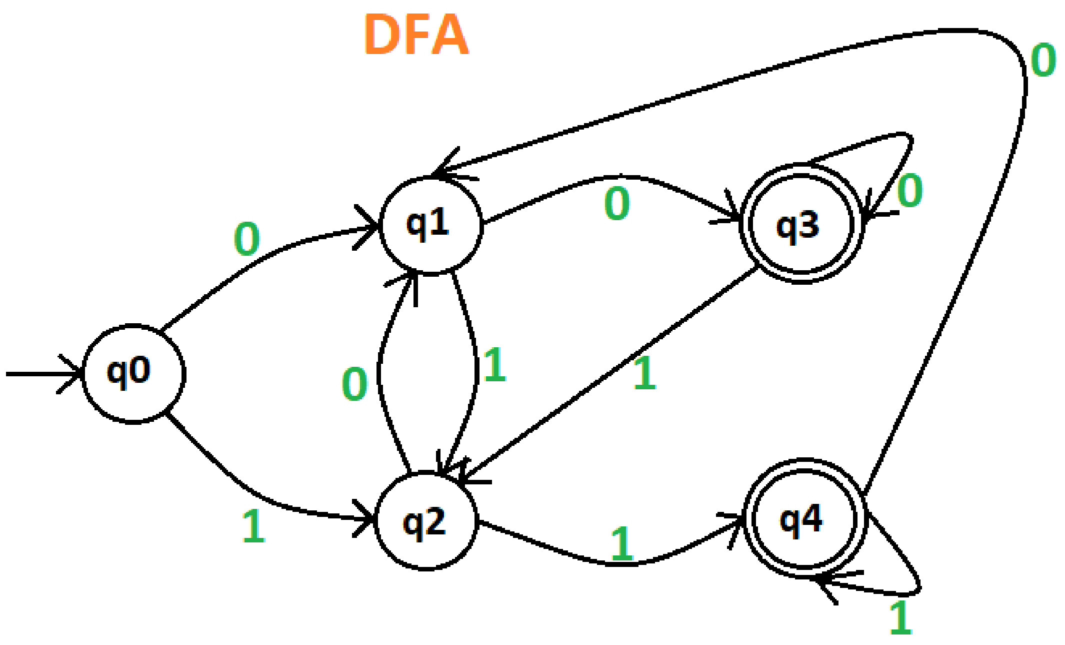
Graph Algorithms on DFAs



[1]

Liam Bugeja Douglas 17002L
Supervisors: Mr Kristian Guillaumier & Prof. John Abela
ICS2210

Table of Contents

Contents

Statement Of Completion

Item	**Completed**
Create a random DFA	Yes
Correctly Computed the depth of the DFA	Yes
Correctly implemented DFA minimization	Yes
Correctly computed the depth of the minimized DFA	Yes
Correctly implemented Tarjan’s algorithm	Yes
Printed number and size of SCCs	Yes
Provide a good discussion on Johnson’s algorithm	Yes
Included a good evaluation in your report	Yes

Johnson’s Algorithm

Johnson’s algorithm is used to find the shortest path between each pair of vertices in a direct weighted Graph. This algorithm makes use of Dijkstra and Bellman-Ford algorithm as subroutines and is very similar to Floyd-Warshall algorithm but, Johnson’s has a better time complexity when a graph has a few edges. The Bellman-Ford algorithm is used to input new weights on the graph as well as remove negative edges and cycles whilst Dijkstra’s algorithm calculates the shortest path between all pairs of vertices.

Johnson’s Algorithm can be split up into 3 main steps:

1. Add a new node in the graph and connect it to all other nodes in the graph with zero weight edges.

2. Next, using the Bellman-Ford algorithm, start from the new node which was added in the graph in the previous step and calculate all the minimum edge weights to all the nodes. If a negative cycle is detected the algorithm is terminated. All the edges are given new values which are always positive, these are calculated by the previous step.

3. Finally, the node added in the beginning is removed and Dijkstra’s algorithm is used to calculate the new shortest paths using the new edge weights.

Johnson’s algorithm has an average time complexity of O (V^2log(V) + V.E) where v is the number of vertices and e is the number of edges. We can open this up by further examining the time complexity of each step.

The first step in the algorithm is pretty simple, by adding one new node it only takes O(V) time. Next, the second step consists of using the Bellman-Ford algorithm and rewriting the weight of the edges, both cases take O(VE) time to complete. Finally, the third step uses Dijkstra's algorithm which has a time complexity of O (V log(V)+ E).

The following information was found on [2][3]

Implementation

Class DFA

Contains all the functions needed to create and manipulate a deterministic finite state automaton (DFA): creating a DFA, calculating the depth of the DFA, minimizing the DFA and finding the strongly connected components (SCCs).

Class main

The main class is only used to initialize the DFA and use the functions found in the DFA class.

Creating the DFA

A Deterministic Finite State Automaton (DFA) can be defined as a 5-tuple (Q, Σ, δ, s, F) as explained in [4], which can be further derived as:

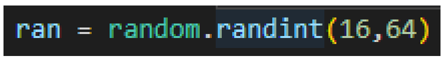
Q = Set of states
Σ = The input alphabet
δ = The transitions
s = The start state
F = The final states

So, to create the DFA the following attributes where needed: A list which should store all the states of the DFA, a dictionary with the state name as the key value and a list containing the destination of the transition, a variable which holds the start state and finally a list containing all the final states. Note that in this assignment the input alphabet was not saved since it only consisted of two values (a,b), these were saved as the position in the values in the dictionary. Meaning that:

transitions[key_vlaue] = [transition with value a, transition with value b]

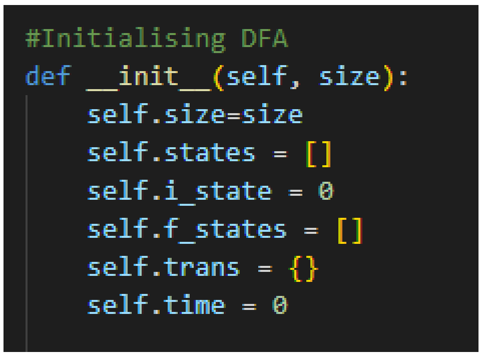
Question 1

To construct the DFA *A* the class DFA was instantiated in the main class since the constructor of the DFA class contains all the necessary variables needed to build *A*. Since the function randint is inclusive the range is between 16 and 64, to create a random number of states in *A* the random class was used:



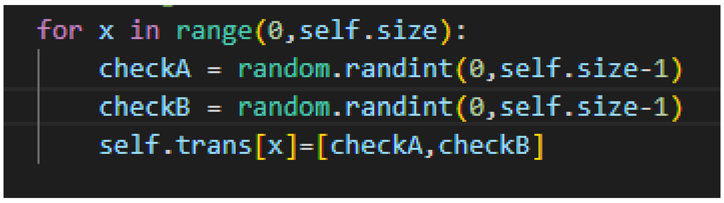
```
ran = random.randint(16,64)
```

As previously said the constructor class contains all the variables which will be initialized



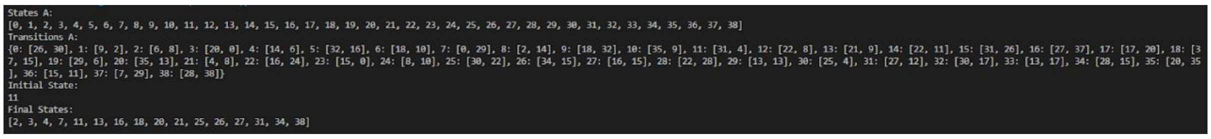
```python
#Initialising DFA
def __init__(self, size):
    self.size=size
    self.states = []
    self.i_state = 0
    self.f_states = []
    self.trans = {}
    self.time = 0
```

The size of *A* will be the number given by the random function, the states are created by looping through the range of the size and inputting the numerical values in the states list. The initial state is chosen randomly by using the random function as well, final states are chosen by each state having a 50% chance of being a final state by also using the randint function. Finally, transitions are created by looping through the size of the DFA and inputting the key value with random values using the randint function. Since each key must have a two pair value a list is used as the value of each key containing the pair value.



```python
for x in range(0,self.size):
    checkA = random.randint(0,self.size-1)
    checkB = random.randint(0,self.size-1)
    self.trans[x]=[checkA,checkB]
```

The below image shows a dry run of creating the DFA *A*



```
States A:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]
Transitions A:
{0: [26, 30], 1: [9, 2], 2: [6, 8], 3: [20, 0], 4: [14, 6], 5: [32, 16], 6: [18, 10], 7: [0, 29], 8: [2, 14], 9: [18, 32], 10: [35, 9], 11: [31, 4], 12: [22, 8], 13: [21, 9], 14: [22, 11], 15: [31, 26], 16: [27, 37], 17: [17, 20], 18: [3
7, 15], 19: [29, 6], 20: [35, 13], 21: [4, 8], 22: [16, 24], 23: [15, 0], 24: [8, 10], 25: [30, 22], 26: [34, 15], 27: [16, 15], 28: [22, 28], 29: [13, 13], 30: [25, 4], 31: [27, 12], 32: [30, 17], 33: [13, 17], 34: [28, 15], 35: [20, 35
], 36: [15, 11], 37: [7, 29], 38: [28, 38]}
Initial State:
11
Final States:
[2, 3, 4, 7, 11, 13, 16, 18, 20, 21, 25, 26, 27, 31, 34, 38]
```

Note that the variable time is used in Tarjan's algorithm which we will later discuss.

Question 2

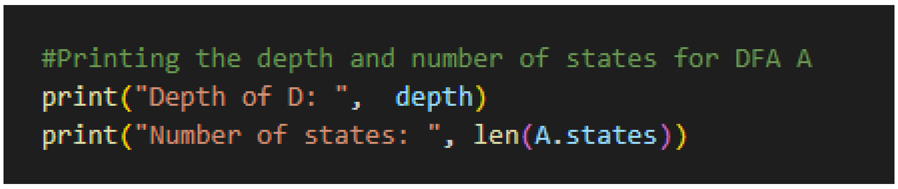
To compute the depth of the DFA *A* the breadth first search algorithm was used, however this was changed a bit by adding a list in which it saves the depth of each value of the states in the DFA. This was then used to find the longest depth in *A*.

Two functions were used for finding the maximum depth, the breadth first search algorithm was taken from [5].

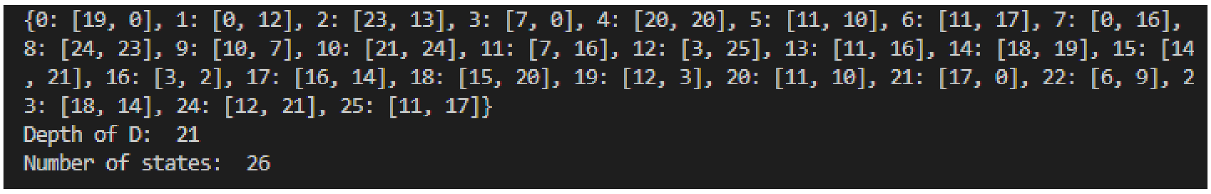
The first function called search() works by first creating three lists (visit, queue, cost). We then add the root node or in our case the initial node in the visit and queue lists. Finally, we loop the queue list until it's empty and check the transitions of the current state, if the state that is taking from the transition is not in visited, we append it to the queue and next state. However, we also added the cost list which keeps track of the depth, this is done by first checking if the current state is already in cost, if not then append the state in cost and add its value. The function returns the cost list.

The depth function then finds the largest number in the cost list and checks if the state is also a final state, if not it then uses the 2nd largest number and so on. Therefore, it finds the largest depth of the state which is also a final state.

After all of this is done the depth of *A* is printed (d_A) as well as the size.



```
#Printing the depth and number of states for DFA A
print("Depth of D: ",  depth)
print("Number of states: ", len(A.states))
```

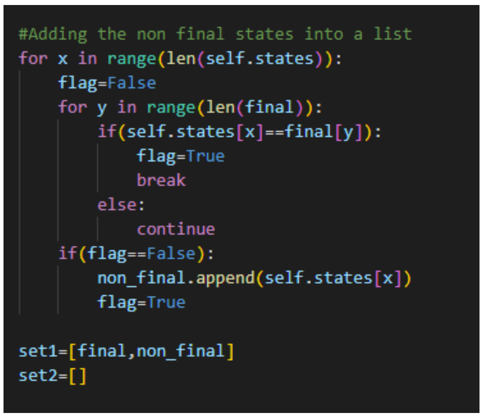


```
{0: [19, 0], 1: [0, 12], 2: [23, 13], 3: [7, 0], 4: [20, 20], 5: [11, 10], 6: [11, 17], 7: [0, 16],
8: [24, 23], 9: [10, 7], 10: [21, 24], 11: [7, 16], 12: [3, 25], 13: [11, 16], 14: [18, 19], 15: [14
, 21], 16: [3, 2], 17: [16, 14], 18: [15, 20], 19: [12, 3], 20: [11, 10], 21: [17, 0], 22: [6, 9], 2
3: [18, 14], 24: [12, 21], 25: [11, 17]}
Depth of D:  21
Number of states:  26
```

Question 3

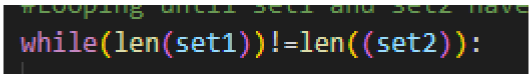
To remove duplicate states, we used Hopcroft's algorithm in which it minimises the DFA. Hopcroft works by identifying states which lead to the same transition and merges them, this is done by first splitting the states into final states and non-final states. Next it continuously partitions the sets into smaller subsets until no more partitions can be made, to implement this algorithm [6] was followed.

As we discussed we first had to partition the final states and non-final-states:



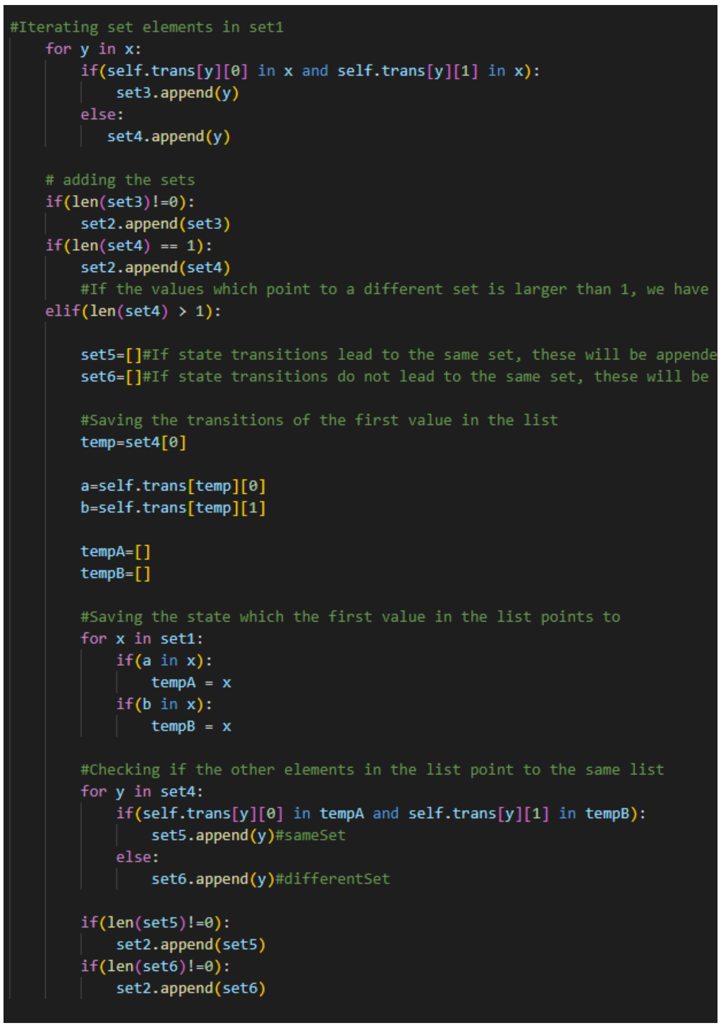
```
#Adding the non final states into a list
for x in range(len(self.states)):
    flag=False
    for y in range(len(final)):
        if(self.states[x]==final[y]):
            flag=True
            break
        else:
            continue
    if(flag==False):
        non_final.append(self.states[x])
        flag=True

set1=[final,non_final]
set2=[]
```

Next, we checked if the current sets are equal to the previous sets (if no change is done than no more partitions can be made).



```
#Looping until set1 and set2 have
while(len(set1))!=len((set2)):
```

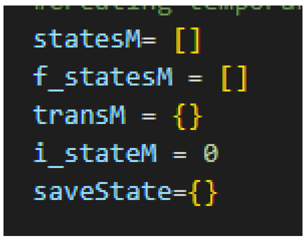
We then checked the transitions of the set, if the values pointed towards different sets these where split.



```
#Iterating set elements in set1
    for y in x:
        if(self.trans[y][0] in x and self.trans[y][1] in x):
            set3.append(y)
        else:
           set4.append(y)

    # adding the sets
    if(len(set3)!=0):
        set2.append(set3)
    if(len(set4) == 1):
        set2.append(set4)
        #If the values which point to a different set is larger than 1, we have
    elif(len(set4) > 1):

        set5=[]#If state transitions lead to the same set, these will be appende
        set6=[]#If state transitions do not lead to the same set, these will be

        #Saving the transitions of the first value in the list
        temp=set4[0]

        a=self.trans[temp][0]
        b=self.trans[temp][1]

        tempA=[]
        tempB=[]

        #Saving the state which the first value in the list points to
        for x in set1:
            if(a in x):
                tempA = x
            if(b in x):
                tempB = x

        #Checking if the other elements in the list point to the same list
        for y in set4:
            if(self.trans[y][0] in tempA and self.trans[y][1] in tempB):
                set5.append(y)#sameSet
            else:
                set6.append(y)#differentSet

        if(len(set5)!=0):
            set2.append(set5)
        if(len(set6)!=0):
            set2.append(set6)
```

After the Hopcroft algorithm was ready, we now had to create a new DFA which we called *M*. This needs to be done since some states may have merge and must be removed from the states list, final states list and removed from the transition dictionary.

A function called DFAM took care of the above by initialising new values and changing them to the new ones in *M*.



```
statesM= []
f_statesM = []
transM = {}
i_stateM = 0
saveState={}
```

If two or more states merged the name for the new state would be the first name in the list, for example if states 1,2 and 3 merged [1,2,3] the state would be named [1]. This is done since it removes the possibility of duplicate states. The new states where than added to the new states list.

The final states were chosen by checking if the current states have any values which are final states. The initial state was chosen by checking if any of the current states has the previous value of the initial state.

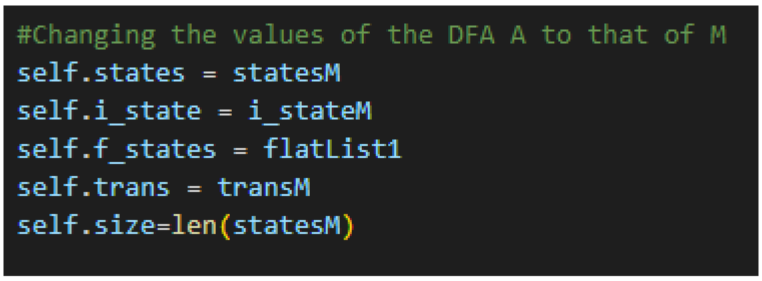
Finally, the new transitions were changed as followed, if a state has merged the values are added to a dictionary as a list with the key value of the first value. For example:

[2,3]: [4,9]

saveState{2} = [2,3]

Then a nested for loop is used to change the values if they are found in the transition dictionary. This was done since the state was previously removed.

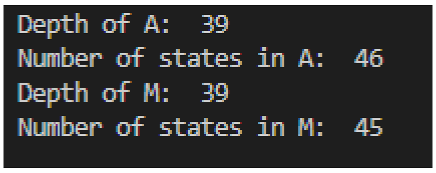
After all of this done. the initial values are change to the new ones:



```
#Changing the values of the DFA A to that of M
self.states = statesM
self.i_state = i_stateM
self.f_states = flatList1
self.trans = transM
self.size=len(statesM)
```

Question 4

To calculate the depth and the new size of the DFA *M* the same procedure was used as we did in question 2. The image below shows the DFA *A*'s depth and size and DFA *M*'s depth and size, in other words this is the same DFA however *M* is minimised.



```
Depth of A:  39
Number of states in A:  46
Depth of M:  39
Number of states in M:  45
```

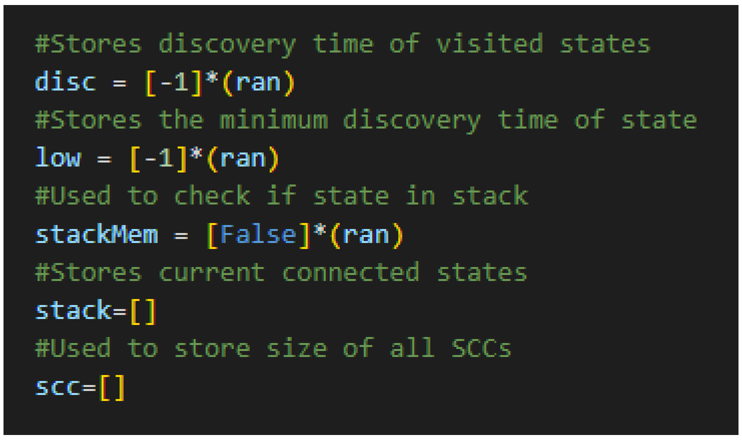
As we can see only two states merged together this instance and the depth remained the same.

Moreover, a function called removeNodes() was written but later removed, this function removed island states in DFA, however for some reason this function was interfering with the Tarjan's algorithm.

Question 5

To implement Tarjan's Algorithm the following web page was followed [7]. Tarjan's Algorithm is an algorithm used in graph theory to find strongly connected components of a directed graph or in our case a DFA, in linear time by using depth first search traversal.

This was implemented by first initialising the following variables:



```
#Stores discovery time of visited states
disc = [-1]*(ran)
#Stores the minimum discovery time of state
low = [-1]*(ran)
#Used to check if state in stack
stackMem = [False]*(ran)
#Stores current connected states
stack=[]
#Used to store size of all SCCs
scc=[]
```

Where:

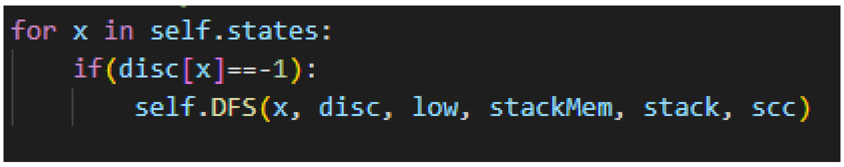
disc = is a list containing the discovery time of each visited state

low = is a list containing the minimum discovery time of each state

stackMem = is a stack to check if the state was visited stack = stores the current connected states we are checking

scc = stores the size of all the SCCs in the DFA.

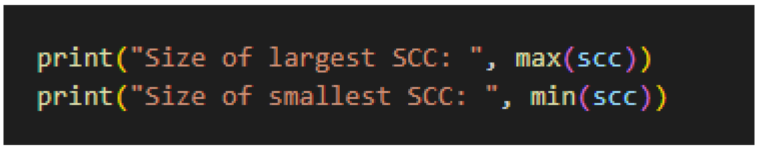
Next a function is recursively called in range of all the states



```
for x in self.states:
    if(disc[x]==-1):
        self.DFS(x, disc, low, stackMem, stack, scc)
```

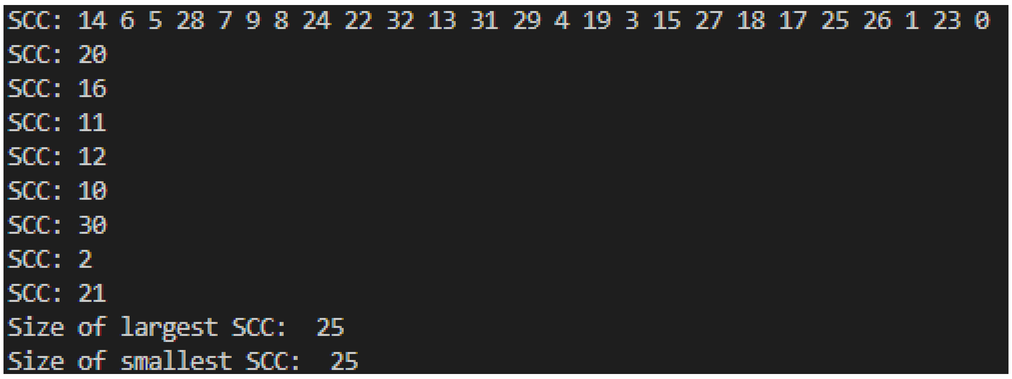
In the function DFS the transitions of the state are checked to see whether a cycle is found or not. All cycles are than stored and the size of the are saved in the scc list.

Finally, the smallest and largest values in the DFA are printed:

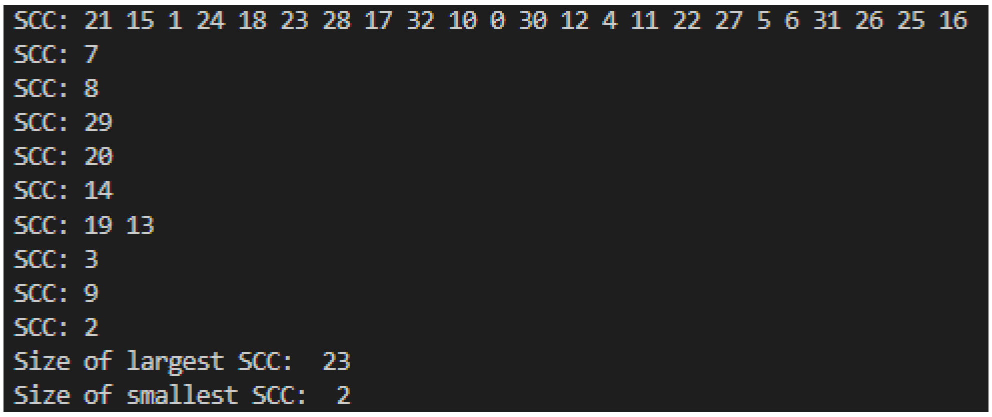


```
print("Size of largest SCC: ", max(scc))
print("Size of smallest SCC: ", min(scc))
```

Below a dry run of the Tarjan's algorithm ca be seen, sometimes DFA does contain a few SCCs but mostly there are no SCCs in the DFA.



```
SCC: 14 6 5 28 7 9 8 24 22 32 13 31 29 4 19 3 15 27 18 17 25 26 1 23 0
SCC: 20
SCC: 16
SCC: 11
SCC: 12
SCC: 10
SCC: 30
SCC: 2
SCC: 21
Size of largest SCC:  25
Size of smallest SCC:  25
```



```
SCC: 21 15 1 24 18 23 28 17 32 10 0 30 12 4 11 22 27 5 6 31 26 25 16
SCC: 7
SCC: 8
SCC: 29
SCC: 20
SCC: 14
SCC: 19 13
SCC: 3
SCC: 9
SCC: 2
Size of largest SCC:  23
Size of smallest SCC:  2
```

Question 6

Johnson’s algorithm is already explained in the beginning of this documentation.

Conclusion

Overall, in this assignment I believe that I understood how to create and manipulate DFA’s as well as using algorithms to minimise and find certain characteristics of a given DFA. I have understood the necessary algorithms I had to implement which are Hopcroft’s, Tarjan’s, Breadth First Search and Depth First Search. Finally, I also understood the use of Johnson’s algorithm and why it is preferred when the number of edges in a graph/DFA is small.

Sources

[1] (28 Aug, 2019). Practice problems on finite automata. Available: https://www.geeksforgeeks.org/practice-problems-finite-automata/.

[2] (22 Mar, 2017). Johnson’s algorithm for All-pairs shortest paths. Available: https://www.geeksforgeeks.org/johnsons-algorithm/

[3] (24 May. 2022). Johnson’s Algorithm. Available: https://brilliant.org/wiki/johnsons-algorithm/

[4] (). Deterministic Finite Automaton. Available: https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm.

[5] (28 Apr, 2022). Breadth First Search or BFS for a Graph. Available: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[6] DavidWesselsVIU, "Compilers: Hopcroft's algorithm," Jan 11, 2021.

[7] (14 Jan, 2022). Tarjan’s Algorithm to find Strongly Connected Components. Available: https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/.

Plagiarism Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person,whether or not such work has been published" (Regulations Governing Conduct at Examinations,1997, Regulation 1 (viii), University of Malta).

I, the undersigned, declare that the assignment submitted is my work, except where acknowledged and referenced.

I understand that the penalties for making a false declaration may include, but are not limited to, 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.

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

Liam Bugeja Douglas   LiamBD

Student Name

ICS2210   Data Structures and Algorithms 2, Course Project 2022

Course Code   Title of work submitted

19/05/2022

Date