), (15, 3, 1), (15, 4, 0), (13, 1, 0), (17, 5, 0), (4, 10, 0), (18, 3, 1), (18,
6, 0), (18, 4, 0), (9, 9, 0), (15, 8, 0), (19, 9, 1), (12, 4, 0), (13, 9, 1), (10, 7,
0), (12, 8, 0), (13, 3, 0), (17, 7, 0), (7, 7, 0), (4, 1, 0), (14, 1, 1), (18, 5, 1),
(8, 2, 0), (21, 1, 1), (21, 2, 0), (11, 2, 0)}
States with agreement: 272
States with disagreement: 8
Agreement percentage: 97.14%
==================================================
Comparison between Sutton & Barto's policy and the SARSA policy
Common states: {(15, 5, 0), (18, 10, 0), (12, 10, 0), (8, 6, 0), (21, 6, 0), (7, 9, 0),
(10, 4, 0), (16, 5, 0), (17, 10, 1), (20, 6, 0), (10, 8, 0), (21, 4, 0), (11, 4, 0), (1
2, 9, 1), (5, 7, 0), (13, 4, 1), (6, 2, 0), (21, 8, 0), (14, 3, 0), (18, 7, 0), (11, 8,
0), (16, 6, 1), (19, 2, 0), (16, 7, 0), (20, 7, 1), (10, 10, 0), (15, 6, 1), (9, 3, 0),
(5, 9, 0), (20, 1, 0), (21, 7, 1), (4, 2, 0), (14, 5, 0), (11, 10, 0), (15, 10, 1), (1
7, 1, 0), (20, 9, 1), (4, 6, 0), (9, 5, 0), (13, 5, 1), (19, 8, 0), (19, 5, 1), (14, 4,
1), (13, 10, 0), (17, 3, 0), (7, 3, 0), (4, 8, 0), (18, 1, 1), (8, 9, 0), (12, 2, 0),
(18, 2, 0), (13, 7, 1), (19, 7, 1), (21, 9, 0), (14, 6, 1), (15, 1, 0), (12, 6, 0), (1
7, 2, 1), (7, 5, 0), (14, 10, 1), (16, 1, 0), (17, 6, 1), (20, 2, 0), (21, 10, 1), (12,
5, 1), (16, 9, 1), (5, 3, 0), (17, 4, 1), (6, 9, 0), (18, 3, 0), (19, 9, 0), (16, 3,
0), (20, 3, 1), (17, 8, 1), (20, 4, 0), (14, 7, 1), (15, 2, 1), (12, 7, 1), (5, 5, 0),
(20, 8, 0), (21, 3, 1), (14, 1, 0), (11, 6, 0), (15, 7, 0), (16, 2, 1), (20, 5, 1), (1
0, 6, 0), (9, 1, 0), (19, 1, 1), (15, 4, 1), (13, 2, 0), (13, 1, 1), (18, 4, 1), (15,
8, 1), (13, 6, 0), (17, 10, 0), (4, 4, 0), (18, 8, 1), (8, 5, 0), (18, 9, 0), (12, 9,
0), (13, 3, 1), (19, 3, 1), (13, 4, 0), (21, 5, 0), (14, 2, 1), (16, 4, 0), (13, 8, 0),
(17, 9, 1), (7, 1, 0), (18, 10, 1), (8, 7, 0), (16, 8, 0), (5, 6, 0), (21, 6, 1), (6,
1, 0), (21, 7, 0), (11, 7, 0), (15, 10, 0), (16, 5, 1), (12, 1, 1), (20, 6, 1), (6, 5,
0), (10, 9, 0), (19, 5, 0), (16, 10, 0), (20, 10, 1), (14, 3, 1), (14, 4, 0), (11, 9,
0), (15, 9, 1), (12, 3, 1), (5, 1, 0), (6, 7, 0), (14, 8, 0), (18, 1, 0), (15, 3, 0),
(19, 7, 0), (19, 4, 1), (20, 1, 1), (10, 2, 0), (14, 6, 0), (9, 8, 0), (13, 9, 0), (17,
2, 0), (4, 7, 0), (14, 10, 0), (19, 6, 1), (17, 6, 0), (8, 1, 0), (18, 5, 0), (9, 10,
0), (19, 10, 1), (12, 5, 0), (13, 10, 1), (21, 1, 0), (7, 4, 0), (4, 9, 0), (14, 9, 1),
(18, 2, 1), (17, 5, 1), (17, 8, 0), (21, 9, 1), (7, 8, 0), (10, 3, 0), (18, 6, 1), (12,
4, 1), (12, 7, 0), (5, 2, 0), (8, 3, 0), (21, 3, 0), (11, 3, 0), (12, 8, 1), (16, 1,
1), (16, 2, 0), (20, 2, 1), (17, 7, 1), (10, 5, 0), (7, 10, 0), (19, 1, 0), (16, 6, 0),
(5, 4, 0), (20, 7, 0), (21, 2, 1), (11, 5, 0), (15, 5, 1), (12, 10, 1), (15, 6, 0), (5,
8, 0), (16, 3, 1), (20, 4, 1), (6, 3, 0), (18, 8, 0), (19, 3, 0), (20, 8, 1), (20, 9,
0), (21, 4, 1), (14, 2, 0), (15, 7, 1), (9, 4, 0), (13, 5, 0), (17, 9, 0), (5, 10, 0),
(21, 8, 1), (4, 3, 0), (18, 7, 1), (8, 4, 0), (9, 2, 0), (19, 2, 1), (13, 2, 1), (16,
7, 1), (8, 8, 0), (12, 1, 0), (9, 6, 0), (13, 6, 1), (13, 7, 0), (4, 5, 0), (14, 5, 1),
(18, 9, 1), (20, 10, 0), (21, 5, 1), (17, 1, 1), (15, 9, 0), (12, 3, 0), (16, 4, 1),
(8, 10, 0), (13, 8, 1), (19, 8, 1), (6, 4, 0), (21, 10, 0), (7, 2, 0), (16, 8, 1), (19,
4, 0), (16, 9, 0), (17, 3, 1), (6, 8, 0), (17, 4, 0), (10, 1, 0), (7, 6, 0), (12, 2,
1), (20, 3, 0), (6, 6, 0), (14, 7, 0), (11, 1, 0), (15, 1, 1), (12, 6, 1), (15, 2, 0),
(19, 6, 0), (16, 10, 1), (6, 10, 0), (9, 7, 0), (19, 10, 0), (20, 5, 0), (14, 8, 1), (1
4, 9, 0), (15, 3, 1), (15, 4, 0), (13, 1, 0), (17, 5, 0), (4, 10, 0), (18, 3, 1), (18,
6, 0), (18, 4, 0), (12, 4, 0), (13, 9, 1), (9, 9, 0), (15, 8, 0), (19, 9, 1), (10, 7,
0), (12, 8, 0), (13, 3, 0), (17, 7, 0), (7, 7, 0), (4, 1, 0), (14, 1, 1), (18, 5, 1),
(8, 2, 0), (21, 1, 1), (21, 2, 0), (11, 2, 0)}
```

```
States with agreement: 240
States with disagreement: 40
Agreement percentage: 85.71%
==================================================
```

In [71]:
```
# AGAIN, USED CHAT GPT

policies = [s_and_b, naive_policy, policy_optimalMC_dict, policy_optimal_sarsa]

# Lists to store player sums and dealer showings for each policy, usable and non-usabl
hit_player_sums_usable = [[] for _ in range(4)]
hit_dealer_showings_usable = [[] for _ in range(4)]
stick_player_sums_usable = [[] for _ in range(4)]
stick_dealer_showings_usable = [[] for _ in range(4)]

hit_player_sums_non_usable = [[] for _ in range(4)]
hit_dealer_showings_non_usable = [[] for _ in range(4)]
stick_player_sums_non_usable = [[] for _ in range(4)]
stick_dealer_showings_non_usable = [[] for _ in range(4)]

# Separate the states into hit and stick regions for each policy
for idx, Q in enumerate(policies):
    for state, action in Q.items():
        player_sum, dealer_showing, usable_ace = state
        if usable_ace:  # Usable ace case
            if action == 0:  # Stick if action is 0
                stick_player_sums_usable[idx].append(player_sum)
                stick_dealer_showings_usable[idx].append(dealer_showing)
            else:  # Hit if action is 1
                hit_player_sums_usable[idx].append(player_sum)
                hit_dealer_showings_usable[idx].append(dealer_showing)
        else:  # Non-usable ace case
            if action == 0:  # Stick if action is 0
                stick_player_sums_non_usable[idx].append(player_sum)
                stick_dealer_showings_non_usable[idx].append(dealer_showing)
            else:  # Hit if action is 1
                hit_player_sums_non_usable[idx].append(player_sum)
                hit_dealer_showings_non_usable[idx].append(dealer_showing)

# Set up a 2x4 grid of subplots for the four policies
fig, axs = plt.subplots(2, 4, figsize=(20, 10))
titles = ["S&B", "Naive", "Monte Carlo", "SARSA"]

for i in range(4):
    # Plot for usable ace (first row)
    axs[0, i].scatter(hit_dealer_showings_usable[i], hit_player_sums_usable[i], color=
    axs[0, i].scatter(stick_dealer_showings_usable[i], stick_player_sums_usable[i], co
    axs[0, i].set_title(f"{titles[i]} (Usable Ace)")
    axs[0, i].set_xlabel("Dealer's Showing")
    axs[0, i].set_ylabel("Player's Sum")
    axs[0, i].set_xticks(np.arange(1, 11))
    axs[0, i].set_xticklabels(['A', '2', '3', '4', '5', '6', '7', '8', '9', '10'])  #
    axs[0, i].set_yticks(np.arange(4, 22))  # Player sums from 4 to 21
    axs[0, i].grid(True)
    axs[0, i].legend()

    # Plot for non-usable ace (second row)
    axs[1, i].scatter(hit_dealer_showings_non_usable[i], hit_player_sums_non_usable[i]
    axs[1, i].scatter(stick_dealer_showings_non_usable[i], stick_player_sums_non_usabl
    axs[1, i].set_title(f"{titles[i]} (Non-Usable Ace)")
    axs[1, i].set_xlabel("Dealer's Showing")
    axs[1, i].set_ylabel("Player's Sum")
    axs[1, i].set_xticks(np.arange(1, 11))
    axs[1, i].set_xticklabels(['A', '2', '3', '4', '5', '6', '7', '8', '9', '10'])  #
```

```
    axs[1, i].set_yticks(np.arange(4, 22))  # Player sums from 4 to 21
    axs[1, i].grid(True)
    axs[1, i].legend()

# Adjust layout and show plot
plt.tight_layout()
plt.show()
```



The Naive policy is a straight line dividing the hits and sticks since the only number it takes into account is the player's sum. If we compare it to the Sutton and Barto policy (usable ace), they are not so different. They do take different actions when the player's sum is 18 and 19. This means that the strategy in naive policy very conservative but still not optimal. it does not exploit favorable situations where lower sums could also be favorable.

Regarding Monte Carlo, the non-usable ace plot seems to resemble quite well the optimal one. it sticks for lower values when the dealer's showing card is of a value close to 5 or 6 just like the optimal one does. However, the plot where there is a usable ace differs a little bit. It has some inconsistencies in the middle values.

Lastly, the SARSA method seems to converge a little bit and it faintly resembles the optimal policy. It performs better than the naive policy but may be slightly less consistent than Monte Carlo in approximating the optimal policy, particularly in lower-stakes situations.

4.2. Influence of the Number of Episodes

Conduct a study by varying the number of episodes in each of the algorithms.

Questions (**1 point**):

- Train each algorithm multiple times with 100,000, 1,000,000, and 5,000,000 episodes and average the results.
- Indicate how the **number of episodes** influences the convergence of each algorithm by calculating the number of states where the policy differs from the optimal one, as well as the average return obtained after playing 100,000 games following each training.

NOTE: I will print the whole stats because i am working with the function i've used for the whole notebook

MONTE CARLO

In [72]:
```
Q_first_visit_decay_weighted_avg100000, policy_optimalMC100000 = mc_control_on_policy_
print()
print("100000 episodes")

# Convert the policy in dictionary format
policy_optimalMC_dict100000 = {}
for state in Q_first_visit_decay_weighted_avg100000.keys():
    action_probabilities = policy_optimalMC100000(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict100000[state] = 0
    else:
        policy_optimalMC_dict100000[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict100000[state]:
        disagree += 1
print(f"States with disagreement: {disagree}")

run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_decay_weighted_avg100000
print("#" * 50)

#####################################################################################

Q_first_visit_decay_weighted_avg1000000, policy_optimalMC1000000 = mc_control_on_polic
print()
print("1000000 episodes")

# Convert the policy in dictionary format
policy_optimalMC_dict1000000 = {}
for state in Q_first_visit_decay_weighted_avg1000000.keys():
    action_probabilities = policy_optimalMC1000000(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict1000000[state] = 0
    else:
        policy_optimalMC_dict1000000[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict1000000[state]:
        disagree += 1
print(f"States with disagreement: {disagree}")

run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_decay_weighted_avg100000
print("#" * 50)


#####################################################################################

Q_first_visit_decay_weighted_avg5000000, policy_optimalMC5000000 = mc_control_on_polic
print()
print("5000000 episodes")

# Convert the policy in dictionary format
policy_optimalMC_dict5000000 = {}
for state in Q_first_visit_decay_weighted_avg5000000.keys():
```

```
    action_probabilities = policy_optimalMC5000000(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict5000000[state] = 0
    else:
        policy_optimalMC_dict5000000[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict5000000[state]:
        disagree += 1
print(f"States with disagreement: {disagree}")

run_episodes_and_get_stats (num_games=100000, q=Q_first_visit_decay_weighted_avg500000
```

```
Episode 100/100000 - Average reward -0.39990234375
Episode 99900/100000 - Average reward -0.170043945312557555
100000 episodes
States with disagreement: 24

Statistics for 100000 games
   Win percentage (counting the natural wins): 42.888999999999996%
   Natural win percentage: 4.069%
   Loss percentage: 47.704%
   Draw percentage: 9.407%
   Average return: -0.027805
##################################################
Episode 999900/1000000 - Average reward 0.0249938964843755555
1000000 episodes
States with disagreement: 11

Statistics for 100000 games
   Win percentage (counting the natural wins): 43.230000000000004%
   Natural win percentage: 4.19%
   Loss percentage: 47.848%
   Draw percentage: 8.921999999999999%
   Average return: -0.02523
##################################################
Episode 4999900/5000000 - Average reward -0.094970703125875555
5000000 episodes
States with disagreement: 16

Statistics for 100000 games
   Win percentage (counting the natural wins): 42.532%
   Natural win percentage: 4.118%
   Loss percentage: 48.246%
   Draw percentage: 9.222%
   Average return: -0.03655
```

When comparing the number of episodes, one milion epsiodes seems to be the best one given that it has less different states compared with the optimal policy. Thus, the average return is also the same. however, the statistics don't change that much, so more episodes means a better policy but to a certain extent.

SARSA

In [73]:
```
_, q_sarsa100000 = SARSA_decay(env, episodes=100000, learning_rate=0.3, discount=1, ep
policy_optimal_sarsa100000 = extract_policy(q_sarsa100000, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa100000[state]:
        disagree += 1
```

```python
print(f"States with disagreement: {disagree}", end=" ")

run_episodes_and_get_stats (num_games=100000, q=q_sarsa100000, env=env, debug=False)

print("#" * 50)

##########################################################################################

_, q_sarsa1000000 = SARSA_decay(env, episodes=1000000, learning_rate=0.3, discount=1,
policy_optimal_sarsa1000000 = extract_policy(q_sarsa1000000, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa1000000[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")

run_episodes_and_get_stats (num_games=100000, q=q_sarsa1000000, env=env, debug=False)

print("#" * 50)

##########################################################################################

_, q_sarsa5000000 = SARSA_decay(env, episodes=5000000, learning_rate=0.3, discount=1,
policy_optimal_sarsa5000000 = extract_policy(q_sarsa5000000, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa5000000[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")

run_episodes_and_get_stats (num_games=100000, q=q_sarsa5000000, env=env, debug=False)
```

```
Episode    99900/  100000 - Average reward -0.12States with disagreement: 35
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.434%
   Natural win percentage: 4.245%
   Loss percentage: 48.844%
   Draw percentage: 9.722%
   Average return: -0.052875
##################################################
Episode   999900/ 1000000 - Average reward -0.09States with disagreement: 33
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.641%
   Natural win percentage: 4.149%
   Loss percentage: 48.791000000000004%
   Draw percentage: 9.568%
   Average return: -0.050755
##################################################
Episode  4999900/ 5000000 - Average reward -0.11States with disagreement: 42
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.88%
   Natural win percentage: 4.208%
   Loss percentage: 49.629%
   Draw percentage: 9.491%
   Average return: -0.06645
```

The same exact thing happens with SARSA. 1000000 episodes is the best option, it has the best average return and it has the least number of different states.

4.3. Influence of the Discount Factor

Conduct a study by varying the *discount factor* in each of the algorithms.

Questions (**1 point**):

- Run the algorithms with *discount factor* = 0.1, 0.5, 0.9 and the rest of the parameters the same as in previous exercises.
- Describe the changes in the optimal policy, comparing the result obtained with the result of previous exercises (*discount factor* = 1).

MONTE CARLO

In [74]:
```
Q_first_visit_decay_weighted_avg01, policy_optimalMC01 = mc_control_on_policy_epsilon_
print()
print("Discount Factor = 0.1")

# Convert the policy in dictionary format
policy_optimalMC_dict01 = {}
for state in Q_first_visit_decay_weighted_avg01.keys():
    action_probabilities = policy_optimalMC01(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict01[state] = 0
    else:
        policy_optimalMC_dict01[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict01[state]:
        disagree += 1
print(f"States with disagreement: {disagree}")
print("#" * 50)

##########################################################################################

Q_first_visit_decay_weighted_avg05, policy_optimalMC05 = mc_control_on_policy_epsilon_
print()
print("Discount Factor = 0.5")

# Convert the policy in dictionary format
policy_optimalMC_dict05 = {}
for state in Q_first_visit_decay_weighted_avg05.keys():
    action_probabilities = policy_optimalMC05(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict05[state] = 0
    else:
        policy_optimalMC_dict05[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict05[state]:
        disagree += 1
print(f"States with disagreement: {disagree}")
print("#" * 50)

##########################################################################################

Q_first_visit_decay_weighted_avg09, policy_optimalMC09 = mc_control_on_policy_epsilon_
print()
print("Discount Factor = 0.9")
```

```
# Convert the policy in dictionary format
policy_optimalMC_dict09 = {}
for state in Q_first_visit_decay_weighted_avg09.keys():
    action_probabilities = policy_optimalMC09(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict09[state] = 0
    else:
        policy_optimalMC_dict09[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict09[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
print("#" * 50)

##########################################################################################

Q_first_visit_decay_weighted_avg1, policy_optimalMC1 = mc_control_on_policy_epsilon_gr
print()
print("Discount Factor = 1")

# Convert the policy in dictionary format
policy_optimalMC_dict1 = {}
for state in Q_first_visit_decay_weighted_avg1.keys():
    action_probabilities = policy_optimalMC1(state)
    if action_probabilities[0] > action_probabilities[1]:
        policy_optimalMC_dict1[state] = 0
    else:
        policy_optimalMC_dict1[state] = 1

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimalMC_dict1[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
```

```
Episode 99900/100000 - Average reward 0.08001708984375555575
Discount Factor = 0.1
States with disagreement: 27
##################################################
Episode 99900/100000 - Average reward 0.1850585937506258755
Discount Factor = 0.5
States with disagreement: 26
##################################################
Episode 99900/100000 - Average reward -0.010002136230468755
Discount Factor = 0.9
States with disagreement: 30
##################################################
Episode 99900/100000 - Average reward -0.099975585937568755
Discount Factor = 1
States with disagreement: 41
```

When comparing the discount factor, the best average reward is with 0.9, however, 0.5 has the least different states. I would opt to follow the latter rather than the former since the policy is closer to the optimal one which means that the difference in average reward is only due to the fact that each episode is random. However, if we run again this piece of code, the averages would probably change and maybe in some occasions, other discount factors would have the best average reward. Instead, if we stick to the best policy (the one with least different states), and run an infinite number of episodes, that would be the best policy.

SARSA

In [75]:
```
_, q_sarsa01 = SARSA_decay(env, episodes=100000, learning_rate=0.3, discount=0.1, epsi
policy_optimal_sarsa01 = extract_policy(q_sarsa01, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa01[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
print("#" * 50)

#########################################################################################

_, q_sarsa05 = SARSA_decay(env, episodes=100000, learning_rate=0.3, discount=0.5, epsi
policy_optimal_sarsa05 = extract_policy(q_sarsa05, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa05[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
print("#" * 50)

#########################################################################################

_, q_sarsa09 = SARSA_decay(env, episodes=100000, learning_rate=0.3, discount=0.9, epsi
policy_optimal_sarsa09 = extract_policy(q_sarsa09, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa09[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
print("#" * 50)

#########################################################################################

_, q_sarsa1 = SARSA_decay(env, episodes=100000, learning_rate=0.3, discount=1, epsilon
policy_optimal_sarsa1 = extract_policy(q_sarsa1, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa1[state]:
        disagree += 1

print(f"States with disagreement: {disagree}")
```

```
Episode      100/  100000 - Average reward -0.31
Episode    99900/  100000 - Average reward -0.15States with disagreement: 44
##################################################
Episode    99900/  100000 - Average reward -0.20States with disagreement: 43
##################################################
Episode    99900/  100000 - Average reward -0.15States with disagreement: 34
##################################################
Episode    99900/  100000 - Average reward -0.28States with disagreement: 35
```

In the case of SARSA, after running a few times the code, when the discount is 0.9, it works best and it has less states with disagreement. (Nevertheless, Monte Carlo still seems to be performing better).

4.4. Influence of the Learning Rate

Conduct a study by varying the learning rate in the *SARSA* algorithm.

Questions (**1 point**):

- Run the *SARSA* algorithm with the following *learning rate* values: 0.001, 0.01, 0.1, and 0.9.
- Analyze the differences with the results obtained previously in terms of the number of errors relative to the optimal policy and the accumulated reward for every 100,000 episodes played.

In [76]:

```
_, q_sarsa0001 = SARSA_decay(env, episodes=100000, learning_rate=0.001, discount=1, ep
policy_optimal_sarsa0001 = extract_policy(q_sarsa0001, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa0001[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")
run_episodes_and_get_stats (num_games=100000, q=q_sarsa0001, env=env, debug=False)
print("#" * 50)

##########################################################################################

_, q_sarsa001 = SARSA_decay(env, episodes=100000, learning_rate=0.01, discount=1, epsi
policy_optimal_sarsa001 = extract_policy(q_sarsa001, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa001[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")
run_episodes_and_get_stats (num_games=100000, q=q_sarsa001, env=env, debug=False)
print("#" * 50)

##########################################################################################

_, q_sarsa01 = SARSA_decay(env, episodes=100000, learning_rate=0.1, discount=1, epsilo
policy_optimal_sarsa01 = extract_policy(q_sarsa01, env)

disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa01[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")
run_episodes_and_get_stats (num_games=100000, q=q_sarsa01, env=env, debug=False)
print("#" * 50)

##########################################################################################

_, q_sarsa09 = SARSA_decay(env, episodes=100000, learning_rate=0.9, discount=1, epsilo
policy_optimal_sarsa09 = extract_policy(q_sarsa09, env)
```

```
disagree = 0
for state in s_and_b.keys():
    if s_and_b[state] != policy_optimal_sarsa09[state]:
        disagree += 1

print(f"States with disagreement: {disagree}", end=" ")
run_episodes_and_get_stats (num_games=100000, q=q_sarsa09, env=env, debug=False)
```

```
Episode    99900/  100000 - Average reward -0.13States with disagreement: 27
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.916%
   Natural win percentage: 4.243%
   Loss percentage: 49.589%
   Draw percentage: 8.495%
   Average return: -0.055515
##################################################
Episode    99900/  100000 - Average reward -0.07States with disagreement: 41
Statistics for 100000 games
   Win percentage (counting the natural wins): 42.295%
   Natural win percentage: 4.168%
   Loss percentage: 48.563%
   Draw percentage: 9.142%
   Average return: -0.04184
##################################################
Episode    99900/  100000 - Average reward 0.014States with disagreement: 33
Statistics for 100000 games
   Win percentage (counting the natural wins): 41.879%
   Natural win percentage: 4.19%
   Loss percentage: 48.847%
   Draw percentage: 9.274000000000001%
   Average return: -0.04873
##################################################
Episode    99900/  100000 - Average reward -0.13States with disagreement: 38
Statistics for 100000 games
   Win percentage (counting the natural wins): 40.875%
   Natural win percentage: 4.194%
   Loss percentage: 49.858999999999995%
   Draw percentage: 9.266%
   Average return: -0.06887
```

Lastly, the best learning rate is 0.01. It has the best average reward and the least states with disagreement. Again, even without having the best win rate, it would perform better overall.

In conclusion, between the SARSA and Monte Carlo methods, the latter performs better. First visit also seems to give better results than every-visit. Exploring starts could be beneficial, but i don't really see how it makes a lot of difference in the Blackjack.

Regarding the parameters, if we count that we would be performing an infinite number of episodes to train the agent (and thus guide ourselves by the least number of differing states), the best discount factor would be of 0.5 and the best learning rate for SARSA would be 0.01.

Regardless, it would be very beneficial to perform cross-examination and see what parameters work better with what other parameters. because maybe another discount factor is optimal for episodes with another learning rate. Maybe plotting the 3D landscape would help determine how to fine-tune the parameters.