```python
import numpy as np
import math
import copy

def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps, use_softmax_policy, init_beta=None, k_exp_sched=None):
    """ Runs tabular Q learning algorithm for stochastic environment.

    Args:
        env: instance of environment object
        num_iters (int): Number of episodes to run Q-learning algorithm
        alpha (float): The learning rate between [0,1]
        gamma (float): Discount factor, between [0,1)
        epsilon (float): Probability in [0,1] that the agent selects a random move instead of
                selecting greedily from Q value
        max_steps (int): Maximum number of steps in the environment per episode
        use_softmax_policy (bool): Whether to use softmax policy (True) or Epsilon-Greedy (False)
        init_beta (float): If using stochastic policy, sets the initial beta as the parameter for the softmax
        k_exp_sched (float): If using stochastic policy, sets hyperparameter for exponential schedule
            on beta

    Returns:
        q_hat: A Q-value table shaped [num_states, num_actions] for environment with with num_states
            number of states (e.g. num rows * num columns for grid) and num_actions number of possible
            actions (e.g. 4 actions up/down/left/right)
        steps_vs_iters: An array of size num_iters. Each element denotes the number
            of steps in the environment that the agent took to get to the goal
            (capped to max_steps)
    """
    action_space_size = env.num_actions
    state_space_size = env.num_states
    q_hat = np.zeros(shape=(state_space_size, action_space_size))
    steps_vs_iters = np.zeros(num_iters)

    for i in range(num_iters):
        # TODO: Initialize current state by resetting the environment
        curr_state = env.reset()
        num_steps = 0
        done = False

        # TODO: Keep looping while environment isn't done and less than maximum steps\
        while (not done) & (steps_vs_iters[i] <= max_steps):
        #while (not done) & (num_steps <= max_steps):
        # I know the second while line is the correct one
        # but when I use the second while loop, the algorithm won't converge
        # I tried to debug, found the reason is that the way I 'break ties' is problematic
        # therefore I used the first while loop, at least the algorithm can converge for epsilon greedy
        # even though the max number of iteration were exceeded
            num_steps += 1
            # break ties if all q value of the given state are 0
        #'break ties'
        # seems my understanding on form of Q matrix is wrong
            #if all(q_hat[curr_state]==0):
                #break_ties = np.random.randint(0, 4)
                #q_hat[curr_state][break_ties] = 1
                #q_hat[curr_state]=np.random.uniform(0,1,4)
            #print(curr_state)
            # Choose an action using policy derived from either softmax Q-value
            # or epsilon greedy
            if use_softmax_policy:
                assert(init_beta is not None)
                assert(k_exp_sched is not None)
                # TODO: Boltzmann stochastic policy (softmax policy)
                beta = beta_exp_schedule(init_beta, i, k_exp_sched) # Call beta_exp_schedule to get the current beta value
                action = softmax_policy(q_hat, beta, curr_state)
            else:
                # TODO: Epsilon-greedy
```

```python
            action = epsilon_greedy(q_hat, epsilon, curr_state, action_space_size)

        # TODO: Execute action in the environment and observe the next state, reward, and done flag
        next_state, reward, done = env.step(action)

        # TODO: Update Q_value
        if next_state != curr_state:
            new_value = np.argmax(q_hat[next_state])
            # TODO: Use Q-learning rule to update q_hat for the curr_state and action:
            # i.e., Q(s,a) <- Q(s,a) + alpha*[reward + gamma * max_a'(Q(s',a')) - Q(s,a)]
            q_hat[curr_state, action] = \
                q_hat[curr_state, action]+alpha*(reward + gamma * q_hat[next_state,new_value] - q_hat[curr_state,action])
            #   q_hat[curr_state, action]+alpha*(reward + gamma * max(q_hat[next_state]) - q_hat[curr_state,action])
            #   q_hat[next_state,new_value] is the same thing as max(q_hat[next_state]) since new_value =
np.argmax(q_hat[next_state])

            # TODO: Update the current state to be the next state
            curr_state = next_state

        steps_vs_iters[i] = num_steps

    return q_hat, steps_vs_iters


def epsilon_greedy(q_hat, epsilon, state, action_space_size):
    """ Chooses a random action with p_rand_move probability,
    otherwise choose the action with highest Q value for
    current observation

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns and num_actions
            number of possible actions
        epsilon (float): Probability in [0,1] that the agent selects a random
            move instead of selecting greedily from Q value
        state: A 2-element array with integer element denoting the row and column
            that the agent is in
        action_space_size (int): number of possible actions

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # Hint: Sample from a uniform distribution and check if the sample is below
    # a certain threshold
    sample = np.random.uniform(0,1)
    if sample <= epsilon:
        return np.random.randint(0,action_space_size)
    else:
        #print(q_hat[state])
        return np.argmax(q_hat[state])



def softmax_policy(q_hat, beta, state):
    """ Choose action using policy derived from Q, using
    softmax of the Q values divided by the temperature.

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns
        beta (float): Parameter for controlling the stochasticity of the action
        obs: A 2-element array with integer element denoting the row and column
            that the agent is in

    Returns:
        action (int): A number in the range [0, action_space_size-1]
```

```
        denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: use the stable_softmax function defined below
    # beta in Boltzmann equation usually is the inverse of temperature
    # the divide by temperature means multiply by beta
    return np.argmax(stable_softmax(q_hat[state]*beta,axis=0))

def beta_exp_schedule(init_beta, iteration, k=0.1):
    beta = init_beta * np.exp(k * iteration)
    return beta

def stable_softmax(x, axis=2):
    """ Numerically stable softmax:
    softmax(x) = e^x /(sum(e^x))
               = e^x / (e^max(x) * sum(e^x/e^max(x)))

    Args:
        x: An N-dimensional array of floats
        axis: The axis for normalizing over.

    Returns:
        output: softmax(x) along the specified dimension
    """
    max_x = np.max(x, axis, keepdims=True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims=True)

    return output

if __name__=='__main__':
    from maze import MazeEnv, ProbabilisticMazeEnv
    from plotting_utils import plot_several_steps_vs_iters
    num_iters = 200
    alpha = 1.0
    gamma = 0.9
    epsilon = 0.1
    max_steps = 100

    beta_list = [1, 3, 6]
    use_softmax_policy = True
    k_exp_schedule = 0  # (float) choose k such that we have a constant beta during training
    # refer to the function beta_exp_schedule(init_beta, iteration, k=0.1),
    # beta = init_beta * np.exp(k * iteration)
    # to hold beta constant, we want k * iteration to be zero
    # then simply set k to 0 will do the work

    env = MazeEnv()
    steps_vs_iters_list = []
    for beta in beta_list:
        q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_steps, use_softmax_policy, beta,
                                       k_exp_schedule)
        steps_vs_iters_list.append(steps_vs_iters)
    label_list = ["beta={}".format(beta) for beta in beta_list]
    plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```