Data Structures and Algorithms

Course Project 2019

Student Name : Valerija Holomjova

Course Code : ICS2210-SEM2-A-1819

Date : 22/04/19

Table of Contents

Description of the Code Layout

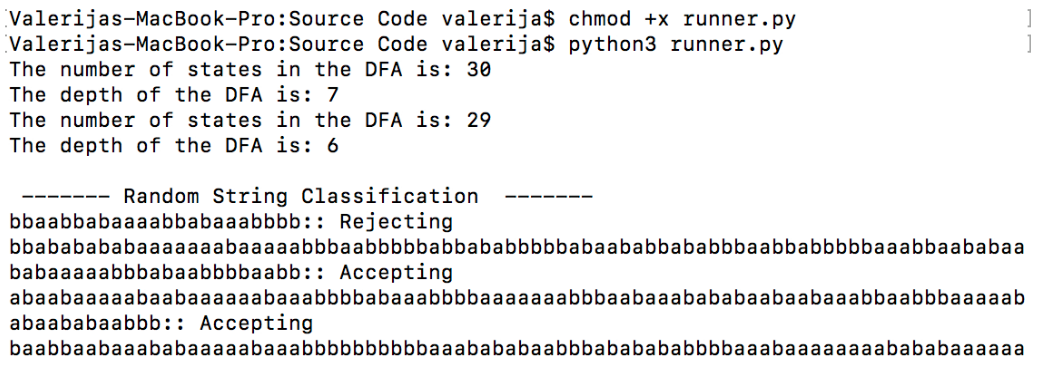
I have coded this assignment using python. The only standard library module that was used was ‘random.py’ to help generate random integer, floats and characters accordingly. The ‘dfa.py’ file includes the DFA class which stores the information of the automaton and has the functions needed to minimize, check, find strong connected components and find the depth. The ‘runner.py’ file is where the DFA values and object are first created and each function of the DFA object as required by the assignment questions are called. Please note that the DFA functions work even with DFA’s that have an incomplete transition function (states that only transition once or none times to another state).

Output of the Code

The code will create a DFA with random values as indicated in the assignment question. It will then print the depth of the newly created DFA. It proceeds by minimizing the DFA and printing the depth again. It then checks the DFA and displays the strings used to check it along with whether they are accepted or rejected. Finally, it will find the DFA’s strongly connected components and display the number of them as well as the size of the largest and smallest strongly connected component.

Running the Code

To run the code, I have used the PyCharm IDE [4] by running the ‘runner.py’ file. Alternatively, the ‘runner.py’ file can be run from the terminal as shown below.

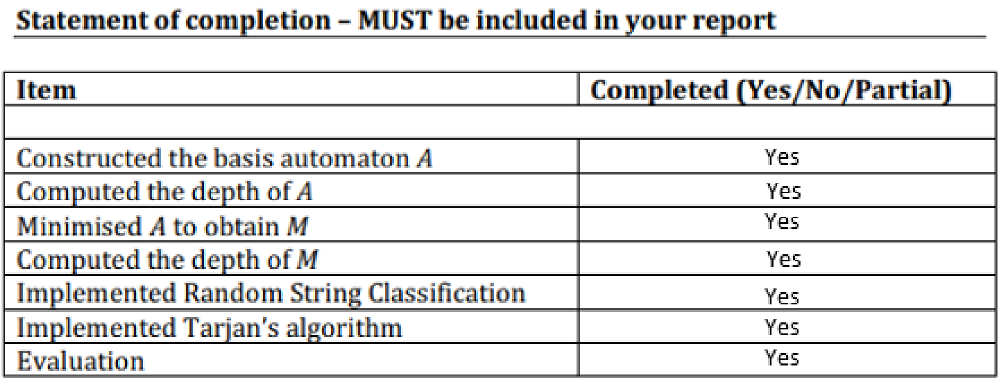


```
Valerijas-MacBook-Pro:Source Code valerija$ chmod +x runner.py
Valerijas-MacBook-Pro:Source Code valerija$ python3 runner.py
The number of states in the DFA is: 30
The depth of the DFA is: 7
The number of states in the DFA is: 29
The depth of the DFA is: 6

 ------- Random String Classification  -------
bbaabbabaaaabbabaaabbbb:: Rejecting
bbabababaaaaaaabaaaaabbbaabbbbbabbababbbbbabaababbababbbaabbabbbbbaaabbaababaa
babaaaaabbbabaabbbbaabb:: Accepting
abaabaaaaabaabaaaaaabaaabbbbabaaabbbbaaaaaaabbbaabaaabababaabaabaaabbaabbbaaaaab
abaababaabbb:: Accepting
baabbaabaaababaaaaabaaabbbbbbbbbaaabababaabbbababababbbbaaabaaaaaaaabababaaaaaa
```

Statement of Completion

Please note that all sections of the assignment have been completed.



Statement of completion – MUST be included in your report

Item	Completed (Yes/No/Partial)
Constructed the basis automaton *A*	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 - Creating the Automaton

The DFA is created using the 'createDFSA()' function which creates a new DFA object with randomly generated rejecting states, accepting states and transitions using the alphabet consisting of 'a' and 'b'. Please note that I have stored the information of the automaton using class variables. The transitions are represented in an adjacency list form using a dictionary where the key is a state and the value is another transition consisting of a character as a key and the state it transitions to when given the character as a value.

```
# DFA Initialization.
def __init__(self, n_states, start, accept, reject, alphabet, transition):
    self.n_states = n_states  # Number of states - int
    self.start = start  # Starting state - int
    self.accept = accept  # Accepting states - List of int
    self.reject = reject  # Rejecting states - List of int
    self.alphabet = alphabet  # Alphabet - List of char
    self.transition = transition  # Transitions - Dictionary of Dictionaries
```

Why did I choose to use an Adjacency List?

I have used an adjacency list because it seems more efficient memory wise for when the DFA after it is minimized and thus the time complexity to iterate through each transition could be faster. To elaborate on my point, it is faster to iterate over all the transitions because neighbours can be accessed directly.

createDFSA() function

The function begins by created a random number of states between 16 and 64. It the creates the DFA's transitions by going through each state and choosing to random states to transition 'a' and 'b' to.

```
# Find a random number between 16-64.
no_states = randint(16, 64)
# Create a dictionary with no_states where each state has a and b leading to a
random two other states
transitions = {}
# Go through each state and create a transition two a random another state.
for i in range(0, no_states):
    state1 = randint(0, no_states - 1)
    state2 = randint(0, no_states - 1)
    transitions[i] = {'a': state1, 'b': state2}
```

The accepting states were then chosen by choosing 20% of the given states at random and these were then sorted. The initial state was chosen randomly from the states and the rejecting states were taken by find the set difference between all the states and the accepting states. These variables where then used to create a DFA object which is then returned.

```
accepting = sample(transitions.keys(), k=(round(.2 * no_states)))
accepting.sort()
# Choose a random starting state
initial = randint(0, no_states - 1)
# Get rejecting states
all_states = [i for i in range(0, no_states)]
rejecting = list(set(all_states).symmetric_difference(accepting))
return DFA(no_states, initial, accepting, rejecting, alphabet, transitions)
```

Question 2 and 4 - Finding the Depth of the Automaton Algorithm

To find the depth of the DFA, the class function ‘getDepth()’ is called which uses breadth first search to find the path longest from the start node to a certain node and then the longest path from that node to another node.

getDepth() function

This class function uses the ‘breadthFirstSearch()’ function to find the depth of its automaton. It does this by finding the nodes with the maximum distance from the start node of the DFA using the ‘breadthFirstSearch()’. Two variables are created to store the final depth and max nodes of the automaton, initially they are set to store the results of the longest path from the start state.

```
# Find the node(s) with longest path from the start state of the DFA.
maximums, max_depth = self.breadthFirstSearch(self.start)
maximum, final_depth = maximums, max_depth  # To store the max depth node and
depth.
```

The algorithm proceeds by finding the longest path from each of the found nodes to other nodes. If one of the nodes has a longer path to another node, the variable storing the depth of the automaton is updated. At the end of the function, the depth of the DFA is displayed.

```
# Find the max distance from each of the max depth states and update the max depth
accordingly.
for state in maximums:
    # Find the node with longest path from the previously found node of the DFA.
    temp_maximum, temp_depth = self.breadthFirstSearch(state)
    # Update the max depth and node if a larger dpeth value is found.
    if temp_depth > final_depth:
        maximum = temp_maximum
        final_depth = temp_depth
# Displays the depth of the DFA.
# Displays the depth of the DFA.
print("The number of states in the DFA is:", self.n_states)
print("The depth of the DFA is:", final_depth)
```

breadthFirstSearch() algorithm

This algorithm implements a typical breadth first search algorithm to find the maximum path from a given node to any another node to help find the depth of the DFA. It has a queue to store the order of which states to check next and a list of visited states. In the beginning the start state is appended to the queue and list of visited states. A dictionary of depth value is created to store the depth for each state where the key is the state and the value is its depth.

```
# Queue - a list to check immediate child of states first, then check their
children (what state to check next).
# Visited - a list to keep track of the states visited.
queue, visited = [], []
queue.append(start)
visited.append(start)
# Depth - a dictionary to store the depths or distance for each node from the start
state.
depth = {i: 0 for i in range(self.n_states)}
```

Until the queue is empty, an item is popped from the start of the queue and set as the current state to be checked. For each state transitioned from the current state, if the neighbouring state is not in the visited list, it’s appended to the list and the queue and the depth of it is set as an increment of its parent’s depth (the current state taken from the queue). This process repeats until all the states have been checked and the depths for each state have been set (the queue is empty).

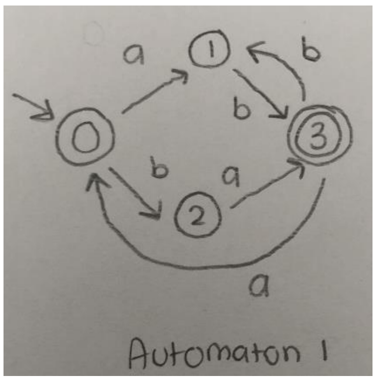
```
while (len(queue) != 0):
    # Store current as a node from the start of the queue and remove it from the
queue - FIFO.
    current = queue[0]
    queue.pop(0)
    # For each state transitioned from the the current state.
    for item, value in self.transition[current].items():
        # If the child was not visited, append it to the visited and queue, and
update it's depth.
        if value not in visited:
            visited.append(value)
            queue.append(value)
            # The depth the state is an increment of the depth of it's parent
state.
            depth[value] = depth[current] + 1
```

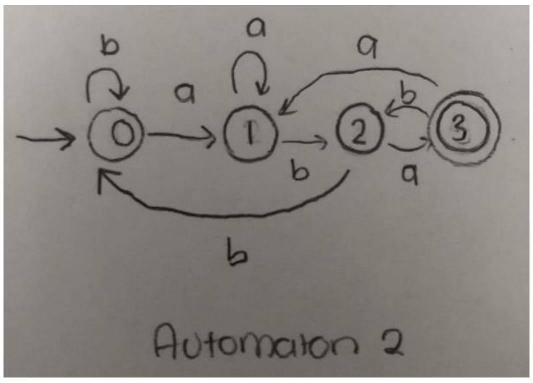
In the end of the algorithm, the maximum depth is obtained as well as a list of states that have the maximum depth so that each one could be checked for the final depth in the 'getDepth()' function. These two variables are then returned.

```
# Get the maximum depth and the states with the maximum depth.
max_depth = max(depth.values())
maximums = [i for i,diam in depth.items() if diam == max_depth]
return maximums, max_depth
```

Evaluation

To test out the algorithm I picked out a few random examples of some DFA's and calculated their depth. Below are images displaying two of the DFA's that I used for testing and a table showing the outputted results for each of the DFA's.





A visual representation of the DFA's being tested.

	Automaton 1	**Automaton 2**
Outputted Results	The number of states in the DFA is: 4 The depth of the DFA is: 2	The number of states in the DFA is: 4 The depth of the DFA is: 3

As shown above the algorithm yields accurate results despite any loops or incomplete transition functions. The only thing I have noticed is that once a DFA is minimized the depth is sometimes the same, I have then found out that this is due to there not being many partitions found during the minimization stage, so the algorithm works well, and steps were taken during implementation to ensure the longest path was chosen.

Question 3 – Minimization using Hopcroft’s Algorithm

Time complexity

For the implementation of the Hopcroft’s algorithm in the ‘hopcroftsAlgorithm()’ function, given that n is the number of states and s is the size of the alphabet, the time complexity of this algorithm is O(s*n*log n) in the worst case. This is due to the fact the algorithm undergoes O(n*s) transitions and partition refinement only occurs conditionally hence the O(log n).

doMinimization() function

This is the class function called to minimize a DFA. It functions by calling the ‘hopcroftsAlgorithm()’ function to obtain a list of partitions that are inequivalent from one another. It then takes these partitions to and replaces it’s DFA values with a minimized version of itself.

This process is done by creating temporary variables to store the new start state, rejecting states, accepting states and transitions. Please note that the index of the partition will be used to represent new states. Hence, the first partition in the list will be state 0 and the last partition will become state n-1 in the DFA (where n is the number of partitions). Next, it checks whether one of the states in the partition is a start state. If this condition is true, the index of the partition is set as the start state of the new DFA.

```
# Minimize the DFA using Hopcroft's algorithm.
partitions = self.hopcroftsAlgorithm()
# To store the new DFA features.
new_transitions, new_accepting, new_rejecting, new_start = {}, [], [], self.start

# For each partition.
for i, partition in enumerate(partitions):
    # Check each state in the partition to check if there is a start state.
    if self.start in partition:
        new_start = i  # Setting the partition to a new start state.
```

Next, the algorithm takes a random state from a partition (since all states should transition to the same partition for any given input sequence). It then checks whether this state is in the DFA’s accepting or rejecting states and appends the partition index accordingly to the new accepting or rejecting states. At the end of the loop, the function calls the ‘getNewTransition()’ function to obtain the new transition for the chosen state in relation to other partitions. In other words, it obtains the transitions from the current state to other partitions. Hence, the obtained transition is appended to the new transition adjacency list and will represent the transition from the current partition (or new state) since all states in the partition should lead to the same ones for any input sequence.

```
A = partition[0]  # Take a random state from the partition - (they should
transition to the same partition).
# Check whether the partition is an accepting or rejecting state.
if (A in self.accept):
    new_accepting.append(i)
else:
    new_rejecting.append(i)
# Get the transitions from the current partition to other partitions.
new_transitions[i] = self.getNewTransition(A, partitions)
```

At the end of the function the variables obtained through the loop are used to replace the current DFA’s values as a means to minimize it.

getNewTransition() function

This function obtains new transitions from a given state ‘A’ from one partition to other partitions. It functions by looping through each input sequence ‘c’ in the alphabet and then obtaining the ‘state’ it transitions to using ‘c’ from the DFA transitions.

```
# The old transition table.
old_transitions = self.transition[A]
# To store the transitions from A to other partitions.
new_transitions = {}
# For each character in the alphabet.
for c in self.alphabet:
    # If transition exists.
    if c in old_transitions.keys():
        state = old_transitions[c]  # State that is reached from A using c.
```

It then goes through each partition and if the ‘state’ is found in one of the partitions, the character and partition index are added as a transition for the given state ‘A’. Please note that if the ‘state’ is not found in any of the partitions, a transition will not be added as it is not part of the newly minimized DFA states. At the end of the function a dictionary of valid transitions from ‘A’ is returned to be used to minimize a DFA in the ‘doMinimization()’ function.

```
# Go through each new partition.
        for i, partition in enumerate(partitions):
            # If the state is found in a partition, set the transition from A using
c as it's index.
            if state in partition:
                new_transitions[c] = i
                break
return new_transitions
```

hopcroftsAlgorithm() function

Hopcroft’s algorithm is based on partition refinement – splitting a partition into a larger number of smaller ones. The result of this algorithm is a list of partitions such that each state in a given partition leads to the same partition for any given input sequence. The following report was used help understand and implement the algorithm [1] which displayed pseudocode for Hopcroft’s Algorithm (page no. 5) based on his formal description of the algorithm in his report [2].

The initial ‘partitions’ includes the set of final and non-final states as these two sets are inequivalent to each other. The algorithm chooses a partition (denoted as ‘A’) from the set of distinguishers ‘D’ (these are sets of states that reach or don’t reach a certain partition – to help refine the final partitions into inequivalent partitions). For each input symbol ‘c’, the set of states that lead to the chosen partition ‘A’ are found and these set of states are denoted as ‘X’. Please note that these states are found using the ‘findStates()’ function.

```
# Partitions includes accepting and rejecting states..
partitions = [self.accept, self.reject]
# Distinguishers has only accepting states
D = [self.accept]
while len(D) != 0:
    # Remove a partition from the distinguishers.
    A = D.pop(0)
    # For each input in the DFA alphabet.
    for c in self.alphabet:
        # X is the states on which a transition on c will lead to a state in A.
        X = self.findStates(self.transition, c, A)
```

It then iterates through each of the currently refined partitions and finds the set intersection and set difference of the current partition ‘part’ and the ‘X’ partition. This signifies the set of states that lead to ‘A’ and don’t lead to ‘A’ respectively. If there are elements in both such sets, the partition ‘part’ is refined by being replaced by the two sets. Please note that the algorithm terminates when no more such splits can be found or more specifically when the length ‘D’ becomes 0. This signifies that each of the elements in ‘partitions’ are inequivalent and can’t be split further.

```
# For each partition in the partitions.
for i, part in enumerate(partitions[:]):
    # If the set of states leading A and not leading to A are non empty.
    intersection = list(set(X).intersection(part))
    difference = list(set(part).difference(X))
    if (len(intersection) != 0 and len(difference) != 0):
        # Refine the partition further by replacing it with the two sets.
        partitions.remove(part)
        partitions.append(intersection)
        partitions.append(difference)
        # If the partition is found in the distinguishers.
```

The algorithm then modifies the distinguisher partitions ‘D’ by checking if ‘part’ is in ‘D’. If ‘part’ is in ‘D’, it has to be replaced by the two sets (intersection and difference) so that they can be split further in future iterations. However, if it is not in ‘D’, the smaller of the two sets is appended to ‘D’. This is done because only one of the sets are needed to refine the whole partition and by choosing the smaller one, the algorithm is faster.

```
if part in D:
    # Replace the partition in distinguishers with the two sets.
    D.remove(part)
    D.append(intersection)
    D.append(difference)
else:
    # Append the intersection or difference depending on which is smaller.
    if len(intersection) <= len(difference):
        D.append(intersection)
    else:
        D.append(difference)
```

After the there are no more distinguisher partitions left, the algorithm terminates and ‘partitions’ variable is returned containing the final list of partitions. These will be used to convert states of the current DFA into a minimized one in the ‘doMinimization()’ function.

findStates() function

The following function finds a list of states in the current DFA that reach some state in a list ‘A’ from a given character ‘c’. It works by looping through each state in the transition function variable and checking whether the transition with a character ‘c’ leads to any state in ‘A’. If this condition is true, the state is added to the list which is returned at the end of the function. It is used for partition refinement in the ‘hopcroftsAlgorithm()’ function.

```
# Finds all the states in the DFA that reach A from a given character c.
def findStates(self, transition, c, A):
    found = []
    for state, values in transition.items():
        if (c in values.keys()) and (values[c] in A):
            found.append(state)
    return found
```

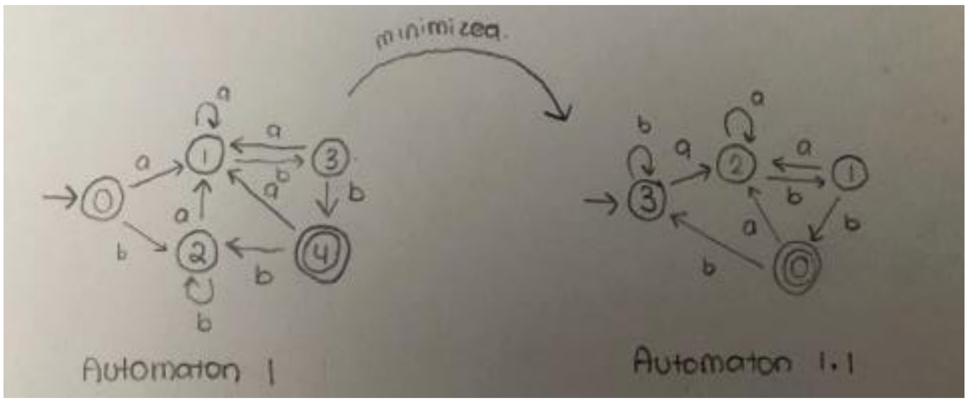
Evaluation

I tested out the algorithm by finding some online examples and comparing the results to those obtained by other minimization methods. Below is the result of one of these examples that was used to test the algorithm. Below is a table indicating the value of the DFA before and after minimization. The values after the minimization display the values of the newly minimized DFA where each partition of the old DFA have been converted to states. The image below is a visual representation before minimization (Automaton 1) and after minimization (Automaton 1.1).

Please note that the partitions found by the Hopcroft's Algorithm was [[4], [3], [1], [0, 2]] for the DFA before its partitions were transformed into states and represented as new DFA. The only flaw I noticed about this algorithm is the partitions are given a state ID based on the position they are placed after being found in the Hopcroft's Algorithm, which may be a bit confusing for other users (eg. in this case the partition 4 is given the new state ID 0).

Variables	**Before Minimization**	**After Minimization**
Number of States	5	4
Start State	0	3
Accepting States	[4]	[0]
Rejecting States	[0,1,2,3]	[1, 2, 3]
Transitions	{ 0: {'a': 1, 'b': 2}, 1: {'a': 1, 'b': 3}, 2: {'a': 1, 'b': 2}, 3: {'a': 1, 'b': 4}, 4: {'a': 1, 'b': 2}}	{ 0: {'a': 2, 'b': 3}, 1: {'a': 2, 'b': 0}, 2: {'a': 2, 'b': 1}, 3: {'a': 2, 'b': 3}}

The values before and after the DFA is minimized.



A visual representation of the DFA minimized.

Question 5 - Checking the Accepted Strings of the Automaton

The DFA is checked by calling the class variable 'checkDFA()'. It generates 100 strings of a random length between 0 and 128 and checks whether the DFA accepts or rejects it.

Time Complexity

The time complexity of the algorithm is O(s*c) where s is the number of strings (set as 100) and c is the number of characters in the string (set as any random number up to 128). This is because in every iteration in the range of 100 (the string amount), another iteration occurs of a random amount of times up to 128 (the string length). It is the worst-case time complexity considering each of the 100 strings have a length of 128.

checkDFA() function

At the start of the function, the string length and string number are initialized. The function proceeds by looping through the string number amount of times (100 times) and for each iteration another iteration is used to generate each character of the string for a random number of times from 0 to 128. Please note that the strings and characters were created after iterations to decrease the time complexity of the algorithm and make it faster rather then created them first and then iterating through them.

The algorithm then uses a variable called current to keep track of the state currently being transitioned into and picks a random character to add onto the string from the alphabet for each string iteration. For each character, the algorithm checks if there is transition from the current state to another state using the character. If this condition is false, the current state is set to none and the iteration is broken so that the entire string would be rejected. If there is an existing state that can be transitioned to from the current state using the character, the current state is updated to that existing state. This process repeats until the number of iterations is fully completed, or in other words when each character of the forged string is checked. Otherwise, it is terminated if a transition does not exist. Hence, the negative but also positive aspect of this algorithm is that the string will not continue making characters if a dead end is reached and will simply reject the entire string. It is negative cause the user will not be able to see the entire string, just the section until it was terminated but positive because the time complexity is reduced.

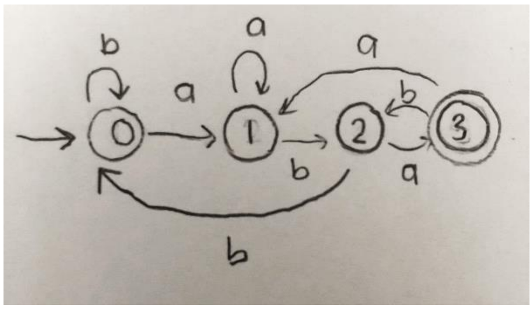
```
for i in range(string_n):
    current = self.start  # Test each string from the starting state.
    string = ''  # To append each character onto.
    # Iterating through a random number between 0 and string_len and generating a
random character each time.
    for j in range(0, random.randint(0, string_len)):
        character = random.choice(self.alphabet) # Choosing a random character form
the alphabet.
        string += character # Appending character to current string.
        # If transition with character doesnt exist, reject immediately.
        if character not in self.transition[current]:
            current = None
            break
        # Transition to the next state with the given character.
        current = self.transition[current][character]
```

Then when each character of the string is processed the final reached state is checked on whether it is an accepting state of the DFA or a rejecting one and this information is outputted to the user accordingly. Please note that when a transition from a state wasn't reached, the state would have been converted to a 'None' object and would be rejected since there is no 'None' objects in the accepting states of the DFA.

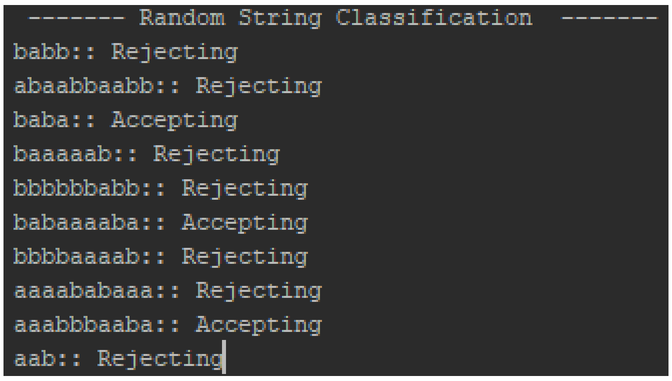
```python
# Check if the state is in accepting states or rejecting states.
if current in self.accept:
    print(string + ":: Accepting")
else:
    print(string + ":: Rejecting")
```

Evaluation

To test out the algorithm I picked out a few random examples of some DFA's and checked whether the results from the algorithm were accurate. Below is one of the DFA's I tested. I lowered the string number to 10 and the max string length to 10 for this run only so that I can easily evaluate the results. The results obtained from the following automaton is displayed below. As can be seen from the results, only strings ending in 'aba' are accepted. Please note that if a transition didn't exist for one of the characters the string would be rejected immediately.



A visual representation of the DFA being tested.



```
 ------- Random String Classification  -------
babb:: Rejecting
abaabbaabb:: Rejecting
baba:: Accepting
baaaaab:: Rejecting
bbbbbbabb:: Rejecting
babaaaaba:: Accepting
bbbbaaaab:: Rejecting
aaaababaaa:: Rejecting
aaabbbaaba:: Accepting
aab:: Rejecting
```

The results of the algorithm for a string number of 10 and a string length of 10 of the DFA being tested.

Question 6 - Tarjan’s Algorithm

For the last question of the assignment, the strongly connected components of the DFA were found by implementing Tarjan’s Algorithm using Depth First Search. The components are found by calling the DFA’s class function ‘findSCC()’. Please note that the following video [3] was used to help me understand the algorithm before implementing it.

Time Complexity

Given that n is the number of nodes and s is the size of the alphabet, the worst case time complexity of the algorithm is O(n + (n*s)) since the algorithm is called once per state and then considers each transition (n*s) provided that the transition function is complete (each state goes to another state given s).

The ‘findSCC()’ function

At the start of the function, two class variables ‘state_id’ and ‘SCC’ are reset. The ‘state_id’ variable is used to keep track of the discovery time of each state (the order in which the states are discovered) during the Depth First Search. The ‘SCC’ variable will be used to store the found strongly connected components.

Next, each state in the DFA is marked as unvisited by initializing the ‘state_ids’ and ‘low_links’ list as variables a size of the number of states and declaring each element to have a value of -1 The ‘state_ids’ variable will store the discovery time (state id) of each state and the ‘low link’ variable will store the low link values for each state (the smallest node id reachable from that state when doing a DFS). A stack is created to store the states that are visited during the DFS as well as a list called ‘on_stack’ to update which state is currently in the stack.

```
# Reset Tarjan's Algorithm state_id and SCC class variables.
self.state_id = 0
self.SCC = []
state_ids = [-1] * self.n_states  # Stores the state ids of a state when
discovered.
low_links = [-1] * self.n_states  # Stores the low link values.
stack = []  # Stack to keep track of visited states.
on_stack = [False] * self.n_states  # Keep track of what states are in the stack.
```

Next, the function goes through all the states in the DFA and calls the recursive ‘tarjanDepthFirstSearch()’ function to find the strongly connected components in the DFA using all states that are not discovered yet. This is done to overcome potential dead ends or cycles.

```
# To ensure all states are explored in case of cycles.
for i in range(self.n_states):
    # If the state isn't discovered yet, find it's SCC.
    if state_ids[i] == -1:
        self.tarjanDepthFirstSearch(i, on_stack, state_ids, low_links, stack)
```

After all nodes are explored and all strongly connected components are found, the number of SCC are printed. The largest and smallest SCC are obtained and the number of states within each of them are printed.

The ‘tarjanDepthFirstSearch()’ function

This function implements the recursive Depth First Search algorithm in order to locate strongly connected components. The variable ‘state’ keeps signifies the current state being discovered. At the start of the function, the current discovered state is assigned a unique state ID and a low link value from the ‘node_id’ variable. The low link value will be updated later in the function if a smaller state ID can be reached from the current variable. The current node is also appended to the stack.

```
# Upon discovering a state assign it an ID and low-link value.
state_ids[state] = self.state_id
low_links[state] = self.state_id
self.state_id += 1
# Append the state to the stack.
on_stack[state] = True
stack.append(state)
```

Next, the function iterates through each of the children nodes of the state (or more technically, each state than can be reached from the current state for any given character). If one of the children hasn’t been discovered yet (it hasn’t been given a state ID), ‘tarjanDepthFirstSearch()’ is called recursively on the child node. When backtracking, the low link of the current state is updated if its child is on the stack and has a lower low link value.

```
# Iterate through the neighbouring states of the state.
for child in self.transition[state].values():
    # If a child isn't discovered, recursively call function (DFS) to seek it's
neighbours.
    if state_ids[child] == -1:
        self.tarjanDepthFirstSearch(child, on_stack, state_ids, low_links, stack)
    # After backtracking, update the low link value of the state if the child is on
the stack.
    if on_stack[child]:
        low_links[state] = min(low_links[child], low_links[state])
```

If the low link value of the current state is equal to state ID of the current state - this signifies that the current state is a head of a strongly connected components. Hence the algorithm will pop each state off the stack till the head is reached. Each of the states being popped are appended to a temporarily list which represents one strongly component and later will be added to the class variable ‘SCC’ to be used in the ‘findSCC()’ function (so that it’s information could be displayed).

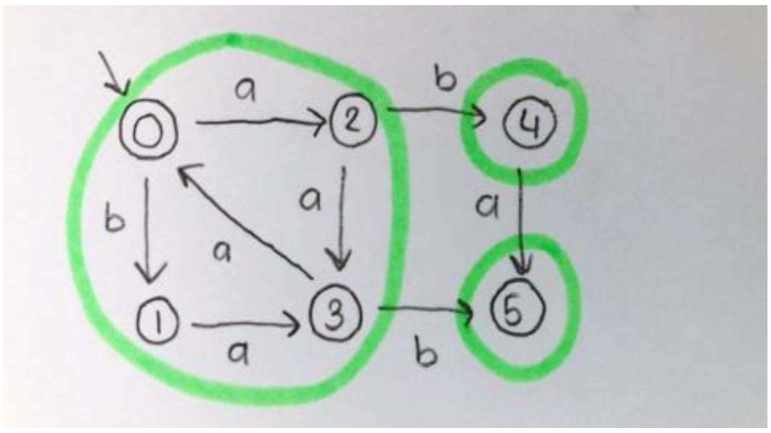
```
# After visiting all neighbours if current state is a head of a SCC.
if low_links[state] == state_ids[state]:
    current = -1
    components = []  # To store the states of the SCC.
    # Pop the states from the stack till the head of the SCC is reached.
    while current != state:
        current = stack.pop()
        on_stack[current] = False
        components.append(current)  # Appending each state of the current SCC.
    self.SCC.append(components)  # Appending the SCC list to the class variable.
```

Evaluation

The algorithm was tested finding some online examples and comparing the results to those obtained by others testing graphs to find strongly connected components. Below is the result of one of these examples that was used to test the algorithm. Below is an image of the automaton that was given to the algorithm with the strongly connected components circled. Tarjan’s algorithm found the following strongly connected components [[5], [4], [1, 3, 2, 0]] which was accurate to the example given and displayed the following results to the user:

The number of SCC is: 3
The largest SCC has size: 4
The smallest SCC has size: 1

Please note that the algorithm worked despite having a non-complete transition function (some of the states had only one transition or none).



A visual representation of the Automaton and it's SCC being evaluated.

References

[1]: XU, Yingjie. *Describing an n Log n Algorithm for Minimizing States in Deterministic Finite Automaton*. p. 5, *Describing an n Log n Algorithm for Minimizing States in Deterministic Finite Automaton,* https://www.irif.fr/~carton/Enseignement/Complexite/ENS/Redaction/2008-2009/yingjie.xu.pdf.

[2]: Hopcroft, John. *An n Log n Algorithm for Minimizing States in a Finite Automaton*. pp. 6–7, *An n Log n Algorithm for Minimizing States in a Finite Automaton, http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf*.

[3]: WilliamFiset, director. *YouTube*. *YouTube*, YouTube, 21 Mar. 2018, www.youtube.com/watch?v=TyWtx7q2D7Y.

[4]: “PyCharm: the Python IDE for Professional Developers by JetBrains.” *JetBrains*, www.jetbrains.com/pycharm/.