DEVELOPING APPROACHES FOR TESTING AUTONOMOUS CARS

Deniz Cangı denizcangi@sabanciuniv.edu
Computer Science and Engineering, Sophomore

G. Görkem Köse gkose@sabanciuniv.edu
Computer Science and Engineering, Sophomore

Cemal Yılmaz
Computer Science and Engineering, FENS

Abstract

This research project focused on reinforcement learning on randomized models. All the models considered have in common that they all can be defined as Finite States Machines and strongly connected directed graphs.

In the first phase of the project, the model is generated based on the given number of nodes, the density of the graph, the possible number of actions. Gradually, a different number of error states are determined randomly and the reinforcement learning algorithm is tested on finding the shortest path to these erroneous states.

Keywords: Reinforcement Learning, Finite State Machine, Erroneous State, Q-Learning, Reward

1 Introduction

Reinforcement Learning is learning how to associate situations to actions in order to maximize the reward in turn of action. The implemented learning algorithm initially has no information about the data set, instead, it discovers the data set and takes the actions that yield the biggest reward by trial-and-error (Sutton, 2017).

The project aims to implement a Q-learning algorithm that finds the erroneous states through the shortest path in a Finite-State machine. Q-learning is a value-based algorithm to learn how to take an action optimally in the given environment (Watkins,1989). Q-learning technique uses the Q-function to select the optimal action from one state to another while maximizing the reward. To achieve this goal, the Q-learning algorithm uses a table called Q-table. The aim of the Q-table is to store the reward of each action from the corresponding state and find the maximum of them which leads to the best action.

Roughly speaking, Finite State Machine (FSM) sequentially reads a string input and accepts or rejects its computation. The reason behind the FSM used as a model for reinforcement learning in this project is that Finite-State Machine is a well-established abstraction for computational modeling (Pigem, 2013). In this context, randomly generated Finite-State Machines are specialized as strongly connected directed graphs since we try to avoid the possible back edges that would cause a vicious cycle in the model in order to prevent the algorithm from stuck, where a graph is strongly connected directed if there is a path from each node to every other node and the edges have a direction. In this context, the graph has weighted edges since the edges also represent the possible actions to take from each node and the input symbols in the finite state machine.

To implement the project, a programming language called Python is used.

2 Progress

2.1 Randomly Generating The Finite-State Machine

In the first phase of the project, the aim was creating a random finite-state machine with given the number of states, the density of the finite-state machine, the possible number of actions to take from each state. In this project, the finite-state machine is specialized as a strongly connected directed graph since we aimed to avoid any vicious cycles caused by a back edge in the finite state machine that prevents the algorithm to get stuck in some state and not explore the other states. The reason behind why the finite-state machine is generated with a given density, where the density is defined as the ratio of the edges in the graph and the maximum number of possible edges, is that to test the reinforcement learning algorithm in both sparse graphs where the number of edges is relatively less and in dense graphs where the number of edges is relatively large. But still, the density of the graph should be large enough to make the states of the finite-state machine strongly connected, so we tested the reinforcement algorithm on a finite-state machine that holds these above conditions of strongly connectedness and minimum density. To generate such a finite state machine, we use an open-source Python package called Network X which enables us to create and manipulate a complex network and represent the final graph visually.

In order to make sure that the generated finite-state machine is a strongly connected graph, we first created a cycle that consists of nodes of the given number as shown in Figure 1.

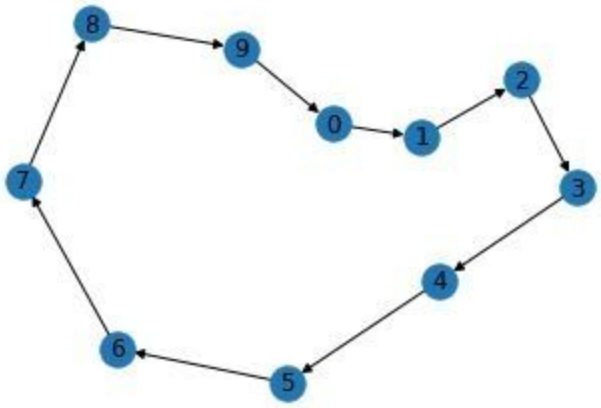

Figure 1: Cycle

Then, we add edges between randomly picked two states if these states are not already connected and that newly created edge does not overshoot the limit of the maximum number of actions to take for each of these states. New edges are continuously added to the graph until we reach the number of edges determined by the density which is implemented as shown in Figure 2.

```
numPossibleEdges=numStates*(numStates-1)

#the number of edges in the graph, that depends on density
numOfEdges=int(numPossibleEdges*density)

G=nx.cycle_graph(numStates, create_using=nx.DiGraph) #creating the cycle

#initially number of edges is equal to the number of states
#since we have just created the cycle
numEdges=numStates

#filling the alphabet of the FSM, the number of symbols in the alphabet
#depends on the maximum number of actions to take from each state
alphabet=[i for i in range(0,maxNumAction)]

#adding an edge between randomly picked two states
while numEdges!=numOfEdges:
    firstNode=randint(0,numStates-1)
    while(True):
        secondNode=randint(0,numStates-1)
        if secondNode!=firstNode:
            break
    if G.has_edge(firstNode, secondNode)==False and G.out_degree(firstNode)< maxNumAction:
        G.add_edge(firstNode,secondNode)
        numEdges=numEdges+1
```

Figure 2: Implementation of an unweighted strongly connected directed graph

Each possible action to take from the states are labeled using the alphabet of the finite-state machine, therefore all edges have a random weight, where the edges originating from the same state have obviously different weights in order to satisfy the deterministic property of the finite-state machine, which is implemented as shown in Figure 3.

```
#assigning random weights to each edge
#in order to satisfy the deterministic property of FSM
for i in G.nodes():
    outedges=[]
    for j in G.neighbors(i):
        found=False
        while found==False:
            random=randint(0,len(alphabet)-1)
            if alphabet[random] not in outedges:
                G[i][j]["weight"]= alphabet[random]
                outedges.append(alphabet[random])
                found=True
```

Figure 3: Weighted graph

In the end, we obtain a finite-state machine which is represented as a strongly connected directed graph on purpose, as shown in Figure 4.

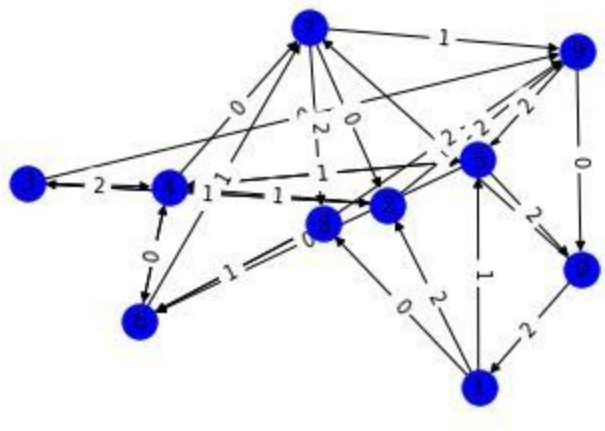

Figure 4: Final Finite-State Machine

2.2 Determining The Erroneous State(s)

In the second phase, after the random finite-state machine is generated, the erroneous state(s) that will hopefully be explored by the Reinforcement Learning algorithm are randomly determined. For initialization, the start state is chosen as the state which is labeled as 0. And the erroneous state(s) are chosen randomly from a state set that excludes the start state and all the states that are neighbors of the start state in order to give a chance and test the Reinforcement Learning algorithm to discover an erroneous state not that close to the start state. Yet, since the Finite-State Machine is strongly connected, that means each node has a valid path that goes to every other node, determining an erroneous state more than 2 states away from the start state is not always guaranteed. In the third phase of the project, the

erroneous state is chosen as a single state in order to see if the Reinforcement Learning algorithm works for our purpose on an abstract model such as a FSM and in the last phase, the erroneous states are chosen as multiple states in order to discover the limits of Reinforcement Learning algorithm. Determining the erroneous states is implemented as shown in Figure 5.

```
errors=[]
for i in range(numOfGoals): #numOfGoals is the number of erroneous states
    #created a list of nodes which are not the neighbors of start state
    notNeighbors=list(nx.non_neighbors(G,starts[i]))
    randomError = randint(0, len(notNeighbors)-1)
    errorState = notNeighbors[randomError]
    while errorState in errors:
      randomError = randint(0, len(notNeighbors)-1)
      errorState = notNeighbors[randomError]
    if errorState not in errors:
      errors.append(errorState)
```

Figure 5: Determining the erroneous state(s)

Furthermore, just to increase the visual understanding, the erroneous states are represented as red and the other states are represented as blue as shown in Figure 6.

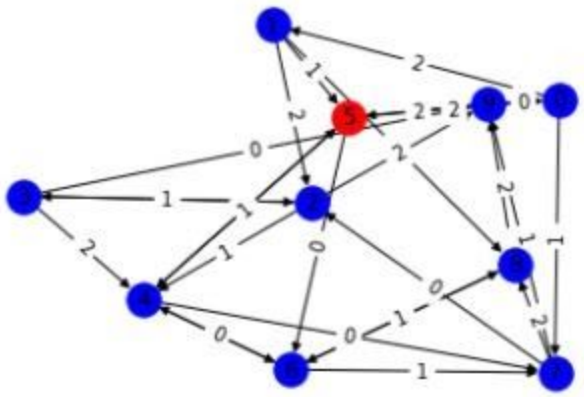

Figure 6: Erroneous state is marked with the color red

2.3 Learning a Single State

In the third phase of the project, the aim was to implement a basic reinforcement learning algorithm with Q-learning technique. To achieve this, we defined a matrix M, which defines the reward system in the algorithm. The number of rows and the number of columns in the matrix is equal to the number of states in the randomly generated finite-state machine. Each row and column represents one state of the machine and each cell in the matrix corresponds to the edge between these two states. Initially all the values in the matrix is equal to -1 which indicates there is no path between these two states. We traced all the edges in the graph if there is an edge between two states and the successive state is the

erroneous state, we changed the value of the cell, which corresponds to the edge from the preceding state to the successive state, to 100. If there is an edge between two states but the successive state is not the erroneous state, then we change the cell to 0. This implementation is shown in Figure 7. Sample final matrix M can be seen in Figure 8.

```
#edges is defined as the all edges in the graph.
edges=G.edges()
#matrix size is equal to the number of states that we've initialized
MATRIX_SIZE=number_states
#created a matrix M which is all 1's with a size MATRIX_SIZE x MATRIX_SIZE
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
#multiply all the cells in the matrix with -1
M *= -1
#trace all the edges in the graph
for point in edges:
    #if the second state is an error state then change the cell in matrix
    #to 100
    if point[1] == goal:
        M[point] = 100
    #if there is an edge between two states but the second state is not the
    #error state then change the cell in matrix to 0.
    else:
        M[point] = 0
```

Figure 7: Implementing the reward matrix M

```
[[ -1.   0.  -1.  -1.  -1.  -1.  -1.   0.  -1.  -1.]
 [ -1.  -1.   0.  -1.  -1. 100.  -1.  -1.   0.  -1.]
 [ -1.  -1.  -1.   0.   0.  -1.  -1.  -1.  -1.   0.]
 [ -1.  -1.   0.  -1.   0.  -1.  -1.  -1.  -1.   0.]
 [ -1.  -1.  -1.  -1.  -1. 100.   0.   0.  -1.  -1.]
 [  0.  -1.  -1.  -1.   0. 100.   0.  -1.  -1.  -1.]
 [ -1.  -1.  -1.  -1.   0.  -1.  -1.   0.   0.  -1.]
 [ -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.   0.   0.]
 [ -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.  -1.   0.]
 [  0.  -1.  -1.  -1.  -1. 100.  -1.  -1.  -1.  -1.]]
```

Figure 8: Sample reward matrix M

Then, we implemented the Q-table which stores the reward of each action. In the project, the Q-table is implemented as a matrix. Each row represents the preceding states and each column represents the succeeding states, each cell. The intersection of the two states stores the reward for the action from the preceding state to a successive state. The number of rows and number of columns in the matrix is equal to the number of states in the finite-state machine. Initially all the values in the matrix is equal to 0. The implementation of the Q matrix is shown in Figure 9.

```
#created the Q table which stores the reward of each action for
#the corresponding states, it has size of MATRIX_SIZE x MATRIX_SIZE
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
```

Figure 9: Initialization of Q-table

Then, we created a function called available_actions, which take a state and return the available actions from that state. First, it gets the given state's row from the matrix M, which shows the edges

between the states, then from that row, it gets the edges which are greater than or equal to zero, get the successive states and return this list. The implementation of the available_actions function can be found in Figure 10.

```
# Determines the available actions for a given state
def available_actions(state):
    #get the row of the state from the matrix M
    current_state_row = M[state]
    #create a list available_action, it gets the cells which are greater
    #than or equal to zero and get the successive state from it
    #and create a list of available states for the given state
    available_action = np.where(current_state_row >= 0)[1]
    return available_action
```

Figure 10: Implementation of available_action function

In the next step, we created a function called sample_next_action which gets a list and randomly chooses an element from the list. We create this function to choose a random state from available states from one state. Then we get the available states of the initial state and put the available_action list to the sample_next_action and randomly chose a state. The implementation is shown in Figure 11.

```
# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

#get the available actions from the initial state
available_action = available_actions(initial_state)
#find a random state from the available_action list.
action = sample_next_action(available_action)
```

Figure 11: Implementation of sample_next_function function

Then, we defined the gamma, discount factor of the learning algorithm. Discount factor determines the weight given to future rewards in the Q function. We took the discount factor as 0.75. Next, we created a function called update which takes the current state, generated random action from the sample_next_action and the discount factor, which is initialized in our code as gamma.

This function randomly chooses a path from the action state and updates the Q table according to the path that it has chosen. Then we tried this function by sending the parameters initial state, random action and gamma. The implementations are shown in Figure 12.

```
#discount factor
gamma = 0.75

# Updates the Q-Matrix according to the path chosen
def update(current_state, action, gamma):

    max_index = np.where(Q[action] == np.max(Q[action]))[1]
    if max_index.shape[0] > 1:
            max_index = int(np.random.choice(max_index, size = 1))
    else:
            max_index = int(max_index)
    max_value = Q[action, max_index]
    Q[current_state, action] = M[current_state, action] + gamma * max_value
    #if chosen states have edges inbetween them, then it updates it with
    #reward, if it's not it updates the Q table with 0.
    if (np.max(Q) > 0):
            return(np.sum(Q / np.max(Q)*100))
    else:
            return (0)
```

Figure 12: Implementation of the update function

In the next step, we created a list called scores and start the training phase. Training iterates for a hundred times the number of states in the graph. It consistently chooses a current state, called current_state, randomly, find the available states, called available_action, from that current state, choose a random action, called action, from that available states and update the Q table and return the score of the current state and action. Then the function append this score to the scores list. At the end of the function, the Q table is trained. The implementation and a sample final result of the Q- table can be seen below in Figure 13 and 14.

```
#scores list
scores = []
for i in range(number_states*100):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

print("Trained Q matrix:")
print(Q / np.max(Q)*100)
```

Figure 13: Training phase

```
Trained Q matrix:
[[  0.          75.           0.           0.           0.
    0.           0.          56.25         0.           0.        ]
 [  0.           0.          56.25         0.           0.
  100.           0.           0.          56.25         0.        ]
 [  0.           0.           0.          56.25        75.
    0.           0.           0.           0.          75.        ]
 [  0.           0.          56.25         0.          75.
    0.           0.           0.           0.          75.        ]
 [  0.           0.           0.           0.           0.
  100.          56.25        56.25         0.           0.        ]
 [ 56.25         0.           0.           0.          75.
   99.96955819  56.25         0.           0.           0.        ]
 [  0.           0.           0.           0.          75.
    0.           0.          56.25        56.25         0.        ]
 [  0.           0.          56.25         0.           0.
    0.           0.           0.          56.25        75.        ]
 [  0.           0.           0.           0.           0.
    0.          56.25         0.           0.          75.        ]
 [ 56.25         0.           0.           0.           0.
  100.           0.           0.           0.           0.        ]]
```

Figure 14: Sample trained Q matrix

In the next step, we did the testing part, to find the most efficient path from the initial state to the error state. First, we created a list called steps and append the initial state to the list. For testing we used the Q table that we have trained. Roughly speaking, it looks up to the Q table and it chooses the indices of the next successive states which has highest reward in the Q table. In the while loop, while the current_state is not the error state chosen randomly, it iterates and find the steps from initial state to the error state one by one. In the end in the steps list have the steps from initial state to the error state. Then, we printed it and find out the actions that brings the initial state to the error state and called it the Language. Implementation and sample results are given in Figure 15 and Figure 16.

```
# Testing part of the Q-learning
current_state = initial_state
#keep track of steps
steps = [current_state]

while current_state != goal:

    next_step_index = np.where(Q[current_state] == np.max(Q[current_state]))[1]
    if next_step_index.shape[0] > 1:
            next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
            next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

print("Language:")
Language=[]

for i in range(len(steps)-1):
    Language.append(G[steps[i]][steps[i+1]]['weight'])
print(Language)
```

Figure 15: Finding the shortest path

```
Most efficient path:
[0, 1, 5]
Language:
[2, 1]
```

Figure 16: Sample result

In the last step we plot the training phase of the Q-learning. In the x axis of the graph, the number of iterations is plotted and in the y axis reward gained at each iteration is plotted. The plotted sample chart is given in Figure 17.

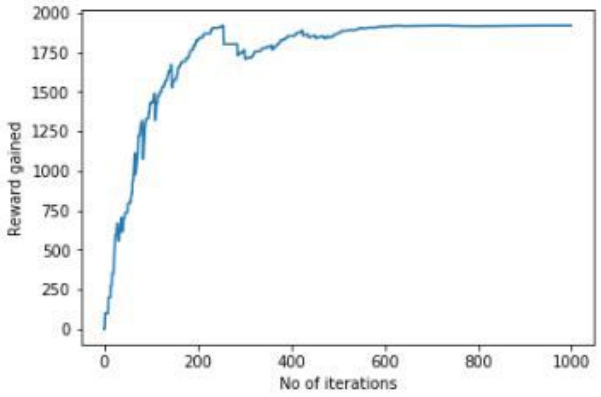

Figure 17: No of iterations vs Reward gained graph

2.4 Learning Multiple States

In the last phase of the project, multiple erroneous states are determined randomly as explained in section 2.2 and the aim is to reach the last erroneous state through all of the other erroneous states using the shortest path possible. To implement this, a "Divide & Conquer" strategy is carried out. The algorithm is as follows when the number of erroneous states is n: We first explore the shortest path starting from the initial state to the erroneous state 1 using the technique explained in section 2.3, add this path into a list so that we can update the path when we reach the successive erroneous state and construct a list called language that stores the label of edges through the shortest path that is recently discovered by the algorithm. Then, starting from the erroneous state 1, we explore the shortest path to erroneous state 2, add this subpath into the list, update the language list as well. This process will repeatedly proceed until we reach the last erroneous state, which is erroneous state n. Obviously, this will be repeated for the number of erroneous state times. The implementation is shown in Figure 18.

```
for i in range(numofGoals): #numOfGoals is the number of erroneous states
    first_steps, first_language=findingThePath(G,number_states,errors[i],initials[i])
    initials.append(errors[i])
    joinedList=joinedList+first_steps[1:]
    joined_language=joined_language+first_language

print(joinedList)
print(joined_language)
```

Figure 18: Implementation of the learning multiple erroneous states

3 Discussion and Conclusion

As a result of our studies, in a strongly connected directed graph that can be interpreted as a Finite-State Machine, we implemented a Reinforcement Learning algorithm that finds the shortest path from a given initial state to a randomly determined erroneous state.

Indeed, before developing approaches for testing autonomous cars, we decided to test the reinforcement learning algorithm on a much more abstract model such as a Finite-State Machine. Future of this project may be focused on autonomous cars and definitely, section 2.4 needs some improvements and more efficient algorithms since in the future, for autonomous cars, the erroneous factors will be multiple.

References

Pigem, B. (2013). *Learning Finite-State Machines: Statistical and Algorithmic Aspects.* Barcelona, Spain.

Sutton, R., Barto, A. (2017). *Reinforcement Learning: An Introduction.* London, England.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. Thesis, University of Cambridge, England.