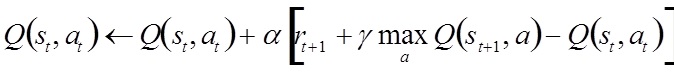
The q-learning algorithm is a useful tool in reinforcement learning. Q-learning gives us a better way of maximizing our reward without getting stuck in one location, or continuously taking the greedy path towards the goal state. Q-learning gives a better idea of the immediate reward from taking a specific action each step of the way when heading towards a goal state. In this particular implementation, our goal is to implement the Q-learning algorithm efficiently so that we can reach the goal state in a small number of steps. We are given a grid, where the agent must go from the top left corner to the bottom right corner in as few steps as possible. Each trip from the initial state to the goal state takes a certain number of steps, and each set of steps taken by the agent is classified as an episode. The episode ends once the goal state is reached. Q learning makes use of various different parameters, just like other types of learning methods. These parameters include the discount factor, learning rate, and rewards. All are found in the Q-learning equation below:



The learning rate will determine how much of an effect does proceeding into another state have on the current state. A higher learning rate will cause the agent to learn quicker, and see which action will provide the optimal Q value. The discount factor determines the importance of future rewards. A smaller discount factor makes the agent short sighted, and focused on more immediate rewards and gains. A larger discount factor puts the focus on future rewards, and how will going forward affect the agent. Rewards are simply what the benefits of moving forward. The magnitudes of these rewards affect the agent’s behavior as well. Since the Q value is to be maximized, rewards will play a big role in that. A higher reward will draw the agent towards it, whereas a lower reward will have the opposite effect. This determines whether an agent will be encouraged to explore, or simply exploit. Exploration is when the agent will venture all over the world, whereas exploitation encourages getting to the goal from the best way possible. Another key factor that plays in Q-learning is whether an agent proceeds greedily by always picking the action with the highest reward, or to occasionally explore and randomly pick an action instead. Implementing this into the program helps to keep the agent from constantly remaining in 1 state, or to constantly run into the walls/ends of the grid.

All the factors listed above are included in my implementation of the Q-learning algorithm. I chose to make the learning rate 0.2, the discount factor 0.9, and have a 10% randomly chosen action to promote a bit more exploration. The rewards are all set to -1 for the inner parts of my grid, but the goal state has a reward of 100, whereas the start state has a reward of -100. The reason I chose the goal to be so high is to draw the agent there somehow, and the initial state has a low reward simply to make sure the agent doesn’t venture back into the initial state. All the inner rewards are set to -1 to drive the agent towards the goal by attempting to maximize the Q-value, making it more positive rather than negative, and the overall reward obtained by the agent. Basically attempting to begin exploitation as opposed to exploration. I did attempt to use positive values, but that ended up in exploration instead, as suspected. The agent would randomly wander around the grid rather than try to get to the goal state. The purpose of this implementation is exploitation, so the agent gets to the goal as soon as possible. At each step, there are only 4 possible actions, which are the typical compass actions of up, down, left, and right. I have encoded them as 0, 1, 2, 3 respectively. The agent can move in any of those direction, but if the agent attempts to run into a wall, it will simply remain in the state it is currently in. Now, the grid is setup so that each entry represents a state. Each state contains its x, y coordinates, a set of Q-values (Q(s, a)) for each action that can be taken from that state, and the reward for being in that state. My implementation works exactly as shown in slide 21 of the temporal difference lecture slides provided. The only catch is that rather than the agent saves the information, the information for all the states are embedded in the grid itself. The code for my implementation is provided at the end of the report.

Observing the learning rate of the agent is of key importance. I have produced 4 learning curves of iterations before reaching the goal state vs number of episodes the agent takes. They are provided below.

Rather than produce 1 curve that shows all the iterations, I chose to produce different curves for different episodes. The first curve for 10 episodes show there is no clear cut trend as to whether the agent is actually learning or not. The amount of exploration (iterations the agent takes) is random and all over the place. Once 100 episodes were added, a trend begins to appear. The amount of exploration the agent does begins to drop steadily. The points are still all over the place, but the overall trend is a decrease in the number of steps the agent takes to reach the goal. Once 600 episodes were used, it is clear that there is a trend. After about 300 episodes, the agent has learned through Q-learning how to get to the goal state in the minimum number of iterations. The amount of iterations to reach the goal remains consistent after about 300. There is one jump in the iterations at around 350 episodes, which could be attributed to the fact that exploration is still encouraged even though exploitation is the goal. In order to ensure that there is an overall downward trend of the number of iterations to reach the goal state, I plotted for 100 episodes as well. Clearly, the trend stays, and after about 300 episodes, the agent consistently reaches the goal state in the same number of steps. The average is about 32 ± 2 steps from episodes 400 to 1000. This overall trend shows that the agent did achieve exploitation as opposed to exploration. The standard deviation of 2 must have occurred from the occasional random exploration that was created 10% of the time. Overall, the agent was able to successfully learn the grid, continually improve the Q-values, and in the end find a successful path from the initial position to the goal state.

***Full Code***

I chose to use Java simply because array manipulation is simple, and I am most comfortable with that language. The code is fully commented, and explains all aspects of the algorithm.

import java.awt.\*;

import java.io.PrintWriter;

import java.util.ArrayList;

/\*\*

\* Created by abhi on 11/5/2014.

\*/

public class QLearn {

public static State[][] world;

public static int size = 15;

private static double discount = 0.9;//discount for traveling through the grid

private static double learningRate= 0.2;//the learning rate

private static int episodes = 600;//total number of episodes

private static ArrayList<Point> data = new ArrayList<Point>();

public static void main(String args[]){

initializeTable();

//implement the q-learning algorithm

for(int i = 0; i < episodes; i++){

Agent agent = new Agent(0, 0);

int steps = 0;

//iterate until the agent is in the goal state

while((agent.getRow() != size-1) || (agent.getCol() != size-1)){

//compute the q value, and set it

State s = world[agent.getRow()][agent.getCol()];

//choose an action

int action = chooseAction(s, steps);

//System.out.println(action);

double currState\_qVal = s.getQ(action);

//make a copy of the agent so that we can use that to get the new state values we need

Agent copyAgent = new Agent(agent);

//move the copy agent first and get the new values to compute Q

copyAgent.moveAgent(action);

State nextOne = world[copyAgent.getRow()][copyAgent.getCol()];

//get all componenets needed to compute the second term of the q-learning equation

double reward = nextOne.getReward();

double maxQ = (nextOne.getMaxQ())\*(discount);

double secondTerm = (learningRate)\*(reward + maxQ - currState\_qVal);

//the resulting q value

double finalQ = currState\_qVal + secondTerm;

//set the new Q value for the movement

s.setQ(action, finalQ);

//move the agent accordingly

agent.moveAgent(action);

steps++;

}

data.add(new Point(i, steps));//episodes, steps to reach end

}

try {

PrintWriter writer = new PrintWriter("hw7\_data\_600.txt", "UTF-8");

for(int i = 0; i < episodes; i++){

writer.println(data.get(i).getX()+"\t"+data.get(i).getY());

}

writer.close();

} catch (Exception e){e.printStackTrace();}

//printWorld();

}

private static void printWorld(){

for(int i = 0; i < size; i++){

for(int j = 0; j < size; j++){

System.out.println(i+"; "+j+"; "+world[i][j]);

}

}

}

private static int chooseAction(State s, int steps){

//0 = up, 1 = down, 2 = left, 3 = right for the actions

//must choose based on epsilon value

//normally pick the greedy choice

int action = s.getMaxAction();

//want to pick a random choice every 10% of the time

if(steps % 10 == 0) {

action = (int) (Math.random() \* 4);

}

return action;

}

private static void initializeTable(){

double qValue = 2;

world = new State[size][size];

for(int i = 0; i < size; i++){

for(int j = 0; j < size; j++){

//rewards are randomly set for all states except:

//1. the last one (most positive)

//2. the first one (most negative)

int reward = -1;

world[i][j] = new State(i, j, qValue, reward);

}

}

//set greatest reward for the final state

world[size-1][size-1].setReward(100);

//set smallest reward for the initial state

world[0][0].setReward(-100);

}

}

/\*\*

\* Created by abhi on 11/5/2014.

\*/

public class Agent {

private int row = 0;

private int col = 0;

public Agent(int row, int col) {

this.row = row;

this.col = col;

}

public Agent(Agent agent) {

this.row = agent.getRow();

this.col = agent.getCol();

}

public void moveAgent(int action){

//0 = up, 1 = down, 2 = left, 3 = right for the actions

if(action == 0){

this.moveUp();

} else if(action == 1){

this.moveDown();

} else if(action == 2){

this.moveLeft();

} else if(action == 3){

this.moveRight();

}

}

public void moveLeft(){

if(col > 0){

col--;

}

}

public void moveRight(){

if(col < (QLearn.size -1)){

col++;

}

}

public void moveUp(){

if(row > 0){

row--;

}

}

public void moveDown(){

if(row < (QLearn.size - 1)){

row++;

}

}

public int getRow() {

return row;

}

public void setRow(int row) {

this.row = row;

}

public int getCol() {

return col;

}

public void setCol(int col) {

this.col = col;

}

public String toString(){

return row+", "+col;

}

}

/\*\*

\* Created by abhi on 11/5/2014.

\*/

public class State {

private int row;

private int col;

//0 = up, 1 = down, 2 = left, 3 = right for the actions

private double qValues[];

private int reward;

public State(int row, int col, double qValue, int reward) {

this.row = row;

this.col = col;

qValues = new double[4];

for(int i = 0; i < 4; i++){

this.qValues[i] = qValue;

}

this.reward = reward;

}

public double getQ(int action){

return qValues[action];

}

public void setQ(int action, double val){

qValues[action] = val;

}

public double getMaxQ(){

double max = qValues[0];

for(int i = 0; i < 4; i++){

if(qValues[i] > max){

max = qValues[i];

}

}

return max;

}

public int getMaxAction(){

if(this.row == 0 && this.col == 0){

int action;

if(qValues[3] > qValues[1]){

action = 3;

} else {

action = 1;

}

return action;

} else if(this.row == 0 && this.col > 0){

double max = qValues[1];

int action = 1;

for(int i = 1; i < 4; i++){

if(qValues[i] > max){

max = qValues[i];

action = i;

}

}

return action;

} else if(this.col == 0 && this.row > 0){

double max = qValues[0];

int action = 0;

for(int i = 0; i < 4; i++){

if(i == 2){

continue;

}

if(qValues[i] > max){

max = qValues[i];

action = i;

}

}

return action;

} else {

double max = qValues[0];

int action = 0;

for(int i = 0; i < 4; i++){

if(qValues[i] > max){

max = qValues[i];

action = i;

}

}

return action;

}

//System.out.println(action);

}

public int getRow() {

return row;

}

public void setRow(int row) {

this.row = row;

}

public int getCol() {

return col;

}

public void setCol(int col) {

this.col = col;

}

public double[] getqValues() {

return qValues;

}

public void setqValue(double[] qValue) {

this.qValues = qValue;

}

public int getReward() {

return reward;

}

public void setReward(int reward) {

this.reward = reward;

}

public String toString(){

String toRet = "";

for(int i = 0; i < 4; i++){

toRet += qValues[i] +", ";

}

return toRet+ "" +reward;

}

}