Doncey Albin | DMU, Spring 2023 | Homework 4

In [1]:
```
using DMUStudent.HW4
using DMUStudent.HW4.RL: actions, act!, observe, reset!, observations, terminated, clo
using StaticArrays: SA
using Statistics: mean
```

In [2]:
```
# Instantiate gridworld from HW4
m = HW4.gw

# Discover available actions
move_right = actions(m)[1]   # first action (move right)
move_left = actions(m)[2]    # second action (move left)
move_up = actions(m)[3]      # third action (move up)
move_down = actions(m)[4]    # fourth action (move down)

# turn actions into dictionary for ease of use
const action2SA = Dict("up" => SA[0,1], "left" => SA[-1,0], "down"=>SA[0,-1], "right"=

# Show action as an arrow
@show HW4.aarrow[action2SA["left"]]

# test actions
@show act!(m, action2SA["right"]) # Switch move_left for another available action to t
@show terminated(m)  # Tell if MDP is terminated

# Observe the state of the agent
observe(m)

# clone the GridWorldEnv
m_copy = clone(m)

# see observations
#@show HW4.RL.observations(m) # shows states as [row, column] 10x10 matrix

# Set state of agent
new_state = SA[1, 5]
HW4.RL.setstate!(m, new_state)
render(m)

HW4.reset!(m)
```

Out[2]:
```
HW4.aarrow[action2SA["left"]] = '←'
act!(m, action2SA["right"]) = -0.14936554675956898
terminated(m) = false
2-element StaticArraysCore.SVector{2, Int64} with indices SOneTo(2):
 1
 1
```

In [3]:
```
# SARSA
using LinearAlgebra: I
using SparseArrays
import POMDPTools
using Plots
```

SARSA Algorithm

In [4]:
```
function sarsa_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.4)
    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]

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

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

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

    Q[(s,a)] += α*(r - Q[(s, a)])

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

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

    for i in 1:n_episodes
        reset!(env)

        if i < 1000
            eps = 1
        else
            eps = max(0.002, 1-i/n_episodes)
        end

        alpha = 0.2*(1 - 0.99*i/n_episodes)
        push!(episodes, sarsa_episode!(Q, env; ϵ=eps, α=alpha))
    end

    return episodes
end
```

Out[4]:
```
sarsa! (generic function with 1 method)
```

Q-Learning Algorithm

In [5]:
```
function qlearning_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.4)
    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]

    while !terminated(env)
        # maximize Q[(sp, ap)] over available actions (act_prime)
        ap = argmax(a->Q[(sp, a)], actions(env))
        Qp_max = Q[(sp, ap)]

        Q[(s,a)] += α*(r + γ*Qp_max - Q[(s, a)])

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

    Q[(s,a)] += α*(r - Q[(s, a)])

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

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

    for i in 1:n_episodes
        reset!(env)
        eps = max(0.1, 1-i/n_episodes)
        alpha = 0.2*(1 - 0.99*i/n_episodes)
        push!(episodes, qlearning_episode!(Q, env; ϵ=eps, α=alpha))
    end

    return episodes
end
```

Out[5]:
```
qlearning! (generic function with 1 method)
```

In [6]:
```
function doubleQlearning_episode!(Q, QA, QB, env; ϵ=0.10, γ=0.99, α=0.4)
    start = time()

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

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

    while !terminated(env)
        rand_num = rand()

        if rand_num <= 0.5
            a_star = argmax(a->QA[(sp, a)], actions(env))
            QA[(s,a)] += α*(r + γ*QB[(sp, a_star)] - QA[(s, a)])
        else
            b_star = argmax(a->QB[(sp, a)], actions(env))
            QB[(s,a)] += α*(r + γ*QA[(sp, b_star)] - QB[(s, a)])
        end

        Q[(s,a)] = QA[(s,a)] + QB[(s,a)]

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

    QA[(s,a)] += α*(r - QA[(s, a)])
    QB[(s,a)] += α*(r - QB[(s, a)])
    Q[(s,a)] = QA[(s,a)] + QB[(s,a)]

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

function doubleQlearning!(env; n_episodes)
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    QA = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    QB = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    episodes = []

    for i in 1:n_episodes
        reset!(env)
        eps = max(0.2, 1-i/n_episodes)
        alpha = 0.1*(1 - 0.99*i/n_episodes)
        push!(episodes, doubleQlearning_episode!(Q, QA, QB, env; ϵ=eps, α=alpha))
    end

    return episodes
end
```

Out[6]:
```
doubleQlearning! (generic function with 1 method)
```

In [7]:
```
env = clone(m)
num_episodes = 300000
sarsa_episodes = sarsa!(env, n_episodes=num_episodes);
qlearning_episodes = qlearning!(env, n_episodes=num_episodes);
doubleQlearning_episodes = doubleQlearning!(env, n_episodes=num_episodes);
```

In [8]:
```
using Interact

function get_q(state, ep)
    if get(env.rewards, state, 0.0) == 0
        #return maximum(map(a->ep.Q[(state,a)], actions(env)))
        return mean(map(a->ep.Q[(state,a)], actions(env)))
    end
    return get(env.rewards, state, 0.0)
end

episodes_alg = sarsa_episodes
#episodes_alg = qlearning_episodes
#episodes_alg = doubleQlearning_episodes
@manipulate for episode in 1:length(episodes_alg), step in 1:maximum(ep->length(ep.his
    ep = episodes_alg[episode]
    i = min(step, length(ep.hist))
    setstate!(env, ep.hist[i]) # set where marker is on map

    render(env; color=s->get_q(s, ep), policy=s->actions(env)[argmax(map(a->ep.Q[(s,a)
end
```

The WebIO Jupyter extension was not detected. See the WebIO Jupyter integration documentation for more information.

Out[8]:

WebIO not detected.

Please read the troubleshooting guide for more information on how to resolve this issue.

https://juliagizmos.github.io/WebIO.jl/latest/troubleshooting/not-detected/

In [9]:
```
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
```

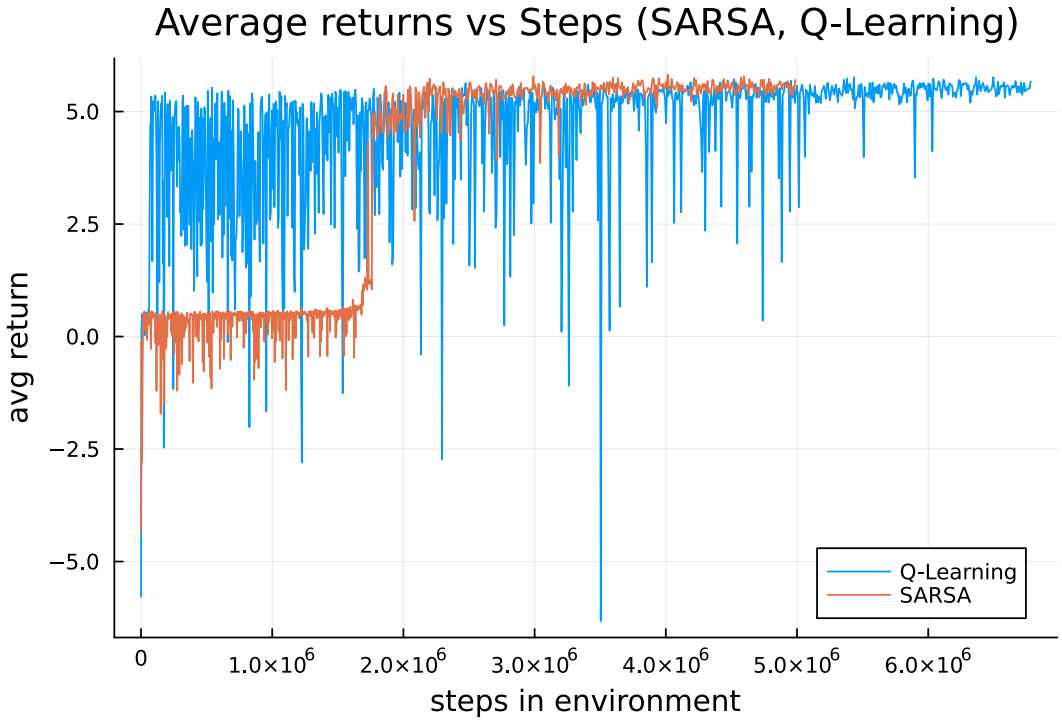
Out[9]:
```
evaluate (generic function with 4 methods)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Plots 1: SARSA vs Q-Learning

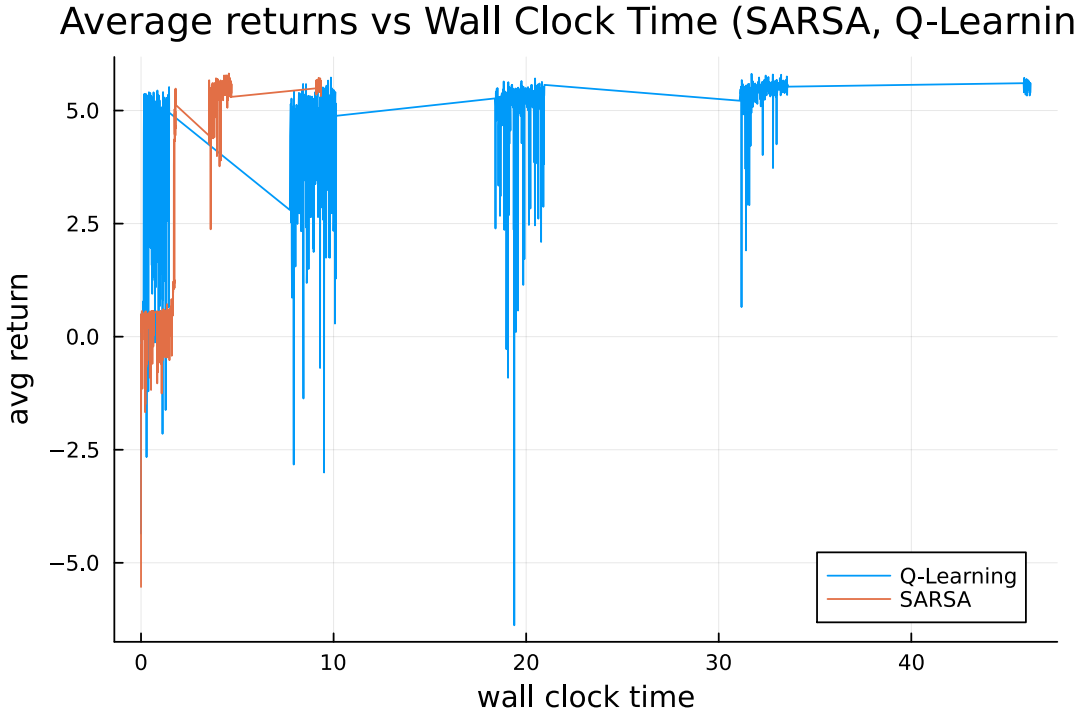
In [10]:
```
episodes = Dict("SARSA"=>sarsa_episodes, "Q-Learning"=>qlearning_episodes)

p = plot(xlabel="steps in environment", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name, eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Steps (SARSA, Q-Learning)")
end
p
```

Out[10]:



In [11]:
```
p = plot(xlabel="wall clock time", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name,eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Wall Clock Time (SARSA, Q-Learning)")
end
p
```

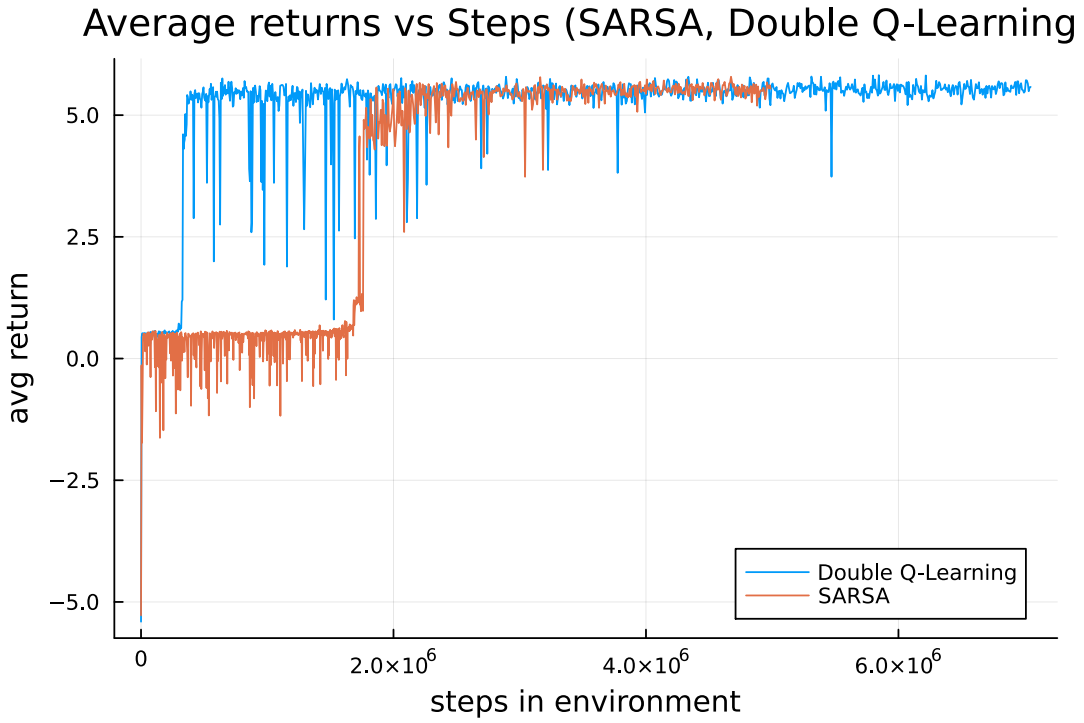
Out[11]:



Plots 2: SARSA vs Double Q-Learning

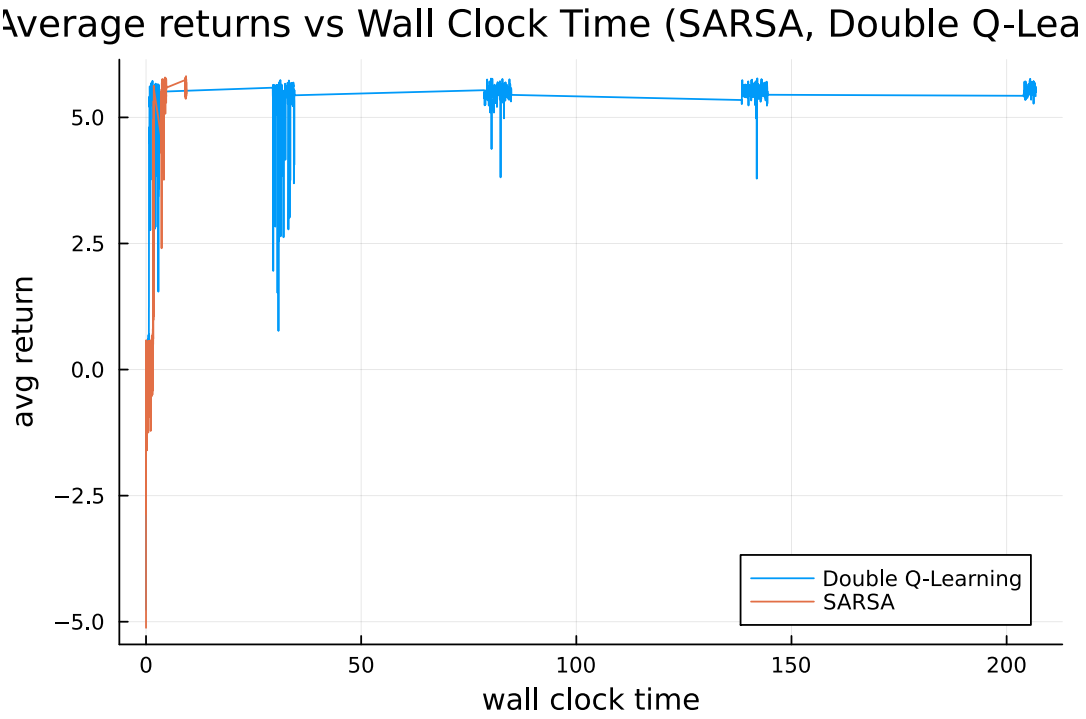
In [12]:
```
episodes = Dict("SARSA"=>sarsa_episodes, "Double Q-Learning"=>doubleQlearning_episodes

p = plot(xlabel="steps in environment", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name, eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Steps (SARSA, Double Q-Learning)")
end
p
```

Out[12]:



In [13]:
```
p = plot(xlabel="wall clock time", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name,eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Wall Clock Time (SARSA, Double Q-Learning)")
end
p
```

Out[13]:

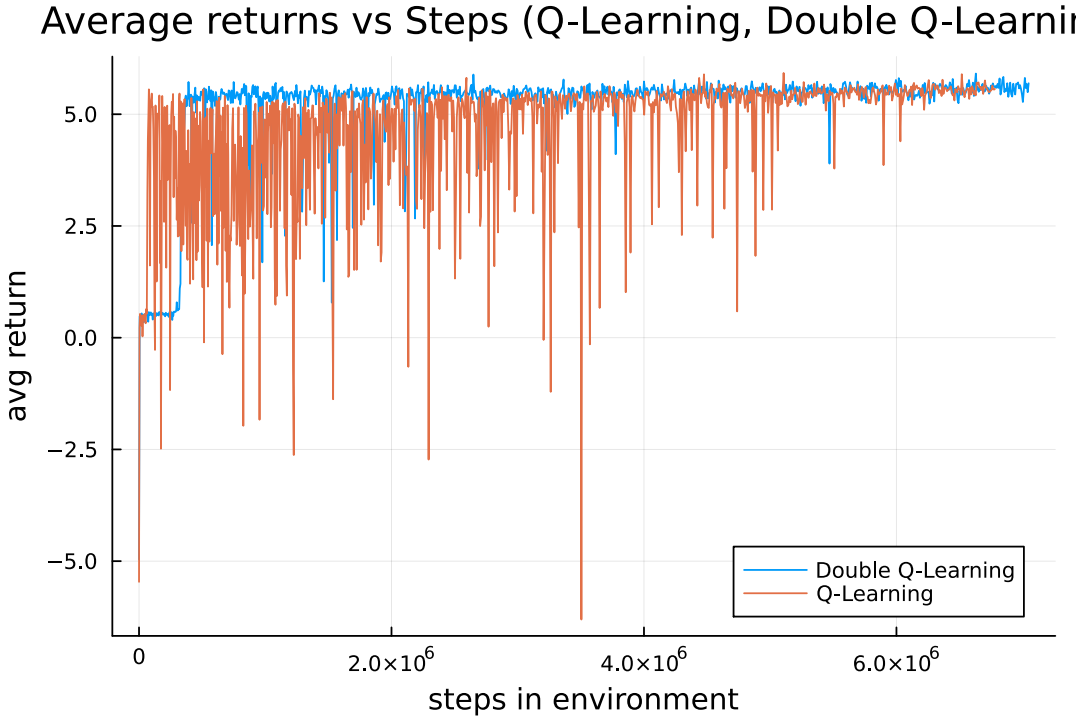


Plots 3: Q-Learning vs Double Q-Learning

In [14]:

```
episodes = Dict("Q-Learning"=>qlearning_episodes, "Double Q-Learning"=>doubleQlearning

p = plot(xlabel="steps in environment", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name, eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Steps (Q-Learning, Double Q-Learning)")
end
p
```

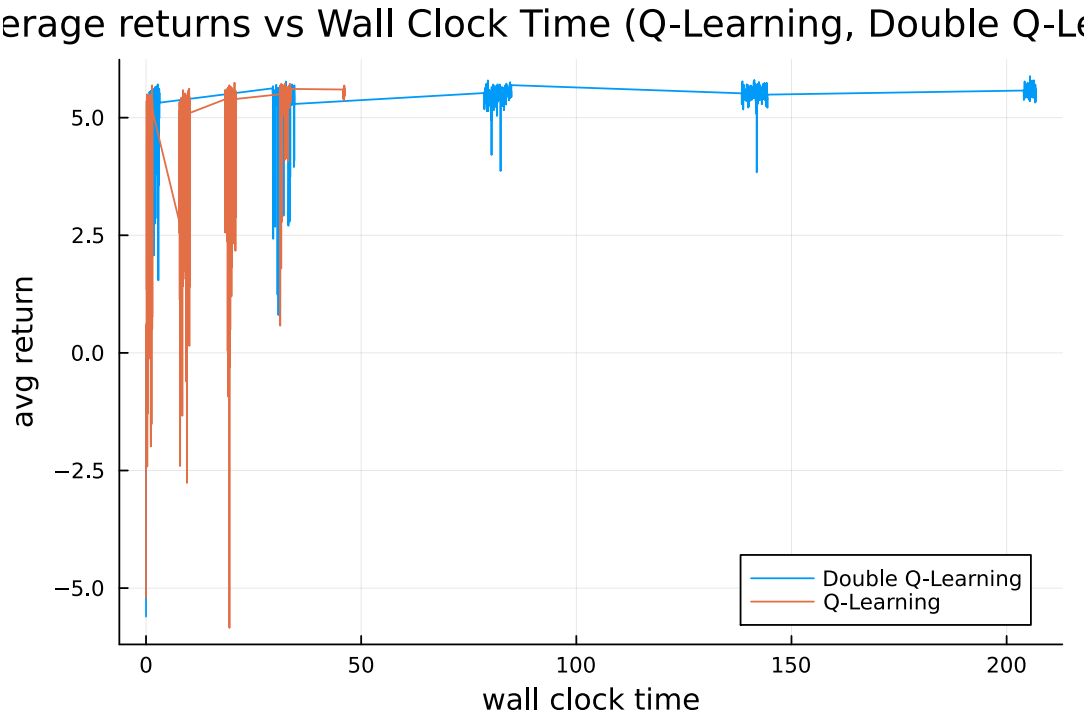
Out[14]:



In [15]:

```
p = plot(xlabel="wall clock time", ylabel="avg return")
n = convert(Int64, num_episodes/1000)
stop = num_episodes
for (name,eps) in episodes
    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
    plot!(p, xs, ys, label=name)
    title!("Average returns vs Wall Clock Time (Q-Learning, Double Q-Learning)")
end
p
```

Out[15]:



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js