**Intelligent Robotics Systems**

**Exercise 3: Markov-Decision-Process**

Arseni Pertzovsky (317377372), Rotem Galiki (305328296)

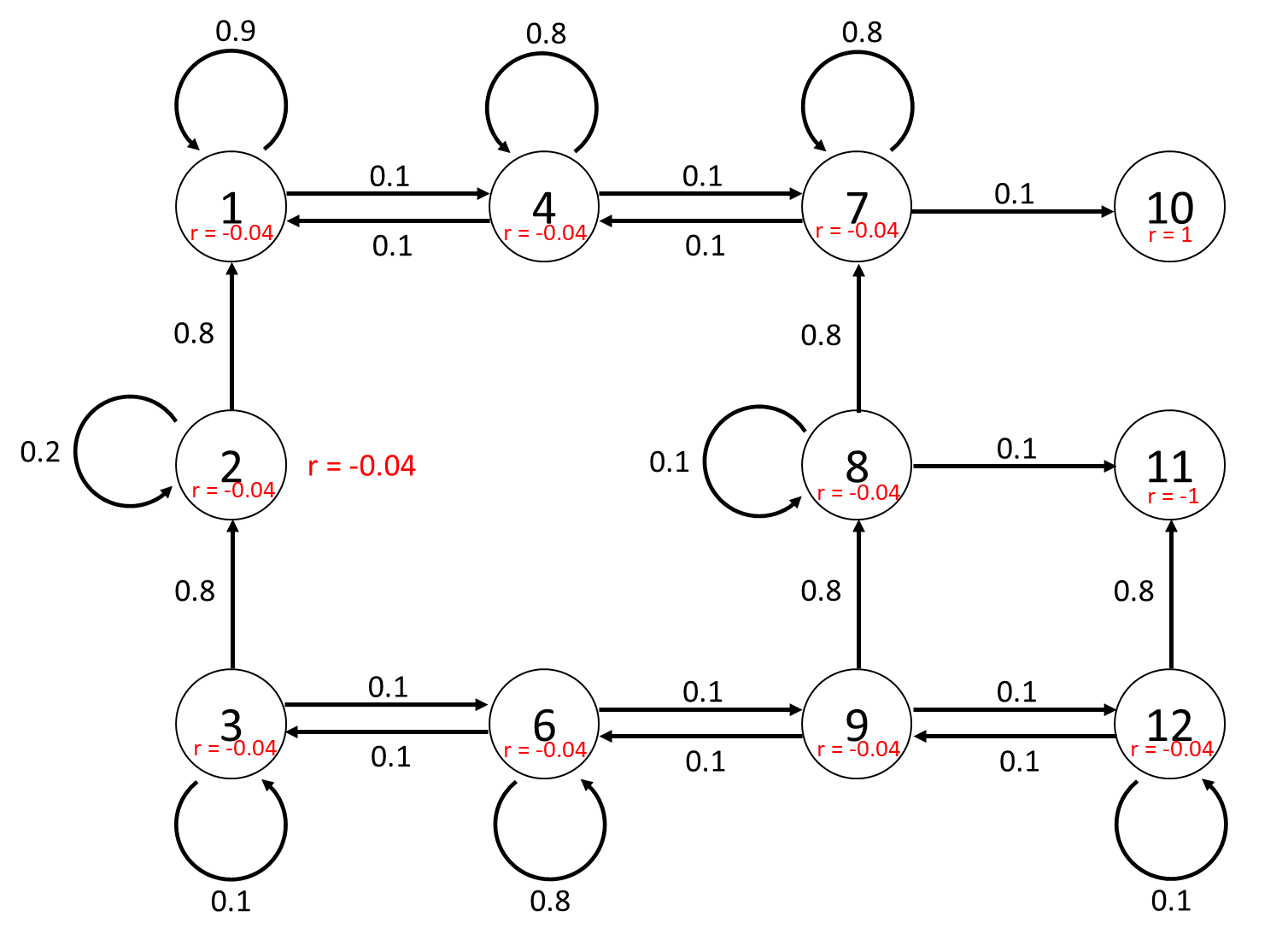
**Question 1:**

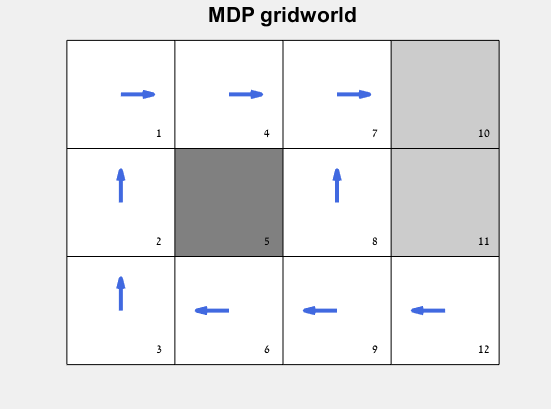
1. **Markov Decision Process (MDP)** is a way to format discrete time stochastic control process based on a model, in order to calculate the and find the optimal set of actions towards a specified goal. Problems that can be solved with an MDP include several properties: model, known goal, uncertain environment (hence the stochastic quality of MDP). An example for a problem that can be solved with MDP can be any navigation process, in which each action preformed is a little bit noisy and can lead to other actions being performed.
2. **State-value function** - The state value function v(s) of an MRP is the expected return starting from state s: .
3. **Action-value function** - The action-value function is the expected return (total discounted future reward) starting from state s, taking an (arbitrary) action a. It describes how good it is to take a particular action a from a given state s.
4. **Policy** – a decision of which action to perform from the set of actions in each possible state.
5. **Dynamic programming** is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively.
6. **Value iteration** makes use of the Bellman optimality equation in order to find the optimal value-function, and then derive the optimal policy from the optimal value-function. In each step of the algorithm the estimate of the value-function is updated until a termination condition is met.
7. **Policy iteration** makes use of the Bellman expectation equation in order to find the optimal policy. The policy iteration algorithm is composed out of a policy evaluation and policy improvement step. For a given policy π the value function v of π is estimated (policy evaluation). From the value function v of π an improved policy is derived following a greedy strategy (policy improvement). This cycle is repeated until convergence to the optimal policy π∗ and optimal value function v∗ is obtained. The algorithm has a definite stopping condition: when the policy does not change in the course of policy improvement step to all states, the algorithm is completed. Policy iteration is usually slower than value iteration for a large number of possible states.
8. **Reinforcement learning (RL)** is an area of machine learning concerned with how agents ought to take actions in an environment to maximize some notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. Reinforcement learning is particularly well-suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, backgammon, checkers and go (AlphaGo). The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible. The MDP can be learned simulating different actions from each state until you have a high degree of confidence in the learned transition function and the learned reward function. Unfortunately this is frequently not possible because the MDP is too large or it would be too expensive to learn the MDP. RL algorithms such as Q-Learning try to do both things at the same time: learn the MDP and solve it to find the optimal policy. So RL is a technique to learn an MDP and solve it for the optimal policy at the same time. Thus, every case where we do not no MDP is a good example to describe the difference between the two.

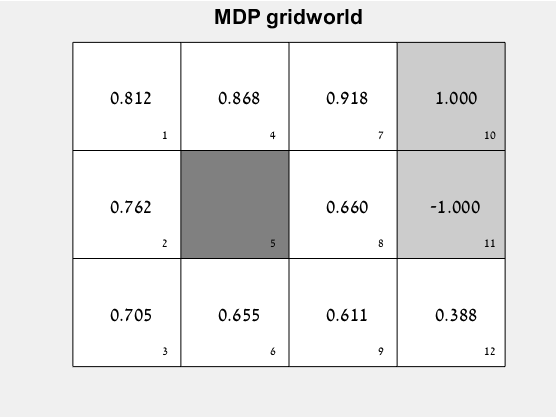
**Question 2:**

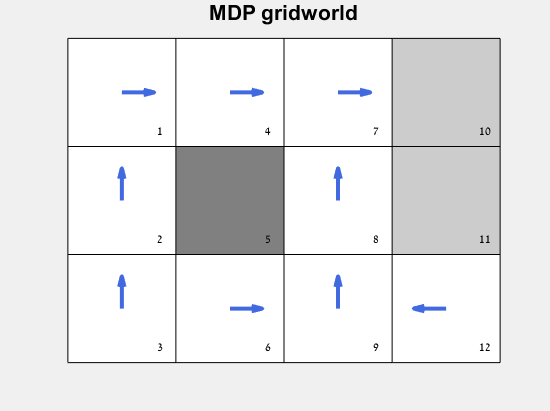
**Part a:**

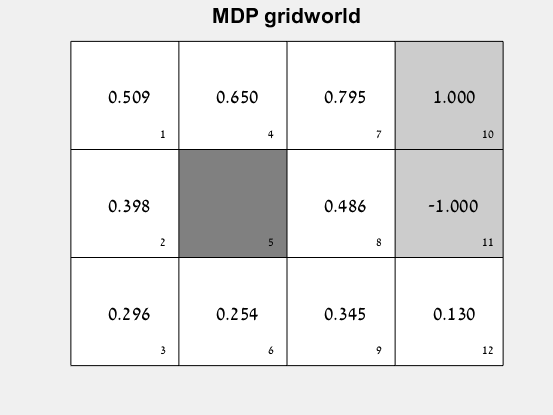
A graphical model of the MDP for a = N showing all states, transition probabilities and rewards:



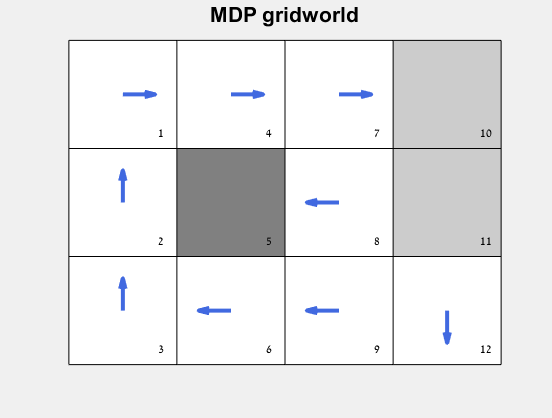
**Part b:**

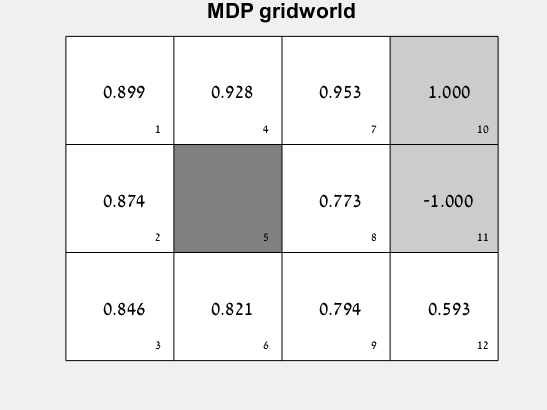


**Part c:**

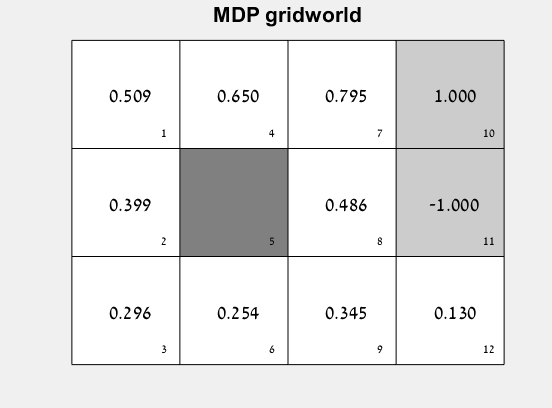
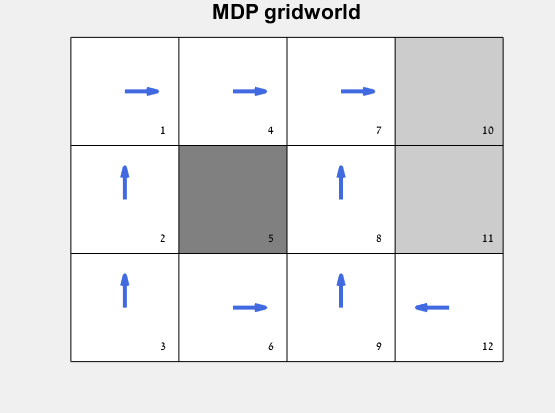


The difference between part b and part c is in the value of the discount factor, gamma, which is equal to 1 in part b and to 0.9 in part c. It means that the algorithm relies less on the future rewards it will gain, and more on current and close rewards collected in the current sell. Because of this, we can see that all the weights decreased in comparison to part b. The policy itself also changed due to prioritizing closer rewards. In part b algorithm prefers to be as far as possible from a dangerous cell (no. 11), however in part be the algorithm wants to finish the ride as soon as possible, in the shortest way.

**Part d:**



In this part gamma equals to 1 and the rewards for every mid-state equals to -0.02 (less than in previous parts). In this case, because each step costs less, the algorithm does not mind to be a little bit longer on the field and to distance as far as possible from -1 cell.

**Part e:**

Part c and part e share the same gamma and reward settings, while part c was solved using value iteration and part e was solved using policy iteration. When comparing the figures presented in these parts, we can see that the results are identical (i.e. they have the same estimated values for each state and the same final chosen policy).

**Matlab code:**

% ------------------------------- part a ------------------------------- %

global prob\_matrix;

prob\_matrix = zeros([12 12 4]);

% ------------ 1 ------------ %

prob\_matrix(1,1,1) = 0.9;

prob\_matrix(4,1,1) = 0.1;

prob\_matrix(1,2,1) = 0.8;

prob\_matrix(2,2,1) = 0.2;

prob\_matrix(2,3,1) = 0.8;

prob\_matrix(6,3,1) = 0.1;

prob\_matrix(3,3,1) = 0.1;

prob\_matrix(4,4,1) = 0.8;

prob\_matrix(7,4,1) = 0.1;

prob\_matrix(1,4,1) = 0.1;

prob\_matrix(6,6,1) = 0.8;

prob\_matrix(9,6,1) = 0.1;

prob\_matrix(3,6,1) = 0.1;

prob\_matrix(7,7,1) = 0.8;

prob\_matrix(10,7,1) = 0.1;

prob\_matrix(4,7,1) = 0.1;

prob\_matrix(7,8,1) = 0.8;

prob\_matrix(11,8,1) = 0.1;

prob\_matrix(8,8,1) = 0.1;

prob\_matrix(8,9,1) = 0.8;

prob\_matrix(12,9,1) = 0.1;

prob\_matrix(6,9,1) = 0.1;

prob\_matrix(11,12,1) = 0.8;

prob\_matrix(12,12,1) = 0.1;

prob\_matrix(9,12,1) = 0.1;

% ------------ 2 ------------ %

prob\_matrix(4,1,2) = 0.8;

prob\_matrix(2,1,2) = 0.1;

prob\_matrix(1,1,2) = 0.1;

prob\_matrix(2,2,2) = 0.8;

prob\_matrix(3,2,2) = 0.1;

prob\_matrix(1,2,2) = 0.1;

prob\_matrix(6,3,2) = 0.8;

prob\_matrix(3,3,2) = 0.1;

prob\_matrix(2,3,2) = 0.1;

prob\_matrix(7,4,2) = 0.8;

prob\_matrix(4,4,2) = 0.2;

prob\_matrix(9,6,2) = 0.8;

prob\_matrix(6,6,2) = 0.2;

prob\_matrix(10,7,2) = 0.8;

prob\_matrix(8,7,2) = 0.1;

prob\_matrix(7,7,2) = 0.1;

prob\_matrix(11,8,2) = 0.8;

prob\_matrix(9,8,2) = 0.1;

prob\_matrix(7,8,2) = 0.1;

prob\_matrix(12,9,2) = 0.8;

prob\_matrix(9,9,2) = 0.1;

prob\_matrix(8,9,2) = 0.1;

prob\_matrix(12,12,2) = 0.9;

prob\_matrix(11,12,2) = 0.1;

% ------------ 3 ------------ %

prob\_matrix(2,1,3) = 0.8;

prob\_matrix(1,1,3) = 0.1;

prob\_matrix(4,1,3) = 0.1;

prob\_matrix(3,2,3) = 0.8;

prob\_matrix(2,2,3) = 0.2;

prob\_matrix(3,3,3) = 0.9;

prob\_matrix(6,3,3) = 0.1;

prob\_matrix(4,4,3) = 0.8;

prob\_matrix(1,4,3) = 0.1;

prob\_matrix(7,4,3) = 0.1;

prob\_matrix(6,6,3) = 0.8;

prob\_matrix(3,6,3) = 0.1;

prob\_matrix(9,6,3) = 0.1;

prob\_matrix(8,7,3) = 0.8;

prob\_matrix(4,7,3) = 0.1;

prob\_matrix(10,7,3) = 0.1;

prob\_matrix(9,8,3) = 0.8;

prob\_matrix(8,8,3) = 0.1;

prob\_matrix(11,8,3) = 0.1;

prob\_matrix(9,9,3) = 0.8;

prob\_matrix(6,9,3) = 0.1;

prob\_matrix(12,9,3) = 0.1;

prob\_matrix(12,12,3) = 0.9;

prob\_matrix(9,12,3) = 0.1;

% ------------ 4 ------------ %

prob\_matrix(1,1,4) = 0.9;

prob\_matrix(2,1,4) = 0.1;

prob\_matrix(2,2,4) = 0.8;

prob\_matrix(1,2,4) = 0.1;

prob\_matrix(3,2,4) = 0.1;

prob\_matrix(3,3,4) = 0.9;

prob\_matrix(2,3,4) = 0.1;

prob\_matrix(1,4,4) = 0.8;

prob\_matrix(4,4,4) = 0.2;

prob\_matrix(3,6,4) = 0.8;

prob\_matrix(6,6,4) = 0.2;

prob\_matrix(4,7,4) = 0.8;

prob\_matrix(7,7,4) = 0.1;

prob\_matrix(8,7,4) = 0.1;

prob\_matrix(8,8,4) = 0.8;

prob\_matrix(7,8,4) = 0.1;

prob\_matrix(9,8,4) = 0.1;

prob\_matrix(6,9,4) = 0.8;

prob\_matrix(8,9,4) = 0.1;

prob\_matrix(9,9,4) = 0.1;

prob\_matrix(9,12,4) = 0.8;

prob\_matrix(11,12,4) = 0.1;

prob\_matrix(12,12,4) = 0.1;

% --------------------------- %

% ------------------------------- part b ------------------------------- %

delta = 1;

tetta = 10^-4;

V = zeros(1,12);

gamma = 1;

% Value iteration

while delta > tetta

delta = 0;

for s = 1:12

v = V(s);

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum + gamma\*prob\_matrix(i,s,a)\*V(i);

end

sum = sum + r(s);

max\_array(a) = sum;

end

V(s) = max(max\_array);

delta = max([delta, abs(v-V(s))]);

end

end

% plot the optimal values

myworld = cWorld();

myworld.plot;

myworld.plot\_value(transpose(V));

% calculate the optimal policy

policy = zeros(1, 12);

for s = 1:12

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum+prob\_matrix(i,s,a)\*V(i);

end

max\_array(a) = r(s)+ gamma\*sum;

end

[val, idx] = max(max\_array);

policy(s) = idx;

end

% plot the optimal policy

myworld = cWorld();

myworld.plot;

myworld.plot\_policy(transpose(policy));

% ------------------------------- part c ------------------------------- %

delta = 1;

tetta = 10^-4;

V = zeros(1,12);

gamma = 0.9;

% Value iteration

while delta > tetta

delta = 0;

for s = 1:12

v = V(s);

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum + gamma\*prob\_matrix(i,s,a)\*V(i);

end

sum = sum + r(s);

max\_array(a) = sum;

end

V(s) = max(max\_array);

delta = max([delta, abs(v-V(s))]);

end

end

% plot the optimal values

myworld = cWorld();

myworld.plot;

myworld.plot\_value(transpose(V));

% calculate the optimal policy

policy = zeros(1, 12);

for s = 1:12

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum+prob\_matrix(i,s,a)\*V(i);

end

max\_array(a) = r(s)+ gamma\*sum;

end

[val, idx] = max(max\_array);

policy(s) = idx;

end

% plot the optimal policy

myworld = cWorld();

myworld.plot;

myworld.plot\_policy(transpose(policy));

% ------------------------------- part d ------------------------------- %

delta = 1;

tetta = 10^-4;

V = zeros(1,12);

gamma\_d = 1;

% Value iteration

while delta > tetta

delta = 0;

for s = 1:12

v = V(s);

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum + gamma\_d\*prob\_matrix(i,s,a)\*V(i);

end

sum = sum + r(s);

max\_array(a) = sum;

end

V(s) = max(max\_array);

delta = max([delta, abs(v-V(s))]);

end

end

% plot the optimal values

myworld = cWorld();

myworld.plot;

myworld.plot\_value(transpose(V));

% calculate the optimal policy

policy = zeros(1, 12);

for s = 1:12

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum+prob\_matrix(i,s,a)\*V(i);

end

max\_array(a) = r(s)+ gamma\*sum;

end

[val, idx] = max(max\_array);

policy(s) = idx;

end

% plot the optimal policy

myworld = cWorld();

myworld.plot;

myworld.plot\_policy(transpose(policy));

% ------------------------------- part e ------------------------------- %

% 1 - initialization

V\_e = zeros(1,12);

V\_e(10) = 1;

V\_e(11) = -1;

policy\_e(1:4,1:12) = 0.25;

delta = 1;

tetta = 10^-4;

gamma\_e = 0.9;

policy\_stable = false;

while policy\_stable == false

% 2 - Policy iteration

delta = 1;

while delta > tetta

delta = 0;

for s = 1:12

v = V\_e(s);

% calculate reward

sum\_r = r(s);

% calculate values

sum\_v = 0;

for i = 1:12

sum\_p = 0;

for a = 1:4

sum\_p = sum\_p + policy\_e(a,s)\*prob\_matrix(i,s,a);

end

sum\_v = sum\_v + sum\_p\*V\_e(i);

end

V\_e(s) = sum\_r + gamma\_e\*sum\_v;

delta = max([delta, abs(v-V\_e(s))]);

end

end

% 3 - policy improvement

policy\_stable = true;

for s = 1:12

[val, idx] = max(policy\_e(:,s));

old\_action = idx;

max\_array = zeros(1, 4);

for a = 1:4

sum = 0;

for i = 1:12

sum = sum + prob\_matrix(i,s,a)\*V\_e(i);

end

max\_array(a) = r(s)+ gamma\_e\*sum;

end

[val, idx] = max(max\_array);

action = idx;

for i = 1:4

if i == action

policy\_e(i,s) = 1;

else

policy\_e(i,s) = 0;

end

end

if old\_action ~= action

policy\_stable = false;

end

end

end

% plot the optimal values

myworld = cWorld();

myworld.plot;

myworld.plot\_value(transpose(V\_e));

% plot the optimal policy

policy\_show = zeros(1, 12);

for i = 1:12

[val, idx] = max(policy\_e(:,i));

policy\_show(i) = idx;

end

myworld = cWorld();

myworld.plot;

myworld.plot\_policy(transpose(policy\_show));

% ------------------------------ functions ------------------------------ %

% ------------------------------- part a ------------------------------- %

function prob = p(s\_1, s\_0, a)

global prob\_matrix

prob = prob\_matrix(s\_1,s\_0,a);

end

function reward = r(s)

if s == 10

reward = 1;

elseif s == 11

reward = -1;

else

%reward = -0.02;

reward = -0.04;

end

end