**HW4\HW4_5264_AmandaMarlow.jl**

```julia
using DMUStudent.HW4: gw
using LinearAlgebra: I
using CommonRLInterface: render, actions, act!, observe, reset!, AbstractEnv, observations, terminated, clone
using SparseArrays
using Statistics: mean
using Plots
using StaticArrays: SA
using StatsBase

############
# SARSA-λ #
############

function sarsa_lambda_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.05, λ=0.9)

    start = time()

    function policy(s)
        if rand() < ϵ
            return rand(actions(env))
        else
            return argmax(a->Q[(s, a)], actions(env))
        end
    end

    s = observe(env)
    a = policy(s)
    r = act!(env, a)
    sp = observe(env)
    hist = [s]
    N = Dict((s, a) => 0.0)

    while !terminated(env)
        ap = policy(sp)

        N[(s, a)] = get(N, (s, a), 0.0) + 1

        δ = r + γ*Q[(sp, ap)] - Q[(s, a)]

        for ((s, a), n) in N
            Q[(s, a)] += α*δ*n
            N[(s, a)] *= γ*λ
        end

        s = sp
        a = ap
        r = act!(env, a)
        sp = observe(env)
        push!(hist, sp)
    end

    N[(s, a)] = get(N, (s, a), 0.0) + 1
    δ = r - Q[(s, a)]

    for ((s, a), n) in N
        Q[(s, a)] += α*δ*n
        N[(s, a)] *= γ*λ
    end

    return (hist=hist, Q = copy(Q), time=time()-start)
end

function sarsa_lambda!(env; n_episodes=100, kwargs...)
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    episodes = []

    for i in 1:n_episodes
        reset!(env)
        push!(episodes, sarsa_lambda_episode!(Q, env;
                                              ϵ=max(0.01, 1-i/n_episodes),
                                              kwargs...))
    end

    return episodes
end

function evaluate(env, policy, n_episodes=1000, max_steps=1000, γ=1.0)
    returns = Float64[]
    for _ in 1:n_episodes
        t = 0
        r = 0.0
        reset!(env)
        s = observe(env)
        while !terminated(env)
            a = policy(s)
            r += γ^t*act!(env, a)
            s = observe(env)
            t += 1
        end
        push!(returns, r)
    end
    return returns
end


#################
# Policy Gradient
#################

function gradLogPi(env, theta, a)
    A = collect(actions(env))
    if a == A[1]
        gradPolicy = [1 - exp(theta[1])/sum(exp.(theta)), -exp(theta[1])/sum(exp.(theta)), -exp(theta[1])/sum(exp.(theta)), -exp(theta[1])/sum(exp.(theta))]
    elseif a == A[2]
        gradPolicy = [-exp(theta[2])/sum(exp.(theta)), 1 - exp(theta[2])/sum(exp.(theta)), -exp(theta[2])/sum(exp.(theta)), -exp(theta[2])/sum(exp.(theta))]
    elseif a == A[3]
```

```
        gradPolicy = [-exp(theta[3])/sum(exp.(theta)), - exp(theta[3])/sum(exp.(theta)), 1 - exp(theta[3])/sum(exp.(theta)), -exp(theta[3])/sum(exp.(theta))]
    elseif a == A[4]
        gradPolicy = [-exp(theta[4])/sum(exp.(theta)), -exp(theta[4])/sum(exp.(theta)), - exp(theta[4])/sum(exp.(theta)), 1 - exp(theta[4])/sum(exp.(theta))]
    else
        throw(error("not a valid action"))
    end

    return gradPolicy
end

function policyGradEpisode!(env, θ, α)

    start = time()

    A = collect(actions(env))
    function policy(s,θ)
        theta = θ[s]
        tot = sum(exp.(theta))
        P = zeros(Float64, 4)
        for i = 1:4
            P[i] = exp(theta[i])/tot
        end
        samp = rand(Float64,1)[1]
        if samp <= P[1]
            a = A[1]
        elseif samp <= P[2]+P[1]
            a = A[2]
        elseif samp <= P[3]+P[2]+P[1]
            a = A[3]
        else
            a = A[4]
        end
        return a
    end

    update = []
    for i = 1:10
        path = []
        gradPolicy = []
        d = 0
        R = 0
        while !terminated(env)
            d +=1
            s = observe(env)
            a = policy(s, θ)
            r = act!(env, a)
            path = push!(path, (s,a,r))
            R += r
            push!(gradPolicy, gradLogPi(env, θ[s], a))
        end

        for k in 1:d
            gradU = gradPolicy[k]*R
            s = path[k][1]
            push!(update, (s, gradU))
```

```julia
            R = R - path[k][3]
        end
    end
    hist = []
    for q in eachindex(update)
        s = update[q][1]
        gradU = update[q][2]
        θ[s] += α*gradU/10
        push!(hist,[s])
    end

    return (hist=hist, θ = copy(θ), time=time()-start, policy = policy)
end

function policyGradient(env, n_episodes, α)
    θ = Dict((s) => 0.5*ones(4) for s in observations(env))
    episodes = []

    for i in 1:n_episodes
        reset!(env)
        push!(episodes, policyGradEpisode!(env, θ, α))
    end

    return episodes
end

function learningCurve_steps(env,episodes, n_episodes)
    p1 = plot(xlabel="steps in environment", ylabel="avg return")
    n = convert(Int64,floor(n_episodes/10))
    stop = n_episodes
    for (name, eps) in episodes
        if(name == "SARSA-λ")
            Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
            xs = [0]
            ys = [mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env))))]
            for i in n:n:min(stop, length(eps))
                newsteps = sum(length(ep.hist) for ep in eps[i-n+1:i])
                push!(xs, last(xs) + newsteps)
                Q = eps[i].Q
                push!(ys, mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env)))))
            end
        else
            xs = [0]
            thetas = Dict((s) => 0.5*ones(4) for s in observations(env))
            ys = [mean(evaluate(env, s->eps[1].policy(s,thetas)))]
            for i in n:n:min(stop, length(eps))
                newsteps = sum(length(ep.hist) for ep in eps[i-n+1:i])
                push!(xs, last(xs) + newsteps)
                thetas = eps[i].θ
                push!(ys, mean(evaluate(env, s->eps[i].policy(s, thetas))))
            end
        end
        plot!(p1, xs, ys, label=name)
    end
    display(p1)
end
```

```julia
function learningCurve_time(env,episodes, n_episodes)
    p2 = plot(xlabel="wall clock time", ylabel="avg return")
    n = convert(Int64,floor(n_episodes/10))
    stop = n_episodes
    for (name, eps) in episodes
        if(name == "SARSA-λ")
            Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
            xs = [0.0]
            ys = [mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env))))]
            for i in n:n:min(stop, length(eps))
                newtime = sum(ep.time for ep in eps[i-n+1:i])
                push!(xs, last(xs) + newtime)
                Q = eps[i].Q
                push!(ys, mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env)))))
            end
        else
            xs = [0.0]
            thetas = Dict((s) => 0.5*ones(4) for s in observations(env))
            ys = [mean(evaluate(env, s->eps[1].policy(s, thetas)))]
            for i in n:n:min(stop, length(eps))
                newtime = sum(ep.time for ep in eps[i-n+1:i])
                push!(xs, last(xs) + newtime)
                thetas = eps[i].θ
                push!(ys, mean(evaluate(env, s->eps[i].policy(s, thetas))))
            end
        end
        plot!(p2, xs, ys, label=name)
    end
    display(p2)
end

env = gw
n_eps= 150000
alpha=0.6
PolicyGrad_episodes = policyGradient(env, n_eps, alpha)
lambda_episodes = sarsa_lambda!(env, n_episodes=n_eps, α=0.1, λ=0.3)
display(render(env))
episodes = Dict("Policy Gradient"=>PolicyGrad_episodes, "SARSA-λ"=>lambda_episodes)
learningCurve_steps(env, episodes, n_eps)
learningCurve_time(env, episodes, n_eps)
```