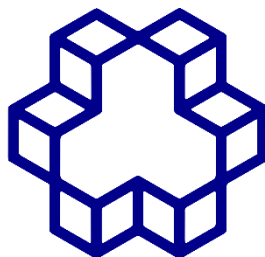


به نام خدا



دانشگاه صنعتی خواجه نصیرالدین طوسی

دانشکده مهندسی برق و کامپیوتر

دوره کارشناسی ارشد مهندسی برق

پروژه نهایی درس یادگیری تقویتی

توسط:

محمد مهدی ناظری اردکانی

۴۰۰۲۸۸۴۴

استاد درس:

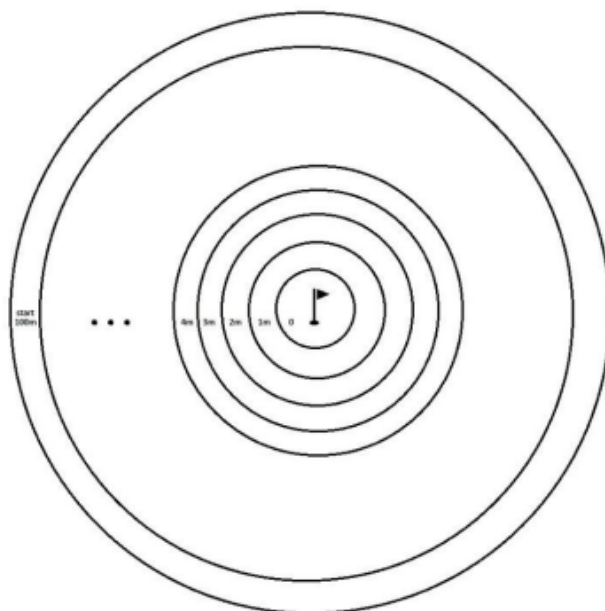
دکتر سید حسین خواسته

بهار ۱۴۰۲





سوال انتخاب شده پروژه.....	۳
مقدمه	۵
قسمت اول سوال.....	۶
شبیه سازی در نرم افزار متلب.....	۶
توضیحات بخش های مختلف کد اصلی متلب	۶
کد اصلی پیاده سازی شده در متلب	۷
شبیه سازی محیط بازی گلف (Environment)	۱۰
تابع پیاده سازی شده برای تولید اپیزود.....	۱۲
تابع پیاده سازی شده برای بهبود پالیسی	۱۶
نتایج شبیه سازی پیاده سازی الگوریتم Q-learning بدون حضور دریاچه	۱۷
نتایج شبیه سازی پیاده سازی الگوریتم SARSA بدون حضور دریاچه	۲۰
قسمت دوم سوال	۲۳
قسمت سوم سوال	۲۴
قسمت چهارم سوال	۲۴
پالیسی بهینه بدست آمده با یک میلیون اپیزود در روش Q-learning	۲۵
پالیسی بهینه بدست آمده با صد هزار اپیزود در روش SARSA	۲۶

سوال انتخاب شده پروژه

زمین گلفی را مانند شکل زیر در نظر بگیرید که در آن شروع بازی از فاصله‌ی صد متری هدف می‌باشد.



به ازای هر ضربه می‌توان از بین چهار چوب Woods، Irons، Hybrids و Putter یک چوب را برای ضربه زدن انتخاب کرد. این چوب‌ها به ترتیب از بی‌دقت‌ترین و پرقدردت‌ترین تا دقیق‌ترین و ضعیف‌ترین مرتب شده‌اند. هر کدام از این چوب‌ها شامل ضریب دقت (precision) و فاصله‌ی پیش‌فرض (default distance) می‌باشند که فاصله‌ی طی شده توسط توپ پس از ضربه با چوب موردنظر را مشخص می‌کند. ضریب دقت و فاصله‌ی پیش‌فرض هر چوب در جدول زیر مشخص شده است.

	Club Type	Default Distance	Precision
	Woods	18 meter	$\mathcal{N}(1, 0.25)$
	Irons	12 meter	$\mathcal{N}(1, 0.15)$
	Hybrids	6 meter	$\mathcal{N}(1, 0.05)$
	Putter	3 meter	1

شدت ضربه (power) با چوب نیز در فاصله‌ای که توپ طی می‌کند تاثیر گذار است که مقدار آن می‌تواند از صفر تا یک با دقت 0.1 (یعنی 10 مقدار) باشد. شدت باد نیز به صورت نرمال می‌تواند در فاصله‌ی طی شده توسط

توپ با مقدار $\mathcal{N}(0, 3)$ تاثیر گذار باشد. در نهایت رابطه‌ی مسافت طی شده توسط توپ پس از ضربه با چوب \hat{t} به صورت زیر محاسبه می‌شود:

$$d = [\text{power} * \text{default distance}_i * \text{precision}_i] + \text{round}((1 - \text{precision}_i) * \text{wind}_{\text{disturbance}})$$

موارد زیر مطلوب است:

1. مسئله‌ی طرح شده را به دو روش SARSA و Q-learning حل کنید، پالیسی بهینه و تابع ارزش state-action را به ازای هر دو روش به دست آورید. (برای هر دو حالت از $\epsilon - \text{soft policy}$ با مقدار یکسانی از ϵ و α استفاده کنید).
2. نحوه‌ی انتخاب پاداش، حالات و اکشن‌ها را شرح دهید؟
3. تفاوت عملکرد این دو روش را شرح داده و مشخص کنید که با توجه به نتایج به دست آمده، به نظر شما کدام روش عملکرد بهتری دارد؟
4. شرایطی را در نظر بگیرید که بخش‌هایی از زمین شامل دریاچه بوده و با افتادن توپ در آب بازی به اتمام می‌رسد. این موضوع چه تغییراتی در شرایط مسئله و رویه‌ی اجرای هر دو الگوریتم ایجاد می‌کند؟

توجه کنید که تمام ضربات به سمت هدف بوده و زاویه‌ی ضربه تأثیری در این مسئله و حل آن ندارد. همچنین فاصله‌ی توپ تا هدف در تمامی مراحل اجرا به صورت گسسته می‌باشد.

مقدمه

برای حل سوال نیاز است که ابتدا مسئله را به صورت یک مسئله یادگیری تقویتی تعریف کنیم. برای تعریف نیاز است که اجزای مختلف یک مسئله یادگیری اعم از حالت‌ها، عمل‌ها و پاداش و محیط اطراف عامل به صورت خاص تعریف شوند. در اینجا این پارامترها به صورت زیر تعریف می‌شوند.

- **حالت‌ها:** فاصله توپ از هدف نهایی که در اینجا می‌تواند از مقدار 0 تا 100 باشد.
- **عمل‌ها:** در اینجا با توجه به صورت سوال 4 نوع عصا یا دسته وجود دارد که هر کدام می‌تواند با قدرت 0.1 تا 1 ضربه وارد کند که در کل 40 نوع اکشن یا عمل را برای ما ایجاد می‌کند.
- **محیط پیرامون:** در اینجا زمین بازی گلف به عنوان محیط پیرامون در نظر گرفته شده است. یکی از نکاتی که در این محیط حائز اهمیت است وجود قدرت باد است که با توجه به صورت سوال در محاسبه فاصله طی شده توسط هر اکشن اعمال می‌شود.

- **پاداش :** هدف از بازی افتادن توپ در حفره وسط می‌باشد. باتوجه به صورت مسئله به دنبال این هدف با کمترین تعداد ضربات به توپ هستیم. بنابراین پاداش را در ادامه به گونه ای در نظر میگیریم که عامل بتواند هدف مورد نظر برسد.

قسمت اول سوال

شبیه سازی در نرم افزار متلب

توضیحات بخش های مختلف کد اصلی متلب

در اینجا برای پیاده سازی الگوریتم های مختلف نیاز است که محیط بازی را در نرم افزار متلب شبیه سازی کنیم. ابتدا به توضیح اجزای برنامه می پردازیم.

در این برنامه، یک بازی گلف با سیاست ϵ -greedy با استفاده از الگوریتم Q-learning تعریف شده است. این الگوریتم به وسیله ی تعیین توان های ارزش عمل می تواند مسیر بهینه را از نقطه شروع تا نقطه هدف در محیطی با موانع و برداری پیدا کند. توضیحات هر بخش از برنامه به شرح زیر است:

• تعریف محیط:

- یک محیط از اندازه game_plane_size با استفاده از آرایه Plane تعریف می شود که از یک صفحه با ابعاد داده شده تشکیل شده است. در این محیط، مقادیر ۱ نمایان گر مکان های آزاد و مقدار ۰ نمایان گر موانع هستند.
- نقطه هدف با مختصات Goal نیز تعریف می شود.

• تعریف حالت ها و اقدامات:

- حالت ها به عنوان مکان های آزاد در Plane مشخص می شوند و در آرایه AllStates ذخیره می شوند.
- اقدامات به عنوان انتخاب های ممکن از نقاط مجاز به عنوان حالت ها تعریف شده و در آرایه AllActions ذخیره می شوند. همچنین، تمام ترکیب های حالت و اقدامات ممکن در آرایه AllStateActionPairs نگهداری می شوند.

• اجرای الگوریتم Q-Learning:

- آرایه های Counter و Q برای ذخیره سازی شمارنده ها و مقادیر Q در هر حالت و اقدام ایجاد می شوند.
- gamma نسبت تخفیف برای پاداش های آینده تعیین می شود.

- با تعداد مشخص شده nEpisode از آموزش و بهروزرسانی Q اقدام می‌شود.
- Q ها از روی سیاست ϵ -greedy بهروزرسانی می‌شوند.
- تولید یک حلقه اجرا با استفاده از سیاست یادگرفته شده:
 - با تعیین نقطه شروع initialState، یک حلقه تکرار برای اجرای سیاست یادگرفته شده با شرایط مشخص (اکسپلوریشن یا اکسپلویتیشن) ایجاد می‌شود.
 - سپس نتیجه‌ی این اجرا بازگردانده می‌شود.

کد اصلی پیاده سازی شده در متلب

در اینجا سعی شده است به صورت کامل کامنت گذاری انجام شود. بنابراین تنها در مواردی که نیاز به توضیح بیشتر است توضیحات به ذیل آن اضافه شده است.

```
% Clear the command window, clear workspace variables, and close all figures
clc;
clear;
close all;
```

Section for setting up the game plane and displaying it

```
game_plane_size = 101; % Define the size of the game plane
Plane = ones(1, game_plane_size); % Initialize the game plane with all free spaces (1:
free, 0: barrier)
imagesc(Plane); % Display the game plane as an image
```

Section for defining the goal location and initializing states

```
Goal = [1, 0]; % Define the coordinates of the goal location

% Find all states (positions) where the game plane is free
[R, C] = find(Plane == 1);
AllStates = [R' C' - 1]; % Store the coordinates of all free states
nStates = size(AllStates, 1); % Get the total number of free states
```

Section for defining all possible actions and creating state-action pairs

Define the possible actions as position offsets (AllActions matrix)

```
AllActions = [1:10; 11:20; 21:30; 31:40]'; % Each row represents a set of actions for
each state

% Calculate the total number of possible actions
nActions = size(AllActions, 1) * size(AllActions, 2);

% Create a matrix (AllStateActionPairs) to store all state-action pairs
AllStateActionPairs = zeros(nStates * nActions, 3);
```

```
% Populate the AllStateActionPairs matrix with state-action pairs
for i = 1:nStates
    for j = 1:size(AllActions, 2)
        % Repeat the current state (coordinates) for each action in the current set
        % and append the corresponding action to create the state-action pair
        AllStateActionPairs(1 + size(AllActions, 1) * (4 * (i - 1) + j - 1) :
size(AllActions, 1) * (4 * (i - 1) + j), :) = [repmat((AllStates(i, :)),
size(AllActions, 1), 1), AllActions(1:size(AllActions, 1), j)];
    end
end
```

در مورد نحوه تعریف اکشن ها نیاز است که ماتریس AllActions را به صورت بالا تعریف کرد به این گونه که ستون اول ، دوم ، سوم و چهارم به ترتیب مربوط به عصا یا دسته با جنس putter ، Irons،Hybrids و Woods است.

همچنین سطر آن نیز به معنای مقدار قدرتی است که انتخاب می شود بنابراین اکشن شماره ۱ تا ۱۰ به معنای انتخاب عصا یا دسته putter با قدرت ۰.۱ تا ۱ میباشد. (اکشن ۱۰ یعنی عصا یا دسته putter با قدرت ۱) بنابراین با استفاده از شماره اکشن میتوان عصا یا دسته انتخابی و قدرت موردنظر انتخاب شده را بدست آورد.

Section for initializing Q-values and other parameters

```
nAllStateActionPairs = size(AllStateActionPairs, 1); % Get the total number of state-
action pairs

Counter = zeros(nAllStateActionPairs, 1); % Initialize a counter for each state-action
pair
Q = zeros(nAllStateActionPairs, 1); % Initialize Q-values for each state-action pair

gamma = 0.99; % Set the discount factor for future rewards
nEpisode = 1000000; % Define the number of episodes for training
epsilon = 0.1; % Define the exploration rate for epsilon-greedy policy

% Initialize the policy randomly by assigning random actions to each state
Policy = randi(nActions, nStates, 1);

alpha = 0.1; % Learning rate for updating Q-values
```

Main loop for running episodes and updating Q-values using Q-learning

```
for e = 1:nEpisode
    % Set the initial state for the current episode
    State = [1, 100];

    % Exploration-exploitation loop within each episode
    while true
        % Find the index of the current state in the AllStates array
        indexOfState = find(ismember(AllStates, State, 'row'));

        % Choose an action based on the policy with epsilon-greedy exploration
        randomNum = rand;
        if randomNum < ((1 - epsilon) + (epsilon / nActions))
            At = Policy(indexOfState); % Exploitation: choose action from the policy
        else
            At = randi(nActions); % Exploration: choose a random action
```



```

end

% Get the new state and the immediate reward based on the selected action
[NewState, Reward] = Environment(AllStates, State, At, Goal);

% Check if the agent has reached the goal state, then exit the loop
if ismember(NewState, Goal, 'row')
    break;
end

% Find the indices of the current state-action pair in the AllStateActionPairs
matrix
indexInPairs = find(ismember(AllStateActionPairs, [State, At], 'row'));

% Find the indices of the next state-action pairs (for the next time step)
indexOfNextStateInPairs = find(ismember(AllStateActionPairs(:, [1, 2]),
NewState, 'row'));

% Update the Q-value for the current state-action pair using the Q-learning
formula
Q(indexInPairs) = Q(indexInPairs) + alpha * (Reward + gamma *
max(Q(indexOfNextStateInPairs)) - Q(indexInPairs));

% Transition to the new state for the next time step
State = NewState;

% Update the policy using the current Q-values
Policy = PolicyImprovementUsingQ(Q, nActions);

% Check if the agent has reached the goal state, then exit the loop
if ismember(State, Goal, 'row')
    break;
end
end

% Display the current episode number
disp(['Episode: ', num2str(e)]);
end

```

در اینجا با توجه به اینکه ابتدا الگوریتم Q-learning را پیاده سازی کردیم، برای بروزرسانی تابع Q طبق فرمول زیر استفاده می کنیم.

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s, a)]$$

Section for generating an episode based on the learned policy

Set the initial state for generating an episode

```

initialState = [1, 100];

display = 1; % Set to 1 for displaying the episode, otherwise set to 0
epsilon = 0; % Exploration rate set to 0 for exploiting the learned policy
qPrediction = 0; % Set to 0 as it is not used in this case

% Generate an episode based on the learned policy
[VisitedStates, Actions, VisitedRewards] = GenerateEpisode(Plane, AllStates,
AllActions, Policy, initialState, Goal, epsilon, display, qPrediction);

```

شبیه سازی محیط بازی گلف (Environment)

```
% Environment - Golf Game Environment Simulator
% [NewState, Reward] = Environment(AllStates, State, Action, Goal)
%
% This function simulates the environment for a basic golf game. It takes the
current
% state, the selected action (representing the golf club and power used), and the
goal
% state. The function calculates the new state of the ball after taking the action
% and computes the reward based on the success of the shot.
%
% Inputs:
% AllStates: An array containing all the possible states of the ball.
% State: A 2-element array representing the current state of the ball [flag,
position].
% Action: An integer representing the selected action (golf club and power).
% Goal: An array representing the goal state of the ball.
%
% Outputs:
% NewState: A 2-element array representing the new state of the ball after the
action [flag, new_position].
% Reward: A scalar value representing the reward earned based on the success of
the shot.
%
function [NewState, Reward] = Environment(AllStates, State, Action, Goal)

% Extract the current position of the ball from the State array
old_state = State(2);

% Extract the power of the action (selected golf club) and its type
power_of_action = mod(Action, 10) * 0.1;
if (power_of_action == 0)
    power_of_action = 1;
end
type_of_action = floor((Action - 1) / 10) + 1;

% Assign default distances and precision based on the type of action (golf club)
switch (type_of_action)
    case 1
        % Putter selected
        default_distance = 3;
        precision = normrnd(1, 0); % Mean of 1, no precision randomness.

    case 2
        % Hybrids selected
        default_distance = 6;
        precision = normrnd(1, 0.05); % Mean of 1, with 0.05 standard deviation.

    case 3
        % Irons selected
        default_distance = 12;
        precision = normrnd(1, 0.15); % Mean of 1, with 0.15 standard deviation.

    case 4
        % Woods selected
        default_distance = 18;
        precision = normrnd(1, 0.25); % Mean of 1, with 0.25 standard deviation.
end

% Generate a random wind disturbance using normal distribution with mean 0 and
standard deviation of 3.
```

```

wind_disturbance = normrnd(0, 3);

% Calculate the new position of the ball after taking the action and considering the
wind disturbance.
d = ceil(power_of_action * default_distance * precision) + round((1 - precision) *
wind_disturbance);
new_state = abs(old_state - d);

% Set the NewState array to [1 new_state], where 1 represents the flag, and new_state
is the updated position of the ball.
NewState = [1 new_state];

% Check if the NewState is one of the possible states in AllStates.
if find(ismember(AllStates, NewState, 'row'))
    % If the NewState is one of the possible states, check if it is the goal state.
    if ismember(NewState, Goal, 'row')
        % If NewState is the goal state, assign a positive reward of +10.
        Reward = +10;
    else
        % If NewState is not the goal state, assign a negative reward of -5.
        Reward = -5;
    end
else
    % If NewState is not one of the possible states, assign a negative reward of -10.
    Reward = -10;
    % Set NewState to [1 100] representing an out-of-bounds state.
    NewState = [1 100];
end

```

در این برنامه، یک شبیه‌سازی ابتدایی از بازی گلف پیاده‌سازی شده است. هدف اصلی برنامه، شبیه‌سازی شرایط حرکت توپ گلف در یک محیط مجازی است. این محیط شامل موقعیت فلگ (flag) و موقعیت کنونی توپ (position) است. با توجه به نوع عملکرد (عصای گلف و قدرت ضربه) و هدف انتخابی کاربر، برنامه حرکت توپ را شبیه‌سازی کرده و امتیاز کسب شده بر اساس ضربه را محاسبه می‌کند.

برنامه متشکل از تابع Environment با ورودی‌ها و خروجی‌های مشخص است. در ادامه به توضیح جزئیات اجزای این تابع می‌پردازیم:

• ورودی‌ها:

- AllStates: آرایه‌ای حاوی تمامی وضعیت‌های ممکن توپ در بازی.
- State: آرایه‌ای با دو عنصر، حاوی وضعیت کنونی توپ [flag, position].
- Action: یک عدد صحیح که نشان‌دهنده عملکرد انتخاب شده (نوع عصای گلف و قدرت ضربه) است.
- Goal: آرایه‌ای حاوی وضعیت هدف توپ.

- **عملکرد:** این تابع با توجه به نوع عصای گلف انتخابی توسط کاربر، مسافت پیشروی معمولی (default_distance) و دقت ضربه (precision) را تعیین می‌کند. سپس از توزیع نرمال، مقدار اختلال باد (wind_disturbance) را تولید می‌کند. با در نظر گرفتن این مقادیر، شدت ضربه را محاسبه و موقعیت جدید توپ پس از ضربه را محاسبه می‌کند.

• خروجی‌ها:

- NewState: آرایه‌ای با دو عنصر، حاوی وضعیت جدید توپ پس از ضربه [flag, new_position].
- Reward: مقدار امتیاز کسب شده بر اساس موفقیت ضربه.

برنامه به صورت تصادفی اختلال باد و دقت ضربه را محاسبه می‌کند که باعث تنوع در شبیه‌سازی‌ها می‌شود.

- اگر وضعیت جدید توپ (NewState) برابر با یکی از وضعیت‌های ممکن (AllStates) باشد، امتیاز بر اساس موفقیت ضربه تعیین می‌شود. اگر به هدف رسیده باشد، امتیاز مثبتی (۱۰ امتیاز) نسبت داده می‌شود؛ اگر به هدف نرسیده باشد، امتیاز منفی (۵ امتیاز) نسبت داده می‌شود.
- اگر وضعیت جدید توپ (NewState) در میان وضعیت‌های ممکن نباشد، یک امتیاز منفی بزرگتر (۱۰ امتیاز) نسبت داده می‌شود و وضعیت جدید توپ به عنوان خارج از محدوده بازی (out-of-bounds) تعیین می‌شود.

نتیجه گیری:

این برنامه یک شبیه‌سازی ساده و ابتدایی از بازی گلف است که به صورت تصادفی اختلالاتی مانند اختلال باد و دقت ضربه را در نظر می‌گیرد. شبیه‌سازی‌های متعدد از این برنامه می‌توانند جهت ارزیابی عملکرد استراتژی‌های مختلف بازیکنان در بازی گلف استفاده شوند. این برنامه می‌تواند به عنوان یک ابزار آزمون و ارزیابی نسبت به مدل‌ها و الگوریتم‌های مختلف کنترل بازی استفاده شود.

تابع پیاده سازی شده برای تولید اپیزود

```
% Generate an episode (sequence of states, actions, and rewards) for a given
% plane navigation problem using a given policy and environment.
% Inputs:
% - Plane: The 2D plane represented as a matrix, where 1 indicates a free cell
%           and 0 indicates a barrier.
% - AllStates: Matrix containing all possible states (coordinates) in the plane.
% - AllActions: Matrix containing all possible actions.
% - Policy: A vector containing the chosen action for each state based on the policy.
% - initialStateActionPair: A pair containing the initial state [x, y] and initial
%                           action [dx, dy] for the episode.
% - Goal: The target/goal state that the agent aims to reach.
% - epsilon: Exploration parameter used for epsilon-greedy action selection.
```

```

% - display: A boolean flag to determine whether to visualize the episode during
generation.
% - qPrediction: A boolean flag indicating whether the action-value function should
%               be predicted using the given policy (1) or not (0).

% Outputs:
% - States: A matrix containing the sequence of states visited in the episode.
% - Actions: A matrix containing the sequence of actions taken at each time step.
% - Rewards: A matrix containing the sequence of rewards received at each time step.

function [States, Actions, Rewards] = GenerateEpisode(Plane, ...
    AllStates, AllActions, Policy, initialStateActionPair, Goal, ...
    epsilon, display, qPrediction)

    % Generate an episode using the given policy from the initial state.
    % The episode consists of states, actions, and rewards encountered during the
simulation.

    FinalTimeStep = 1e6; % Maximum number of time steps allowed in the episode.
    States = zeros(FinalTimeStep, 2); % Array to store states at each time step.
    Actions = zeros(FinalTimeStep, 1); % Array to store actions at each time step.
    Rewards = zeros(FinalTimeStep, 1); % Array to store rewards at each time step.

    isFinalState = 0; % Flag to indicate if the final state (goal state) is reached.
    t = 1; % Time step counter.

    States(t, :) = initialStateActionPair(1, [1 2]); % Set the initial state in the
episode.

    while 1
        if display
            % Visualization (optional): Display the current state, goal, and plane
with obstacles.
            Plane2 = Plane; % 1: free, 0: barrier
            Plane2(States(t, 1), States(t, 2)) = 3; % Mark the current state as '3'
for visualization.
            Plane2(Goal(1), Goal(2) + 1) = 2; % Mark the goal state as '2' for
visualization.
            imagesc(Plane2);
            title(['Time Step (' num2str(t) ')'], 'fontsize', 20);
            pause(0.1);
        end

        % Get the index of the current state in the list of all possible states.
        indexOfState = find(ismember(AllStates, States(t, :), 'row'));

        % Determine the action to take based on the epsilon-greedy policy.
        if rand > epsilon % Choose the action based on the learned policy with
probability 1 - epsilon.
            At = Policy(indexOfState); % Select the action from the learned policy.
        else
            At = randi(size(AllActions, 1)); % Choose a random action with probability
epsilon.
        end

        % If qPrediction is enabled and it's the first time step, set the initial
action.
        if qPrediction == 1 && t == 1
            Actions(t, :) = initialStateActionPair(1, [3 4]);
        else
            Actions(t, :) = At; % Record the action taken at the current time step.
        end
    end

```

```

% Simulate the environment based on the current state and action.
% Update the next state and reward based on the environment dynamics.
[States(t + 1, :), Rewards(t + 1)] = Environment(AllStates, States(t, :),
Actions(t, :), Goal);

% Check if the next state is the goal state (reached the final state).
if ismember(States(t + 1, :), Goal, 'row')
    isFinalState = 1; % Set the flag to indicate reaching the final state.
end

if isFinalState == 1
    if display
        % Visualization (optional): Display the final state if goal reached.
        Plane2 = Plane;
        Plane2(States(t + 1, 1), States(t + 1, 2) + 1) = 3;
        Plane2(Goal(1), Goal(2) + 1) = 2;
        imagesc(Plane2);
        title(['Time Step (' num2str(t) ')'], 'fontsize', 20);
        pause(0.1);
    end

    break; % Exit the episode generation loop when the goal is reached.
end

t = t + 1; % Increment the time step.
end

% Remove excess zeros from the arrays to get the actual episode data.
States = States(1:t + 1, :);
Actions = Actions(1:t + 1, :);
Rewards = Rewards(1:t + 1, :);

end

```

برنامه‌ی فوق یک تابع است که برای تولید یک "حلقه" از وضعیت‌ها، اعمال، و پاداش‌ها در یک مسأله‌ی مشخص مرتبط با بازی گلف استفاده می‌شود. این تابع به عنوان یک بخش از یادگیری تقویتی عمل می‌کند که مسئله را به صورت بلند مدت حل می‌کند تا عامل بتواند در آن محیط بهترین رفتارها را یاد بگیرد.

توضیحات مفصل در مورد اجزای مختلف تابع به شرح زیر است:

• ورودی‌ها:

- Plane: یک ماتریس دو بعدی که نقشه محیط را نشان می‌دهد؛ اعداد ۱ نشان‌دهنده‌ی سلول‌های آزاد و اعداد ۰ نشان‌دهنده‌ی موانع (دریاچه) در مسیر بازی هستند.
- AllStates: ماتریسی حاوی تمام حالت‌های ممکن (مختصات) در محیط.
- AllActions: ماتریسی حاوی تمام حرکت‌های ممکن.
- Policy: یک بردار حاوی حرکت انتخابی برای هر حالت بر اساس سیاست (روش عمل) مشخص شده.

- `initialStateActionPair`: یک جفت اولیه حاوی حالت اولیه $[x, y]$ و حرکت اولیه $[dx, dy]$ برای آغاز حلقه.
- `Goal`: حالت هدف/نقطه‌ی مقصدی که عامل هدف آن را تصمیم به رسیدن گرفته است.
- `epsilon`: پارامتر کاوش استفاده شده برای انتخاب عمل‌های ϵ -حریصانه (`epsilon-greedy`).
- `display`: یک متغیر که تعیین می‌کند آیا در حین تولید، حلقه‌ی مسیر بازی را نمایش داده شود یا خیر.
- `qPrediction`: یک متغیر که نشان‌دهنده‌ی این است که آیا تابع ارزش عمل باید با استفاده از سیاست مشخص شده پیش‌بینی شود (۱) یا خیر (۰).

• خروجی‌ها:

- `States`: ماتریسی که حاوی دنباله‌ای از وضعیت‌هایی است که در حین حلقه‌ی مسیر بازی بازدید شده‌اند.
- `Actions`: ماتریسی که حاوی دنباله‌ای از اعمالی است که در هر گام انجام شده است.
- `Rewards`: ماتریسی که حاوی دنباله‌ای از پاداش‌های دریافت شده در هر گام از مسیر بازی است.

• اجرای الگوریتم

- این تابع یک حلقه‌ی بی‌نهایت اجرا می‌کند تا عامل از وضعیت اولیه به سمت وضعیت هدف حرکت کند.
- در هر مرحله از حلقه، موقعیت توپ بر روی نقشه نمایش داده می‌شود (اگر پارامتر `display` روشن باشد).
- اعمالی که عامل انجام می‌دهد، به صورت ϵ -حریصانه انتخاب می‌شوند؛ به عبارت دیگر، بر اساس مقدار `epsilon`، عامل با احتمال $1 - \epsilon$ از سیاست استفاده می‌کند و با احتمال `epsilon` اعمال رندوم (تصادفی) انجام می‌دهد.
- مقدار پاداش در هر گام نیز بر اساس حالت فعلی و اعمال انجام شده تعیین می‌شود.

این تابع بر اساس اطلاعات و تابع پاداش محیط به عنوان یک "محیط بدون مدل" عمل می‌کند؛ به عبارت دیگر، عملکرد محیط (موانع و پاداش‌ها) به صورت غیر قابل تغییر برای عامل تعریف شده است، و اطلاعات تجربی از طریق اجرای حلقه‌ی تولید گرفته می‌شود.

تابع پیاده سازی شده برای بهبود پالیسی

```
function Policy = PolicyImprovementUsingQ(QValues, nActions)
    % PolicyImprovementUsingQ performs policy improvement using Q-values

    % Determine the number of states
    nStates = size(QValues, 1) / nActions;

    % Initialize the policy vector
    Policy = zeros(nStates, 1);

    for state = 1:nStates
        % Extract the Q-values for the current state
        stateQValues = QValues(1 + nActions * (state - 1):nActions * state);

        % Find the index of the maximum Q-value(s)
        maxIndices = find(stateQValues == max(stateQValues));

        % Randomly select an action from the indices with maximum Q-value(s)
        selectedAction = maxIndices(randi(numel(maxIndices))));

        % Assign the selected action to the current state in the policy vector
        Policy(state) = selectedAction;
    end
end
```

به طور کلی، این برنامه در فرایند بهبود سیاست (Policy Improvement) با استفاده از مقادیر Q (Q-values) استفاده می‌شود.

- تابع **PolicyImprovementUsingQ**: این تابع وظیفه بهبود سیاست را با استفاده از مقادیر Q دارد.
- ورودی‌های تابع عبارتند از:
 - **QValues**: ماتریس مقادیر Q که شامل Q-values برای هر جفت وضعیت-عمل است.
 - **nActions**: تعداد عمل‌های ممکن در هر وضعیت.
- خروجی تابع عبارت است از:
 - **Policy**: بردار سیاست که برای هر وضعیت، عملی که باید انجام شود را نشان می‌دهد.
- **تعیین تعداد وضعیت‌ها**: در ابتدا، تعداد وضعیت‌ها به صورت $nStates$ محاسبه می‌شود. این تعداد برابر با تقسیم تعداد ردیف‌های **QValues** بر **nActions** است.
- **مقداردهی اولیه بردار سیاست**: بردار **Policy** به طول $nStates$ به صورت یک بردار ستونی از صفرها مقداردهی اولیه می‌شود.

- **حلقه اصلی:** حلقه for برای هر وضعیت از ۱ تا nStates اجرا می‌شود. در هر مرحله، مقادیر Q مربوط به وضعیت فعلی از QValues استخراج می‌شود و در متغیر stateQValues ذخیره می‌شود.
- **یافتن شاخص بیشینه مقدار Q:** با استفاده از تابع find، شاخص‌هایی که مقادیر بیشینه در stateQValues دارند، استخراج می‌شوند. این شاخص‌ها در بردار maxIndices قرار می‌گیرند.
- **انتخاب تصادفی عمل:** با استفاده از تابع randi و با در نظر گرفتن numel(maxIndices) به عنوان حداکثر مقدار، یک شاخص تصادفی در محدوده بردار maxIndices انتخاب می‌شود. این شاخص انتخاب شده به عنوان عملی که باید در وضعیت فعلی انجام شود، در متغیر selectedAction ذخیره می‌شود.
- **اختصاص عمل انتخاب شده به بردار سیاست:** عمل انتخاب شده در متغیر selectedAction به وضعیت فعلی در بردار سیاست Policy(state) اختصاص داده می‌شود.
- با پایان حلقه اصلی، بردار سیاست Policy به عنوان خروجی تابع برگردانده می‌شود. در اینجا QValues یک ماتریس معتبر با ابعاد $1 \times (nStates * nActions)$ است، که در آن مقادیر Q برای هر جفت وضعیت-عمل به صورت پی در پی ذخیره شده‌اند.

نتایج شبیه سازی پیاده سازی الگوریتم Q-learning بدون حضور دریاچه

ابتدا با توجه به الگوریتم Q-learning که به صورت زیر در کتاب مرجع نیز ذکر شده است، مقدار Q طبق فرمول و کد زیر بروزرسانی میشود.

Q-Learning: Off-Policy TD Control

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal
  
```

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

% Update the Q-value for the current state-action pair using the Q-learning formula

```
Q(indexInPairs) = Q(indexInPairs) + alpha * (Reward + gamma *
max(Q(indexOfNextStateInPairs)) - Q(indexInPairs));
```

باتوجه به شرایط مسئله ، انتظار داریم در صورتی که مانع (دریاچه) وجود نداشته باشد، ابتدا عامل، باید عصا یا دسته با جنس woods که بیشترین فاصله را به صورت default طی میکند را با قدرت ۱ انتخاب کند و سپس هر چه به هدف نزدیک می شود با توجه به فاصله ای که تا هدف دارد، دسته با دقت بالاتر و قدرت کمتر را انتخاب کند تا با کمترین حرکت به هدف برسد . میدانیم هرچه جنس دسته طوری باشه که مسافت بیشتری را طی کند دقت کمتری دارد و به همین دلیل باد اثر بیشتری روی آن میگذارد و همچنین هر چه دسته به سمت دقت بالاتر می رود اثر باد روی آن کمتر می شود.

حال کد اصلی بالا را با در نظر گرفتن پارامتر های زیر برای ۱۰۰۰۰ اپیزود اجرا کرده ایم. نتایج پالیسی به صورت زیر بدست آمده است.

```
gamma = 0.99; % Set the discount factor for future rewards
nEpisode = 10000; % Define the number of episodes for training
epsilon = 0.15; % Define the exploration rate for epsilon-greedy policy

% Initialize the policy randomly by assigning random actions to each state
Policy = randi(nActions, nStates, 1);

alpha = 0.1; % Learning rate for updating Q-values
```

```
optim_policy = Policy'
```

```
optim_policy =
```

```
Columns 1 through 13
```

```
39    7    1    32    18    16    26    19    30    26    36    35    29
```

```
Columns 14 through 26
```

```
30    28    35    38    39    40    39    40    38    40    39    40    39
```

```
Columns 27 through 39
```

```
40    40    40    40    40    39    39    40    40    39    40    39    40
```

```
Columns 40 through 52
```

```
40    40    40    40    40    40    40    40    40    39    40    40    40
```

```
Columns 53 through 65
```

40 39 40 40 39 40 40 40 40 40 39 40 40

Columns 66 through 78

40 40 40 40 40 40 39 40 39 40 39 40 40

Columns 79 through 91

40 40 40 40 40 40 40 30 39 24 40 27 30

Columns 92 through 101

34 11 25 17 28 14 35 7 16 40

با توجه به نتایج بالا و انتظاراتی که داشتیم در زمانی که در فاصله های زیاد مثل حالت های ۳۰ به بالا هستیم باید اکشن ۴۰ (انتخاب دسته چوبی با قدرت ۱) انتخاب میشد و در فاصله های کم باید اکشن های زیر ۱۰ انتخاب می شد که در اینجا این انتظارات برآورده نشده است . بنابراین نیاز است که تعداد اپیزودها و یا مقادیر پارامترها را به گونه ای تغییر بدهیم تا عامل بتواند بهتر فراگیرد. در اینجا ما با افزایش اپیزود به صد هزار، نتایج پالیسی بهینه را به صورت زیر بدست آورده ایم.

```
gamma = 0.99; % Set the discount factor for future rewards
nEpisode = 100000; % Define the number of episodes for training
epsilon = 0.15; % Define the exploration rate for epsilon-greedy policy

% Initialize the policy randomly by assigning random actions to each state
Policy = randi(nActions, nStates, 1);

alpha = 0.1; % Learning rate for updating Q-values
```

```
optim_policy = Policy'
```

```
optim_policy =
```

Columns 1 through 13

0 1 6 9 16 17 18 20 20 20 27 29 30

Columns 14 through 26

30 30 30 38 39 40 40 39 39 40 40 40 40

Columns 27 through 39

40 40 40 40 40 40 40 40 40 40 40 40 40

Columns 40 through 52

40 40 40 40 40 40 39 39 40 40 40 40 40

Columns 53 through 65

40 40 40 40 39 40 39 40 40 40 40 40 40

Columns 66 through 78

40 40 40 40 39 40 40 40 40 40 40 40 40

Columns 79 through 91

40 40 40 40 40 40 40 40 40 40 40 40 40

Columns 92 through 101

40 40 39 40 40 40 40 40 40 40

حال پالیسی بهینه بدست آمده بالا طبق انتظارات ظاهر شده است. در زمانی که فاصله تا هدف نسبتاً زیاد است دسته با کمترین دقت و بیشترین قدرت انتخاب میشود و زمانی که به هدف نزدیک میشویم، دسته ای با قدرت کمتر و دقت بیشتر انتخاب شده است. بنابراین عامل به خوبی توانسته یادبگیرد و با کمترین تعداد ضربه توپ را در داخل هدف قرار بدهد. که در اینجا با نهایتاً ۸ حرکت این کار انجام میشود.

نتایج شبیه سازی پیاده سازی الگوریتم SARSA بدون حضور دریاچه

ابتدا با توجه به الگوریتم SARSA که به صورت زیر در کتاب مرجع نیز ذکر شده است، مقدار Q طبق فرمول و کد زیر بروزرسانی میشود.

Sarsa: On-Policy TD Control

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
 Initialize s
 Choose a from s using policy derived from Q (e.g., ϵ -greedy)
 Repeat (for each step of episode):
 Take action a , observe r, s'
 Choose a' from s' using policy derived from Q (e.g., ϵ -greedy)
 $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
 $s \leftarrow s'; a \leftarrow a';$
 until s is terminal

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

```
% Find the indices of the next state-action pairs (for the next time step)
indexOfNextStateInPairs = find(ismember(AllStateActionPairs(:, [1, 2]),
NewState, 'row'));

a_prime = Policy(find(ismember(AllStates, NewState, 'row'))); %#ok
indexOfNextStateInPairs = indexOfNextStateInPairs(a_prime);

% Update the Q-value for the current state-action pair using the Q-learning
formula
Q(indexInPairs) = Q(indexInPairs) + alpha * (Reward + gamma *
Q(indexOfNextStateInPairs) - Q(indexInPairs));
```

باتوجه به شرایط مسئله ، انتظار داریم در صورتی که مانع (دریاچه) وجود نداشته باشد، ابتدا عامل، باید عصا یا دسته با جنس woods که بیشترین فاصله را به صورت default طی میکند را با قدرت ۱ انتخاب کند و سپس هر چه به هدف نزدیک می شود با توجه به فاصله ای که تا هدف دارد، دسته با دقت بالاتر و قدرت کمتر را انتخاب کند تا با کمترین حرکت به هدف برسد . میدانیم هرچه جنس دسته طوری باشد که مسافت بیشتری را طی کند دقت کمتری دارد و به همین دلیل باد اثر بیشتری روی آن میگذارد و همچنین هر چه دسته به سمت دقت بالاتر می رود اثر باد روی آن کمتر می شود.

حال کد اصلی بالا را با در نظر گرفتن پارامتر های زیر برای ۱۰۰۰۰ اپیزود اجرا کرده ایم. نتایج پالیسی به صورت زیر بدست آمده است.

```
gamma = 0.99; % Set the discount factor for future rewards
nEpisode = 100000; % Define the number of episodes for training
epsilon = 0.15; % Define the exploration rate for epsilon-greedy policy

% Initialize the policy randomly by assigning random actions to each state
Policy = randi(nActions, nStates, 1);

alpha = 0.1; % Learning rate for updating Q-values

optim_policy = Policy'
```

optim_policy =

Columns 1 through 13

0	2	4	8	16	17	24	18	20	20	28	30	30
---	---	---	---	----	----	----	----	----	----	----	----	----

Columns 14 through 26

30	30	37	30	38	39	15	40	40	40	40	36	40
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 27 through 39

40	40	40	40	40	40	39	40	40	40	40	40	24
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 40 through 52

40	39	39	40	39	39	40	39	40	40	37	40	1
----	----	----	----	----	----	----	----	----	----	----	----	---

Columns 53 through 65

40	39	40	40	36	40	40	32	2	40	38	25	38
----	----	----	----	----	----	----	----	---	----	----	----	----

Columns 66 through 78

38	12	4	25	23	34	20	29	21	33	6	28	26
----	----	---	----	----	----	----	----	----	----	---	----	----

Columns 79 through 91

32	15	18	34	32	32	9	2	3	39	4	14	23
----	----	----	----	----	----	---	---	---	----	---	----	----

Columns 92 through 101

18	13	10	11	29	38	16	34	26	37
----	----	----	----	----	----	----	----	----	----

چیزی که در اینجا قابل مشاهده است این است که همانند روش Q-learning برای رسیدن به پالیسی بهینه نیاز است که تعداد اپیزود ها را افزایش دهیم در اینجا شبیه سازی را با ۱۰۰۰۰۰ اپیزود اجرا میکنیم. نتایج شبیه سازی و پالیسی بهینه به دست آمده به صورت زیر است.

```
gamma = 0.99; % Set the discount factor for future rewards
nEpisode = 100000; % Define the number of episodes for training
epsilon = 0.15; % Define the exploration rate for epsilon-greedy policy

% Initialize the policy randomly by assigning random actions to each state
Policy = randi(nActions, nStates, 1);

alpha = 0.1; % Learning rate for updating Q-values
```

```
optim_policy = Policy'
```

```
optim_policy =
```

Columns 1 through 13

0	3	5	10	16	7	19	20	20	20	27	28	29
---	---	---	----	----	---	----	----	----	----	----	----	----

Columns 14 through 26

30 30 30 37 39 38 39 40 40 40 40 40 40

Columns 27 through 39

40 40 40 40 40 40 40 40 40 40 40 40 39

Columns 40 through 52

40 40 40 40 40 40 40 40 40 40 39 40 40

Columns 53 through 65

40 40 40 40 40 40 40 40 40 40 39 40 40

Columns 66 through 78

40 40 40 40 40 39 40 40 40 40 39 40 39

Columns 79 through 91

40 40 40 40 40 40 40 40 39 40 40 40 40

Columns 92 through 101

40 40 40 40 40 40 40 40 40 40

حال پالیسی بهینه بدست آمده بالا طبق انتظارات ظاهر شده است. در زمانی که فاصله تا هدف نسبتاً زیاد است دسته با کمترین دقت و بیشترین قدرت انتخاب میشود و زمانی که به هدف نزدیک میشویم، دسته ای با قدرت کمتر و دقت بیشتر انتخاب شده است. بنابراین عامل به خوبی توانسته یادبگیرد و با کمترین تعداد ضربه توپ را در داخل هدف قرار بدهد. که در اینجا با نهایتاً ۸ حرکت این کار انجام میشود.

قسمت دوم سوال

2. نحوه‌ی انتخاب پاداش، حالات و اکشن‌ها را شرح دهید؟

این قسمت را به صورت کامل در قسمت اول سوال به همراه نحوه تعریف آنها در کد توضیح داده شده است.

قسمت سوم سوال

تفاوت عملکرد این دو روش را شرح داده و مشخص کنید که با توجه به نتایج به دست آمده، به نظر شما کدام روش عملکرد بهتری دارد؟

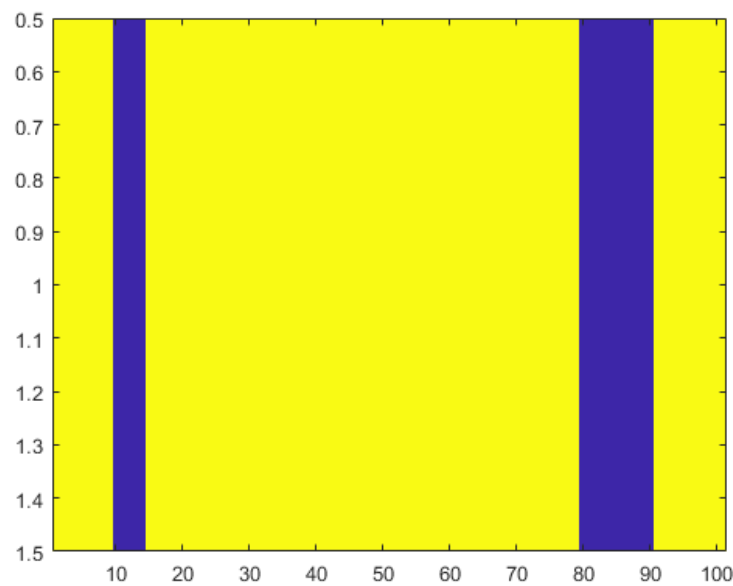
طبق نتایج بالا مشخص است که هر دو روش ممکن است جواب خوبی بدهد اما چیزی که مهم است این است که در روش Q-learning با توجه به این که تابع ارزش-عمل به صورت آفلاین (بیشینه گیری از تابع ارزش-عمل) بروز رسانی میشود ممکن است نتایج بهتری بدهد اما زمان آموزش آن نیز بیشتر طول میکشد و زمان اجرای آن در مقایسه با روش SARSA متفاوت است. بنابراین با توجه به این که روش SARSA به صورت آنلاین تابع ارزش-عمل را به روز میکند و تنها به دو خانه از تابع ارزش عمل نیاز ندارد تا بروز شود (در روش Q-learning نیاز است تا بیشینه یک بردار اندازه گیری شود) بنابراین زمان اجرای آن کوتاه تر است اما به دیتای نسبتا بیشتری نیاز دارد به دلیل آن که دیدی که از محیط دارد سطحی تر است.

قسمت چهارم سوال

برای پیاده سازی این دریاچه ها در زمین نیاز است که در جایی که زمین بازی را تعریف می کنیم کد آن به صورت زیر تغییر پیدا کند.

Section for setting up the game plane and displaying it

```
game_plane_size = 101; % Define the size of the game plane
Plane = ones(1, game_plane_size); % Initialize the game plane with all free spaces (1:
free, 0: barrier)
Plane(1,10:14) = 0;
Plane(1,80:90) = 0;
imagesc(Plane); % Display the game plane as an image
```

در اینجا فرض شده است که در فاصله های ۱۰ تا ۱۴ و ۸۰ تا ۹۰ متری هدف دریاچه وجود دارد و اگر توپ داخل این مناطق قرار بگیرد طبق تعریفی که از محیط در قسمت اول نیز کرده بودیم، عامل امتیاز ۱۰- گرفته و به فاصله ۱۰۰ متری برخواهد گشت.

حال پالیسی بهینه بدست آمده توسط دو الگوریتم خواسته شده را بدست می آوریم.

پالیسی بهینه بدست آمده با یک میلیون اپیزود در روش Q-learning

```
optim_policy = Policy'
```

```
optim_policy =
```

```
Columns 1 through 13
```

```
0    1    5   10   15   15    7   18   20   37   29   38   38
```

```
Columns 14 through 26
```

```
7    15   17   20   20   20   20   27   26   26   26   27   27
```

```
Columns 27 through 39
```

```
28   29   29   30   29   30   30   38   37   39   38   38   37
```

```
Columns 40 through 52
```

```
38   38   39   39   39   39   40   40   40   40   40   40   39
```

Columns 53 through 65

40	40	40	40	40	40	40	40	39	39	40	40	40
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 66 through 78

40	40	40	40	40	40	40	40	40	40	39	5	7
----	----	----	----	----	----	----	----	----	----	----	---	---

Columns 79 through 85

40	17	19	20	33	26	20
----	----	----	----	----	----	----

با توجه به این که تعداد حالت هایی که ربات میتواند در آن قرار بگیرد $85 = 101 - 5 - 11$ تا است بنابراین شماره حالت ها دیگر با فاصله ای که از هدف دارند مقداری متفاوت است اما موضوع همان است و ما انتظار داریم در صورتی که نزدیک دریاچه باشیم، دسته ای انتخاب بشود که بتواند طول دریاچه را عبور کند حتی اگر دقت خوبی نداشته باشد. این نکته نیز حائز اهمیت است که در اینجا نباید دریاچه ها به گونه ای تعریف بشوند که با هیچ ضربه ای عامل نتواند از آن عبور کند. به عنوان مثال در اینجا نباید از ۱۸ متر بیشتر دریاچه داشته باشیم. در حالی که ما در اینجا دو دریاچه ۵ متری و ۱۱ متری در فاصله های ۱۰ تا ۱۴ و ۸۰ تا ۹۰ متری هدف قرار دادیم.

پالیسی بهینه بدست آمده با صد هزار اپیزود در روش SARSA

با تعریف موانع همانند روش Q-learning نتایج بدست آمده به صورت زیر می باشد.

```
optim_policy = Policy'
```

```
optim_policy =
```

Columns 1 through 13

0	1	4	8	16	20	19	19	20	30	30	6	7
---	---	---	---	----	----	----	----	----	----	----	---	---

Columns 14 through 26

10	17	17	18	20	20	20	17	20	25	26	27	28
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 27 through 39

30	30	30	30	30	30	36	37	37	39	38	38	38
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 40 through 52

39	38	39	39	39	40	40	40	40	40	39	40	40
----	----	----	----	----	----	----	----	----	----	----	----	----

Columns 53 through 65

39 39 40 40 40 40 39 40 40 40 40 40 40

Columns 66 through 78

40 40 40 40 40 40 40 40 40 40 40 13 9

Columns 79 through 85

16 17 18 20 20 32 20

در اینجا نیز پالیسی بدست آمده طبق انتظار ما شده است. به عنوان مثال در صورتی که در ۹۱ تا ۱۰۰ متری هدف باشیم هر چه به ۹۰ نزدیک میشویم دسته ای انتخاب میشود که قدرت بیشتری داشته باشد و یا ابتدا دسته ای با دقت بیشتر و قدرت کمتر انتخاب شده تا به نزدیکی دریاچه برسیم و مطمئن باشیم توپ قبل از دریاچه می ایستد و سپس نزدیک دریاچه به توپ ضربه قوی تر با دقت کمتری خواهیم زد.

طبق نتایج بالا مشخص است که هر دو روش ممکن است جواب خوبی بدهد اما چیزی که مهم است این است که در روش Q-learning با توجه به این که تابع ارزش-عمل به صورت آفلاین (بیشینه گیری از تابع ارزش-عمل) بروز رسانی میشود ممکن است نتایج بهتری بدهد اما زمان آموزش آن نیز بیشتر طول میکشد و زمان اجرای آن در مقایسه با روش SARSA متفاوت است. از طرفی در صورتی که این موانع متغیر باشد آنگاه روش SARSA جواب بهتری در زمان آموزش کمتر خواهد داد به دلیل آن که این روش به صورت آنلاین تابع ارزش عمل را بروز میکند.