RL Homework 2

Chris Swetenham (s1149322)

April 2, 2012

1 MDP

Similar to the previous assignment, we operate in a 3 by 4 grid world surrounded by walls with no internal obstacles. Actions are movement in the 4 cardinal directions. The start state is in one corner and the goal state is one of the center states (state 7 in the standard numbering). 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 from the actions. 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.

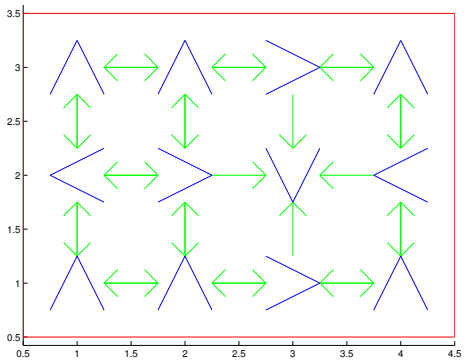

Figure 1: Random Policy and State Transitions

We use the value iteration code from last assignment. We have also modified it to return the Q-function directly which will be used later. The value iteration procedure converges rapidly after 4 iterations, giving the following value function and policy:

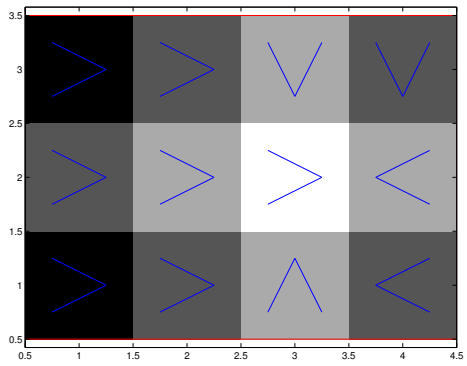

Figure 2: Optimal policy from value iteration

```
%% Setup
clc;
clear all;
close all;
dbstop if error;
dbstop if infnan;

%% Part I - MDP

%% World Setup

stream0 = RandStream('mt19937ar','Seed',0);
%RandStream.setGlobalStream(stream0);
RandStream.setDefaultStream(stream0);

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 = 4;
MapHeight = 3;
MapSize = [MapWidth MapHeight];
NStates = prod(MapSize);
GoalState = 7;

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


%% Walls

% Compute state transition matrix

function T = computeStateTransitionTable()
    T = zeros([NStates, NActions]);
```

```
    for LA = 1:NActions
        for LS = 1:NStates
            PS2 = posFromState(LS);
            NP = Actions(LA,:) + PS2;
            NX = NP(1);
            NY = NP(2);

            if (NX < 1 || NX > MapWidth || NY < 1 || NY > MapHeight)
                T(LS, LA) = LS;
            else
                T(LS, LA) = stateFromPos(NP);
            end
        end
    end
end

% [State x Action] -> State
StateTransitionTable = computeStateTransitionTable();

% Make final state absorbing
StateTransitionTable(GoalState, :) = repmat(GoalState, [NActions 1]);

%% Visualisation

% Policy representation: [State] -> Action
% from the goal state
StartPolicy = floor(rand(NStates, 1) * 4) + 1;

% 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;

VizMinX = 0.5;
VizMaxX = MapWidth+0.5;
VizMinY = 0.5;
VizMaxY = MapHeight+0.5;
```

```
function drawWalls()
    line([VizMinX VizMaxX], [VizMinY VizMinY], 'Color', 'red');
    line([VizMinX VizMaxX], [VizMaxY VizMaxY], 'Color', 'red');
    line([VizMinX VizMinX], [VizMinY VizMaxY], 'Color', 'red');
    line([VizMaxX VizMaxX], [VizMinY VizMaxY], 'Color', 'red');
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 LA = 1:NActions
        for LS = 1:NStates
            PS2 = posFromState(LS);
            if (StateTransitionTable(LS, LA) ~= LS)
                drawGlyph(squeeze(Arrow(LA,:,:,:)), PS2);
            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] = numel(Policy);

    for LS = 1:NStates
        PS2 = posFromState(LS);
        X = PS2(1);
        Y = PS2(2);
        drawActionImpl(ActionGlyphs, X, Y, Policy(LS));
    end
end

function vizPolicy(Fig, V, Policy)
     figure(Fig);
        imagesc(reshape(V, MapSize)');
        colormap('gray');
        drawWalls();
        drawPolicy(Policy);
    axis([VizMinX, VizMaxX, VizMinY, VizMaxY], 'xy', 'equal');
end

figure(1);
    drawWalls();
    drawStateTransitions();
    drawPolicy(StartPolicy);
axis([VizMinX, VizMaxX, VizMinY, VizMaxY], 'xy', 'equal');
writeFigureEPS('StartingPolicy.eps');

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

%% We implement state transition probabilities PP(a, s, s') as a function
%% PP(s, a) -> [States, Probs]; which returns a list of possible next
%% states and their probabilities.

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

%% Value iteration
```

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

function [NV NQ] = valueIterationStep(V, StateTransitions, Reward, Discount)
    NV = zeros([NStates, 1]);
    NQ = zeros([NStates, NActions]);
    for LS = 1:NStates
        Q = zeros([NActions, 1]);
        for LA = 1:NActions;
            [S2 Pr] = StateTransitions(LS, LA);
            for LI = 1:size(S2, 2)
                Q(LA) = Q(LA) + Pr(LI) * (Reward(LS, LA, S2(LI)) + Discount * V(S2
                    (LI)));
            end
        end
        [V2 ~] = max(Q);
        NQ(LS, :) = Q';
        NV(LS) = V2;
    end
end

function [V Q] = valueIteration(Fig, Discount, StateTransitions,
     MaxPolicyIterations)
    V = zeros([NStates 1]);
    for Iter = 1:MaxPolicyIterations
        % Evaluate policy
        [NewV Q] = 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);
    end
    fprintf('Value Iteration: Iterations before policy convergence: %d\n', Iter);
end
fprintf('Value Iteration\n');
Discount = 1;
MaxPolicyIterations = 1000;
[~, Q] = valueIteration(4, Discount, NormalStateTransitions, MaxPolicyIterations);
writeFigureEPS('ValueIteration.eps');
```

Listing 1: part1.m

2 Bayes Filter

We now wish to formulate a Bayes Filter for localisation starting from an uncertain state (maintaining deterministic state transitions). We will evaluate this using a policy which chooses random actions with equal probability, and the goal state is not taken into account. The only signal available is a bump sensor which is triggered if the robot bumps into a wall. In the usual formulation of the Bayes Filter, the signal is a function of the current state, but here it is a function of the state transition. We implement a version of the Bayes Filter where the signal is a function of the previous state and the action taken. We could have instead chosen to augment the state with an extra bit to indicate the bumper status in the last transition, but this seems like a less elegant formulation and it is desirable to avoid needlessly increasing the number of states.

```
function [NewBel] = bayesFilter(Bel, A, Z, StateTransitions, Observations)
% bayesFilter updates the current belief based on an action,
% an observation, the state transition probabilities, and observation
% probabilities.
% INPUT Bel: [NStates x 1]
%            Current belief as a dense discrete distribution.
%       A: [1x1]
%          Action taken.
%       Z: [1x1]
%          Observation.
%       StateTransitions: (State, Action) -> (States [Nx1], Probs [Nx1])
%          Function taking a state action pair, returning sparse
%          distribution of new states.
%       Observations: (State, Action) -> (Obs [Nx1], Probs [Nx1])
%          Function taking a state action pair, returning sparse
%          distribution of observations.
    NStates = size(Bel, 1);
    NewBel = zeros(NStates, 1);
    for NS = 1:NStates
        Temp = 0;
        for S = 1:NStates
            [S2 Pr] = StateTransitions(S, A);
            PS2 = sum(Pr(S2 == NS));

            [Z2 Pr] = Observations(S, A);
            PZ = sum(Pr(Z2 == Z));

            Temp = Temp + PZ * PS2 * Bel(S);
        end
        NewBel(NS) = Temp;
    end
    NewBel = NewBel / sum(NewBel);
end
```

Listing 2: bayesFilter.m

In our test run, the belief converges to one corner of the grid after just 4 steps. The belief state is only reduced when some of the possible states could have run into the wall and some would not have run into the wall, under the latest action.

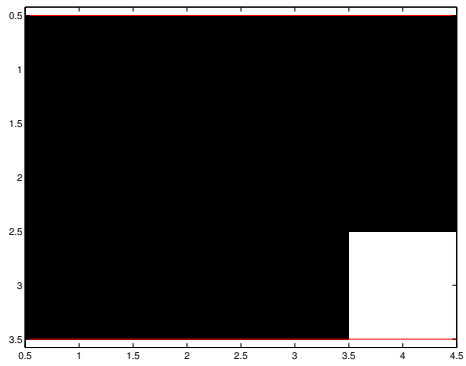

Figure 3: Converged belief from Bayes Filter

```
%% Part II - Bayes filter

% bayes filter has p(z|x) but we bump on action, not position.
% One solution is to modify bayes filter to use p(z|u,x)
% Another is to augment state to be (x, y, bumped).

% for all s'
%  temp = sum_s { p(s'|u, s) p(s) }
%  p(s') = \eta p(z|s', u) temp

% where eta is a normalising constant

function [Z2 Pr] = Observations(S, A, StateTransitions)
    [S2 Pr] = StateTransitions(S, A);
    PS1 = sum(Pr(S2 == S));

    Z2 = [    0   1];
    Pr = [1-PS1 PS1];
end

NormalObservations = @(S, A) Observations(S, A, NormalStateTransitions);

function vizBel(Fig, Bel)
     figure(Fig);
        imagesc(reshape(Bel, MapSize)');
        colormap('gray');
        drawWalls();
    axis([VizMinX, VizMaxX, VizMinY, VizMaxY], 'ij', 'equal');
end

function [X] = sampleDiscrete(Xs, Ps)
    C = cumsum(Ps);
    R = rand(1);
    I = find(C > R, 1);
    % If non-zero prob of R==0.0, C >= R might select sample with P=0.
    % If non-zero prob of R==1.0, C > R might fail to select anything.
    I = sum(I) || numel(C); % handle rare case where R==1.0

    X = Xs(I);
end

fprintf('Bayes Filter\n');
B = ones(NStates, 1) / NStates;
S = floor(rand(1) * NStates) + 1;
for PP = 1:10000
    A = floor(rand(1) * 4) + 1;
```

```
    [S2 Pr] = NormalStateTransitions(S, A);
    NS = sampleDiscrete(S2, Pr);
    Z = (S == NS);
    S = NS;
    B = bayesFilter(B, A, Z, NormalStateTransitions, NormalObservations);

    vizBel(3, B);

    if (max(B) == 1)
        break;
    end
end
fprintf('Bayes Filter: Iterations before belief convergence: %d\n', PP);
writeFigureEPS('BayesFilter.eps');
```

Listing 3: part2.m

3 QMDP

We now combine the code from the previous sections in order to implement QMDP. The initial state is unknown, but we compute the optimal value function of the underlying MDP and then pick the optimal action according to our belief and the value function.

As implied by the problem formulation we change the state transition table for the QMDP run so that the goal state is no longer absorbing, and instead terminate an episode when the belief has converged and the robot is in the goal state.

We observe that the resulting process often does not converge. Trying all possible starting states, we find that it converges only starting from states 3 and 4, converging in 5 and 4 states respectively. From other states, it will end up in a belief state where it will never take an action that further constrains its belief state. Shown here is the result after 100 iterations starting from state 12:

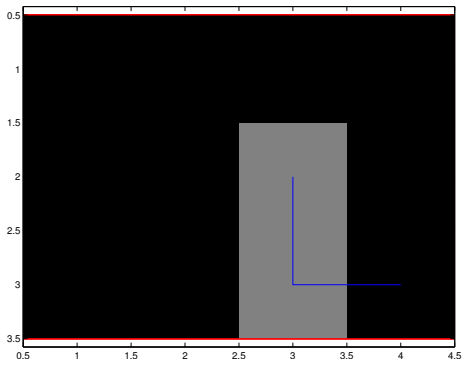

Figure 4: Non-convergence, starting from state 12

The non-converging situations have 2 or 4 possible states and hover around the goal state without ever taking an action that might bump into the wall and

thereby reduce the belief state.

```
%% Part III - QMDP

fprintf('QMDP\n');
function [As Pr] = computeAction(B, Q)
    QQ = zeros(NActions, 1);
    for LA = 1:NActions;
        QQ(LA) = sum(B .* Q(:, LA));
    end
    [Max, ~] = max(QQ);

    As = find(QQ == Max);
    Pr = ones(size(As, 1), 1) / size(As, 1);
end

% Recompute state transition table without making goal state absorbing.
QMDPStateTransitionTable = computeStateTransitionTable();
QMDPStateTransitions = @(S, A) deal(QMDPStateTransitionTable(S, A), 1);
QMDPObservations = @(S, A) Observations(S, A, QMDPStateTransitions);

% Value iteration won't converge if we don't make the goal state absorbing.
% We keep the value function computed with the absorbing goal state.
MaxIter = 100;
for SS = 1:NStates
    fprintf('QMDP: Start at state %d\n', SS);
    B = ones(NStates, 1) / NStates;
    S = SS;
    Path = S;
    for PP = 1:MaxIter
        [As Pr] = computeAction(B, Q);
        A = sampleDiscrete(As, Pr);
        [S2 Pr] = QMDPStateTransitions(S, A);
        NS = sampleDiscrete(S2, Pr);
        Z = (S == NS);
        Path = [Path; NS];
        B = bayesFilter(B, A, Z, QMDPStateTransitions, QMDPObservations);

        % pause(1);

        S = NS;

        % Stop if belief has converged and we are in goal state
        if (max(B) == 1 && S == GoalState)
            break;
        end
    end

    vizBel(5, B);
    hold on;
    PathX = [];
    PathY = [];
    for PI = 1:size(Path, 1)
        P = posFromState(Path(PI));
        PathX = [PathX; P(1)];
        PathY = [PathY; P(2)];
    end
    plot(PathX, PathY, 'b');

    if PP==MaxIter
        fprintf('QMDP: Start at state %d, did not converge\n', SS);
    else
        fprintf('QMDP: Start at state %d, iterations before belief convergence: %d
            \n', SS, PP);
    end
end
writeFigureEPS('QMDP-12.eps');
```

Listing 4: part3.m

4 Discussion

We investigated making the policy of the QMDP algorithm non-deterministic among the maximal-value choices, but this does not help in cases such as the one in Figure 4. It is a fundamental limitation of QMDP that it does not reason about future belief states when selecting an action.

In this toy environment, the actions and observations are deterministic, so one simple way to fix the policy would be to first run right until a bump, then down until a bump, thereby collapsing the belief state in both dimensions; and then driving straight for the goal.

If the goal signal were made available to the bayes filter as an additional input, it could localise the goal state correctly; but this would often not correspond to a realistic situation in the real world.

We could try using an epsilon-greedy policy, although this would not reliably produce a sequence of actions that reduce our belief state in general.

We could also solve the full POMDP problem in this setting, using point-based value iteration over the 12-dimensional belief state; obviously this would be more computationally expensive but it is the 'correct' solution.

Appendix A - Additional Code

```
function [] = writeFigureEPS(Fig, FileNameEPS)
% writeFigureEPS writes the current figure to an eps 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
    FileNameEPS = Fig;
    Fig = gcf;
end
SavedBreakpoints = dbstatus;
dbclear if infnan; % Yes, this is a thing
print(Fig, '-depsc', FileNameEPS);
dbstop(SavedBreakpoints);
```

Listing 5: writeFigureEPS.m