RL Homework 1

Chris Swetenham (s1149322)

March 1, 2012

1 Grid World Navigation

We are supplied a description of an 8 by 8 grid world with walls. Actions are movement in the 4 cardinal directions. The start state is at bottom left and the goal state is in the top right. We formulate this as an MDP with a state for each coordinate on the grid, represented by a single integer. We calculate possible state transitions based on the wall definitions. We make the goal state a terminal state. The reward is 0 when we are in the goal state, -1 otherwise. We use a deterministic policy. For policy iteration, we fill in a starting policy which is non-optimal:

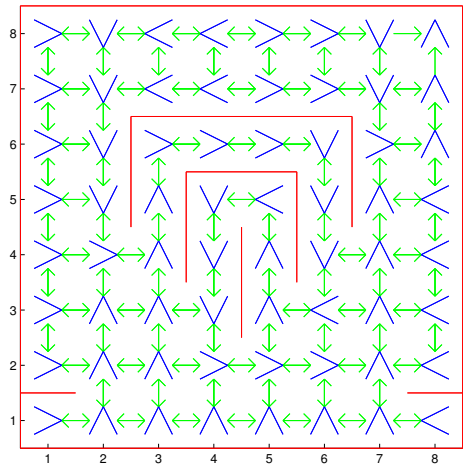

Figure 1: Starting Policy and State Transitions

```
%% World Setup

Rot90 = [
     0 +1;
    -1  0
];

% Action representation: Idx mapped to dX, dY by this table

Actions = [
    +1  0; % 1 E >
     0 -1; % 2 S V
    -1  0; % 3 W <
     0 +1; % 4 N ^
];
NActions = size(Actions, 1);

MapWidth = 8;
MapHeight = 8;
MapSize = [MapWidth MapHeight];
NStates = prod(MapSize);
GoalState = NStates;

stateFromPos = @(P) (P(1) - 1) + MapWidth * (P(2) - 1) + 1;
posFromState = @(S) [mod((S - 1),MapWidth) + 1, floor((S - 1)/MapWidth) + 1];


%% Walls

% Horizontal wall representation: Y, StartX, EndX
Walls_H = [
    0.5, 0.5, 8.5;
    1.5, 0.5, 1.5;
    1.5, 7.5, 8.5;
    5.5, 3.5, 5.5;
    6.5, 2.5, 6.5;
    8.5, 0.5, 8.5
];

% Vertical wall representation: X, StartY, EndY
Walls_V = [
    0.5, 0.5, 8.5;
    2.5, 4.5, 6.5;
    3.5, 3.5, 5.5;
    4.5, 2.5, 4.5;
    5.5, 3.5, 5.5;
    6.5, 4.5, 6.5;
    8.5, 0.5, 8.5
];

% Compute state transition matrix from walls

% [State x Action] -> State
StateTransitionTable = zeros([NStates, NActions]);

for A = 1:NActions
    for S = 1:NStates
        P = posFromState(S);

        X = P(1);
        Y = P(2);
        NP = Actions(A,:) + P;
        MoveOK = 1;

        for W = 1:size(Walls_H, 1)
            WS = [Walls_H(W, 2) Walls_H(W, 1)];
            WE = [Walls_H(W, 3) Walls_H(W, 1)];
```

```
            MoveOK = MoveOK & ~testSegmentSegment(P, NP, WS, WE);
        end

        for W = 1:size(Walls_V, 1)
            WS = [Walls_V(W, 1) Walls_V(W, 2)];
            WE = [Walls_V(W, 1) Walls_V(W, 3)];
            MoveOK = MoveOK & ~testSegmentSegment(P, NP, WS, WE);
        end

        if (MoveOK)
            StateTransitionTable(S, A) = stateFromPos(NP);
        else
            StateTransitionTable(S, A) = S;
        end
    end
end

StateTransitionTable(GoalState, :) = repmat(GoalState, [NActions 1]);

%% Visualisation

% Policy representation: [State] -> Action
% from the goal state
StartPolicy = [
    1 4 4 4 4 4 4 3 ...
    1 4 4 1 1 1 4 3 ...
    1 4 4 2 4 3 4 3 ...
    1 1 4 2 4 2 4 3 ...
    1 2 4 2 3 2 4 3 ...
    1 2 1 1 1 2 1 4 ...
    1 2 3 3 1 1 2 4 ...
    1 2 3 3 1 1 2 4
]';

% Arrows for rendering state transitions
Arrow(1, :, :) = [
    0.25 0;
    0.75 0;
    0.625 -0.125;
    0.75 0;
    0.625 +0.125;
    0.75 0;
];
% Rotate for other directions
for A = 2:4
    for L = 1:size(Arrow, 2)
        Arrow(A,L,:) = Rot90^(A-1) * squeeze(Arrow(1,L,:));
    end
end

% Arrows for rendering policies
% 0 G X
ActionGlyphs(1, 1, :, :) = [-1 +1; -1 +1]';
ActionGlyphs(2, 1, :, :) = [-1 +1; +1 -1]';
% 1 E >
ActionGlyphs(1, 2, :, :) = [-1 +1; -1 +1]';
ActionGlyphs(2, 2, :, :) = [+1  0; -1  0]';
% 2 S V
ActionGlyphs(1, 3, :, :) = [-1  0; +1  0]';
ActionGlyphs(2, 3, :, :) = [+1 -1; +1 -1]';
% 3 W <
ActionGlyphs(1, 4, :, :) = [+1 -1; +1 -1]';
ActionGlyphs(2, 4, :, :) = [+1  0; -1  0]';
% 4 N ^
ActionGlyphs(1, 5, :, :) = [-1  0; +1  0]';
ActionGlyphs(2, 5, :, :) = [-1 +1; -1 +1]';
% Rescale
ActionGlyphs = 0.25 * ActionGlyphs;
```

```
function drawWalls()
    for i = 1:size(Walls_H, 1)
        Y      = Walls_H(i,1);
        StartX = Walls_H(i,2);
        EndX   = Walls_H(i,3);

        line([StartX EndX], [Y Y], 'Color', 'red');
    end
    for i = 1:size(Walls_V, 1)
        X      = Walls_V(i,1);
        StartY = Walls_V(i,2);
        EndY   = Walls_V(i,3);

        line([X X], [StartY EndY], 'Color', 'red');
    end
end

function drawGlyph(Glyph, P)
    XS = squeeze([Glyph(1:2:end,1), Glyph(2:2:end,1)])';
    YS = squeeze([Glyph(1:2:end,2), Glyph(2:2:end,2)])';
    line(P(1) + XS, P(2) + YS, 'Color', 'green');
end

function drawStateTransitions()
    [NStates NActions] = size(StateTransitionTable);
    for A = 1:NActions
        for S = 1:NStates
            P = posFromState(S);
            if (StateTransitionTable(S, A) ~= S)
                drawGlyph(squeeze(Arrow(A,:,:,:)), P);
            end
        end
    end
end

function drawActionImpl(Glyphs, X, Y, A)
    XS = squeeze(Glyphs(1, A+1,:,:));
    YS = squeeze(Glyphs(2, A+1,:,:));
    line(X + XS, Y + YS, 'Color', 'blue');
end

function drawPolicy(Policy)
    [NStates] = size(Policy);

    for S = 1:NStates
        P = posFromState(S);
        X = P(1);
        Y = P(2);
        drawActionImpl(ActionGlyphs, X, Y, Policy(S));
    end
end

function vizPolicy(Fig, V, Policy)
     figure(Fig);
        imagesc(reshape(V, MapSize));
        colormap('gray');
        drawWalls();
        drawPolicy(Policy);
    axis([0.5, 8.5, 0.5, 8.5], 'xy', 'square');
end

figure(1);
    drawWalls();
    drawStateTransitions();
    drawPolicy(StartPolicy);
axis([0.5, 8.5, 0.5, 8.5], 'xy', 'square');
```

```
writeFigurePDF('StartingPolicy.pdf');

reward = @(S, A, S2) -1 * (S ~= GoalState);
```

Listing 1: part1.m

a) Policy Evaluation

We represent state transition probabilities $\mathcal{P}^a_{ss'}$ as a MATLAB function which takes the current state and the action, and returns a list of possible next states and a list of their probabilities. We implement policy evaluation by stepping through these states in a single-step backup, and repeat this until the change in the state value function is sufficiently small. For the starting policy, our evaluation converges after 23 steps. We implement policy improvement by finding the greedy policy for the value function by searching in each state for the action which maximises the expected return. Figure 2 shows the resulting value function and greedy policy.

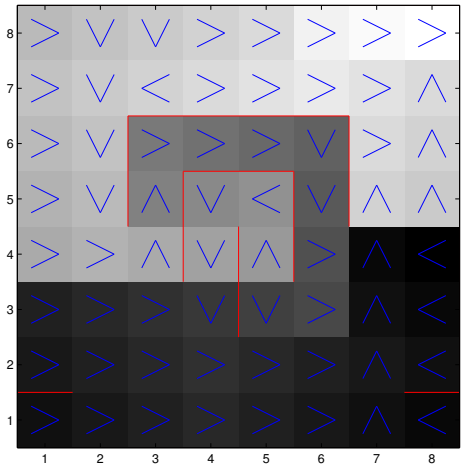

Figure 2: Value function and Greedy Policy

```
NormalStateTransitions = @(S, A) deal(StateTransitionTable(S, A), 1);

%% Policy evaluation
function [V] = evaluatePolicyStep(V, Policy, StateTransitions, Reward, Discount)
    for S = 1:NStates
        A = Policy(S);
        [S2 Pr] = StateTransitions(S, A);
        NV = 0;
        for I = 1:size(S2, 2)
            NV = NV + Pr(I) * (Reward(S, A, S2(I)) + Discount * V(S2(I)));
        end
        V(S) = NV;
    end
end
```

```
function [V] = evaluatePolicy(Policy, StateTransitions, Reward, Discount,
     MaxIterations)
    V = zeros([NStates, 1]);
    for Iteration = 1:MaxIterations
        OldV = V;
        V = evaluatePolicyStep(V, Policy, StateTransitions, Reward, Discount);
        MaxDelta = max(abs(V - OldV));

        if (MaxDelta < 0.001)
            break;
        end
    end
    fprintf('Iterations␣before␣value␣fn␣convergence:␣%d\n', Iteration);
end

function [Policy] = computeGreedyPolicy(V, StateTransitions, Reward, Discount)
    Policy = zeros([NStates, 1]);
    for S = 1:NStates
        Q = zeros([NActions, 1]);
        for A = 1:NActions;
            [S2 Pr] = StateTransitions(S, A);
            for I = 1:size(S2, 2)
                Q(A) = Q(A) + Pr(I) * (Reward(S, A, S2(I)) + Discount * V(S2(I)));
            end
        end
        [~, A] = max(Q);
        Policy(S) = A;
    end
end

fprintf('Policy␣evaluation\n');
Discount = 1;
MaxIterations = 1000;
V = evaluatePolicy(StartPolicy, NormalStateTransitions, reward, Discount,
     MaxIterations);
Policy = computeGreedyPolicy(V, NormalStateTransitions, reward, Discount);
vizPolicy(2, V, Policy);
writeFigurePDF('PolicyEvaluation.pdf');
```

Listing 2: part1a.m

b) Policy Iteration

We interleave policy evaluation and policy improvement until the policy remains unchanged. This converges after 9 iterations of policy iteration. The resulting policy by manual inspection seems to correctly take the shortest path to the goal.

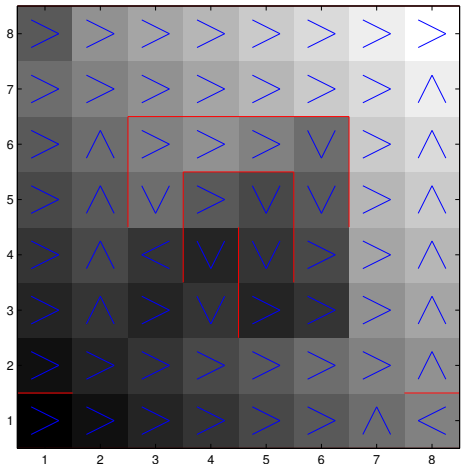

Figure 3: Optimal Value Function and Policy - Policy Iteration

```
%% Policy iteration

function policyIteration(Fig, Policy, Discount, StateTransitions,
     MaxValueIterations, MaxPolicyIterations)
    for PP = 1:MaxPolicyIterations
        % Evaluate policy
        V = evaluatePolicy(Policy, StateTransitions, reward, Discount,
             MaxValueIterations);

        % Compute greedy policy
        NewPolicy = computeGreedyPolicy(V, StateTransitions, reward, Discount);
        if (all(Policy == NewPolicy))
            break;
        end
        Policy = NewPolicy;

        vizPolicy(Fig, V, Policy);

        % refresh;
        % pause(0.1);
    end
    fprintf('Policy␣Iteration:␣Iterations␣before␣policy␣convergence:␣%d\n', PP);
end
fprintf('Policy␣iteration\n');
Discount = 1;
MaxValueIterations = 1000;
MaxPolicyIterations = 1000;
policyIteration(3, StartPolicy, Discount, NormalStateTransitions,
     MaxValueIterations, MaxPolicyIterations);
writeFigurePDF('NormalPolicyIteration.pdf');
```

Listing 3: part1b.m

c) Value Iteration

We now use value iteration to directly compute the value of each state as the value of the action which maximises the expected value of the successor state.

We repeat this until the value of each state remains the same. This takes 15 iterations to converge. We compute the greedy policy for the value function as before, this time only to visualise it.

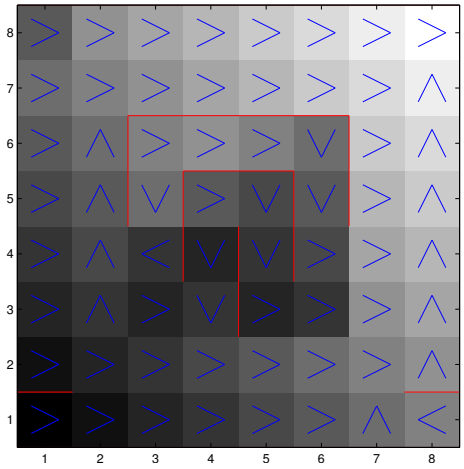

Figure 4: Optimal Value Function and Policy - Value Iteration

```
%% Value iteration

function [NV] = valueIterationStep(V, StateTransitions, Reward, Discount)
    NV = zeros([NStates, 1]);
    for S = 1:NStates
        Q = zeros([NActions, 1]);
        for A = 1:NActions;
            [S2 Pr] = StateTransitions(S, A);
            for I = 1:size(S2, 2)
                Q(A) = Q(A) + Pr(I) * (Reward(S, A, S2(I)) + Discount * V(S2(I)));
            end
        end
        [V2 Dummy] = max(Q);
        NV(S) = V2;
    end
end

function valueIteration(Fig, Discount, StateTransitions, MaxPolicyIterations)
    V = zeros([NStates 1]);
    for PP = 1:MaxPolicyIterations
        % Evaluate policy
        NewV = valueIterationStep(V, StateTransitions, reward, Discount);

        % Compute greedy policy
        Policy = computeGreedyPolicy(NewV, StateTransitions, reward, Discount);

        if (all(V == NewV))
            break;
        end
        V = NewV;

        vizPolicy(Fig, V, Policy);

        % refresh;
        % pause(0.1);
```

```
    end
    fprintf('Value Iteration: Iterations before policy convergence: %d\n', PP);
end
fprintf('Value iteration\n');
Discount = 1;
MaxPolicyIterations = 1000;
valueIteration(4, Discount, NormalStateTransitions, MaxPolicyIterations);
writeFigurePDF('NormalValueIteration.pdf');
```

Listing 4: part1c.m

d) Sticky Walls

We now introduce a sticky right wall. When in one of the states in the rightmost column, there is a probability p that we will stay in the current state no matter which action was taken. We use a new state transition function which returns the current state with probability p, and run the previous policy iteration and value iteration code. We start with $p = 0.4$ and also compare with $p = 0.6$. For states next to the wall and far from the goal, it becomes worthwhile to step away from the wall incurring a -1 penalty rather than try to stay near the wall; even more so in the case where $p = 0.6$. In the policy iteration case, the number of policy iterations required for convergence is still 9, but the value function requires up to 32 steps to converge each iteration. In the value iteration case, the value function now takes 52 iterations to converge, many more than the 15 seen previously.

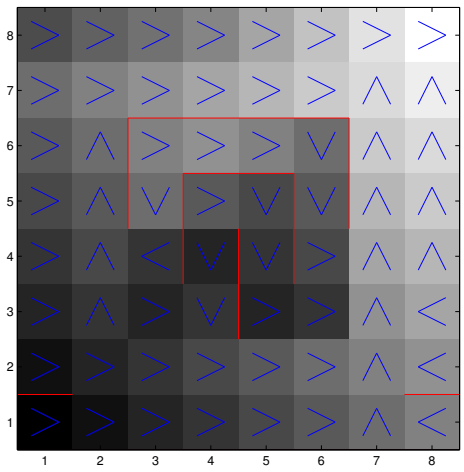

Figure 5: Stick Walls - Policy Iteration

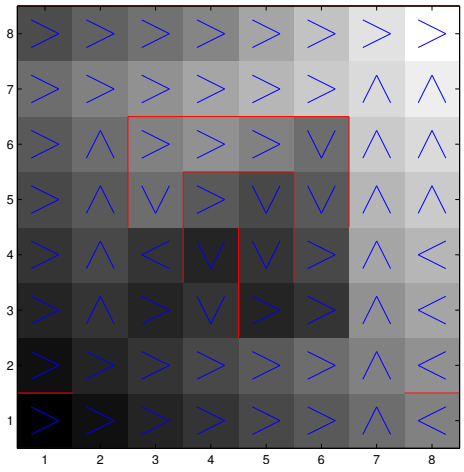

Figure 6: Stick Walls - Value Iteration

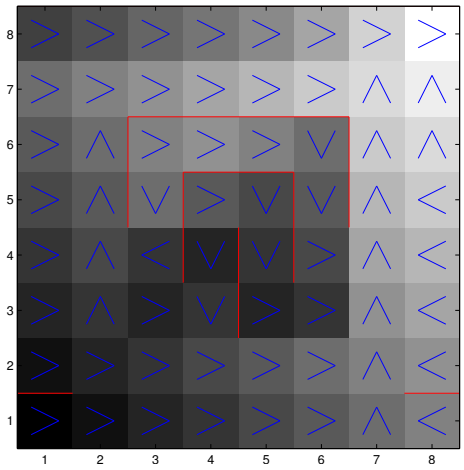

Figure 7: Stick Walls - Policy Iteration, $p = 0.6$

```
% Sticky wall
StickyProb = 0.4;

function [S2 Pr] = stickyStateTransitions(S, A)
    P = posFromState(S);
    X = P(1);
    if X == MapWidth
        S2 = [S, StateTransitionTable(S, A)];
        Pr = [StickyProb, 1 - StickyProb];
    else
        S2 = StateTransitionTable(S, A);
        Pr = 1;
    end
```

```
end

Discount = 1;
MaxValueIterations = 1000;
MaxPolicyIterations = 1000;
fprintf('Sticky world policy iteration\n');
policyIteration(5, StartPolicy, Discount, @stickyStateTransitions,
     MaxValueIterations, MaxPolicyIterations);
writeFigurePDF('StickyPolicyIteration.pdf')
fprintf('Sticky world value iteration\n');
valueIteration(6, Discount, @stickyStateTransitions, MaxPolicyIterations);
writeFigurePDF('StickyValueIteration.pdf');
StickyProb = 0.6;
fprintf('Sticky world policy iteration - p = 0.6\n');
policyIteration(7, StartPolicy, Discount, @stickyStateTransitions,
     MaxValueIterations, MaxPolicyIterations);
writeFigurePDF('StickyPolicyIteration6.pdf');
```

Listing 5: part1d.m

2 Secretary Problem

a) MDP Design

We specify the problem as an MDP as follows:

States: On step K, we have rejected the first $K-1$ applicants, and we observe the ranking R of the Kth candidate relative to them. There are K such possible rankings, positions 1 to K. The rankings of the rejected candidates among themselves is irrelevant. In MATLAB we represent these states in a triangular matrix. In the MATLAB code we will represent the state by the values K and R and functions of those states as tables indexed by K and R.

Terminal states: All states after the Accept action. We do not represent these states explicitly in the MATLAB code.

Actions: Accept or Reject. On the 30th step we can only Accept. Represented as 2 and 1 in the MATLAB code.

The total number of non-terminal states is $1+2+...+30 = \frac{1}{2}N(N+1) = 465$.

Reward: We assign to each candidate a random value $0-1$ according to which they are ranked. The reward is the value of the accepted candidate on entering the terminal state, 0 otherwise.

b) Monte Carlo

We simulate 25 million episodes. Each episode we generate random values for the candidates and step through them in order. We use an ϵ-greedy on-policy monte-carlo algorithm with $\epsilon = 0.1$.

Each step we insert our new candidate into a sorted list and thus determine the candidate's rank. We take the action specified by the ϵ-greedy policy, and record the sequence of states we visit in the episode. The sequence of Rs visited and its length is sufficent to determine the sequence of states and actions taken. At the end of the episode, we use this list to update our estimate of Q and our greedy policy for the visited states. We compute the state-value function V from Q for visualisation.

There is no strict convergence test for MC so we empirically detemined 25 million episodes as giving stable results. After 10 minutes of computation we converge on the following policy, seen in Figure 8. The white area indicates Accept actions. 1 is the highest rank. Right at the start we always reject. A little later we will only accept someone ranked very high. Right before the end, we will accept anyone above average since on the final step we have to accept and expect an average applicant value (0.5).

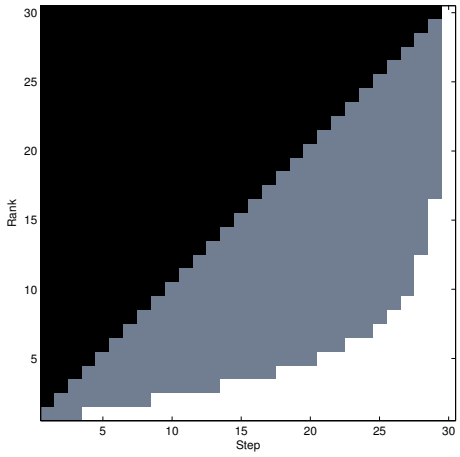

Figure 8: Policy after 25 million iterations

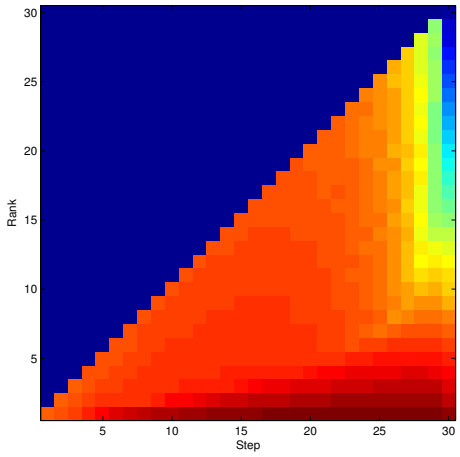

Figure 9: State value function after 25 million iterations

```
NCandidates = 30;

% For the on-policy monte-carlo implementation we will learn the Q(S, A)
% function using an e-greedy strategy.

Epsilon = 0.1;

Q = zeros([NCandidates, NCandidates, 2]);
Policy = zeros([NCandidates, NCandidates]);
TotalReturn = zeros([NCandidates, NCandidates, 2]);
VisitCount = zeros([NCandidates, NCandidates, 2]);

tic;
MaxEpisodes = 25000000;
```

```
for Episode = 1:MaxEpisodes
    % Each episode, we will generate 30 random candidate values
    % Without loss of generality we will interview them in order 1-N

    C = rand([NCandidates, 1]);

    % We always visit the starting state [1,1].
    % We always finish in the terminal state.
    % In each episode we go through states [1, 1], [2, R_2], ... [K, R_K]
    % before hitting the terminal state; so we need only store the sequence
    % of rankings after the first state to determine all the visited
    % states.
    % Similarly we know the sequence of actions taken based on the length
    % of this sequence.
    Rs = zeros([1 NCandidates]);

    SortedSoFar = zeros([1 NCandidates]);

    for K = 1:NCandidates
        % Rank of the candidate among those seen so far
        R = K;
        if K > 1
            for I = (K-1):-1:1
                if SortedSoFar(I) < C(K)
                    SortedSoFar(I + 1) = SortedSoFar(I);
                    R = R - 1;
                else
                    break;
                end
            end
        end
        SortedSoFar(R) = C(K);

        if (K == NCandidates)
            Action = 2;
        elseif (rand(1) < Epsilon)
            Action = (rand(1) > 0.5) + 1;
        else
            Action = Policy(K, R);
        end
        Rs(K) = R;
        if Action == 2
            break;
        end
    end
    MaxK = K;
    Reward = C(K);
    for K = 1:MaxK
        A = (K == MaxK) + 1;
        R = Rs(K);

        NV = VisitCount(K, R, A) + 1;
        VisitCount(K, R, A) = NV;
        NT = TotalReturn(K, R, A) + Reward;
        TotalReturn(K, R, A) = NT;
        Q(K, R, A) = NT / NV;

        Policy(K, R) = (Q(K, R, 1) < Q(K, R, 2)) + 1;
    end

    if (mod(Episode, 10000) == 0)
        fprintf('Episode %d (%03d%%)\n', Episode, floor(100*Episode/MaxEpisodes));
    end
end
for K = 1:NCandidates
    for R = (K+1):NCandidates
        Policy(K, R) = 0;
    end
```

```
end
toc;
figure;
imagesc(Policy');
axis xy;
axis square;
xlabel('Step');
ylabel('Rank');
colormap('gray');
writeFigurePDF('SecretaryMDPPolicy.pdf');
figure;
imagesc(Q(:,:,1)');
axis xy;
axis square;
xlabel('Step');
ylabel('Rank');
colormap('gray');
writeFigurePDF('SecretaryMDPQFunction.pdf');
figure;
[VG, Dummy] = max(Q, [], 3);
VE = 0.5 * sum(Q, 3);
V = (1-Epsilon) * VG + Epsilon * VE;
imagesc(V');
axis xy;
axis square;
xlabel('Step');
ylabel('Rank');
colormap('gray');
writeFigurePDF('SecretaryMDPVFunction.pdf');
end
```

Listing 6: part2b.m

c) Q-Learning

TODO

Appendix A - Additional Code

```
function [] = writeFigurePDF(Fig, FileName)
% writeFigure writes the current figure to a pdf file
% INPUT Fig: [optional] the figure to write; default is the current figure.
%       FileName: a string containing the path of the file to save to.

if nargin < 2
    FileName = Fig;
    Fig = gcf;
end
FileNameRoot = regexp(FileName, '(.*)\.pdf$', 'tokens');
FileNameEPS = [FileNameRoot{1}{1} '.eps'];
print(Fig, '-depsc', FileNameEPS);
[Status, ~] = unix(['epstopdf ' FileNameEPS]);
if Status ~= 0
    fprintf(2, 'Error running epstopdf!\n');
end
```

Listing 7: writeFigurePDF.m

```
function [F, C] = makeLogFile(FileName)
% makeLogFile opens a log file and makes an onCleanup object to close it.
F = fopen(FileName, 'w');
C = onCleanup(@()fclose(F));
```

Listing 8: makeLogFile.m