Bialystok University of Technology

Faculty of Mechanical Engineering

Subject: Interim work project

MPARS16004

Topic: Reinforcement learning-based mobile robot navigation, using Q-learning.

Name and surname: Magdalena Falkowska
Field of study: Automatic Control and Robotics
Specialization: Mobile Robots
Semester: VI
Index numer: 100027
Academic year: 2019/2020

……………………………………………………………

Student's sign

Educational outcomes verification:	
EK1 ……………..	EK5 ……………..
EK2 ……………..	EK6 ……………..
EK3 ……………..	EK7 ……………..
EK4 ……………..	EK8 ……………..
Lecturer's comments:	
Mark: ………………………	……………………………………………………………… *Lecturer's sign*

Abstract

Reinforcement learning algorithms in robot path planning are commonly used today for the research and upgraded to look for better results. In this project the focus is on Dyna-Q algorithm which is based on Q-learning. The main problem in these methods is high usage of memory, time to solve the maze and the path that is not always the shortest. The algorithm was developed and implemented in Matlab with instructions and possibility to easily change parameters and environments. To investigate what mostly depends on the results some comparisons and tests were made to found out that modeling the environment is something that speeds up the whole process. Parameters of the algorithm can be changed depending on what we want to achieve, but if it goes for solving time, the Agent need to have optimal maximum of the steps per episode and high enough planning steps for calculating the future rewards. Planning steps need to be adequate to size and complexity of the environment. Dyna-Q was also compared do A* algorithm to see the difference between two different approaches in path planning and effects of simulations in Matlab.

Contents

List of Figures:

Chapter 1 – Introduction

1.1. Introduction to intelligent mobile robots and machine learning.

Intelligent mobile robots until now, have been used in fields like space exploration, military industries or medical service. Students in the Faculty of Mechanical Engineering have created some projects based on that: Mars rovers, which participated in world competitions, or Bobot, that helps children in hospitals.

Robots in industrial environments can focus on their complex tasks without necessity to move between areas. Autonomous mobile robots in comparison, need to use high performance algorithms for detecting partially or fully unknown environment, where many cases need to be take into consideration. They cannot cause any harm or damage and need to do their tasks with high accuracy, using only sensors and implemented algorithms. Depending on purpose, the environment can be modeled or not. Mobile robots, which have the ability to make decisions are already used in underwater exploration or unmanned flights.

a) Bobot- Mobile robot for kids in hospitals

b) Mars Rover- Mobile robot for space exploration

Figure 1.1: Student's projects in the Faculty of Mechanical Engineering (Bialystok University of Technology)

Why machine learning?

The ability to navigate is important for autonomous mobile robots, which includes learning. Stationary robots doesn't need to deal with unpredicted situations, because their environment is controlled by humans and can be programmed only to specified tasks. Learning algorithms can provide that the robot will know how to adapt to the environment or acquire new skills.

1.2. Mobile robots path planning algorithms

Today, intelligent robots effectively perform the tasks for which they were set up to automate services. However, adding intelligence to a robot does not solve problems such as finding a path.

The main purpose of path planning is to find the shortest way from A to B, the difference in algorithms depends on variety of environments including the number of obstacles. Path planning is one of the most researched problems in mobile robots, because it has a large impact on the overall robot activity.
To successfully complete a navigation task, the robot must know its position in relation to that of its target. In addition, it must take into account the dangers of the surrounding environment and adapt its activities so as to maximize its chances of getting there. The three fundamental navigation functions are: localization, mapping, and motion planning.

Path planning raises a number of issues, for example: reducing calculation time and improving the smooth trajectory of a virtual robot, moving around autonomous robots in complex environments or moving obstacles and navigation of a multi-agent robot. In case of different types of problems, there are three main aspects that every mobile robot must face: efficiency, accuracy, and safety. The full graph showing the most common problems is presented below:

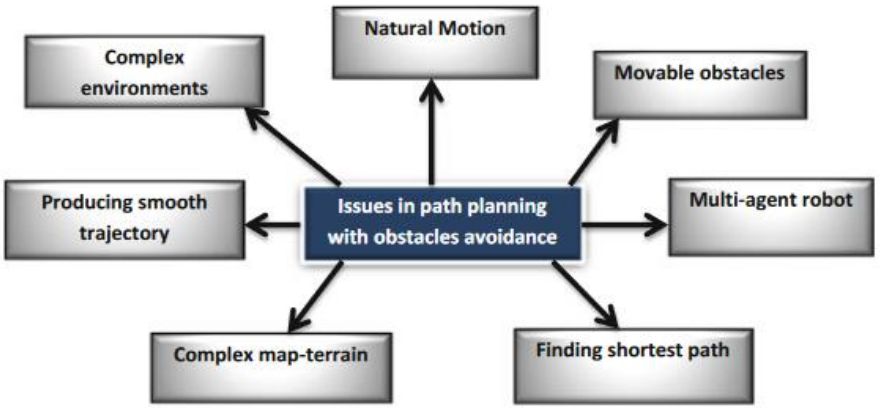


Figure 1.2: Different types of issues in path planning. [2]

Based on relevant state-of-the-art literature, there are three categories of approaches used to solve the path planning problem: classical approaches, graph search approaches, and heuristic approaches, as showed in Figure 1.2. The three of them will be discussed here: Potential field method, A* and Genetic Algorithm.

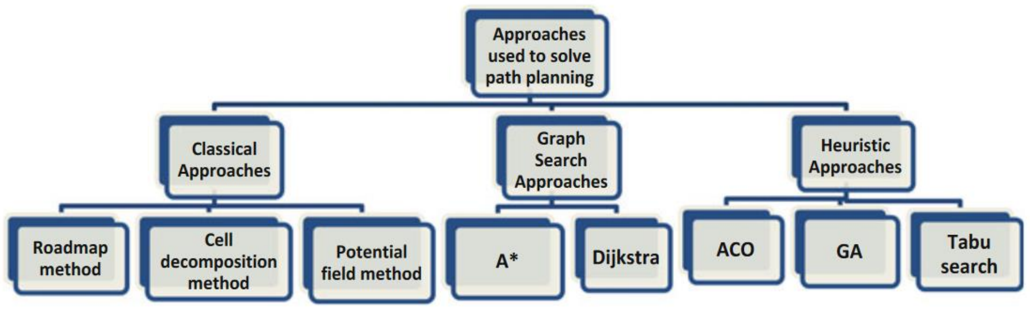


Figure 1.3: Path planning approaches. [2]

Artificial Potential Field Method (PFM)

This method can be described with a field with valleys and hills and a ball. When we make the goal lower than start point and make the obstacles from hills, the ball will fall into desired destination, avoiding obstacles. The same goes with PFM, where obstacles and goal are using repulsive and attractive forces.

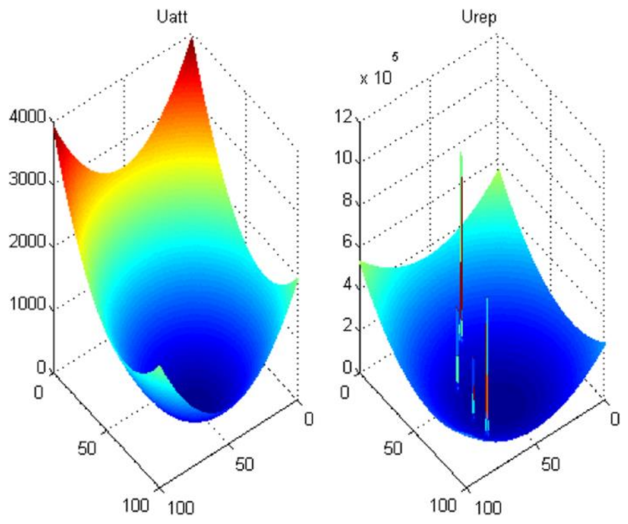


Figure 1.4: Mesh plot of the potential energy of attraction and repulsion in the modified artificial potential field algorithm for circular obstacles. [3]

In PFM, the path is calculated with the gradient of the total artificial potential. However, it is not always a good choice for large and complex environments. There is a possibility that robot can be trapped because of the existence of local minima. Also, classical methods often consume much time to find the goal, especially when the robot must deal with multiple problems.

A* Algorithm

A* algorithm is known as extension of Dijkstra algorithm, which uses graph search method for path planning. It combines the advantages of uniform cost and greedy searches using a fitness function(1.1):

$$F(n) = g(n) + h(n) \tag{1.1}$$

Where:

$g(n)$ - accumulated cost from start node to node n

$h(n)$ – heuristic estimation of the remaining cost to get from node n to the goal nodes

A* uses two lists of nodes, one for nodes to be next considered and second for visited ones. Based on the function, the specific nodes are being expanded and sent to second list, then their neighbors are added to the first list. Whole process is closed when it gets to the final node.

This algorithm is good for finding the shortest path, but also is time consuming and has a trend to work slower with more obstacles.

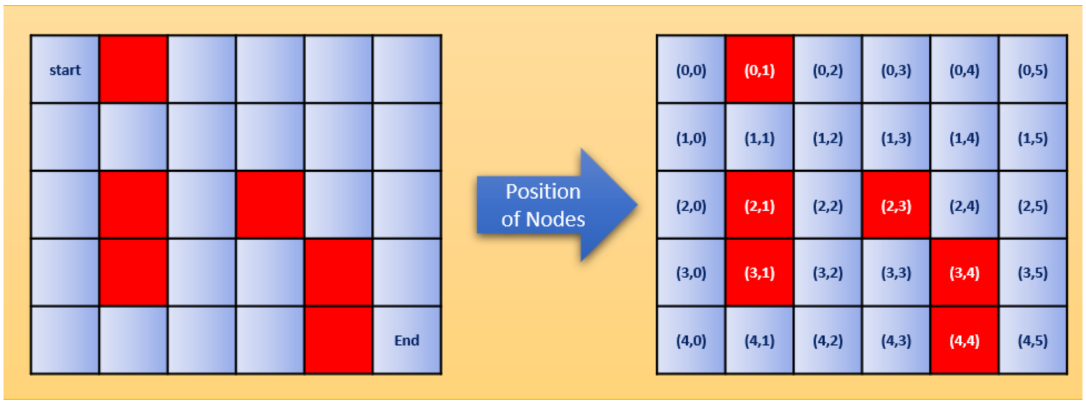


Figure 1.5: Left: Maze problem Right: Position of every node of the Maze. [6]

Genetic Algorithm (GA)

Genetic Algorithm belongs to metaheuristic method algorithms, where words meta and heuristic are from Greek. Meta is translated as beyond and heuristic as find. It uses probabilistic decisions during the search, where random actions are used to find optimal solutions with a proper learning strategy.

GA as its name showing, is based on natural processes. It uses evolution and natural selection as a base to find solutions to the given problems.

It is constructed as a population of solutions, where solutions are called chromosomes. This goes into sequence of processes to do the evolution from one population to another.

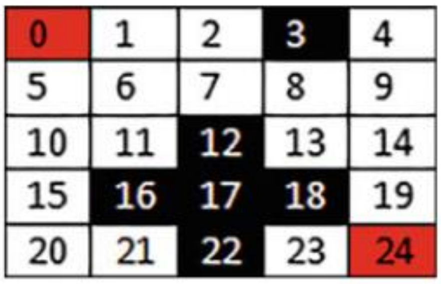


Figure 1.6: Example of GA grid map. [2]

The algorithm in each iteration generates the cost of each path and then do crossover operations.

Chapter 2 – Reinforcement Learning

2.1. Introduction

Reinforcement Learning is a branch of Machine Learning, there are three main sections: Unsupervised learning, Supervised learning and Reinforcement learning. Algorithms are used to solve different types of problems that help to make more natural decisions and increase performance based on results. The learning part means that the algorithm can obtain new information in order to use it. The main idea of this branch is to operate the way the human brain works.

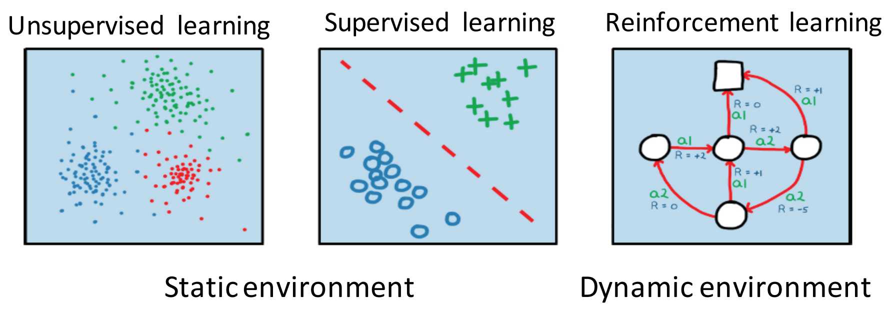


Figure 2.1.1. Machine Learning classification.[9]

Unsupervised learning uses unlabeled data instead of Supervised learning to get results. Reinforcement learning is a different part that can work with dynamic environment instead of other two.

Reinforcement Learning is a method that solves Markov decision processes, that is a random process where the future is independent of the past, given the present. Mathematically it is: $s' = s'(s, a, r)$, where s' is the future state, s is its preceding state and a and r are the action and reward.

Most RL algorithms are based on the model presented in Figure 2.1.2.

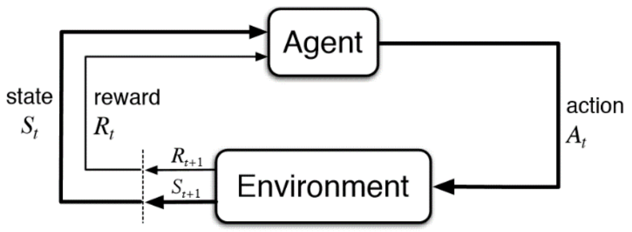


Figure 2.1.2. Reinforcement Learning model. [8]

Environment gives a state, then the Agent is responding to it. Environment also gives a reward based on Agents action and sends the next state. This loop is repeated until environment will send terminate state.

<u>Basic definitions:</u>

- ***ε*-Greedy Policy**: The Agent will do the action that will yield the highest expected reward. To allow also exploration, parameter epsilon within range (0-1) is compared to random number: if random number is higher, then the greedy action is selected, if lower - a random action.
- **Exploitation** - Learning from experience.
- **Exploration** - Random actions.

- **Episode** – Is like one game, the Agent learns from start to the goal receiving rewards.
- **Discount Factor (γ)** – If it is higher, then the Agent will focus on future rewards, if it is not, only on immediate ones.

What is Q-learning?

Q-learning is model-free, off-policy RL algorithm, which is based on Bellman Equation. In classical Q-learning, all possible states of an agent and its possible action in a given state are deterministically known. Q-Function can be presented as below:

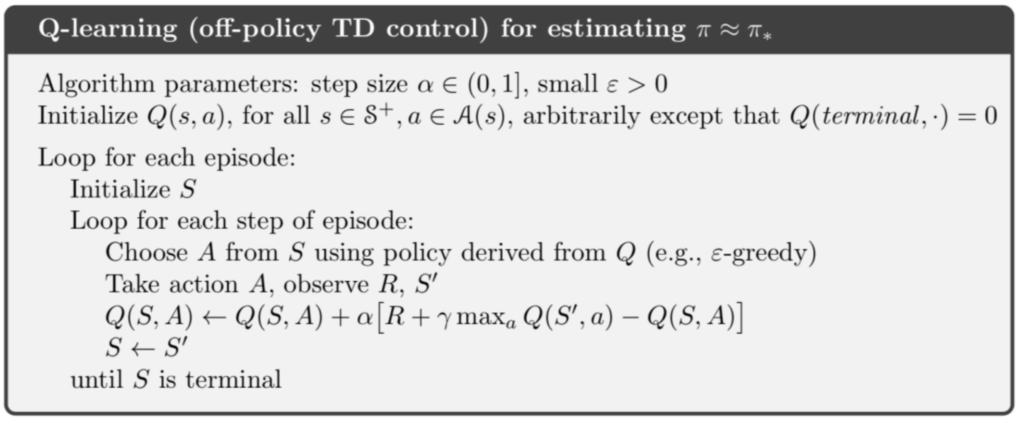
$$new\ Q(s,a) = Q(s,a) + \alpha[\,R(s,a) + \gamma * maxQ'(s',a') - Q(s,a)] \quad (1.2)$$

Where:

$R(s,a)$	- reward for taking action a, from state s
$maxQ'(s',a')$	- maximum expected value from state s’
γ	- discount future rewards
$Q(s,a)$	- previous estimate of value
α	- learning rate

From new best estimated value ($[\,R(s,a) + \gamma * maxQ'(s',a')$) the previous estimation is subtracted. That part is multiplied by learning rate and added to the previous estimation.

Q-learning takes action A on state S, note the reward and the next state S’, then chooses the max Q-Value in state S’. It uses all these info to update Q(S, A) and move it to S’ to execute epsilon greedy action which does not necessarily result in taking action that has the max Q-Value in state S’. The pseudo code for Q-learning is presented in Figure 2.1.3.



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

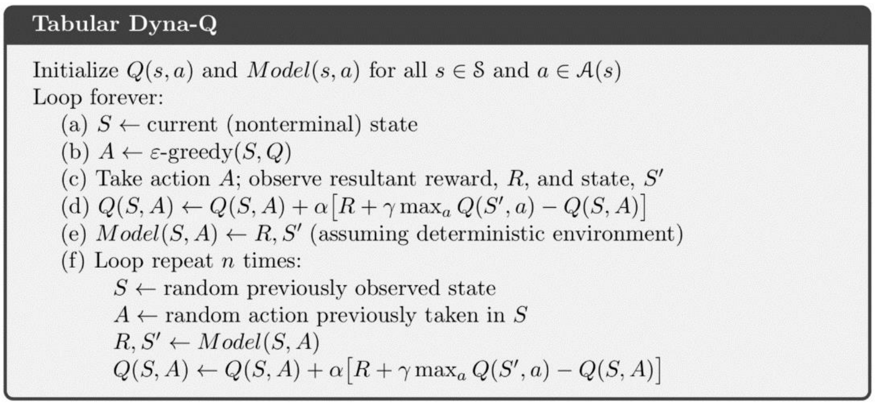
```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 2.1.3. Q-learning pseudo code.[10]

Dyna-Q algorithm

Dyna-Q instead of Q-learning is modeling the environment with the mapping function – from (state, action) to (nextstate, reward). Basically it means that the Agent uses its experience to model environment and then transfer it to the function which will generate future reward.

This algorithm is based on Q-learning and it is only an extended version of it:



Tabular Dyna-Q

```
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon$-greedy$(S,Q)$
  (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) Loop repeat $n$ times:
        $S \leftarrow$ random previously observed state
        $A \leftarrow$ random action previously taken in $S$
        $R, S' \leftarrow Model(S,A)$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
```

Figure 2.1.4. Dyna-Q pseudo code.[12]

Predicting future rewards can speed up the process of policy learning and this will be checked in simulation. Dyna-Q uses real-time experience and future reward with a possibility to change proportion between them with gamma parameter. This is what shows the Figure 2.1.5 below:

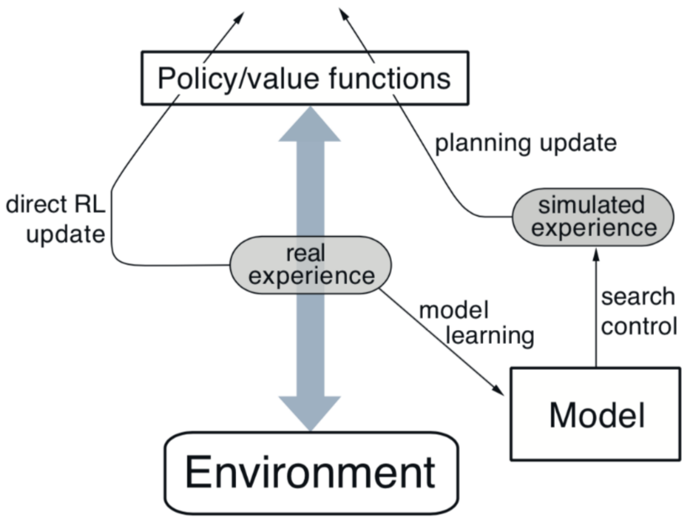


Figure 2.1.5. Dyna-Q architecture.[12]

2.2. Reinforcement Learning methods in path planning: Literature review

1) Knowledge based reinforcement learning robot in maze environment. [13]

In this article we can read about the example of solving the maze by a Line Follower Robot, using Reinforcement Learning algorithm and LSR rule (states Left direction has highest priority compared to straight and right).

LSR is used in a teaching phase, then RL algorithm in a replay phase to find optimal path based on experience from LSR, it means where are the dead ends for example. This method occurred to have some disadvantages like high usage of memory, so because of that robot couldn't solve the maze multiple time in certain moment.

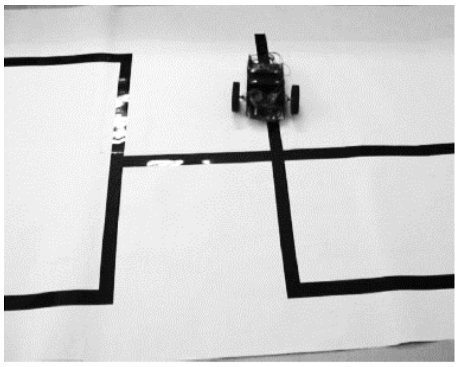


Figure 2.2.1: Robot in a maze based on black line.[13]

2) Comparative Analysis of Reinforcement Learning Methods for Optimal Solution of Maze Problems. [14]

In this article, the comparison of three RL methods is made: discrete Q-Learning, Dyna-CA, FRIQ-Learning (Fuzzy Rule Interpolation-based Q-Learning).

In comparison, Dyna-CA-Learning and FRIQ-Learning are faster than discrete Q-Learning. We learn that the difference in convergence time in discrete Q-Learning results from large state space which leads large state-action value function. If it goes for the maze problem, the fastest solution will be the FRIQ-Learning because it has smaller Q-function representation.

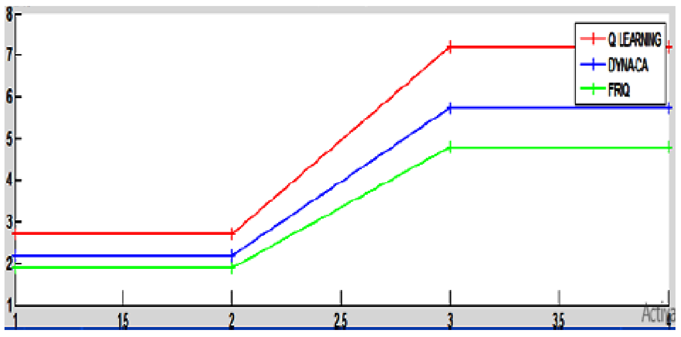
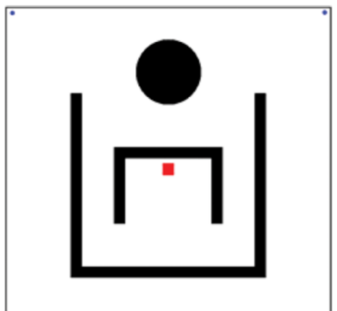


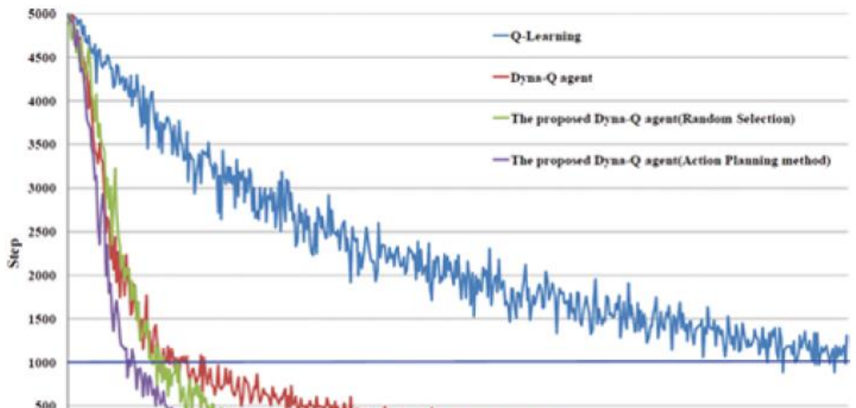
Figure 2.2.2: Convergence Time Comparison without Obstacle Delay.
(Red: Q-leaning, Blue: Dyna-Q, Green: FRIQ)[14]

3) Adaptive Model Learning Based on Dyna-Q Learning. [15]

This article shows how to combine Dyna-Q Learning with a tree model. In this adaptive model learning method, the experience from sensory input vector is used to model the environment.

To show the result it was compared to original Q-Learning method and to Dyna-Q agent. The conclusion is about that models in RL allow to faster learning than model-free algorithms.




Q-Learning
Dyna-Q agent
The proposed Dyna-Q agent(Random Selection)
The proposed Dyna-Q agent(Action Planning method)
Step
5000
4500
4000
3500
3000
2500
2000
1500
1000
500

b) Maze for simulation

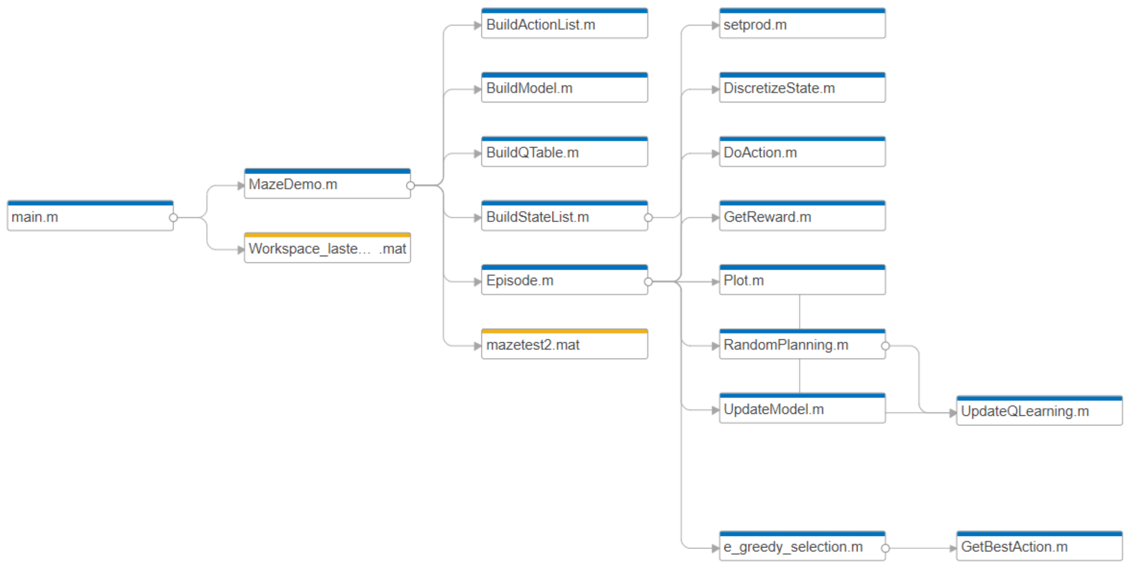
a) Results, proposed model is most effective: violet/green, Q-learning: blue, standard Dyna-Q: red

Figure 2.2.3: Simulation of the proposed method and comparison. [15]

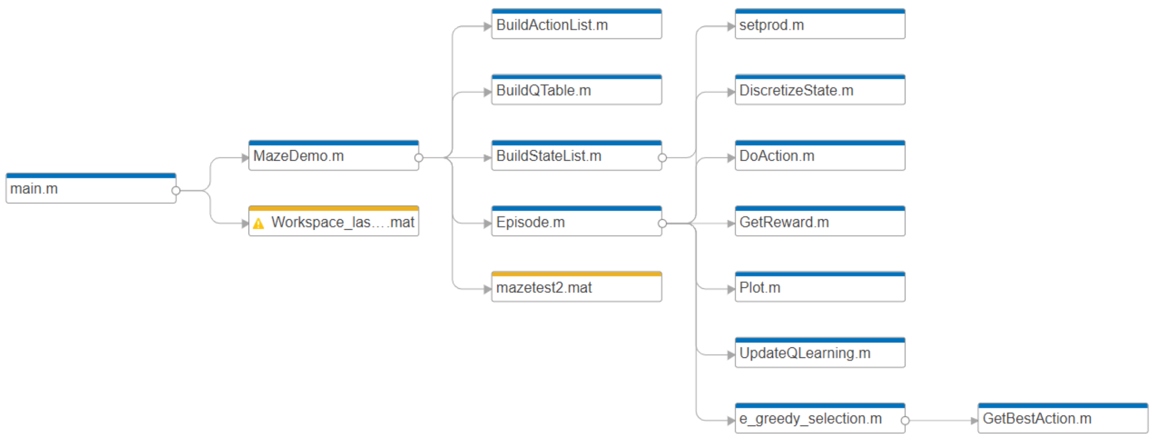
2.3. Implementation in Matlab

Structure of the project

For better understanding of how the code is constructed, built in function called Dependency Analyzer is used (Fig.2.3.1.). This gives a better view on the algorithm and helps to understand it, based on knowledge about Dyna-Q and Q-learning.



a) Dyna-Q structure



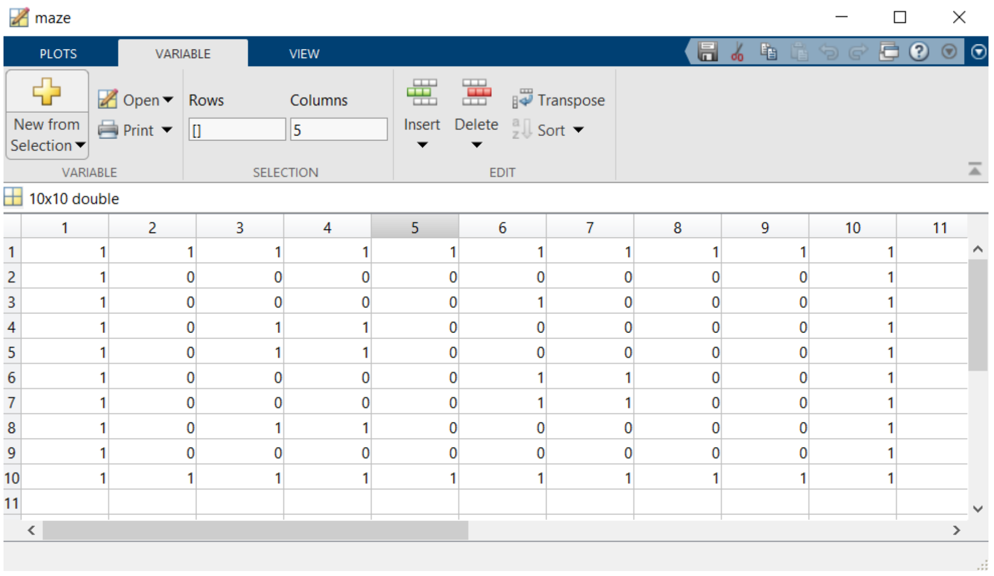
b) Q-learning structure

Figure 2.3.1: Method to visualize the connections between functions and files in the project.

The project also prepared for Q-learning without the functions containing the Model, but it can also be done by reducing planning steps in Dyna-Q to 0.

Maze implementation

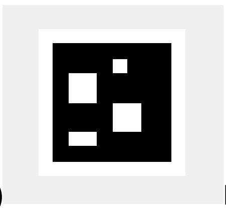
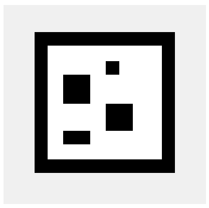
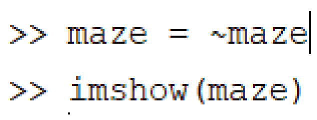
The simplest method to implement the maze into presented algorithm is to create an array with zeros and ones as in the example below, where “1” is an obstacle and “0” means movable area.



	1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	
2	1	0	0	0	0	0	0	0	0	1	
3	1	0	0	0	0	1	0	0	0	1	
4	1	0	1	1	0	0	0	0	0	1	
5	1	0	1	1	0	0	0	0	0	1	
6	1	0	0	0	0	1	1	0	0	1	
7	1	0	0	0	0	1	1	0	0	1	
8	1	0	1	1	0	0	0	0	0	1	
9	1	0	0	0	0	0	0	0	0	1	
10	1	1	1	1	1	1	1	1	1	1	
11											

Figure 2.3.2: Simple maze array.

Every array can be visualized as a picture with an imshow built-in function. In images “1” goes for white and “0” for black, so it’s reverted. With this knowledge every black and white picture can be implemented as a maze and colors can be reverted by using a negation of an array: maze= ~maze. This example was saved as mazetest2.m and is being loaded in the code.

a) 
b) 
c) 

```
>> maze = ~maze
>> imshow(maze)
```

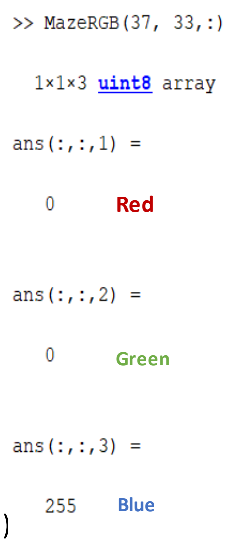
Figure 2.3.3: Example of the maze a) maze based on array from Figure 2.3.2 b) reverted maze c) functions that were used

Following this lead, every picture can be used as a maze, but it needs to meet some cryteria.

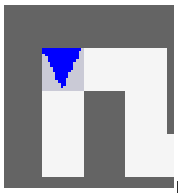
Every pixel in RGB image consists of three values, but for the algorithm, the maze needs to have discrete values 0/1 on each pixel.

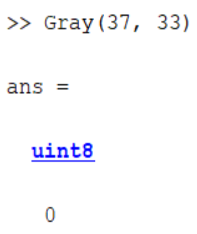
<u>Checking pixel value of RGB image</u>

Each image is a matrix so to check RGB or Grayscale values it is needed to know array parameters that each pixel is described, presented below.

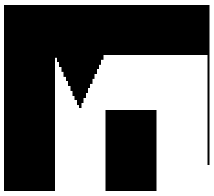


```
>> MazeRGB(37, 33,:)

  1×1×3 uint8 array

ans(:,:,1) =

    0       Red


ans(:,:,2) =

    0       Green


ans(:,:,3) =

  255       Blue
```

a) 

b) 

```
>> Gray(37, 33)

ans =

  uint8

   0
```



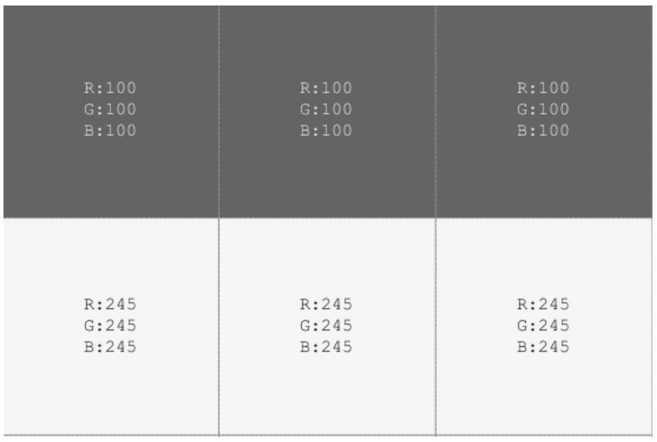
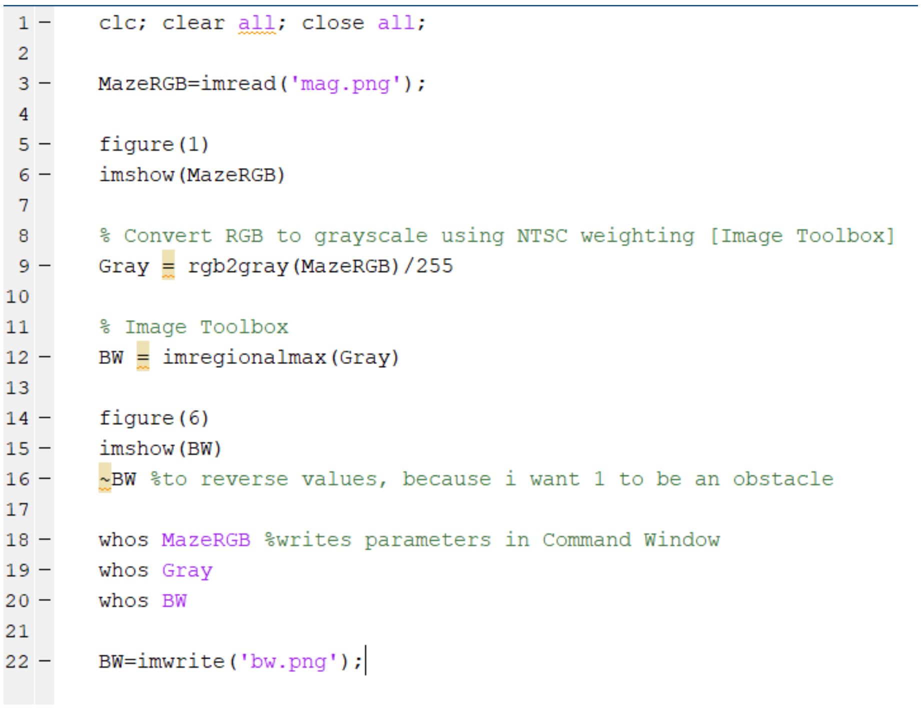
c) 

Figure 2.3.4: Pixel Value a) RGB cell b) Grayscale cell c) Zoom taken on RGB with ImTool function.

The code for preparing any image is based on matlab built-in functions from Image Processing Toolbox that are: rgb2gray, imregionalmax (to get discrete values based on grayscale that has on each pixel values from 0 to 255).



```
clc; clear all; close all;

MazeRGB=imread('mag.png');

figure(1)
imshow(MazeRGB)

% Convert RGB to grayscale using NTSC weighting [Image Toolbox]
Gray = rgb2gray(MazeRGB)/255

% Image Toolbox
BW = imregionalmax(Gray)

figure(6)
imshow(BW)
~BW %to reverse values, because i want 1 to be an obstacle

whos MazeRGB %writes parameters in Command Window
whos Gray
whos BW

BW=imwrite('bw.png');
```

Figure 2.3.5: RGB to black and white image.

If the image has more pixels, the bigger his array will be. That could slow down the entire algorithm. This maze from Figure 2.3.6 was converted from 713x713 to size 31x31 array size and it was done with function presented in Figure 2.3.7b, using the fact that this maze is constructed from squares. Every black/white square size 23x23 pixels can be wrote as one cell in array and so on.

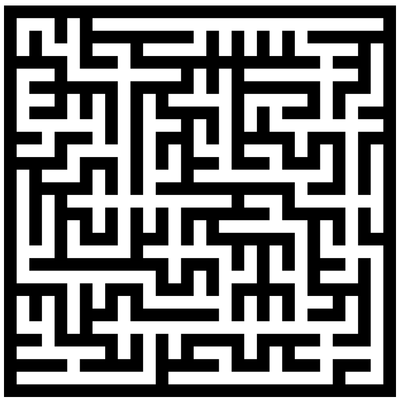
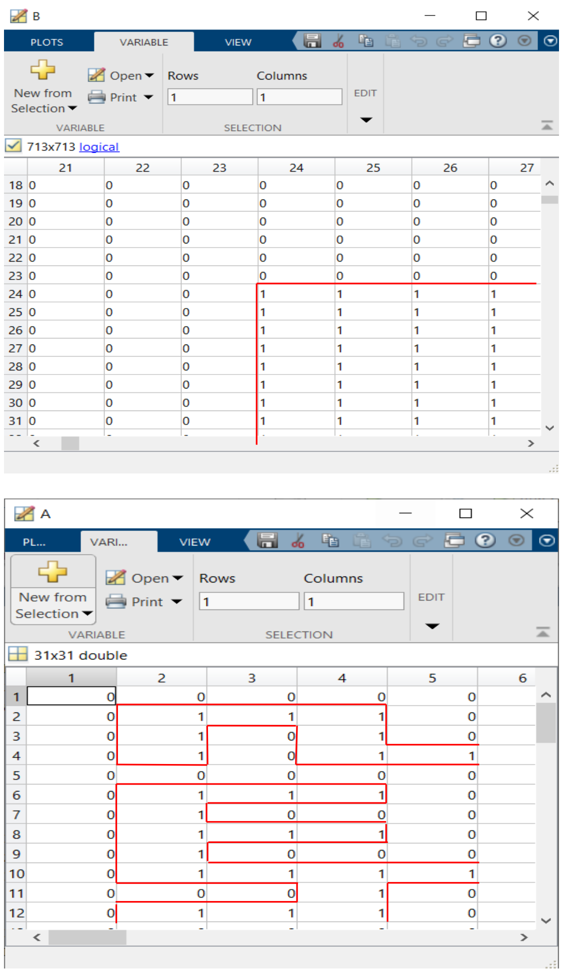


Figure 2.3.6: The maze that was converted.

The Figure below helps to visualize this process:



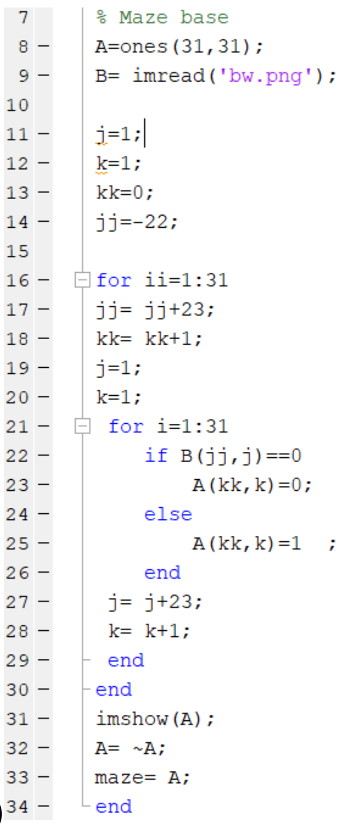
713x713 logical

	21	22	23	24	25	26	27
18	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0
24	0	0	0	1	1	1	1
25	0	0	0	1	1	1	1
26	0	0	0	1	1	1	1
27	0	0	0	1	1	1	1
28	0	0	0	1	1	1	1
29	0	0	0	1	1	1	1
30	0	0	0	1	1	1	1
31	0	0	0	1	1	1	1

31x31 double

	1	2	3	4	5	6
1	0	0	0	0	0	
2	0	1	1	1	0	
3	0	1	0	1	0	
4	0	1	0	1	1	
5	0	0	0	0	0	
6	0	1	1	1	0	
7	0	1	0	0	0	
8	0	1	1	1	0	
9	0	1	0	0	0	
10	0	1	1	1	1	
11	0	0	0	1	0	
12	0	1	1	1	0	

a)



```
% Maze base
A=ones(31,31);
B= imread('bw.png');

j=1;
k=1;
kk=0;
jj=-22;

for ii=1:31
jj= jj+23;
kk= kk+1;
j=1;
k=1;
 for i=1:31
    if B(jj,j)==0
        A(kk,k)=0;
    else
        A(kk,k)=1   ;
    end
 j= j+23;
 k= k+1;
 end
end
imshow(A);
A= ~A;
maze= A;
end
```

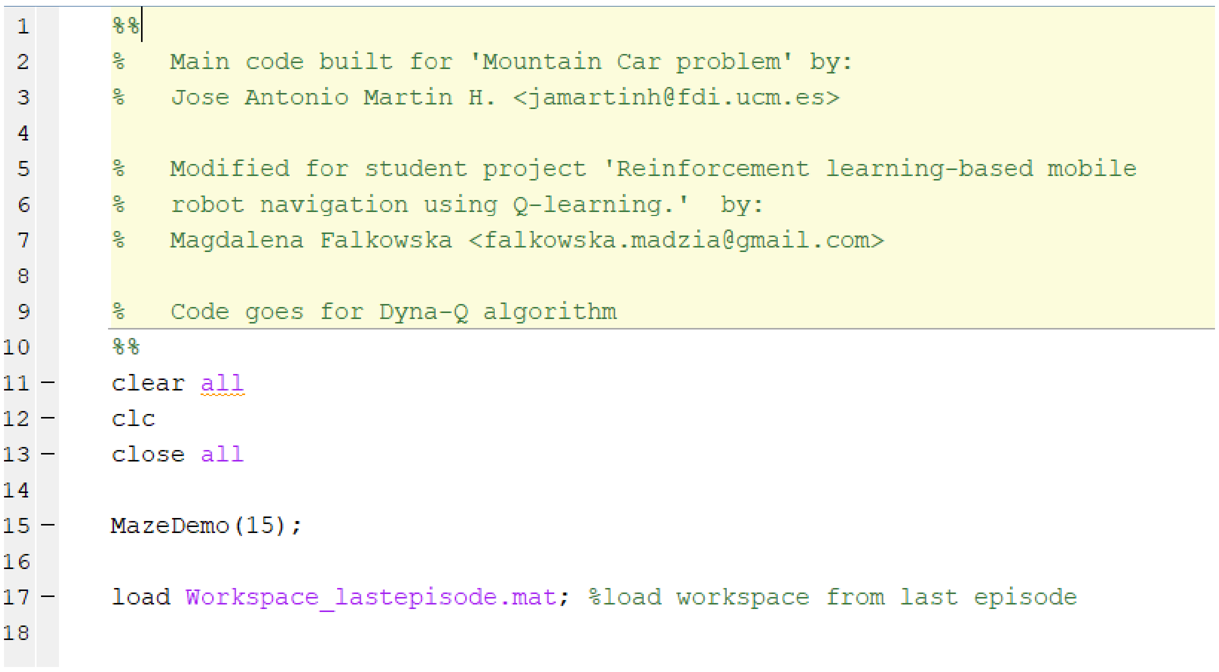
b)

Figure 2.3.7: Maze conversion to smaller array a) A- bigger maze variable B- smaller b) the code prepared for conversion.

To use any maze in the Matlab code, it is needed to include this function in the project or save it as a variable to matlab file. In the main code, all of the mazes were implemented with this method.

The Dyna-Q code

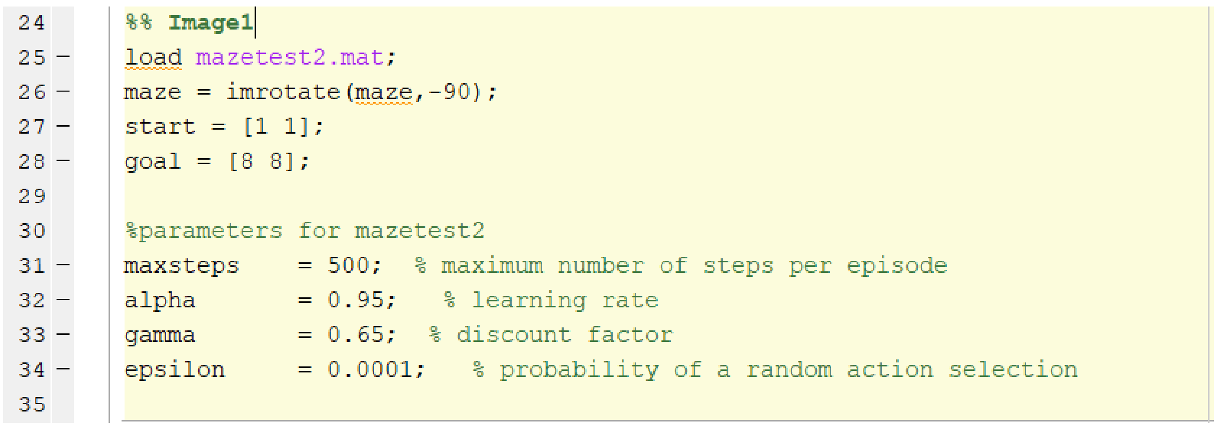
Main.m function is for clearing the memory, starting MazeDemo.m which includes whole algorithm and optionally for additional functions that we want to do after the learning part, like saving variables from workspace or plotting.



```
%%
%   Main code built for 'Mountain Car problem' by:
%   Jose Antonio Martin H. <jamartinh@fdi.ucm.es>

%   Modified for student project 'Reinforcement learning-based mobile
%   robot navigation using Q-learning.'  by:
%   Magdalena Falkowska <falkowska.madzia@gmail.com>

%   Code goes for Dyna-Q algorithm
%%
clear all
clc
close all

MazeDemo(15);

load Workspace_lastepisode.mat; %load workspace from last episode

```

Figure 2.3.8: Main function

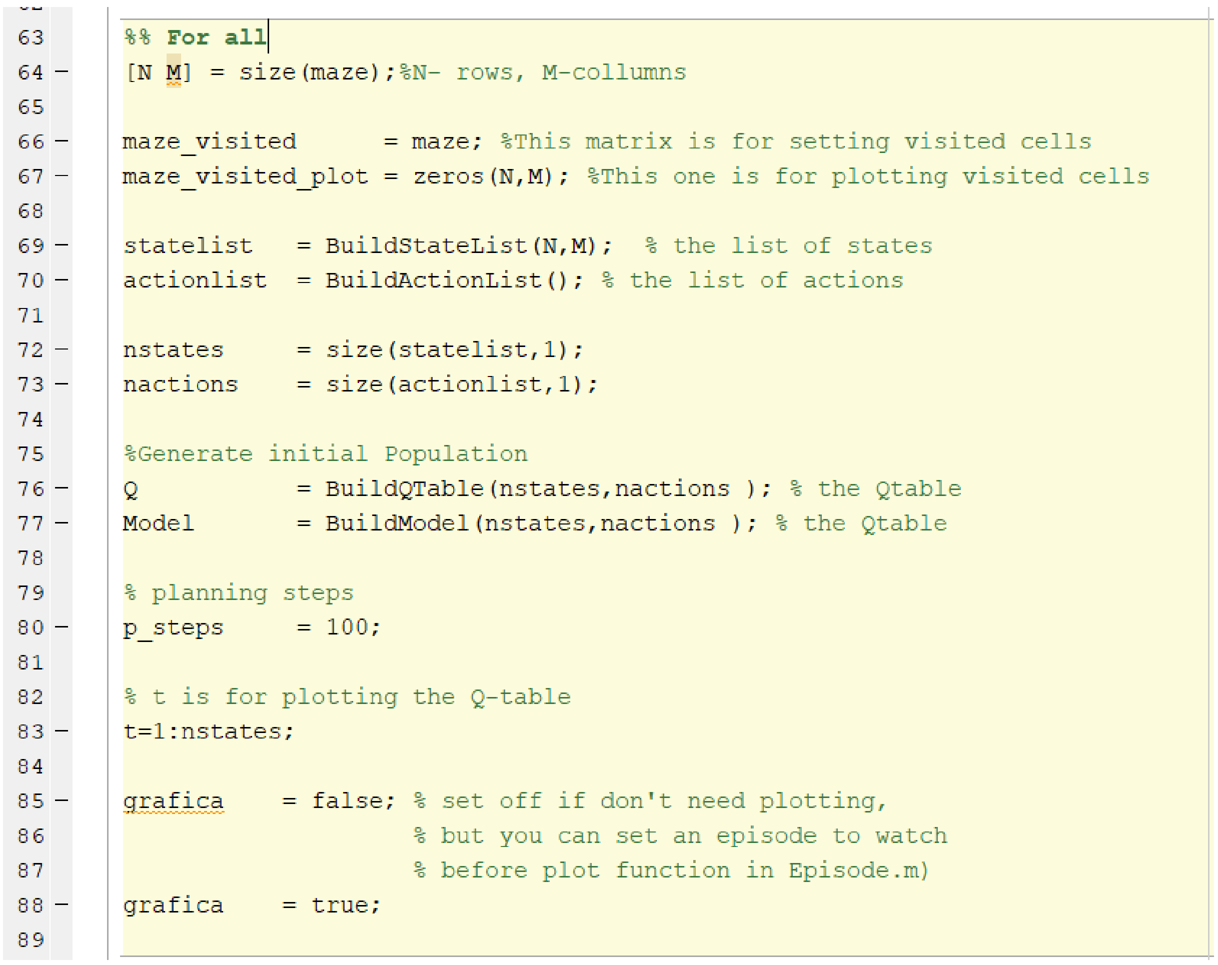
In MazeDemo function we can load our maze and set parameters for it. After that, all needed arrays like states, actions, Q-table and model are built.



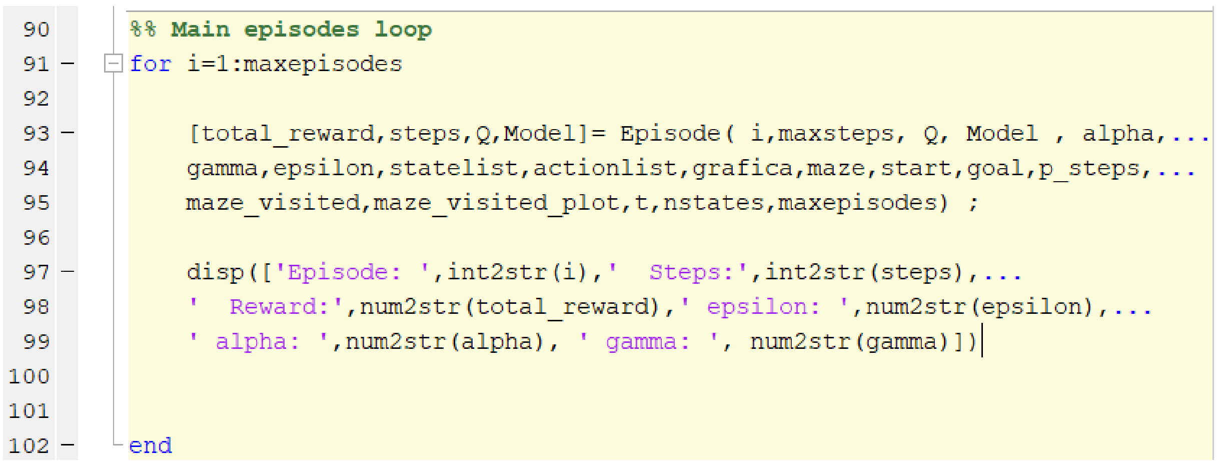
```
%% Image1
load mazetest2.mat;
maze = imrotate(maze,-90);
start = [1 1];
goal = [8 8];

%parameters for mazetest2
maxsteps    = 500;  % maximum number of steps per episode
alpha       = 0.95;   % learning rate
gamma       = 0.65;  % discount factor
epsilon     = 0.0001;   % probability of a random action selection

```

Figure 2.3.9: MazeDemo function (1).

The variables were added to the original code to visualize the path: *maze_visited* and *maze_visited_plot*, and t for plotting the Q-table.



```
%% For all
[N M] = size(maze);%N- rows, M-collumns

maze_visited      = maze; %This matrix is for setting visited cells
maze_visited_plot = zeros(N,M); %This one is for plotting visited cells

statelist   = BuildStateList(N,M);  % the list of states
actionlist  = BuildActionList(); % the list of actions

nstates     = size(statelist,1);
nactions    = size(actionlist,1);

%Generate initial Population
Q           = BuildQTable(nstates,nactions ); % the Qtable
Model       = BuildModel(nstates,nactions ); % the Qtable

% planning steps
p_steps     = 100;

% t is for plotting the Q-table
t=1:nstates;

grafica    = false; % set off if don't need plotting,
                    % but you can set an episode to watch
                    % before plot function in Episode.m)
grafica    = true;

```

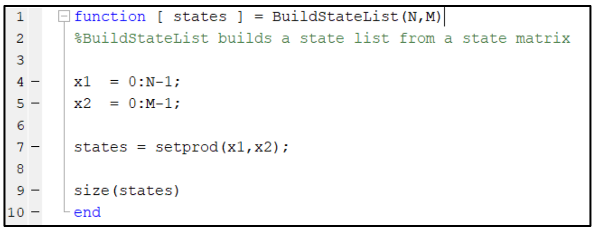


```
%% Main episodes loop
for i=1:maxepisodes

    [total_reward,steps,Q,Model]= Episode( i,maxsteps, Q, Model , alpha,...
    gamma,epsilon,statelist,actionlist,grafica,maze,start,goal,p_steps,...
    maze_visited,maze_visited_plot,t,nstates,maxepisodes) ;

    disp(['Episode: ',int2str(i),'  Steps:',int2str(steps),...
    '  Reward:',num2str(total_reward),' epsilon: ',num2str(epsilon),...
    ' alpha: ',num2str(alpha), ' gamma: ', num2str(gamma)])


end
```

Figure 2.3.10: MazeDemo function (2)

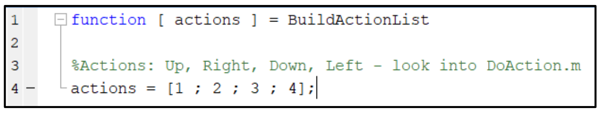
In the Main episodes loop the code will run until the specified number of episodes in main.m function. Every episode will return its number, done steps by the robot, summed given reward and parameters (epsilon, alpha, gamma).

Below functions that run in MazeDemo are presented:



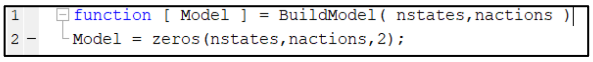
```
function [ states ] = BuildStateList(N,M)
%BuildStateList builds a state list from a state matrix

x1  = 0:N-1;
x2  = 0:M-1;

states = setprod(x1,x2);

size(states)
end
```

b) BuildStateList function



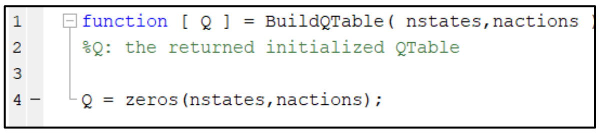
```
function [ actions ] = BuildActionList

%Actions: Up, Right, Down, Left - look into DoAction.m
actions = [1 ; 2 ; 3 ; 4];
```

a) BuildActionList function



```
function [ Model ] = BuildModel( nstates,nactions )
Model = zeros(nstates,nactions,2);
```

d) BuildModel function

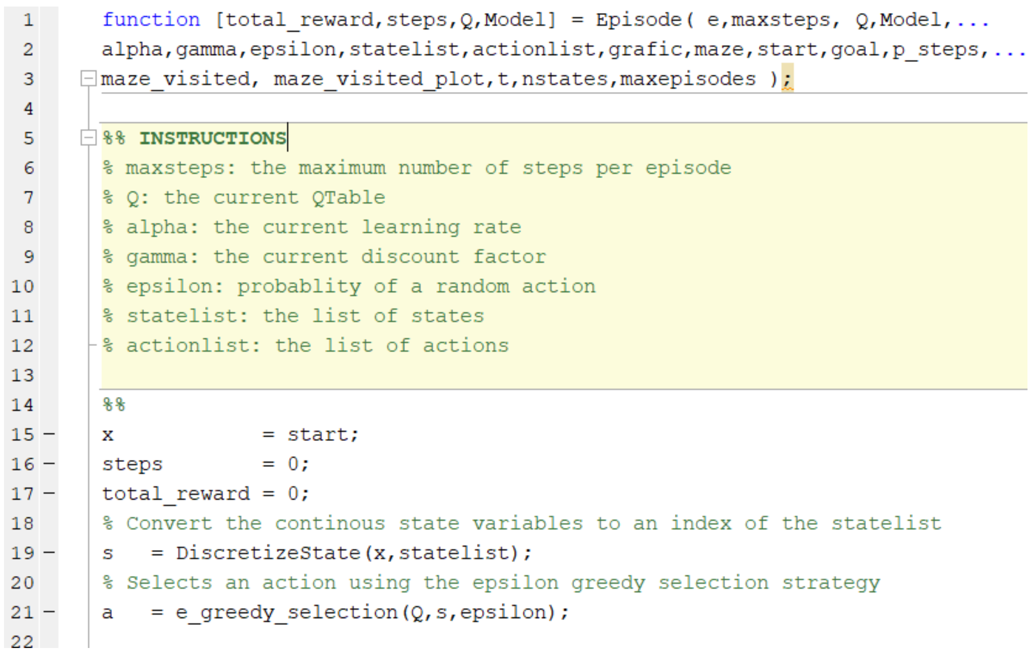


```
function [ Q ] = BuildQTable( nstates,nactions
%Q: the returned initialized QTable

Q = zeros(nstates,nactions);
```

c) BuildQTable function

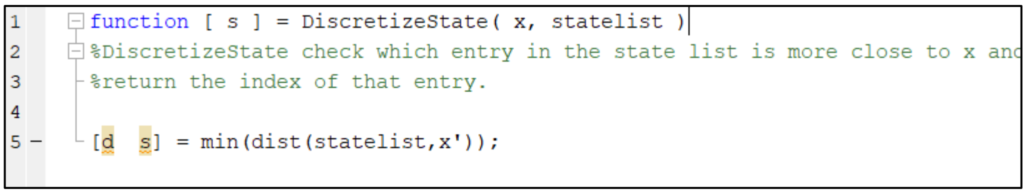
Figure 2.3.11: Functions that prepare arrays in MazeDemo.

States are built using the dimensions of the maze: N, M. Setprod is a function that connects x1 and x2 into one array. Then, the next step is to generate actions from four possible movements – Up, Right, Down and Left. Actions and States are connected in the Q-table and Model arrays to have a place for writing the rewards to each pair state-action.



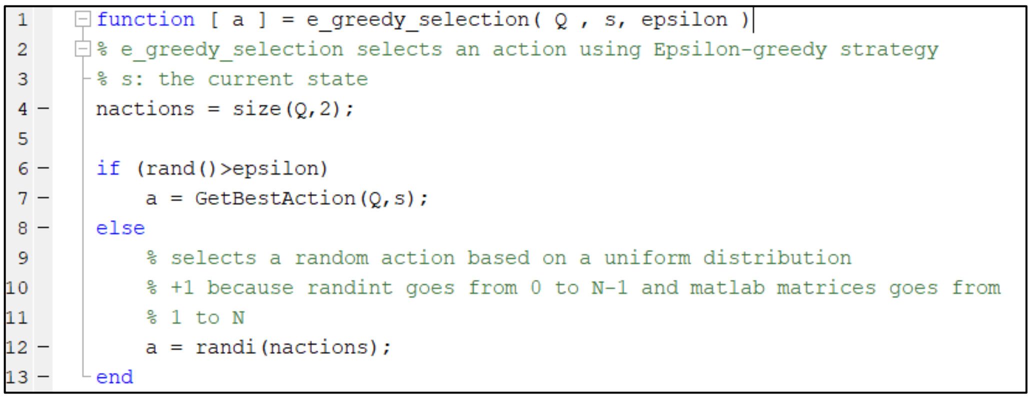
```
function [total_reward,steps,Q,Model] = Episode( e,maxsteps, Q,Model,...
alpha,gamma,epsilon,statelist,actionlist,grafic,maze,start,goal,p_steps,...
maze_visited, maze_visited_plot,t,nstates,maxepisodes );

%% INSTRUCTIONS
% maxsteps: the maximum number of steps per episode
% Q: the current QTable
% alpha: the current learning rate
% gamma: the current discount factor
% epsilon: probablity of a random action
% statelist: the list of states
% actionlist: the list of actions

%%
x            = start;
steps        = 0;
total_reward = 0;
% Convert the continous state variables to an index of the statelist
s    = DiscretizeState(x,statelist);
% Selects an action using the epsilon greedy selection strategy
a    = e_greedy_selection(Q,s,epsilon);
```

Figure 2.3.12: Episode function (part 1).



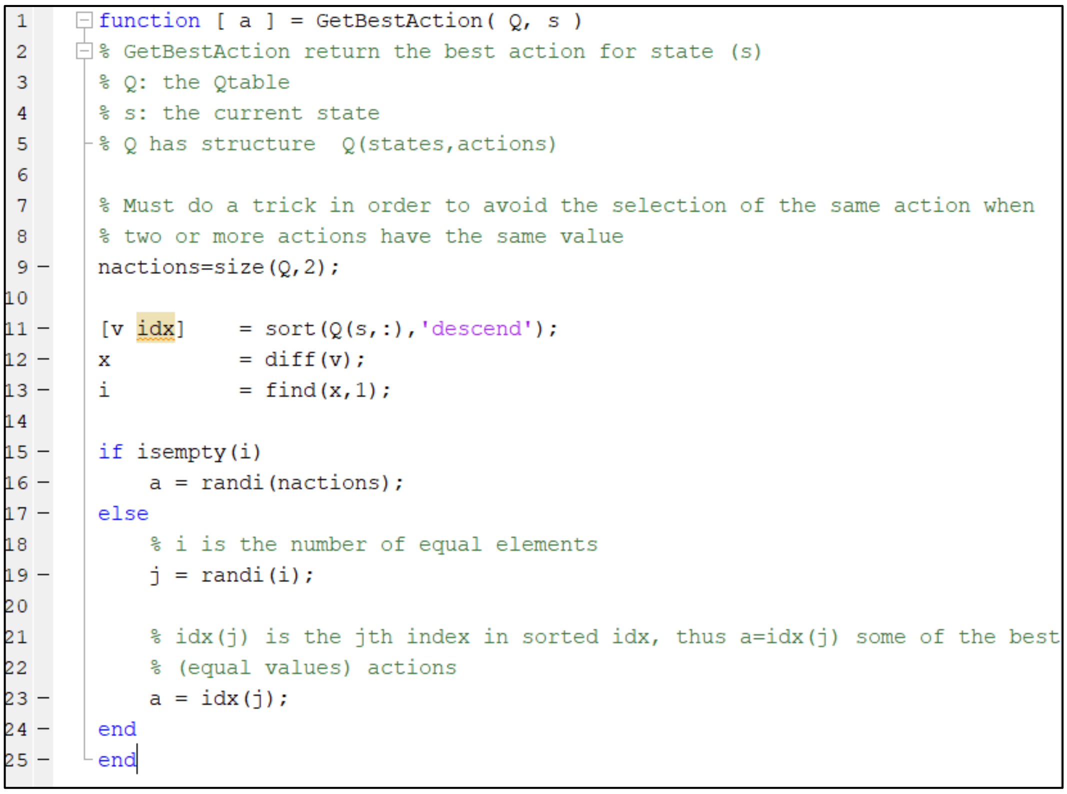
```
function [ s ] = DiscretizeState( x, statelist )
%DiscretizeState check which entry in the state list is more close to x an
%return the index of that entry.

[d  s] = min(dist(statelist,x'));
```

a) DiscretizeState function



```
1    function [ a ] = e_greedy_selection( Q , s, epsilon )
2    % e_greedy_selection selects an action using Epsilon-greedy strategy
3    % s: the current state
4    nactions = size(Q,2);
5
6    if (rand()>epsilon)
7        a = GetBestAction(Q,s);
8    else
9        % selects a random action based on a uniform distribution
10       % +1 because randint goes from 0 to N-1 and matlab matrices goes from
11       % 1 to N
12       a = randi(nactions);
13   end
```

b) e_greedy_selecion function



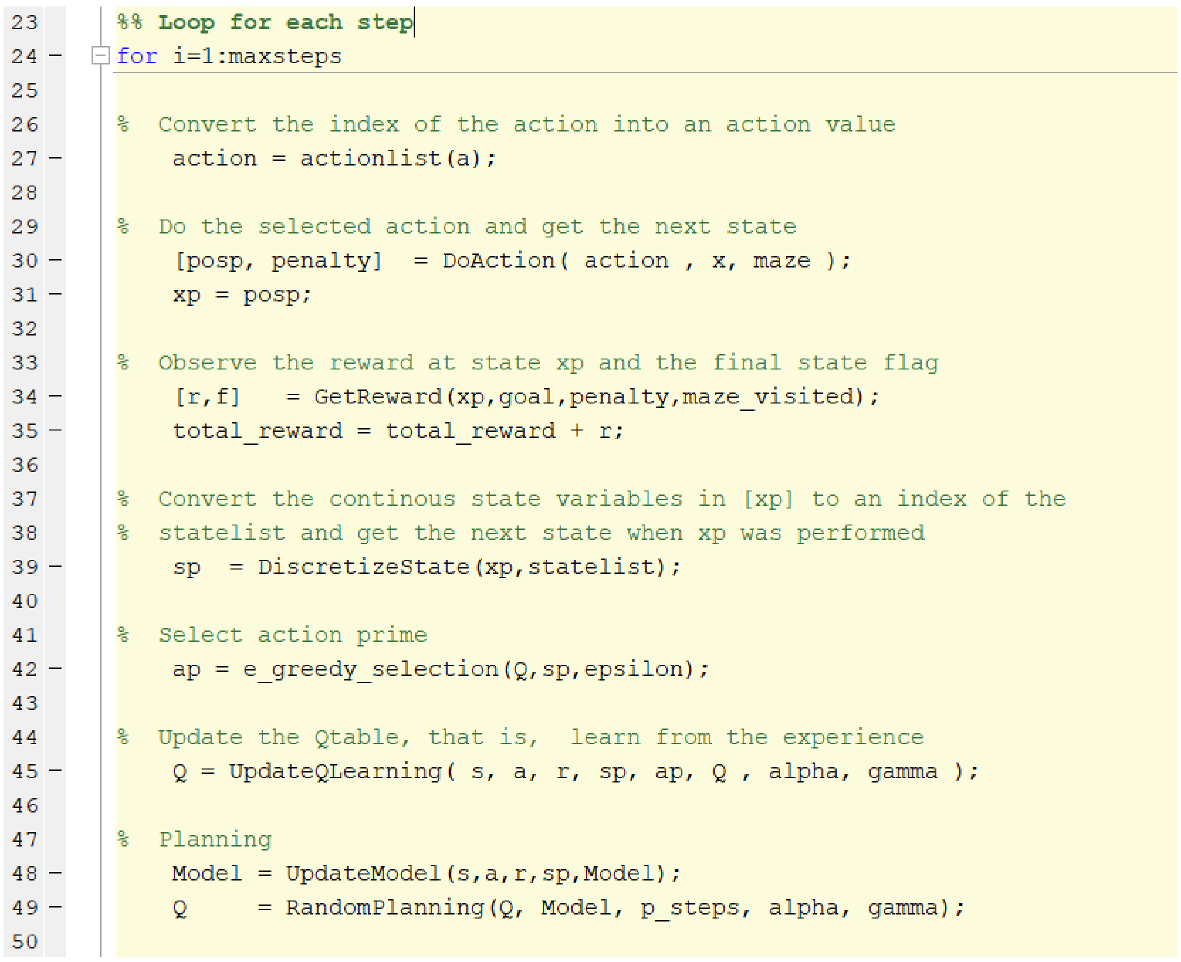
```
function [ a ] = GetBestAction( Q, s )
% GetBestAction return the best action for state (s)
% Q: the Qtable
% s: the current state
% Q has structure  Q(states,actions)

% Must do a trick in order to avoid the selection of the same action when
% two or more actions have the same value
nactions=size(Q,2);

[v idx]     = sort(Q(s,:),'descend');
x           = diff(v);
i           = find(x,1);

if isempty(i)
    a = randi(nactions);
else
    % i is the number of equal elements
    j = randi(i);

    % idx(j) is the jth index in sorted idx, thus a=idx(j) some of the best
    % (equal values) actions
    a = idx(j);
end
end
```

c) GetBestAction function

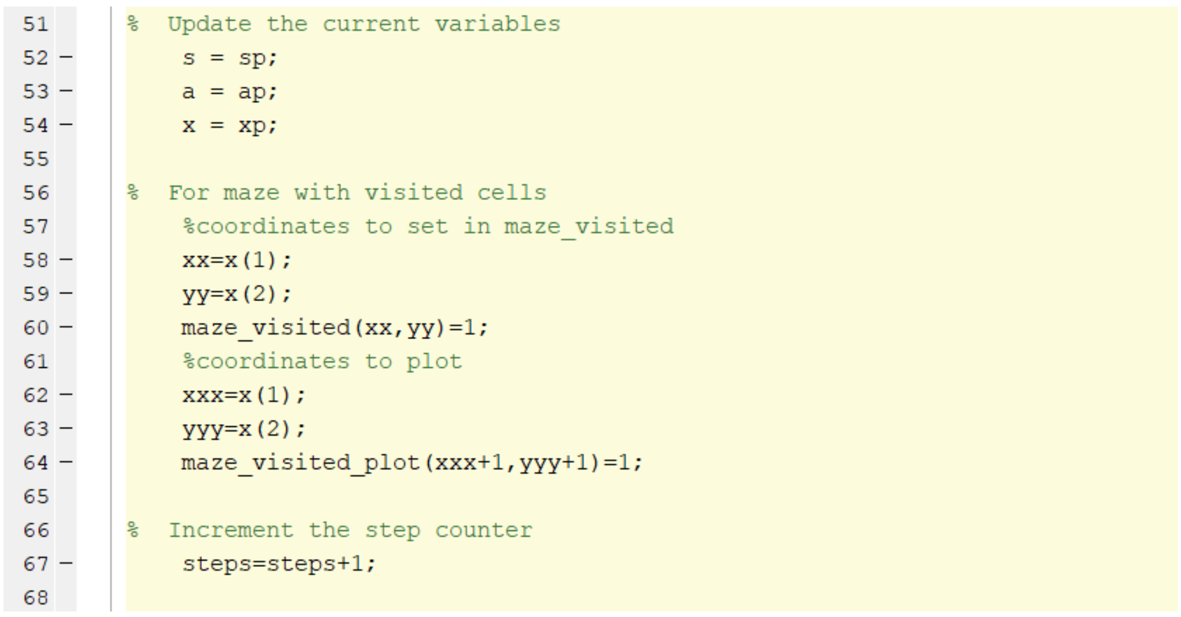
Figure 2.3.13: Functions that run in Episode function.

Here is the part that describes meaning of epsilon variable, which is responsible for exploration. If its value is higher, then the robot will move more randomly and less depend on experience (exploitation). (Line 6-7 in Figure 2.3.13b)

In each episode the robot will try to find its goal and it is done in limits of maxsteps:



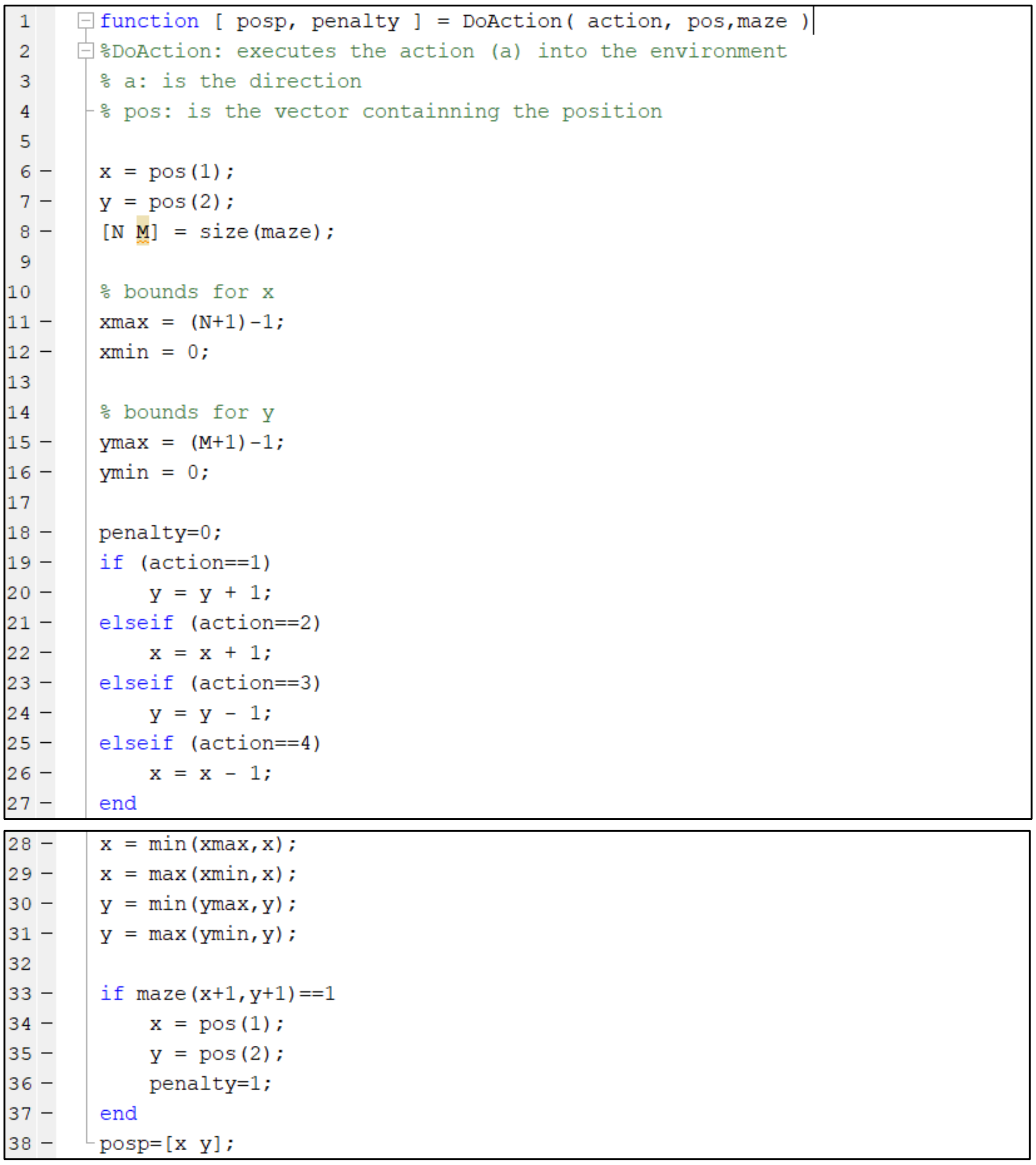
```
%% Loop for each step
for i=1:maxsteps

%  Convert the index of the action into an action value
    action = actionlist(a);

%  Do the selected action and get the next state
    [posp, penalty]  = DoAction( action , x, maze );
    xp = posp;

%  Observe the reward at state xp and the final state flag
    [r,f]    = GetReward(xp,goal,penalty,maze_visited);
    total_reward = total_reward + r;

%  Convert the continous state variables in [xp] to an index of the
%  statelist and get the next state when xp was performed
    sp   = DiscretizeState(xp,statelist);

%  Select action prime
    ap = e_greedy_selection(Q,sp,epsilon);

%  Update the Qtable, that is,  learn from the experience
    Q = UpdateQLearning( s, a, r, sp, ap, Q , alpha, gamma );

%  Planning
    Model  = UpdateModel(s,a,r,sp,Model);
    Q      = RandomPlanning(Q, Model, p_steps, alpha, gamma);

```



```
%  Update the current variables
    s = sp;
    a = ap;
    x = xp;

%  For maze with visited cells
    %coordinates to set in maze_visited
    xx=x(1);
    yy=x(2);
    maze_visited(xx,yy)=1;
    %coordinates to plot
    xxx=x(1);
    yyy=x(2);
    maze_visited_plot(xxx+1,yyy+1)=1;

%  Increment the step counter
    steps=steps+1;

```

Figure 2.3.14: Steps loop in Episode function (part 2).

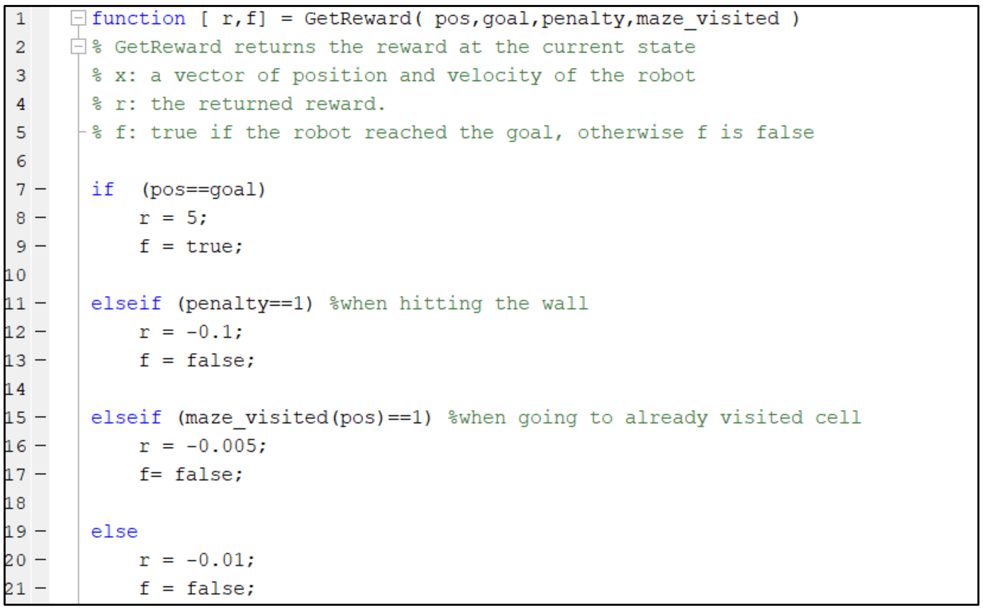
All the functions that take part in this fragment are showed below.



```matlab
function [ posp, penalty ] = DoAction( action, pos,maze )
%DoAction: executes the action (a) into the environment
% a: is the direction
% pos: is the vector containning the position

x = pos(1);
y = pos(2);
[N M] = size(maze);

% bounds for x
xmax = (N+1)-1;
xmin = 0;

% bounds for y
ymax = (M+1)-1;
ymin = 0;

penalty=0;
if (action==1)
    y = y + 1;
elseif (action==2)
    x = x + 1;
elseif (action==3)
    y = y - 1;
elseif (action==4)
    x = x - 1;
end
x = min(xmax,x);
x = max(xmin,x);
y = min(ymax,y);
y = max(ymin,y);

if maze(x+1,y+1)==1
    x = pos(1);
    y = pos(2);
    penalty=1;
end
posp=[x y];
```

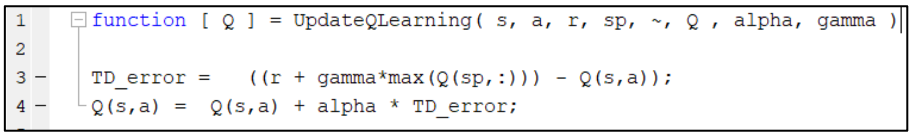
Figure 2.3.15: Functions in steps loop – DoAction function.

Here robot is doing chosen action, if it's the wall it gets penalty=1 and doesn't move. In GetReward function(Figure 2.3.16a) it gets reward with "-" in every move that is not the goal. We can put here the values we want, depending on that what is the priority. I added an optional penalty based on maze_visited and this part also can be omitted.



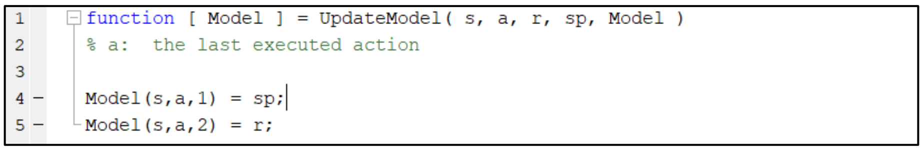
```
function [ r,f] = GetReward( pos,goal,penalty,maze_visited )
% GetReward returns the reward at the current state
% x: a vector of position and velocity of the robot
% r: the returned reward.
% f: true if the robot reached the goal, otherwise f is false

if  (pos==goal)
    r = 5;
    f = true;

elseif (penalty==1) %when hitting the wall
    r = -0.1;
    f = false;

elseif (maze_visited(pos)==1) %when going to already visited cell
    r = -0.005;
    f= false;

else
    r = -0.01;
    f = false;
```

a) GetReward function



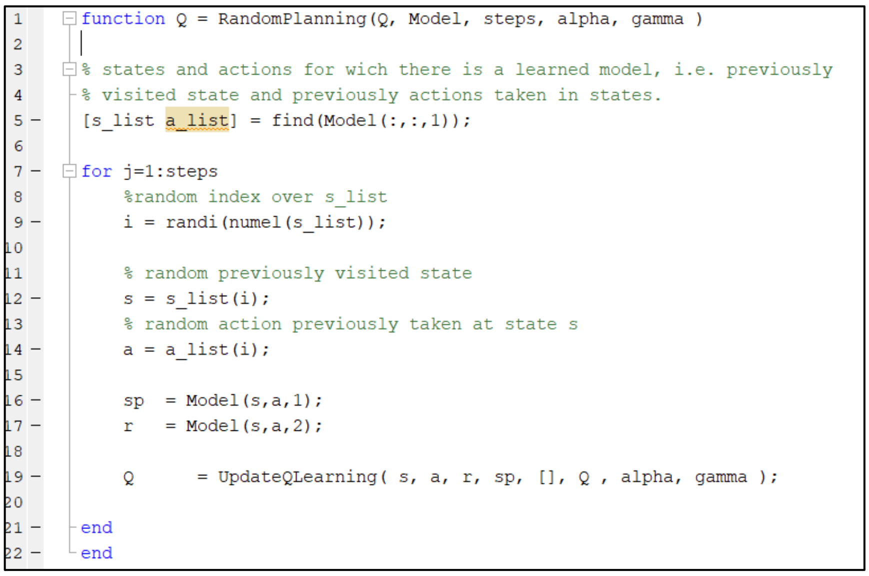
```
function [ Q ] = UpdateQLearning( s, a, r, sp, ~, Q , alpha, gamma )

TD_error =   ((r + gamma*max(Q(sp,:))) - Q(s,a));
Q(s,a) =  Q(s,a) + alpha * TD_error;
```

b) UpdateQLearning function



```
function [ Model ] = UpdateModel( s, a, r, sp, Model )
% a:  the last executed action

Model(s,a,1) = sp;
Model(s,a,2) = r;
```

c) UpdateModel function

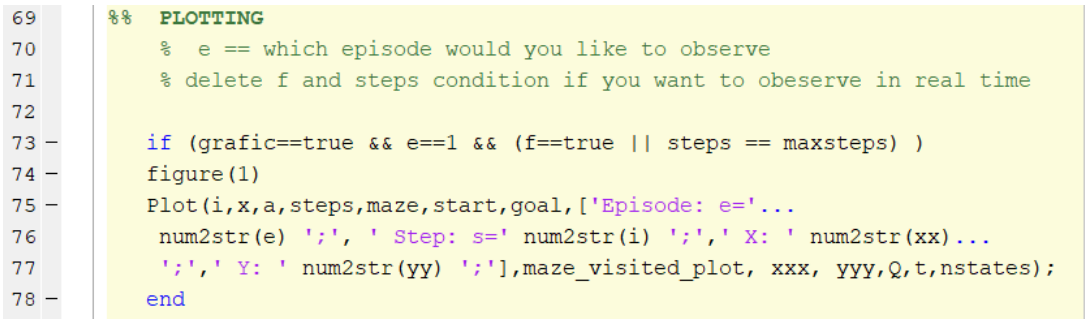


```
function Q = RandomPlanning(Q, Model, steps, alpha, gamma )

% states and actions for wich there is a learned model, i.e. previously
% visited state and previously actions taken in states.
[s_list a_list] = find(Model(:,:,1));

for j=1:steps
    %random index over s_list
    i = randi(numel(s_list));

    % random previously visited state
    s = s_list(i);
    % random action previously taken at state s
    a = a_list(i);

    sp  = Model(s,a,1);
    r   = Model(s,a,2);

    Q      = UpdateQLearning( s, a, r, sp, [], Q , alpha, gamma );

end
end
```

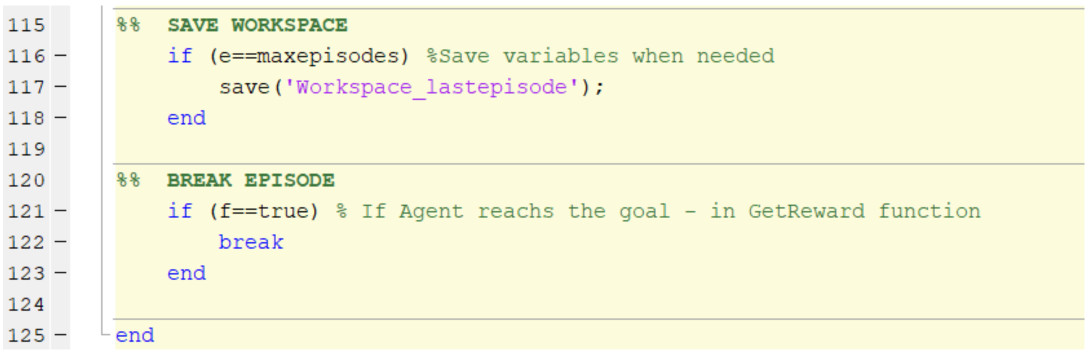
d) RandomPlanning function

Figure 2.3.16: Functions in steps loop.

After action, reward and new state, the Q-table with the Model is being updated and the next step is plotting the maze. We can do it live, choose the episode we want to watch or view it after the robot gets to the goal (or reach maximum of steps). It can be very helpful to analyze the algorithm with different parameters and types of mazes.



```
%% PLOTTING
    % e == which episode would you like to observe
    % delete f and steps condition if you want to obeserve in real time

   if (grafic==true && e==1 && (f==true || steps == maxsteps) )
   figure(1)
   Plot(i,x,a,steps,maze,start,goal,['Episode: e='...
    num2str(e) ';', ' Step: s=' num2str(i) ';',' X: ' num2str(xx)...
    ';',' Y: ' num2str(yy) ';'],maze_visited_plot, xxx, yyy,Q,t,nstates);
   end
```



```
%% SAVE WORKSPACE
    if (e==maxepisodes) %Save variables when needed
        save('Workspace_lastepisode');
    end

%% BREAK EPISODE
    if (f==true) % If Agent reachs the goal - in GetReward function
        break
    end

end
```

Figure 2.3.17: Episode function (part 3)– plot, save workspace and break.

In Figure 2.3.11 it can be seen a statement with variables *grafic, e, f* and *steps*. With *grafic* we can disable all plotting, with *e* we can choose an episode to watch and *f/steps* are for omit the part with watching the whole episode, because sometimes it can be very long to perform, depending on the computer performance.

Saving the variables from workspace is needed here, because without it at the end of the program we can't watch them, they are loaded in main.m function after all episodes. Episode function breaks when variable *f* is true, from GetReward function.

Chapter 3 – Simulation and experimental verification

3.1 Dyna-Q and Q-learning

As it was mentioned earlier, if in Dyna-Q algorithm we set planning steps to 0, then the algorithm will be basic Q-learning, without part with future rewards.

Below it is presented how the simulation looks like in Matlab on 10x10 maze. Maze on the left is from original part of the code, that shows movement of the Agent in real time, without its path colored. Another maze is created to visualize the path and the possibility to omit the real-time part of simulation, because of long executing time.

Under mazes there is a visualization of the Q-table, that shows its values in all possible states with 4 actions. We can see here the state with maximum reward around 78 state. This situation belongs to Dyna-Q with planning equal to 3. The displayed values of episodes, steps and parameters can be seen in the Figure 3.1.2.

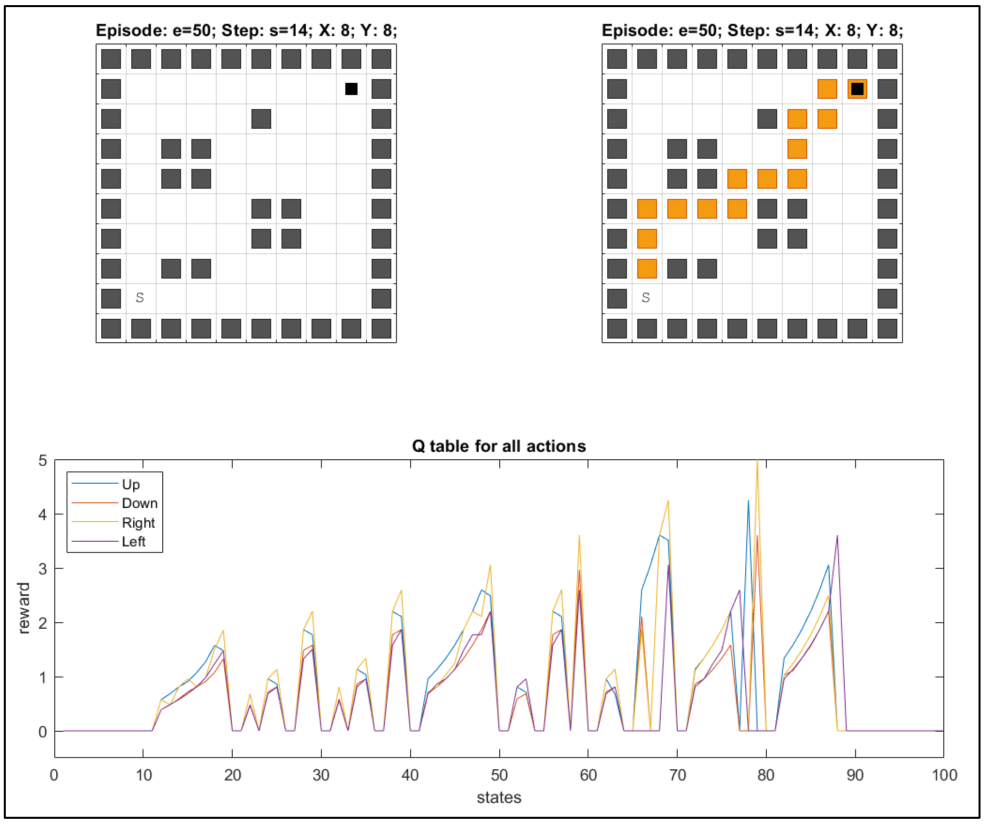
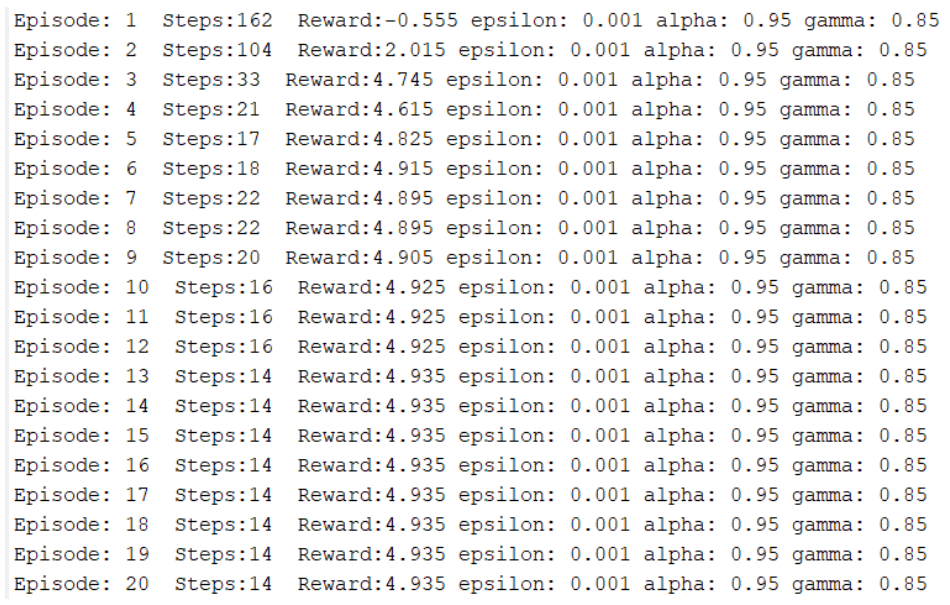


Figure 3.1.1: Path planning for Dyna-Q with p_steps=3, Episode 50 in terminate state.



```
Episode: 1  Steps:162  Reward:-0.555 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 2  Steps:104  Reward:2.015 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 3  Steps:33  Reward:4.745 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 4  Steps:21  Reward:4.615 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 5  Steps:17  Reward:4.825 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 6  Steps:18  Reward:4.915 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 7  Steps:22  Reward:4.895 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 8  Steps:22  Reward:4.895 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 9  Steps:20  Reward:4.905 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 10  Steps:16  Reward:4.925 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 11  Steps:16  Reward:4.925 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 12  Steps:16  Reward:4.925 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 13  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 14  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 15  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 16  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 17  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 18  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 19  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 20  Steps:14  Reward:4.935 epsilon: 0.001 alpha: 0.95 gamma: 0.85
```

Figure 3.1.2: Displayed values in Command Window.

As it can be seen in Figure 3.1.1 the Q-table is very complex on the chart because of modeling the environment. On the next figure the Q-table with the same parameters from Q-learning algorithm is showed, that have also some reward values below 0, like Dyna-Q in the first episode.

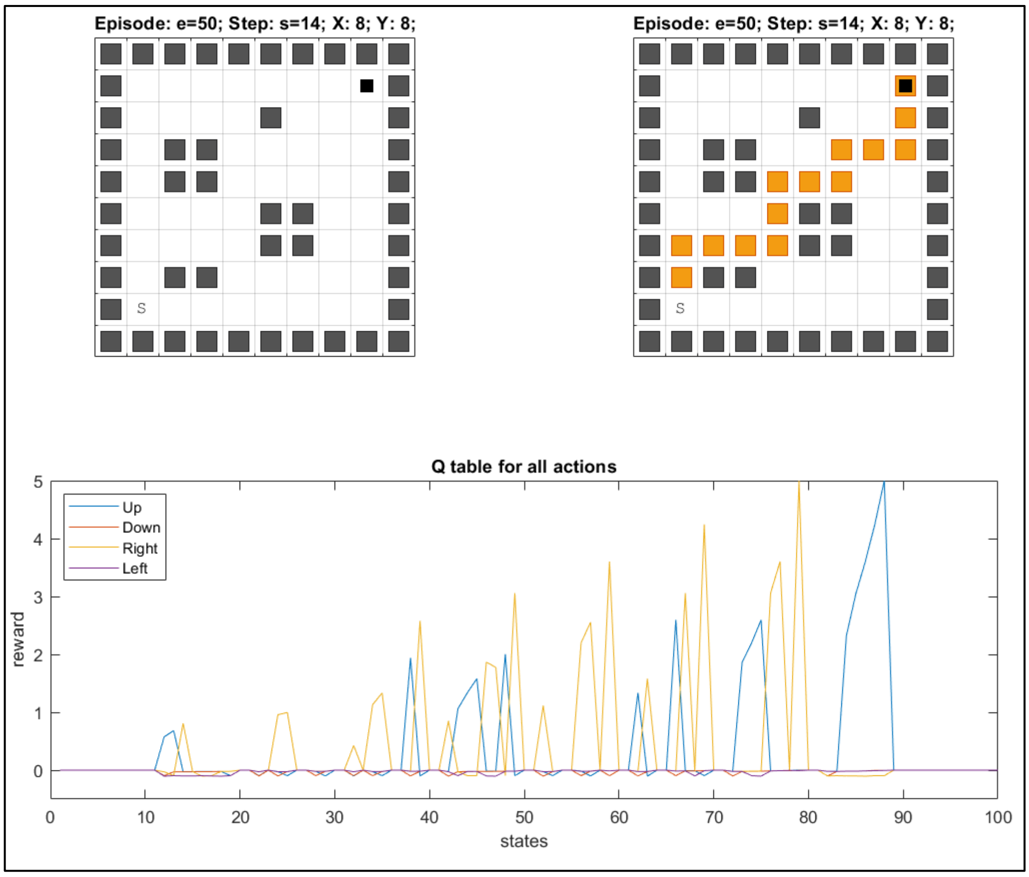


Figure 3.1.3: Path planning for Q-learning (p_steps=0), Episode 50 in terminate state.

Also, the simulations have been made to show the difference in episodes in case of planning steps. The Q-learning and 3 different planning steps for Dyna-Q are presented below. The y-axis shows how many steps the Agent did to get to the goal in each episode, that is on x-axis.

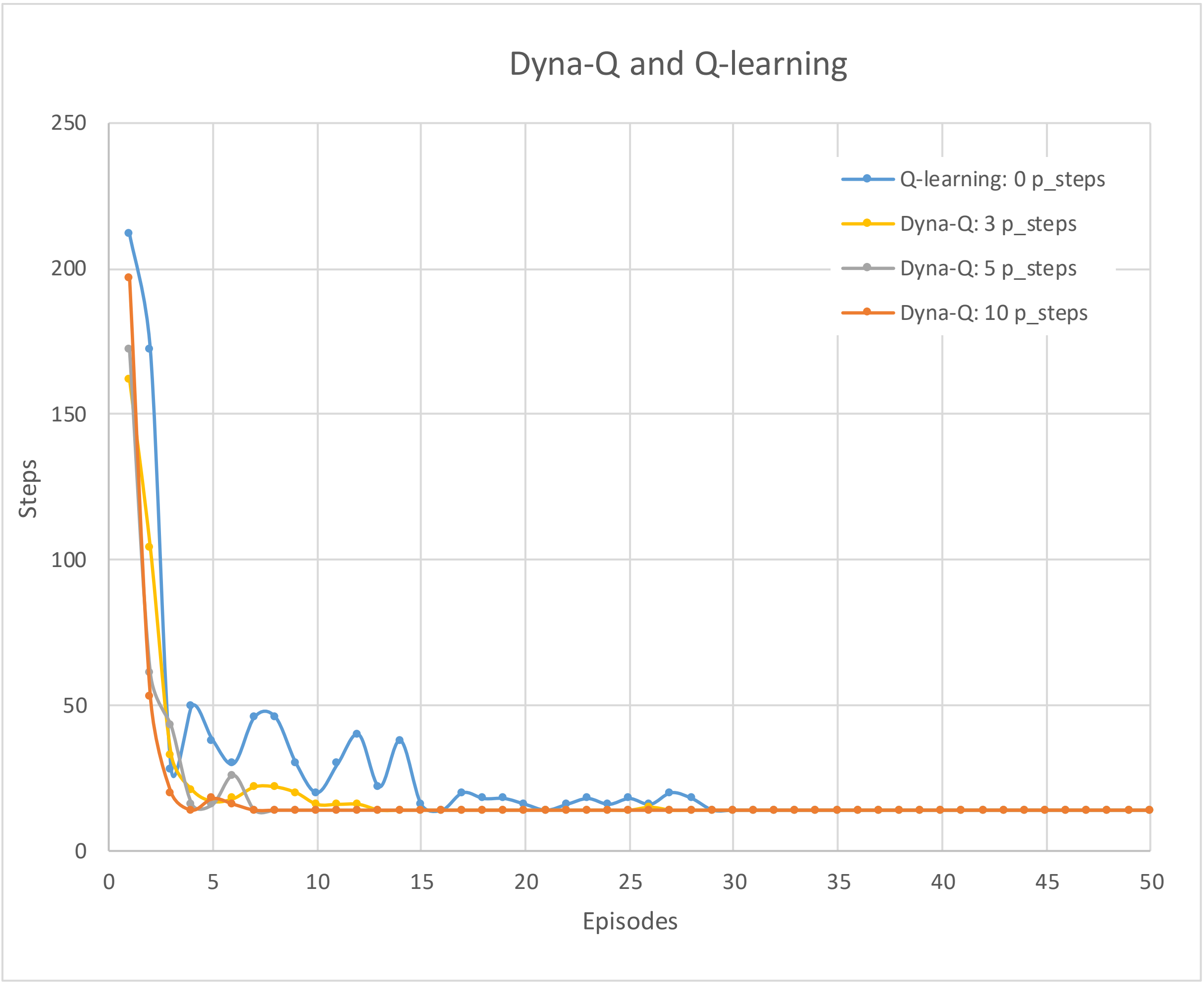
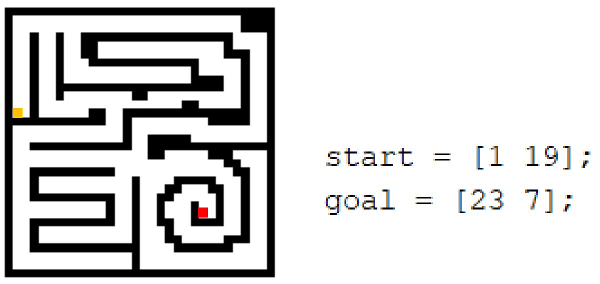


Figure 3.1.4: Comparison of algorithms in case of different planning steps value.

Although Q-learning is simpler than Dyna-Q, there is much difference in effectiveness if we add planning with the model, even in small part. All Dyna-Q charts are similar to each other and Q-learning need at least twice as many episodes to get stabilized the shortest path, which here is equal to 14 steps.

3.2 Dyna-Q and A* algorithm.

This part was done together with Radoslaw Chludzinski which is based on the A* algorithm developed in his Interim work project[16]. The simulations were made on the same maze(64x64) presented below.



```
start = [1 19];
goal = [23 7];
```

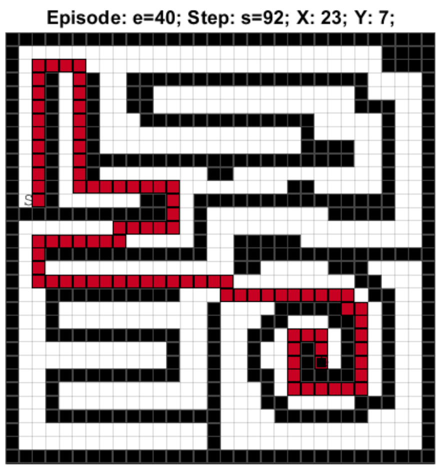
Figure 3.2.1: Maze for the algorithm comparison with coordinates of start/goal.

The A* algorithm has developed a different method for path planning problem with various assumptions, in case of this it was decided to find similarities in both of the algorithms.

Reinforcement learning algorithm has a possibility to do only four actions: Up, down, left, right, but this A* algorithm can perform also biased directions. Despite that, the number of steps to find the shortest path can be compared.

Knowing the number of all cells: 64 x 64 = 4096, the parameter of searched points can be also considered. RL is mainly based on learning part, which is a method that requires to do many trials and errors to develop a solution. Because of that, the first episode has usually the largest number of searched points, that can be more than the number of all cells. In this process of learning, in every episode it tries to decrease this number, because to every step is given a negative reward except the goal. A* algorithm does not have a reward system, but it uses a graph searching. The number of cells it searched increases with the complexity of the labyrinth.

Elapsed time can be a reference to the calculation weight for the algorithm, the more complex it is, the more time Matlab has to spend on solving it.

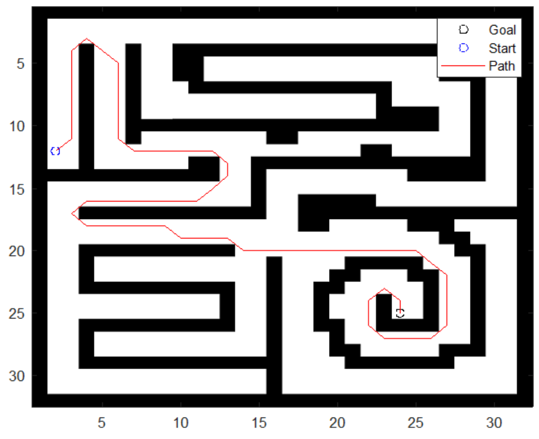


Steps: 92

Searched points: 92-5000 depending on episode

Elapsed time: 1.621s

a) Dyna-Q algorithm,
parameters: p_steps=100, epsilon=0.0001, alpha=0.95, gamma=0.65



Steps: 74

Searched points: 592

Elapsed time: 0.083s

b) A algorithm[16]*

Figure 3.2.2: Comparison of Dyna-Q and A algorithm.*

Both algorithms found the shortest path, based on their structure. Dyna-Q algorithm was prepared to move only in 4 directions, without an oblique movements, so the shortest path is equal to 92 steps, compared to A* where it is equal to 74. It would be the one of the further steps to upgrade Dyna-Q algorithm, to give it more directions to move.

The main difference is that the A* algorithm works in a static, well known environment, and Dyna-Q can be put into any environment without necessity to input a map into the robot. We can also assume that robots with both algorithms have a map of environment and the code of Dyna-Q is done before the final movement, then Dyna-Q takes more memory and more time (about 1,5s more) to make all episodes.

Both methods are effective depending on how we want to use them and what to achieve in case of the robot. After obtaining comparative results and testing the algorithm more times, it turned out that the results on this maze could be better, which is discussed in chapter 3.3.

3.3 Dyna-Q in a complex maze

The maze from comparison that is used to improve the results from Dyna-Q algorithm is presented below, along with Q-table. There was a need to somehow reduce the number of episodes that would find the shortest path.

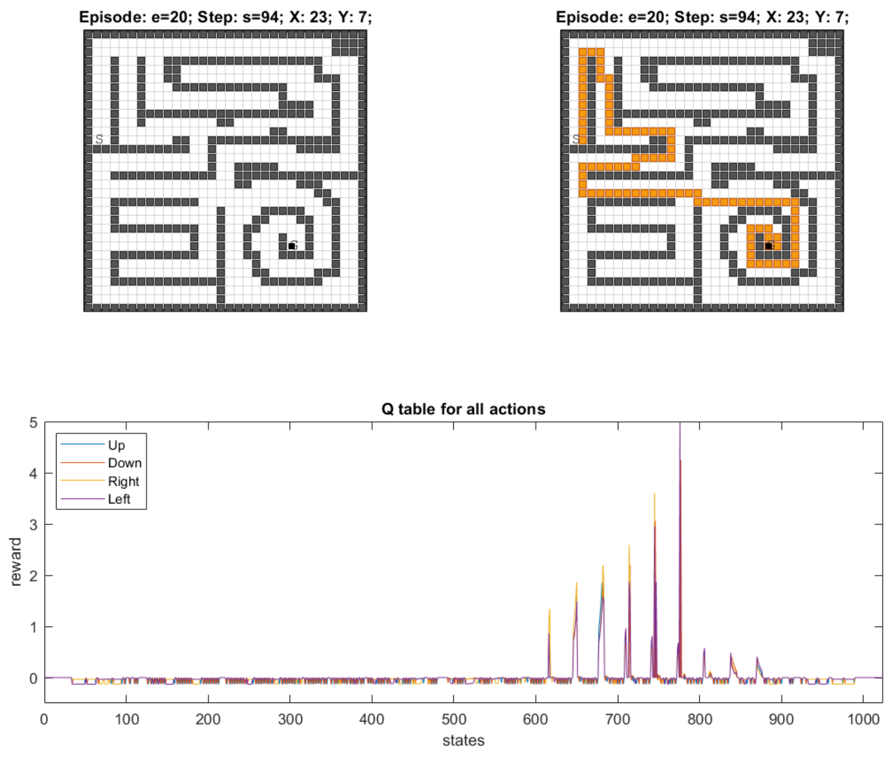
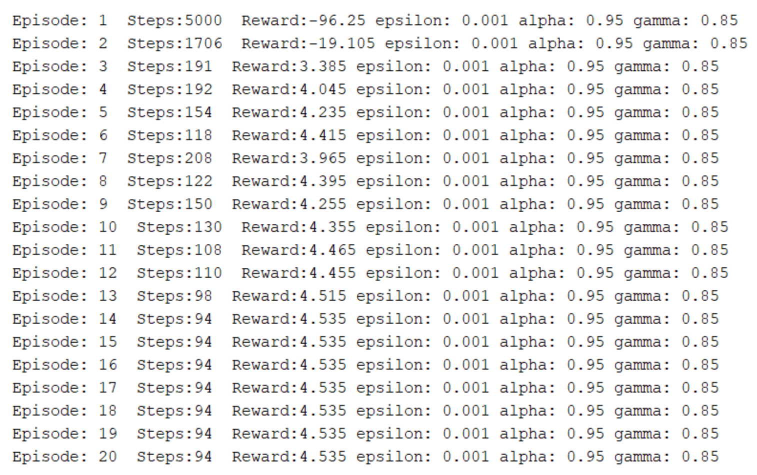


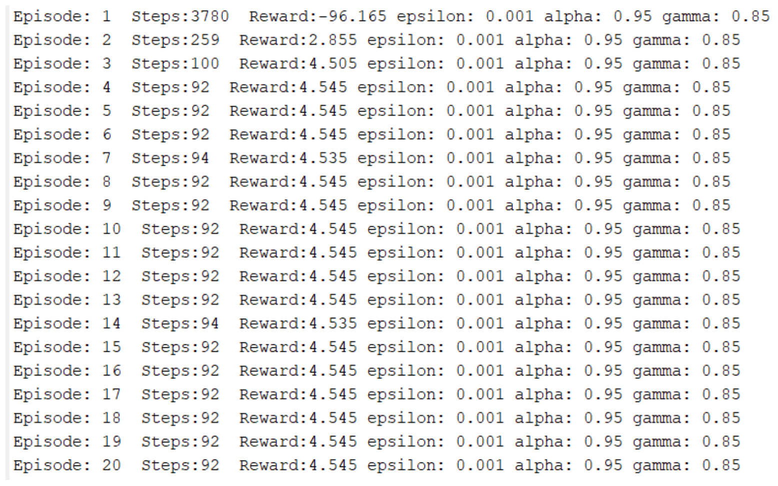
Figure 3.3.1: Simulation with 100 planning steps, episode 20.

In comparison to small mazes, the most significant variable is planning steps. After a lot of simulations with changing the parameters of alpha, gamma and epsilon, based on experience from Dyna-Q and Q-learning at chapter 3.1, it was decided to change the planning steps value much higher- to 1000 steps.



```
Episode: 1  Steps:5000  Reward:-96.25 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 2  Steps:1706  Reward:-19.105 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 3  Steps:191  Reward:3.385 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 4  Steps:192  Reward:4.045 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 5  Steps:154  Reward:4.235 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 6  Steps:118  Reward:4.415 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 7  Steps:208  Reward:3.965 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 8  Steps:122  Reward:4.395 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 9  Steps:150  Reward:4.255 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 10  Steps:130  Reward:4.355 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 11  Steps:108  Reward:4.465 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 12  Steps:110  Reward:4.455 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 13  Steps:98  Reward:4.515 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 14  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 15  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 16  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 17  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 18  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 19  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 20  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
```

a) P_steps = 100



```
Episode: 1  Steps:3780  Reward:-96.165 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 2  Steps:259  Reward:2.855 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 3  Steps:100  Reward:4.505 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 4  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 5  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 6  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 7  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 8  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 9  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 10  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 11  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 12  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 13  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 14  Steps:94  Reward:4.535 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 15  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 16  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 17  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 18  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 19  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
Episode: 20  Steps:92  Reward:4.545 epsilon: 0.001 alpha: 0.95 gamma: 0.85
```

b) P_steps = 1000

Figure 3.3.2: Comparison of simulations with changed value of planning steps.

These results are closer to what was expected from machine learning algorithm. In the real environment we would have put the Agent in an unknown maze and needed to teach him in about 4 episodes, not 15. For a better result, the gamma parameter has been slightly lowered in this maze, to let the Agent explore more so it won't be sticked to 94 steps path and try to find the shortest. Gamma parameter is for balancing the greedy policy and exploration.

Comparison between the episodes and Q-table difference.

Change in paths for different episodes visualized on the maze and Q-table:

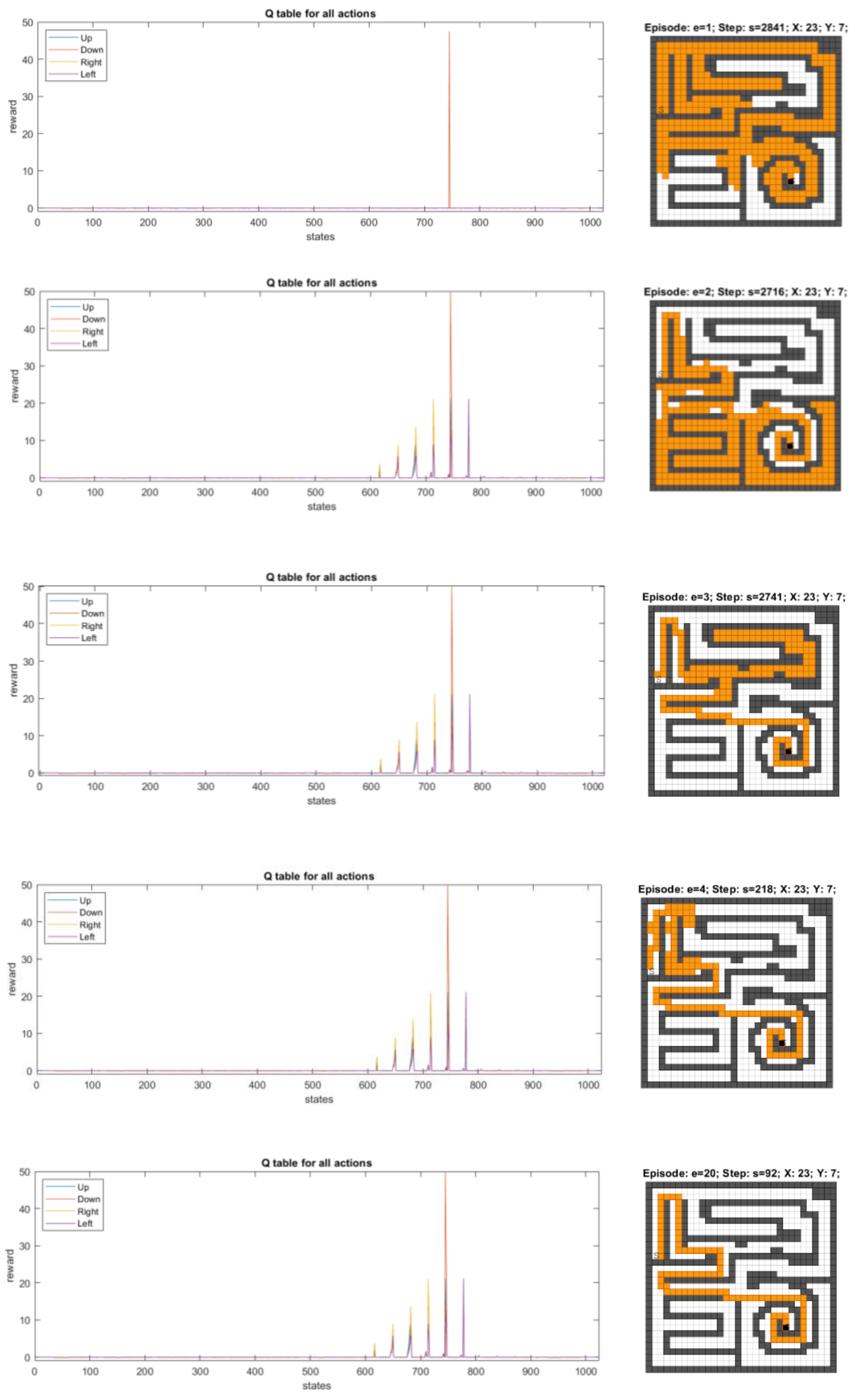
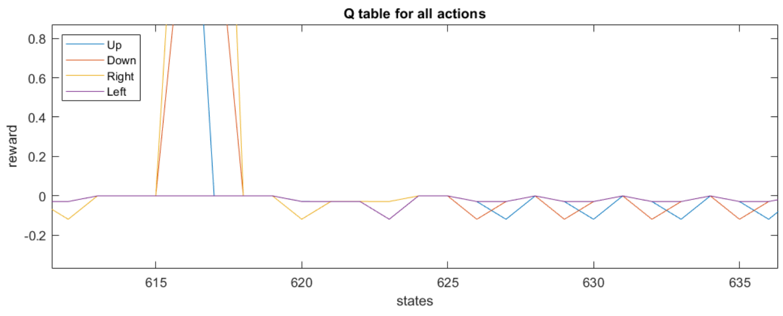


Figure 3.3.3: Comparison of episodes.

It can be seen that mostly the Q-table is changed after the 1'st episode, because after the Agent finds the path, the values of rewards are mostly known. With a low value of planning steps it takes more episodes to calculate the path with the highest reward, so it can be observed as an orange area that was searched. Every orange square is a visited cell in the maze, If the Q-table is not changing that much, it means that the only cause in slow path finding are the parameters, in this case planning steps.

Close up on Q-table chart:



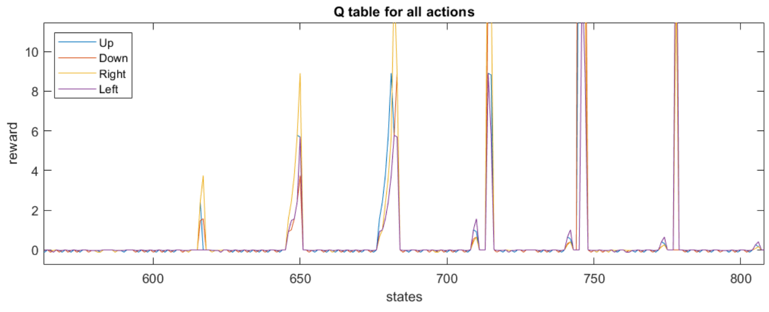
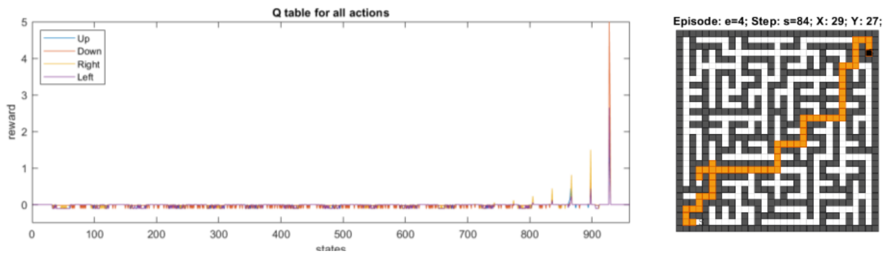


Figure 3.3.4: Comparison of episodes – Q-table close up.

These comparisons were took when the max reward was equal to 50, after that when it was changed that to 5, the charts were more readable and reacted better on changing the parameters.

The difference with low and high gamma value is presented on another maze that was described in the Maze Implementation part:

a)



b)

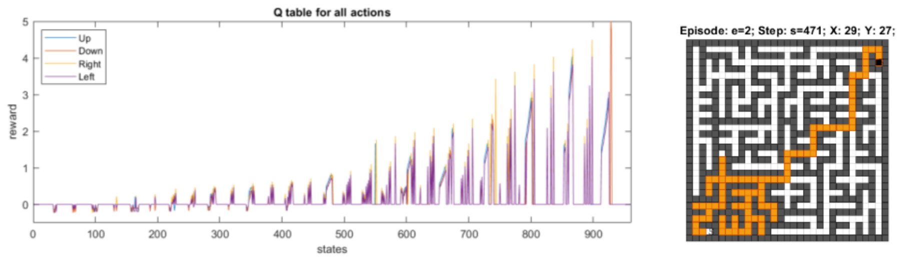


Figure 3.3.5: Comparison in Q-table with low(a) and high(b) gamma.

This maze was not mainly used, because the path built with only one square corridor is more convenience for the algorithm and is not a good choice for testing all the parameters.

Based on these charts we can assume that this is the effect of calculating the future rewards, that the gamma stands for, so at the end we have much more states with higher reward.

Conclusions

Q-learning is the simplest method for understanding the Reinforcement Learning. Based on its algorithm, now we have a lot of ways to upgrade it, basic Dyna-Q and it's mixed forms with vectors for example and other algorithms that want to speed up the learning process and accuracy. In Q-learning there is a huge difference with model-free and model-based approach. Adding a modeling part to the algorithm that can allow the Agent to learn on experience decreases the number of episodes at least at 50%.

It is better to have the values of reward close to each other: -0.2, 0.1, 1, 5; not 50 with -0.1 for penalty like it was done in one of the simulations, because it decreases a chance to proper manipulating the parameters and to see effect between them. Although we can observe a huge difference in reward values with higher gamma, that confirms the theory.

Based on the environment and possibilities of the robot we can choose between Reinforcement learning or another like A* algorithm. Reinforcement learning needs more memory to work but it can be used in dynamic and unknown environments instead of A* that need to have a map of the environment to do the graph search. Of course these algorithms can be integrated, because Reinforcement learning can be used for other purposes than path finding and be a great supplement like it can be read in other works [13].

The more project was described, the better the simulation results were. Making simulations can take time in this method, so it's better to focus also on theory and what are the expected results.

Further research on that topic would be about how to upgrade this algorithm to more complex problems or dynamic obstacles and analyze other algorithms like SARSA to see the difference and their purpose.

References

[1] Chen Xia. Intelligent Mobile Robot Learning in Autonomous Navigation. Signal et Automatique de Lille (CRIStAL) - UMR 9189 , 2015 (p. 1-7)

[2] Anis Koubaa, Hachemi Bennaceur, Imen Chaari, Sahar Trigui, Adel Ammar, Mohamed-Foued Sriti, Maram Alajlan, Omar Cheikhrouhou and Yasir Javed. Robot Path Planning and Cooperation, Springer International Publishing, Vol. 772,(p. 13-53), 2018.

[3] Seyyed Mohammad Hosseini Rostami, Arun Kumar Sangaiah, Jin Wang3 and Xiaozhu Liu. Obstacle avoidance of mobile robots using modified artificial potential field algorithm. EURASIP Journal on Wireless Communications and Networking volume 2019, Article number: 70, 2019 (p. 13)

[4] Farbod Fahimi. AutonomousRobots: Modeling, Path Planning, and Control, 2009 (p. 94)

[5] Oshina Vasishth and Yogita Gigras. Path Planning Problem. International Journal of Computer Applications, 2014, Vol.104(2)

[6] Baijayanta Roy. A-Star (A*) Search Algorithm. Sep 29, 2019 https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb

[7] Random Services, portal related to mathematics. https://www.randomservices.org/random/markov/index.html

[8] Kung-Hsiang, Huang. Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG). Jan 12, 2018 https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

[9] Mathworks.com – Reinforcement Learning with Mathlab and Simulink, E-book. https://www.mathworks.com/content/dam/mathworks/ebook/gated/reinforcement-learning-ebook-part1.pdf

[10] Ziad SALLOUM. Summary of Tabular Methods in Reinforcement Learning: Comparison between the different tabular methods in Reinforcement Learning. Dec 16, 2018 https://towardsdatascience.com/summary-of-tabular-methods-in-reinforcement-learning-39d653e904af

[11] Andre Violante. Simple Reinforcement Learning: Q-learning. Mar 18, 2019 https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[12] Jeremy Zhang . Reinforcement Learning — Model Based Planning Methods. Jul 6, 2019 https://towardsdatascience.com/reinforcement-learning-model-based-planning-methods-5e99cae0abb8

[13] D. Venkata Vara Prasad, Devi. J Chitra and Priyadharsini. D Manju. Knowledge Based Reinforcement Learning robot in maze environment international. Journal of Computer Applications, February 2011, Vol. 14- No.7 (0975 – 8887).

[14] Savita Kumari Sheoran and Poonam Poonam. Comparative Analysis of Reinforcement Learning Methods for Optimal Solution of Maze Problems. International Journal of Advanced Research in Computer Science . Mar/Apr2017, Vol. 8 Issue 3, p957-962. 6p.

[15] Kao-Shing Hwang, Wei-Cheng Jiang and Yu-Jen Chen. ADAPTIVE MODEL LEARNING BASED ON DYNA-Q LEARNING. Proceedings of SICE Annual Conference (SICE), August 2012, pp.1277-1280

[16] Radoslaw Chludzinski. Student Interim Project: Mobile robot navigation based on A* algorithm, Bialystok University of Technology Faculty of Mechanical Engineering, June 2020.