Reinforcement Learning: An Introduction - Chapter 9

Thomas Hopkins

Exercise 9.1

I do not think that the eligibility trace method would perform as well as the model planning method. This is because the trace for each state would fall off for the states that were many steps away from the goal. Furthermore, some of the states involved in the eligibility trace update would point to the state that followed it most recently. This problem does not occur in the model planning method because each transition encountered is stored as part of the model and randomly sampled during simulated learning.

Exercise 9.2

The DynaQ+ agent likely performed better in both phases due to its exploration. It prioritized its exploration based on its model of the environment while the others explored randomly.

Exercise 9.3

The difference between DynaQ+ and DynaQ narrowed because the DynaQ+ agent spends some of its computation exploring states that have not been encountered in a while. This allows the other DynaQ agent to "catch up" in cumulative reward by not exploring nearly as much.

Exercise 9.4

It seems like the DynaQ+ agent with the exploration mechanism performed during action selection performs better at first than the mechanism that alters rewards. But over time, the original method catches up.

In [33]:

```
using Base.Iterators
using Random
using Plots
gr()

Random.seed!(32)

STATES = collect(product(1:6, 1:9))
GRID = zeros(6, 9)
GRID[4, 2:9] .= 1
ACTIONS = [(0, -1), (-1, 0), (0, 1), (1, 0)] # (left, up, right, down)
START = (6, 4)
GOAL = (1, 9)


function select_action(Q_values, state, n_vals; kappa = 0.1, method = false,
    if !greedy
        if rand() < 0.1
            action = rand(ACTIONS)
            return findfirst(x -> all(x .== action), ACTIONS), action
        end
    end
    values = Q_values[state...]
    if method
        values .+= (kappa * sqrt.(n_vals[state...]))
    end
    a = argmax(values)
    return a, ACTIONS[a]
end

function step(state, action)
    new_state = [state[1] + action[1], state[2] + action[2]]
    if new_state[1] <= 0
        new_state[1] = 1
    elseif new_state[1] > 6
        new_state[1] = 6
    end
    if new_state[2] <= 0
        new_state[2] = 1
    elseif new_state[2] > 9
        new_state[2] = 9
    end
    if GRID[new_state...] == 1
        new_state = state
    else
        new_state = (new_state[1], new_state[2])
    end
    reward = 0.0
    terminated = false
    if new_state == GOAL
        reward = 1.0
        terminated = true
    end
    return reward, new_state, terminated
end

function dynaq_plus(Q_values, model, n_vals; n=50, num_time_steps = 100000, k
    cumulative_rewards = []
```

```julia
    cumulative_re = 0.0
    state = START
    a, action = select_action(Q_values, state, n_vals; kappa = kappa, method
    terminated = false
    for e = 1:num_time_steps
        a, action = select_action(Q_values, state, n_vals; kappa = kappa, met
        reward, new_state, terminated = step(state, action)
        na, new_action = select_action(Q_values, new_state, n_vals; greedy =
        if !method
            Q_values[state...][a] += alpha * ((reward + kappa * sqrt.(n_vals[
        else
            Q_values[state...][a] += alpha * (reward + gamma * Q_values[new_s
        end
        for i = 1:6, j = 1:9
            n_vals[i, j] .+= 1
        end
        n_vals[state...][a] = 0
        model[state...][a] = (new_state, reward)
        state = new_state
        if terminated
            state = START
        end
        cumulative_re += reward
        append!(cumulative_rewards, cumulative_re)
        for i = 1:n
            S = findall(x -> any([!ismissing(x[i]) for i = 1:4]), model)
            s = rand(S)
            A = findall(x -> !ismissing(x), model[s])
            a = rand(A)
            sp, r = model[s][a]
            ap, _ = select_action(Q_values, sp, n_vals; greedy = true)
            Q_values[s][a] += alpha * (reward + gamma * Q_values[sp...][ap] -
        end
    end
    return Q_values, cumulative_rewards
end
```

Out[33]:

```
dynaq_plus (generic function with 1 method)
```

In [34]:

```julia
q_values = [[0.0 for a in ACTIONS] for i = 1:6, j = 1:9]
model = [Any[missing for a in ACTIONS] for i = 1:6, j = 1:9]
n_vals = [[0, 0, 0, 0] for i = 1:6, j = 1:9]
Q, reward_original = dynaq_plus(q_values, model, n_vals);
```

In [35]:

```julia
q_values = [[0.0 for a in ACTIONS] for i = 1:6, j = 1:9]
model = [Any[missing for a in ACTIONS] for i = 1:6, j = 1:9]
n_vals = [[0, 0, 0, 0] for i = 1:6, j = 1:9]
Q, reward_new = dynaq_plus(q_values, model, n_vals; method = true);
```

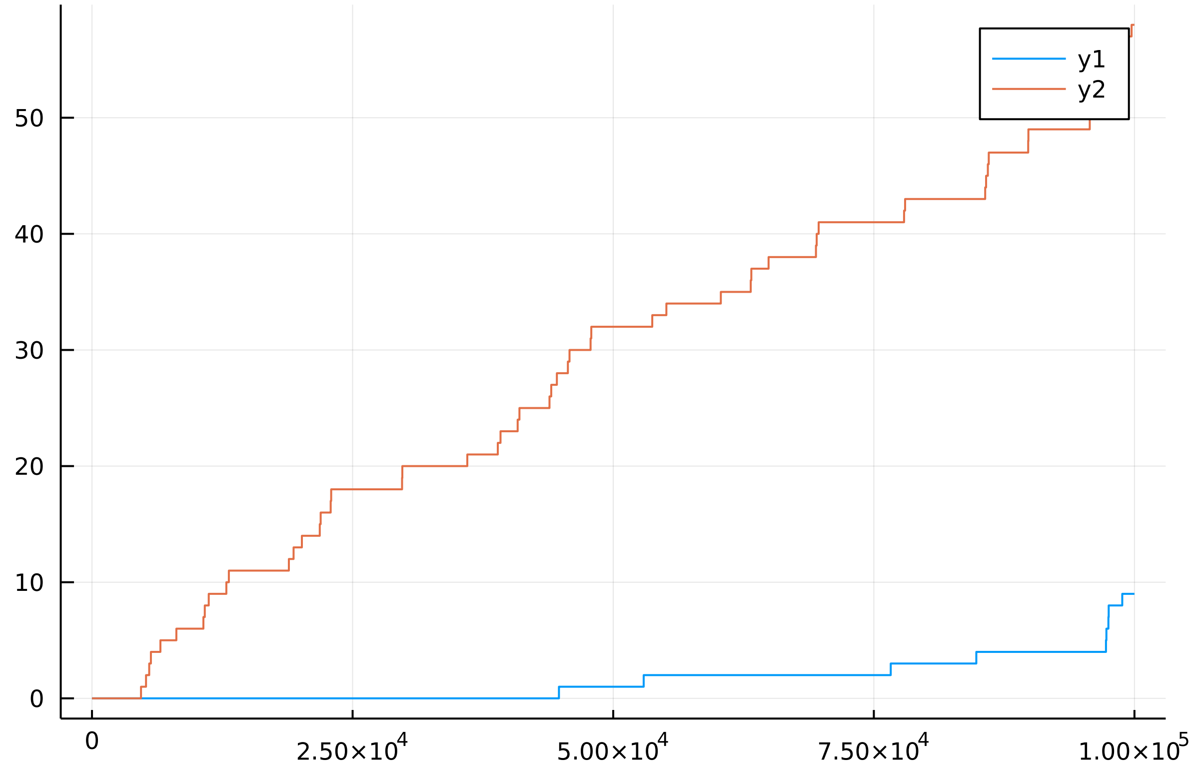
In [39]:

```julia
plot(1:100000, reward_original);
```

In [40]:

```julia
plot!(1:100000, reward_new)
```

Out[40]:



Exercise 9.5

If the distribution of next states were skewed, this would weaken the case for sample backups since the next states with a high probability of occurring would be sampled most often. On the other hand, full backups would take into account all of the next possible states which make this a better choice (depending on the branching factor).

Exercise 9.6

The curves taper off after their initial increase because the values of some of the states that are sampled are already well known. This means that there is diminishing returns for further backups of these states. The on-policy distribution reaches this point faster since it focuses its search to try the best actions more often. The uniform distribution, on the other hand, reaches this point later since it gives equal weight to all actions.