ICS2210-SEM2-A-1819

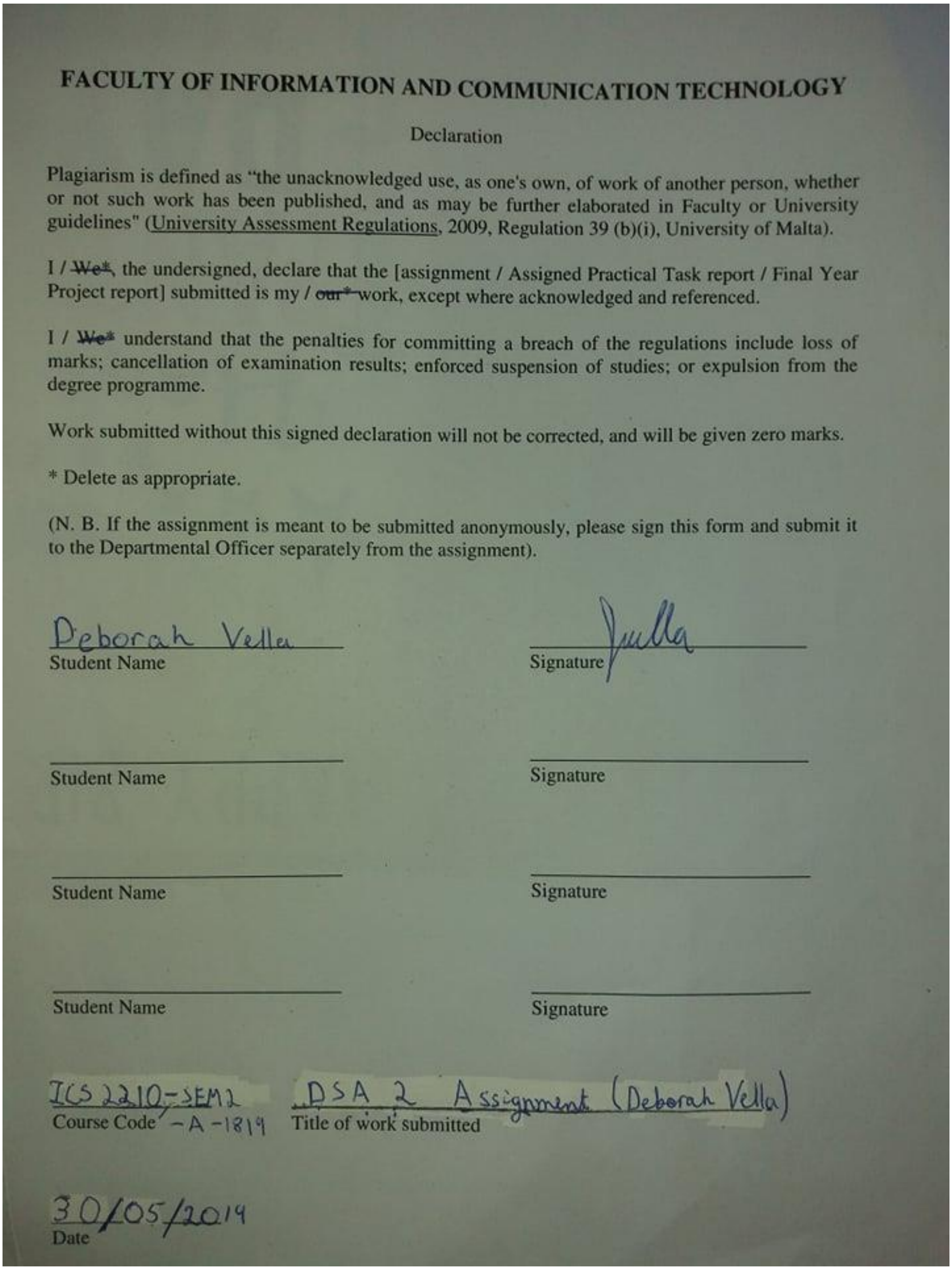
Data Structures and Algorithms 2 Assignment

Name: Deborah Vella

Course: Artificial Intelligence

Table of Contents

Plagiarism Declaration Form



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / ~~We*~~, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / ~~our*~~ work, except where acknowledged and referenced.

I / ~~We*~~ understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Deborah Vella	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature
ICS2210-SEM2-A-1819	DSA 2 Assignment (Deborah Vella)
Course Code	Title of work submitted

30/05/2019
Date

Statement of Completion

Item	Completed (Yes/No/Partial)
Constructed the basis automaton	Yes
Computed the depth of A	Yes
Minimised A to obtain M	Yes
Computed the depth of M	Yes
Implemented Random String Classification	Yes
Implemented Tarjan’s algorithm	Yes
Evaluation	Yes

Question 1

What structure was chosen and why

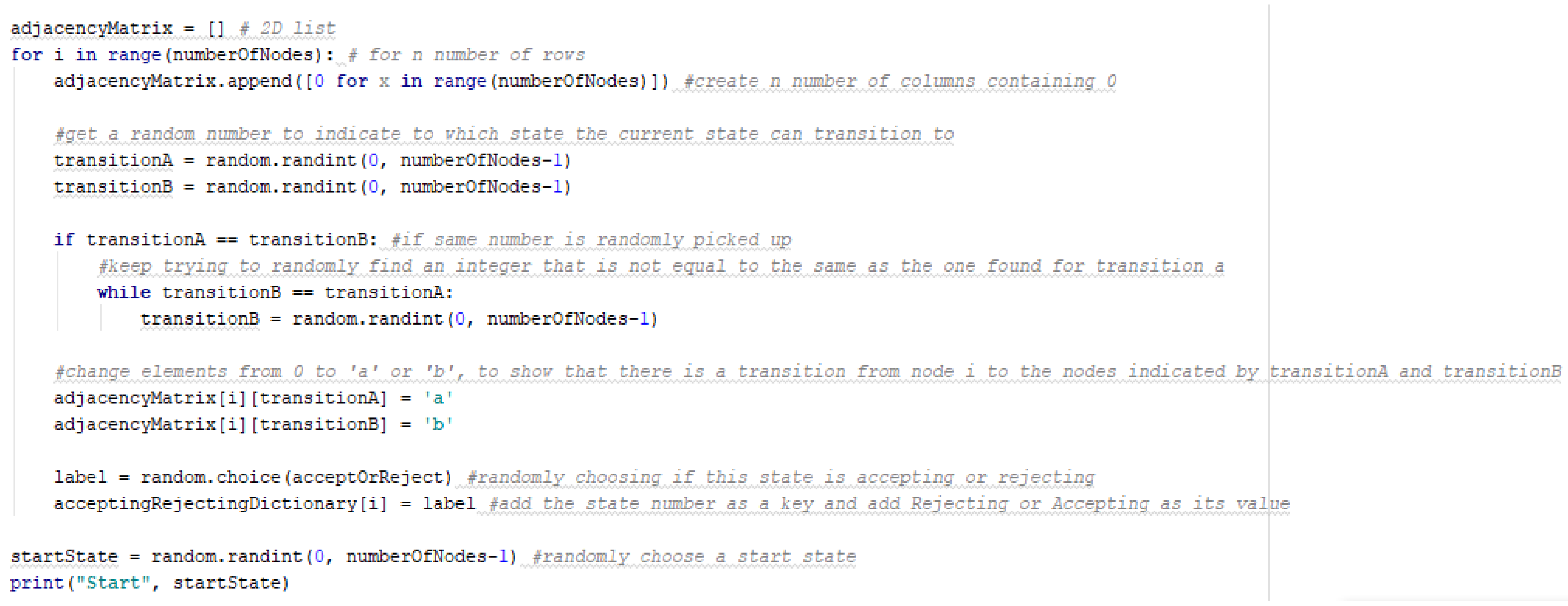
For question 1, an adjacency matrix was chosen to be implemented. The DFA was implemented in an adjacency matrix because it makes it easier to visualize the actual DFA. This is because each index of the rows and columns represent a state in the graph. Furthermore, each element contains a transition which in this case are transitions ‘a’ and ‘b’. It is also very easy to understand, as the position of a transition in the 2D array gives you the state from which the transition is getting out of (i.e row number) and the state it should go to next (i.e column number).

Adjacency matrices also have their disadvantages. In fact, they are not very efficient for big graphs as it takes O(n^2) space and takes O(n^2) time to loop through all the edges (having n being one dimension). On the other hand, it fast in adding new edges, as it takes the position of where it should be added in the matrix, making it use O(1) time complexity.

In contrast, adjacency lists are faster and more space efficient, but are harder to understand or visualize. Which is why an adjacency matrix was chosen.

Implementation overview

In the implementation shown in figure 1 below, a 2D array was chosen to store the adjacency matrix. For each random number of rows, the same numbers of columns are created. It proceeds to randomly choose to which states the two transitions will go, and then add them to the matrix. It finally chooses if the created state is rejected or accepted and the answer is stored in a dictionary. When the DFA is created and stored, the start state is randomly chosen from all possible states and printed out for the user to see.



```python
adjacencyMatrix = [] # 2D list
for i in range(numberOfNodes): # for n number of rows
    adjacencyMatrix.append([0 for x in range(numberOfNodes)]) #create n number of columns containing 0

    #get a random number to indicate to which state the current state can transition to
    transitionA = random.randint(0, numberOfNodes-1)
    transitionB = random.randint(0, numberOfNodes-1)

    if transitionA == transitionB: #if same number is randomly picked up
        #keep trying to randomly find an integer that is not equal to the same as the one found for transition a
        while transitionB == transitionA:
            transitionB = random.randint(0, numberOfNodes-1)

    #change elements from 0 to 'a' or 'b', to show that there is a transition from node i to the nodes indicated by transitionA and transitionB
    adjacencyMatrix[i][transitionA] = 'a'
    adjacencyMatrix[i][transitionB] = 'b'

    label = random.choice(acceptOrReject) #randomly choosing if this state is accepting or rejecting
    acceptingRejectingDictionary[i] = label #add the state number as a key and add Rejecting or Accepting as its value

startState = random.randint(0, numberOfNodes-1) #randomly choose a start state
print("Start", startState)
```

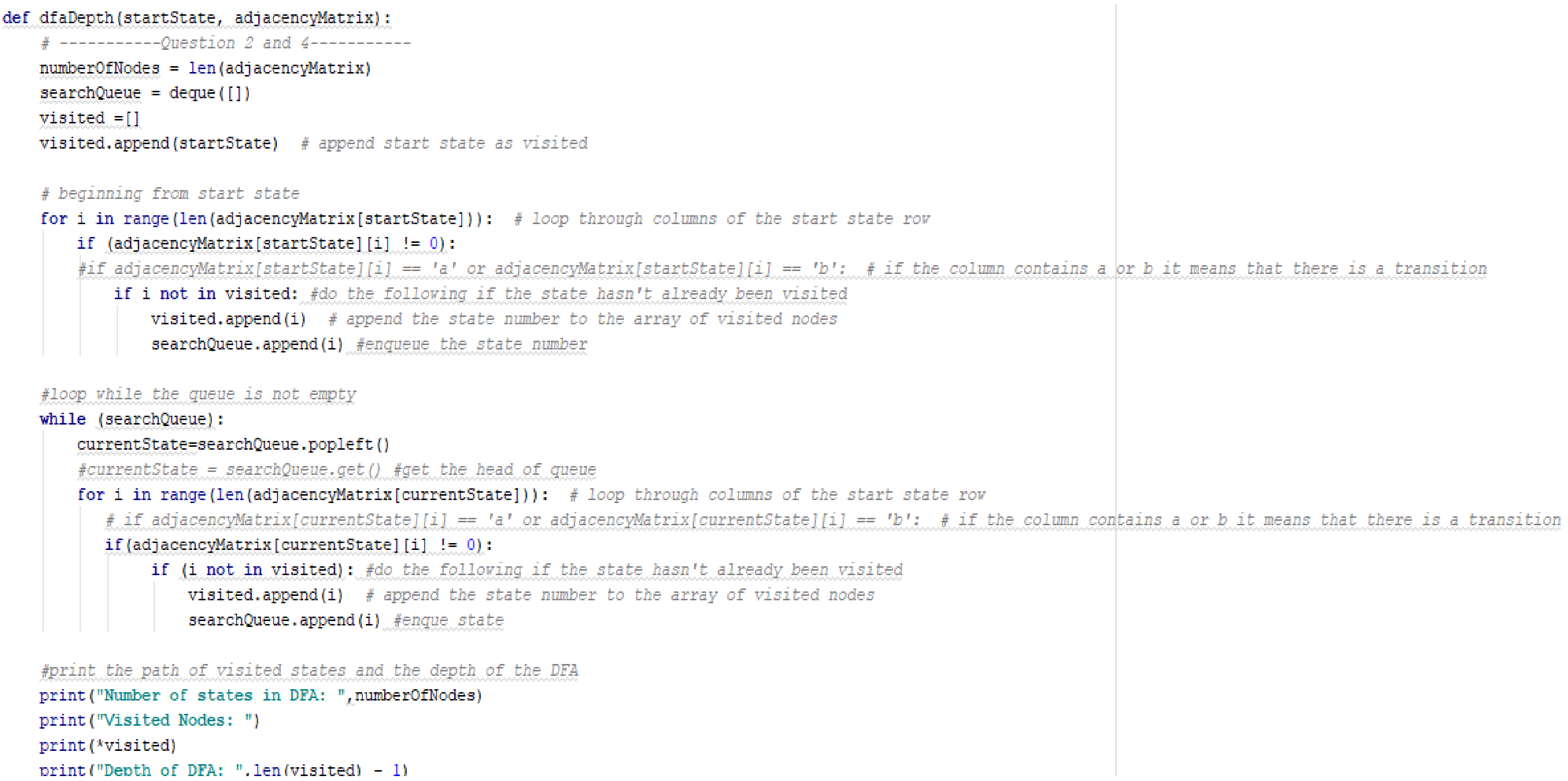
Figure 1: Implementation of the adjacency matrix

Questions 2 and 4

To compute the depth of the DFA, this video about breadth first search was used to understand the algorithm: https://www.youtube.com/watch?v=QRq6p9s8NVg . A breadth first search algorithm was implemented as it eventually gives out the results about the depth of the deterministic finite state automaton, and the visited states.

Implementation Overview

For this algorithm, a queue and an array storing visited states are used. The algorithm starts by visiting the start state, and check to which other states the transitions go to, and add them if they weren't already visited. Then it proceeds to the rest of the DFA. It loops while the queue is not empty. It gets the state at the head of the queue and then loops through the columns of that state's row and checks for transitions 'a' and 'b'. Whenever a transition is met, it checks if the state the transition goes to is already in the list of visited states. If not, insert the state to the list of visited nodes. When all of the above is computed, the depth of DFA and the visited nodes are outputted.



```python
def dfaDepth(startState, adjacencyMatrix):
    # -----------Question 2 and 4-----------
    numberOfNodes = len(adjacencyMatrix)
    searchQueue = deque([])
    visited =[]
    visited.append(startState)  # append start state as visited

    # beginning from start state
    for i in range(len(adjacencyMatrix[startState])):  # loop through columns of the start state row
        if (adjacencyMatrix[startState][i] != 0):
        #if adjacencyMatrix[startState][i] == 'a' or adjacencyMatrix[startState][i] == 'b':  # if the column contains a or b it means that there is a transition
            if i not in visited: #do the following if the state hasn't already been visited
                visited.append(i)  # append the state number to the array of visited nodes
                searchQueue.append(i) #enqueue the state number

    #loop while the queue is not empty
    while (searchQueue):
        currentState=searchQueue.popleft()
        #currentState = searchQueue.get() #get the head of queue
        for i in range(len(adjacencyMatrix[currentState])):  # loop through columns of the start state row
           # if adjacencyMatrix[currentState][i] == 'a' or adjacencyMatrix[currentState][i] == 'b':  # if the column contains a or b it means that there is a transition
           if(adjacencyMatrix[currentState][i] != 0):
                if (i not in visited): #do the following if the state hasn't already been visited
                    visited.append(i)  # append the state number to the array of visited nodes
                    searchQueue.append(i) #enque state

    #print the path of visited states and the depth of the DFA
    print("Number of states in DFA: ",numberOfNodes)
    print("Visited Nodes: ")
    print(*visited)
    print("Depth of DFA: ",len(visited) - 1)
```

Figure 2: Implementation of depth of DFA algorithm

Question 3

To implement Hopcroft algorithm, the pseudo code on Wikipedia was followed: https://en.wikipedia.org/wiki/DFA_minimization

To understand deeper the algorithm, I read the Hopcroft algorithm parts in the following papers:

- https://www.cs.ru.nl/bachelors-theses/2017/Erin_van_der_Veen___4431200___The_Practical_Performance_of_Automata_Minimization_Algorithms.pdf?fbclid=IwAR2RhPY8LIAQlqQx_tx8rmBPqyQv6Mjd59XftsM1Mwxy0-26QMDKSaSVqag
- https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Implementation Over view

The code below is the implemented algorithm of Hopcroft. The comments explain each step.

```
#computes intersaction of two arrays
def intersection(arr1, arr2):
    arr3 = [value for value in arr1 if value in arr2]
    return arr3

#computes the set difference of two arrays
def diff(first, second):
    difference =[value for value in first if value not in second]
    return  difference

#finds the position of a specific element in a 2D array
def index_2d(data, search):
    for i, e in enumerate(data):
        try:
            return i, e.index(search)
        except ValueError:
            pass
    raise ValueError("{} is not in list".format(repr(search)))


def Hopcroft(acceptingRejectingDict, dfa):
    setOfFinalStates = []
    setOfNonFinalStates = []
    alphabet = ['a', 'b']

    #Separating nodes into two sets which are the set of final states and set of
non final states
    for key, value in acceptingRejectingDict.items():
        if value == 'Accepting':
            setOfFinalStates.append(key)
        else:
            setOfNonFinalStates.append(key)

    waitingSet=[]
    partition=[]

    partition.append(setOfFinalStates) #append final and non-final states to
partition P
    partition.append(setOfNonFinalStates)
    waitingSet.append(setOfFinalStates)#append final states to waiting set W

    while(waitingSet): #loop while waiting set is not empty
        set = waitingSet.pop() #remove a set A from waiting set

        for letter in alphabet: #for each possible transition
            X=[] #set this back to an empty array just in case it was populated in
```

```
the previous iteration
        #get the set of states that when processing the transition letter, end up
in one of the states in the array set
            for column in set:#for all states in set
                for row in range(len(dfa)):#find the state that has a transition
letter coming out of it
                    if(dfa[row][column] == letter):
                        if row not in X:
                    # append the row where the transition letter matched (because
the row number is the same as the state number
                            X.append(row)

            count = 0 #keeps track of which position we are currently in, in the
partition array
            for Y in partition:
                intersect = intersection(Y, X) #compute intersection between sets Y
and X
                setDiff = diff(Y,X)#compute set difference Y\X
                if(len(intersect)>0 and len(setDiff)>0): #if both of the sets are
populated with something
                    partition[count] = intersect #replace Y in partition with
intersaction set
                    partition.append(setDiff) #append set difference to partition

                    if(Y in waitingSet):
                        waitingSet.remove(Y) #replace Y in waiting set by the set
differenc and intersaction found above
                        waitingSet.append(intersect)
                        waitingSet.append(setDiff)
                    else: #else append the smallest of the two sets
                        if intersect <= setDiff:
                            waitingSet.append(intersect)
                        else:
                            waitingSet.append(setDiff)
                count+=1

    print("Final Partition: ")
    print(partition)
    return partition
```

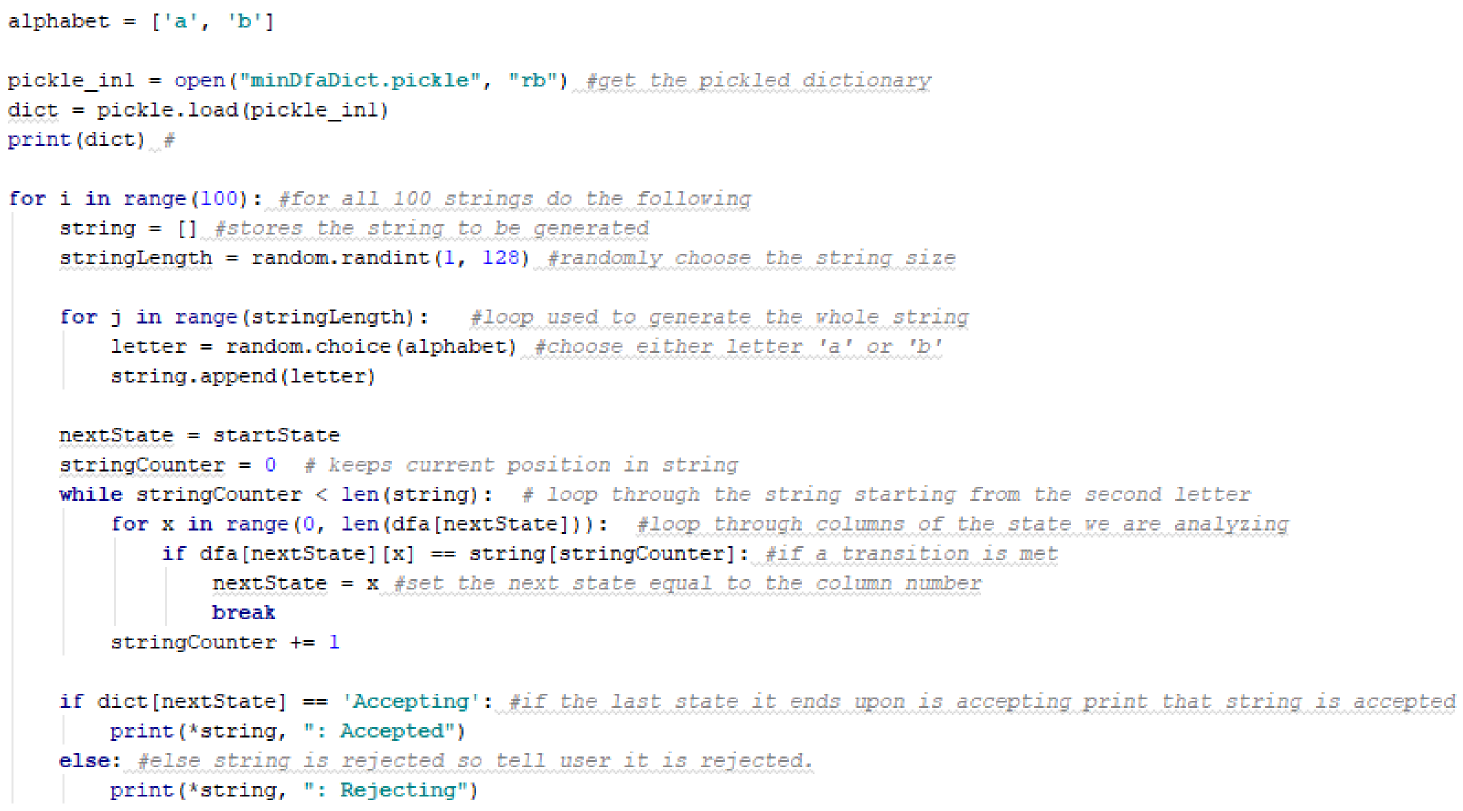
Time complexity

The time complexity of Hopcroft is O(*kn* log *n*) where *k* is the alphabet size and *n* is the number of states. It takes O(log n), because, the algorithm does not search through all the nodes one by one. Instead it splits the sets into two every time, and then it uses only one of these two sets. Therefore, the problem is always being divided into two, and do constant amount of additional computation. It continues refining partitions until it cannot be split any more.

Question 5

Implementation Overview

For all 100 strings that should be generated, generate a random string with a random length, then start the algorithm to check if the string is accepted or rejected. This loops through the whole string, and for each letter it loops through, it gets the next state it transitions to. In the end, the state number is passed as a key in the dictionary and it corresponding values gives back whether the string is accepted or rejected.



```
alphabet = ['a', 'b']

pickle_inl = open("minDfaDict.pickle", "rb") #get the pickled dictionary
dict = pickle.load(pickle_inl)
print(dict) #

for i in range(100): #for all 100 strings do the following
    string = [] #stores the string to be generated
    stringLength = random.randint(1, 128) #randomly choose the string size

    for j in range(stringLength):   #loop used to generate the whole string
        letter = random.choice(alphabet) #choose either letter 'a' or 'b'
        string.append(letter)

    nextState = startState
    stringCounter = 0  # keeps current position in string
    while stringCounter < len(string):  # loop through the string starting from the second letter
        for x in range(0, len(dfa[nextState])):  #loop through columns of the state we are analyzing
            if dfa[nextState][x] == string[stringCounter]: #if a transition is met
                nextState = x #set the next state equal to the column number
                break
        stringCounter += 1

    if dict[nextState] == 'Accepting': #if the last state it ends upon is accepting print that string is accepted
        print(*string, ": Accepted")
    else: #else string is rejected so tell user it is rejected.
        print(*string, ": Rejecting")
```

Figure 3: Code for generating strings and checking if accepted or rejected

Time Complexity

The time complexity of parsing the string and checking if it is correct, is O(*n***m*) + O(1) where *n* is the string size, and *m* is the DFA length (size of one dimension). This is because, there is a for loop, iterating through size *m*, nested inside a while loop of size *n*. To access a state in the dictionary, it takes constant O(1) time.

Question 6

To understand this algorithm, I watched the following video, which explains what strongly connected components are, and how Tarjan's algorithm works to find them:
https://www.youtube.com/watch?v=TyWtx7q2D7Y

Implementation overview

The function loops through all the states and executes the function performing the DFS every time a state is marked as unvisited. Then the function that performs the DFS, calls itself whenever the state we're going to is marked as unvisited. Whenever an element is added on stack, it is marked as True in the array that determines if a node is on stack. Low link values and ids are always updated depending if the DFS is going to start from beginning, or if we are getting the smallest low link value, or popping strongly connected components.

Calling function from main method

```
for i in range(0,numberOfNodes): #loop through states
    if(ids[i] == unvisited): #if state has notbeen visited yet
        dfsForTarjanAlgo(i) #compute depth first search

print("Number of strongly connected graphs: ", len(sccLengths))
largest = max(sccLengths)
print("Largest SCC: " ,largest)
smallest = min(sccLengths)
print("Smallest SCC: " , smallest)
```

Function computing dept first search

```
def dfsForTarjanAlgo(currentNodeId):

    global sccCounter, stack, next, scc, sccLengths
    stack.append(currentNodeId) #push the id of the current node on the stack
    onStack[currentNodeId] = True #mark current node as being on the stack

    ids[currentNodeId] = next #give the positions at currentNodeID an Id value
    lowVals[currentNodeId] = next
    next+=1

    for element in range(len(newDFA[currentNodeId])): #loop through columns of DFA
at row indicated by element
        if(newDFA[currentNodeId][element] !=0): #if it is not equal to zero i.e if
there is a transition
            if (ids[element] == unvisited): #check if the state we're going to is
marked as unvisited
                dfsForTarjanAlgo(element) #call this enforcing recursion
            if onStack[element]: #this line executes after the recursive stage
above is done
                #if the state we came from is on the stack
                lowVals[currentNodeId] =
min(lowVals[currentNodeId],lowVals[element]) #get the minimum between the current
low link value,
                #and the state we have been at.

    if(ids[currentNodeId]==lowVals[currentNodeId]): #check if we are at the
beginning of a scc
        print("Strongly Connected components: ")
        while True:
            element = stack.pop() #pop off the scc from the stack
            print("Node:", element) #print popped element
            scc.append(element) #append it to an array of stringly connected
```

```
components
            onStack[element] = False #mark the node as not being on the stack
anymore
            lowVals[element] = ids[currentNodeId] #giving the popped values the
same ID
            if element == currentNodeId:
                sccLengths.append(len(scc))
                scc=[]
                break

    sccCounter+=1
```

Time Complexity

Tarjan's complexity is O(| V |+ |E |), where V are the vertices and E are the transitions. The outer loop traverses each node only once, taking O(|V|) time. DFS algorithm takes O(|V| + |E|) time, and as Tarjan's algorithm is just a slight modification of DFS, it should also have the same time complexity. The DFS is executed at most once per vertex. For the vertex it has to loop through the length of columns i.e | adjacencyMatrix[vertex] |.

The total in addition to V is: $(\sum_{v \in V} |adjacencyMAtrix[V]|)$

By handshaking lemma, it is equal to O(E).

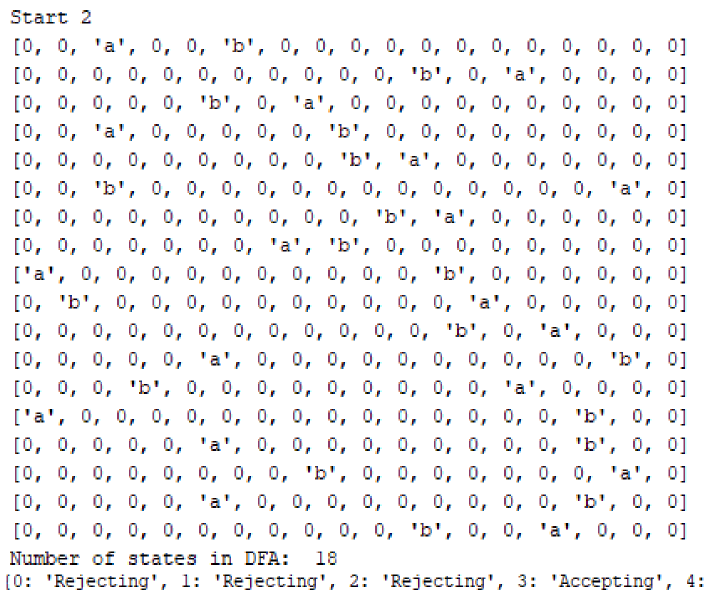
Evaluation

The evaluations where mainly done using small adjacency matrices to be easier to test and track any errors. Hence, each question/algorithm was separately and independently tested, instead of testing one whole program all at once. For most of them, particular adjacency matrices where implemented to be exactly as some DFAs I found when researching. This way I could compare my outputs and results with the papers' results that are certainly correct.

Adjacency Matrix creation

Testing

To test the adjacency matrix creation, the 2D array was outputted on screen to be able to evaluate it.



```
Start 2
[0, 0, 'a', 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 'a', 0, 0, 0, 0]
[0, 0, 0, 0, 0, 'b', 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 'a', 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 'a', 0, 0, 0, 0, 0, 0, 0]
[0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 'a', 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 'a', 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0]
['a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0]
[0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 'a', 0, 0, 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0]
[0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0]
['a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 'a', 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 'a', 0, 0, 0]
Number of states in DFA:  18
[0: 'Rejecting', 1: 'Rejecting', 2: 'Rejecting', 3: 'Accepting', 4:
```

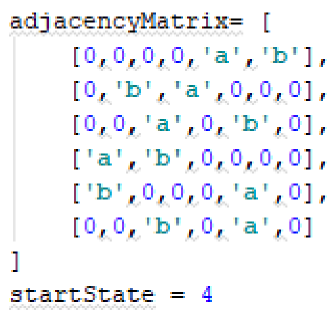
Figure 4: Output of adjacency matrix

Question	Evaluate respective output	Works/Does not work
a. Create n states, where n is a random number between 16 and 64 inclusive.	The output contains 18 nodes which is a random number between 16 and 64	Works
b. Randomly label every state as either accepting or rejecting.	Only part of the dictionary output is shown above. Every state is given one of the two labels randomly.	Works
c. Every one of the n states has two outgoing transitions leading to two other random states; one transition is labelled with the symbol *a*, and the other with the symbol *b*. Transitions from a state to itself are allowed.	All of the 18 states have two transitions coming out of them, and are all one 'a' and one 'b'.	Works
d. Choose any random state as the starting state.	State 2 is chosen a start state. It is a node in the whole DFA.	Works

Depth of DFA

Testing

For testing the depth in its own, I decided to implement a small DFA to start testing with a relatively small automaton. The figures below show the implemented DFA in python and the same DFA as a drawing.



```
adjacencyMatrix= [
        [0,0,0,0,'a','b'],
        [0,'b','a',0,0,0],
        [0,0,'a',0,'b',0],
        ['a','b',0,0,0,0],
        ['b',0,0,0,'a',0],
        [0,0,'b',0,'a',0]
]
startState = 4
```

Figure 5: DFA used to test algorithm

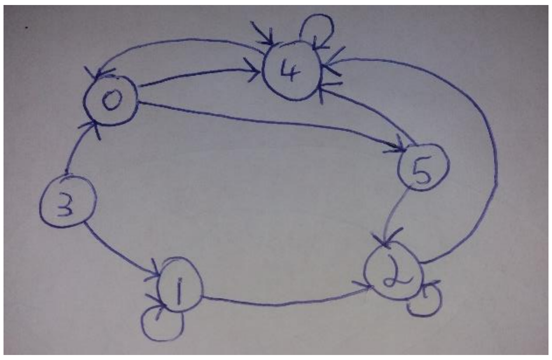


Figure 6: Drawn DFA on paper

Computing depth of DFA by computing breadth first search:

Queue:

4
~~0~~
~~5~~
2

Visited: 4 0 5 2

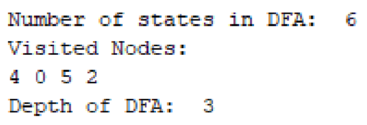
Depth of DFA = 3

Computing depth of DFA by choosing the maximum from the shortest paths of reaching states.

State	Shortest path
0	1
1	/
2	3
3	/
4	0
5	2

Maximum from shortest paths = 3. Matches the depth of DFA computed with BFS.

Actual Output from program using DFS:



```
Number of states in DFA:   6
Visited Nodes:
4 0 5 2
Depth of DFA:   3
```

Figure 7: Output of depth information

The output matches the results calculated in the two methods above. Therefore, the algorithm to find the dept of an automaton must be correct.

Hopcroft’s Algorithm

First, this algorithm was tested with a dry run found in an online paper. Please follow this link for the paper: https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Testing

To do so, I hard coded the DFA found on the paper mentioned above, to be able to compare the results with it. In figure 9, the DFA found in the paper’s dry run is displayed, while in figure 8, the implementation of the same DFA into an adjacency matrix is shown.

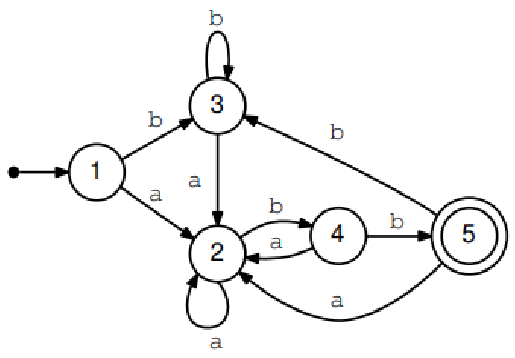
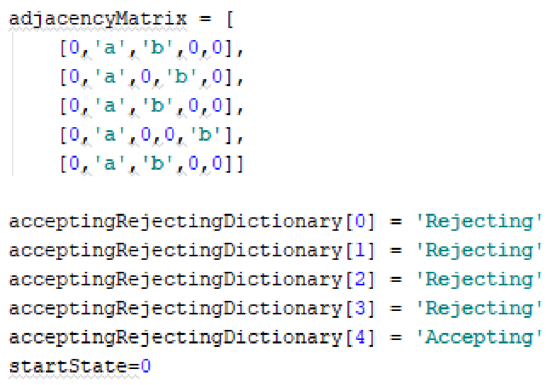


Figure 9: DFA gotten from online paper



```
adjacencyMatrix = [
    [0,'a','b',0,0],
    [0,'a',0,'b',0],
    [0,'a','b',0,0],
    [0,'a',0,0,'b'],
    [0,'a','b',0,0]]

acceptingRejectingDictionary[0] = 'Rejecting'
acceptingRejectingDictionary[1] = 'Rejecting'
acceptingRejectingDictionary[2] = 'Rejecting'
acceptingRejectingDictionary[3] = 'Rejecting'
acceptingRejectingDictionary[4] = 'Accepting'
startState=0
```

Figure 8: Adjacency matrix implementation of DFA

Then, Hopcroft’s algorithm was executed, returning a final partition. This partition should be equivalent to the states in the paper’s final DFA, displayed in figure 10. The difference between my output and the paper’s partition is that my DFA starts from state 0 to 4, and the paper’s starts from 1 till 5. When comparing figures 10 and 11 it can be noted that partitions:

- State 2 in figure 10 = [1] in figure 11
- State 1,3 in figure 10 = [0,2] in figure 11
- State 4 in figure 10 = [3] in figure 11
- State 5 in figure 10 = [4] in figure 11

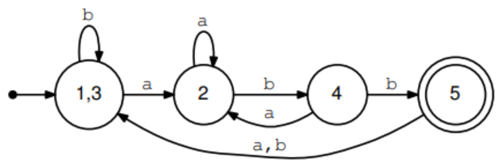
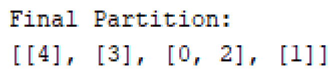


Figure 10: Minimized DFA in paper

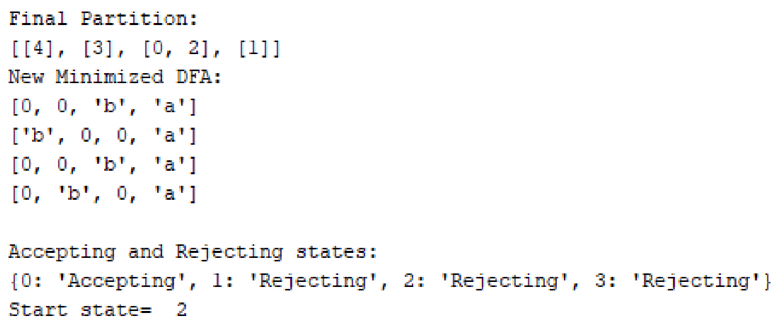


```
Final Partition:
[[4], [3], [0, 2], [1]]
```

Figure 11: Partition output after Hopcroft's algorithm is executed

Then, the program proceeds to create an adjacency matrix of the minimized DFA and displaying the new information of the DFA. Here it was decided that all the information outputted should be drawn in a DFA on a piece of paper, give a clearer picture of the output. These are shown in the figures below.

In figure 12, first state is equivalent to set [4], second state is equivalent to set [3], third state is equal to set [0,2] and last state is equal to set [1].



```
Final Partition:
[[4], [3], [0, 2], [1]]
New Minimized DFA:
[0, 0, 'b', 'a']
['b', 0, 0, 'a']
[0, 0, 'b', 'a']
[0, 'b', 0, 'a']

Accepting and Rejecting states:
{0: 'Accepting', 1: 'Rejecting', 2: 'Rejecting', 3: 'Rejecting'}
Start state=  2
```

Figure 12: Information about minimized DFA's structure

The information shown above, was drawn on paper shown in figure 13. When this DFA is compared to the DFA in figure 10, it can be noted that, the states and transitions are all the same except for one. The different transition is that of ‘a’ going from state 4 to state 1. This is because my DFA was constructed based on the states and transitions of the original DFA. Therefore, given that the final state goes to state 1 in the original automaton, it was implemented that way as well in the minimized construction.

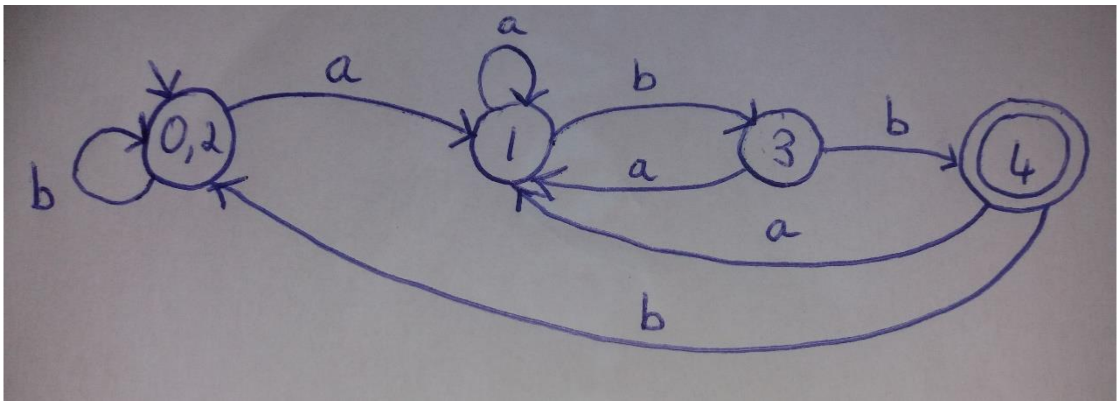
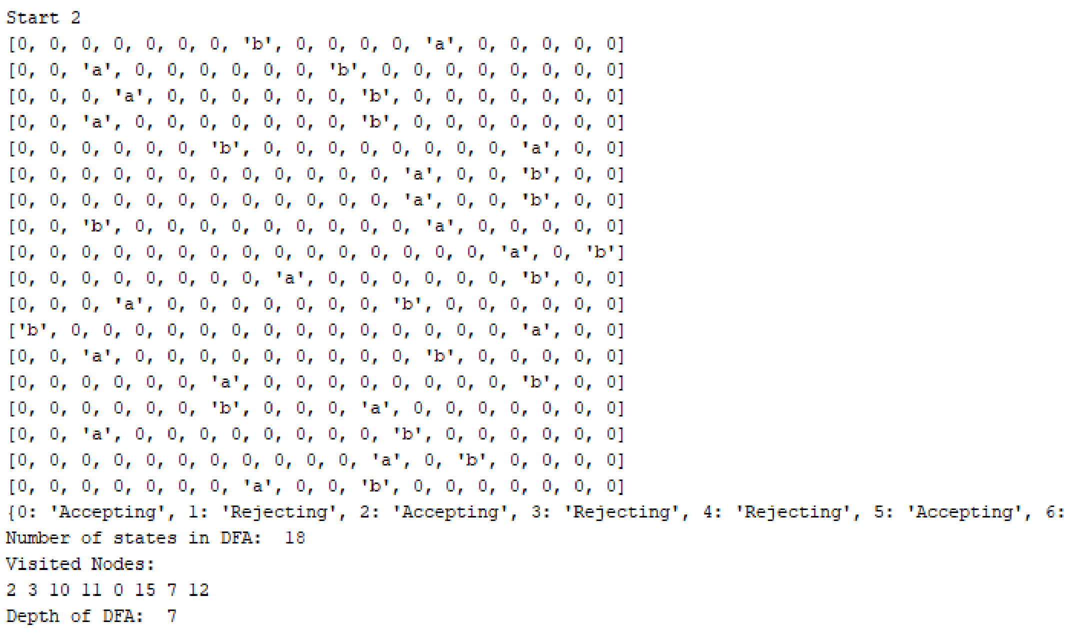


Figure 13: Outputted DFA drawn on a paper

Testing on DFA generated by program

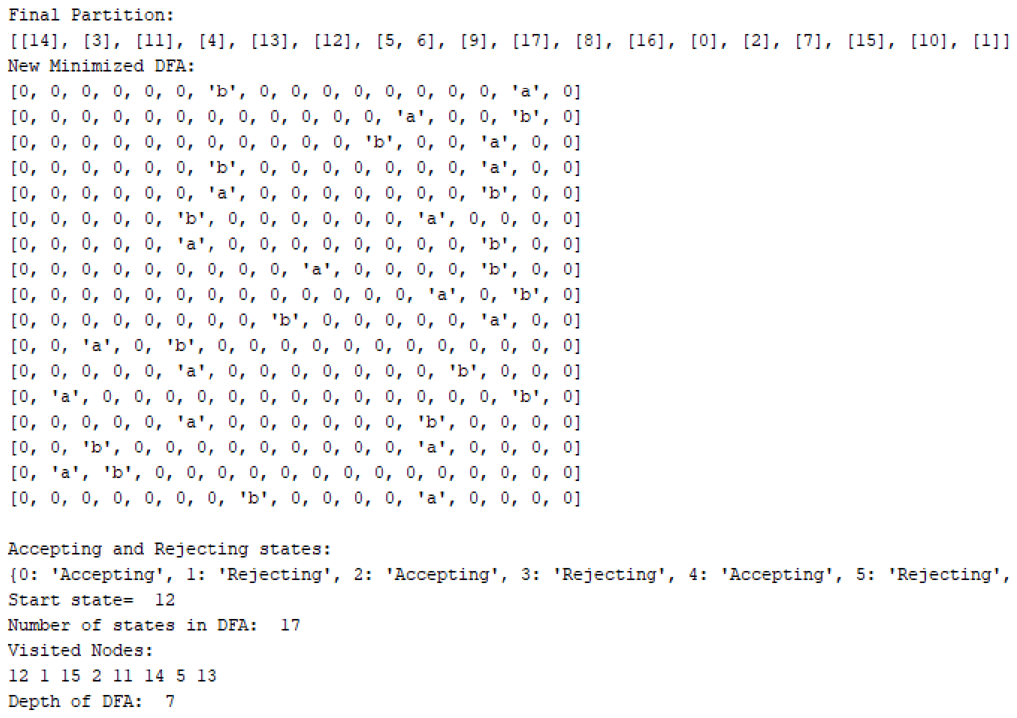
When testing on large DFAs which are randomly generated by my code, the DFA is either barely minimised or not minimised at all.

Example in the following figure, there is an adjacency matrix of 18 states, starting from state 2. The picture after it, is a screenshot of the outcome of Hopcroft, after taking the adjacency matrix as input.



```
Start 2
[0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0]
[0, 0, 'a', 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0]
[0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 'b', 0, 0]
[0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 'b']
[0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0]
['b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0]
[0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0]
[0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 'b', 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0]
{0: 'Accepting', 1: 'Rejecting', 2: 'Accepting', 3: 'Rejecting', 4: 'Rejecting', 5: 'Accepting', 6:
Number of states in DFA:  18
Visited Nodes:
2 3 10 11 0 15 7 12
Depth of DFA:  7
```

Figure 14: Adjacency matrix generated by program



```
Final Partition:
[[14], [3], [11], [4], [13], [12], [5, 6], [9], [17], [8], [16], [0], [2], [7], [15], [10], [1]]
New Minimized DFA:
[0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 'b', 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 'a', 0, 0]
[0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0]
[0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 'b', 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 'b', 0]
[0, 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 0, 'a', 0, 0]
[0, 0, 'a', 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0]
[0, 'a', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'b', 0]
[0, 0, 0, 0, 0, 'a', 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0]
[0, 0, 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 'a', 0, 0, 0, 0]
[0, 'a', 'b', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 'b', 0, 0, 0, 0, 'a', 0, 0, 0, 0]

Accepting and Rejecting states:
{0: 'Accepting', 1: 'Rejecting', 2: 'Accepting', 3: 'Rejecting', 4: 'Accepting', 5: 'Rejecting',
Start state=  12
Number of states in DFA:  17
Visited Nodes:
12 1 15 2 11 14 5 13
Depth of DFA:  7
```

Figure 15: The minimized DFA. There is only one less state than the original one

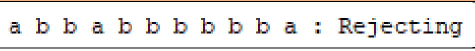
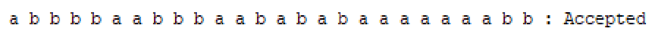
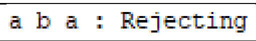
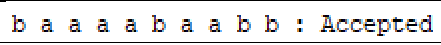
Hopcroft's results conclusion

When testing Hopcroft's algorithm on the first small automaton, it could be concluded that, the equivalence sets where split as should be, and the construction of the minimized DFA could also be considered as correct. Therefore, In that case, Hopcroft worked perfectly fine. On the other hand, when Hopcroft was applied on the very large DFAs created during run time, the minimization is either not happening at all or minimizing very few states e.g one.

String Generation

Testing

This test was done on the minimized DFA displayed in figures 12 and 13 above. For testing if a string is accepted or rejected, some of the smallest strings generated will be shown below.

String generated	Expected output	Actual Output
a b b a b b b b b b a	Rejected	 a b b a b b b b b b a : Rejecting
abbbbaabbbaababab aaaaaaa bb	Accepted	 a b b b b a a b b b a a b a b a b a a a a a a a b b : Accepted
aba	Rejected	 a b a : Rejecting
baaaabaabb	Accepted	 b a a a a b a a b b : Accepted

String Generation Conlusion

All of the expected outputs match the actual outputs in the table. Therefore, the string generation and finding if it is accepted by the automaton or not work as they should.

Tarjan's Algorithm

To test this algorithm, I once again hard coded a graph found in the video linked below. It was tested on that specific graph so that my output could be compared with the results in the video to check if the strongly connected components are correctly found.

Video: https://www.youtube.com/watch?v=TyWtx7q2D7Y

Testing

First of all, the graph in the video was implemented as an adjacency matrix in the python code and was used to compute Tarjan's algorithm on. These are shown in the two figures below.

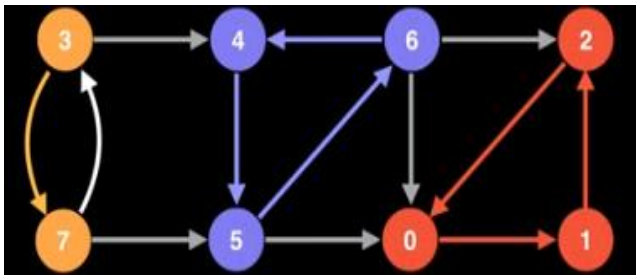
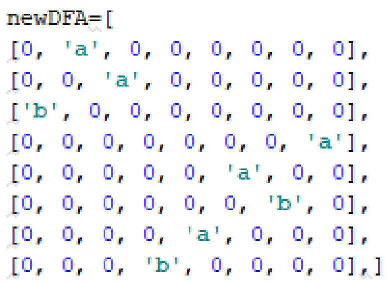


Figure 16: Graph example in video



```
newDFA=[
[0, 'a', 0, 0, 0, 0, 0, 0],
[0, 0, 'a', 0, 0, 0, 0, 0],
['b', 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 'a'],
[0, 0, 0, 0, 0, 'a', 0, 0],
[0, 0, 0, 0, 0, 0, 'b', 0],
[0, 0, 0, 0, 'a', 0, 0, 0],
[0, 0, 0, 'b', 0, 0, 0, 0],]
```

Figure 17: Hardcoded DFA for testing purpose

Then Tarjan's algorithm was executed to find the strongly connected components. The program's output was compared to the results in the example in the video. It was decided that the strongly connected group of nodes should be displayed on screen, to help in knowing if it is working correctly or not.

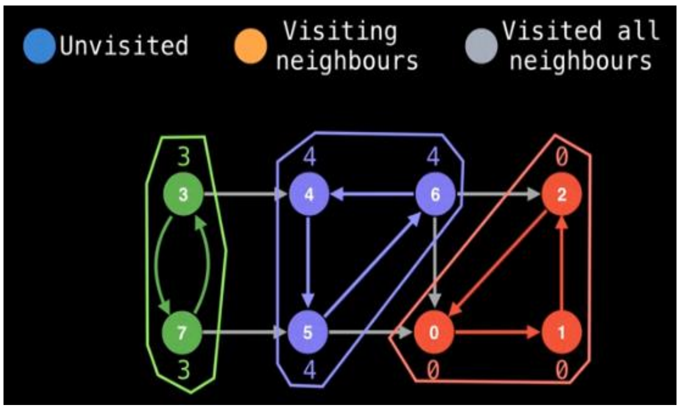
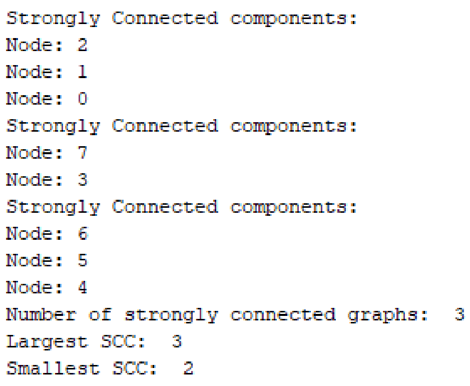


Figure 18: Strongly connected components in video



```
Strongly Connected components:
Node: 2
Node: 1
Node: 0
Strongly Connected components:
Node: 7
Node: 3
Strongly Connected components:
Node: 6
Node: 5
Node: 4
Number of strongly connected graphs:  3
Largest SCC:  3
Smallest SCC:  2
```

Figure 19: SCC outputted from program, with required details

When comparing my results, it could be concluded that, the strongly connected components in my output match the grouped nodes in the video. The largest and smallest numbers of scc and the total of scc are also correct.

Tarjan's results conclusion

As already discussed, the code's output matches the video's answer, implying that the implemented algorithm should be working perfectly as it should.

Conclusion of whole project

All in all, the algorithms work as they should, based on the evaluation done. The small adjacency matrices hardcoded for testing were commented out in the original program, so that only the questions required by the assignment specifications execute. It was also noted, that sometimes, the bigger the DFA, the worst the performance of some algorithms is.