Name: **Jamie Grech**
ID: **123499M**
Course: **B.Sc. Information Technology (Hons.) (Artificial Intelligence)**

Faculty of Information and Communication Technology
Department of Artificial Intelligence

Study-unit: **Data Structures and Algorithms 2**
Code: **ICS2210**

Statement of Completion

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

All the required components of the assignment were completed successfully using the Python programming language.

Question 1

Two classes, State and Graph, defined within their respective files State.py and Graph.py are used in the construction of the DFSA used in this assignment.

The State class

The State class has one constructor which takes three parameters, accepting, transitions and stateid. The accepting parameter is expected to be a Boolean value which specifies whether or not the state is an accepting state/ final state. The transitions parameter is expected to be a dictionary mapping the characters in the alphabet of the DFSA, in our case a and b, to the ids of the states transitioned to from the given state when given the respective character. The stateid parameter is used to assign the numerical id of the state within the DFSA it is used in. Besides the aforementioned properties, the State class also contains a Boolean flag 'onstack' and two integer properties 'order' and 'link' which are used when implementing Tarjan's algorithm later on. The use of dictionaries to map characters in the alphabet to their corresponding transitions is a form of an implementation of an adjacency list, which I deemed to be more appropriate given the nature of the automaton requested. The automaton requested is specified to have only 2 edges for every state, while there can be up to 64 states. This makes our graph sparsely connected which **justifies the use of an adjacency list as being more efficient than an adjacency matrix.**

The Graph class

The Graph class takes two parameters, numofstates and name. The name parameter is simply a string parameter used to assign a name to the graph for the sake of making it easier to differentiate between the automata A and M when printing out their details. The numofstates parameter is an int parameter used to generate a number of states with randomized properties using the python random library. The state's accepting property is determined by using the random.getrandbits function to generate a 0 or 1 value randomly and convert it to a Boolean. The transitions are generated by using random.randint to generate a random integer within the range of states in the graph i.e. between 0 and numofstates -1, both inclusive, for each character in the alphabet, i.e. a and b. After generating the two random numbers, if it is found that both a and b lead to the same state, the number for the transition leading from b is generated again using the same method until a different number is generated. It was assumed that states with both characters transitioning to the same state where not desired since it was not specifically stated to be allowed within the assignment specification, unlike transitions from a state to itself being allowed. After a state's properties are randomly generated and it is assigned its index based on the number of states generated before it, it is added to the graph's list 'states', and if the state is marked as accepting it is also added to the graph's 'finalstates' list. After all the states are generated, a random number between 0 and numofstates-1 is generated to determine which state will be the starting state. The state whose id is the number generated has it's start property set to True and the graph's startid property is set to the generated number. Next, the graph's calculatedepth method which will be described later on is called which returns a list of Booleans, each corresponding to the whether the state in the graph's states list at the respective index is reachable in any way from the starting state. Finally, the state list is iterated through one last time and each state deemed to be reachable is added to another list 'reachablestates'. The graph class also has a 'depth' property to store the depth of the automaton calculated by the calculatedepth method mentioned earlier which is required in question 2/4 of the assignment. The class also has 5 more properties, 'index',

‘stack’, ‘largestSCC’, ‘smallestSCC’ and ‘sccs’ which are used for Tarjan’s algorithm in question 6. When the graph constructor is called in the main file, the numofstates parameter is randomly generated between 16 and 64 as specified in the assignment description.

Question 2/4

The calculatedepth function is defined for the graph class in order to calculate the depth of the graph as well as check which nodes are reachable, using the breadth first search algorithm. This is done as follows:

1. A list of Boolean flags called ‘visitednodes’ is initialized with a number of elements equal to the number of states in the graph and each index being set to false.
2. A list called searchqueue is declared which will store the ids of states in queue to be visited and the initial depth of the graph will be set to 0.
3. The index of the starting node of the graph is added to the searchqueue, followed by a value of -1, and the node corresponding to the index of the starting node in the visitednodes list is set to true. The value of -1 in the searchqueue is used to indicate that all the nodes in the current depth level of the graph have been visited.
4. The first state in the queue is popped from the queue and its neighbours which haven’t yet been visited are pushed to the end of the queue and marked as visited. If the value popped is a -1, another -1 is popped to the end of the queue. The next element in the queue is then checked. If it is found to be a -1, the loop is broken and the list of visitednodes is returned. If not, the depth is increased by 1 and the loop continues.
5. Step 4 is repeated until the aforementioned condition of having two -1s in a row, indicating all reachable nodes have been visited, is satisfied.

The function was evaluated by generating multiple smaller graph (i.e. with less states) and running through them step by step using the PyCharm IDE’s debugger and ensuring that it was working as intended.

Question 3

For the minimization of the automaton, I opted for Hopcroft's algorithm as when comparing efficiency of the two algorithms, I found that although bought Moore's and Hopcroft's algorithms have an average time complexity of O(n log log n), the worst case complexity of Moore's algorithm (O(n log n)) is worse than that of Hopcroft's algorithm(O(n log log n)) [1]. Thus I decided to implement Hopcroft's algorithm, based off the pseudocode below [2]:

```
P := {F, Q \ F};
W := {F};
while (W is not empty) do
     choose and remove a set A from W
     for each c in Σ do
          let X be the set of states for which a transition on c leads to a state in A
          for each set Y in P for which X ∩ Y is nonempty and Y \ X is nonempty do
               replace Y in P by the two sets X ∩ Y and Y \ X
               if Y is in W
                    replace Y in W by the same two sets
               else
                    if |X ∩ Y| <= |Y \ X|
                         add X ∩ Y to W
                    else
                         add Y \ X to W
          end;
     end;
end;
```

The new graph starts out as a deepcopy of the graph to be minimized since simply creating a new graph being equal to it would act as a pointer to it and thus lose the properties of the initial graph. Python's innate set functions were used for the set difference and intersection used in the pseudocode. This algorithm returns a list of sets whereby each set represents one or more states which are deemed equivalent to eachother in the scope of the automaton. Each of these sets were then iterated through to check whether the states inside it are final states and whether it contains the starting state, as well as which of the other sets contains the states which the states inside it should transition to. After all these properties are checked the new state corresponding to that set is created with the respective properties and added to a list 'newstates'. The startid of the new graph is set to the index of the set containing the starting state in the partition and all the properties of the new graph are set based on the newstates list. The minimize function which carries out all the aforementioned processing also takes a string as an optional parameter to set the name property of the minimized graph. The minimization was evaluated by checking the properties of the minimized graph to see that it does indeed have less states than the original graph, then evaluate the 100 strings randomly generated in question 5 with both the minimized and unminimized graphs. After running the program multiple times, it was noted that sometimes a number of odd strings did not give the same result when classified with the minimized and unminimized automata, the reason for which remains unknown. Despite this for the most part all 100 strings generated would give the same result when classified by both automata.

Question 5

Generatestring

The generatestring function was defined to generate a string of random length between 0 and 128 characters. First a random number between 0 and 128 is generated using the python random package to determine how many characters the string will have. The string to be generated would then start as an empty string and a random binary value would be generated, with a being appended to the string if the value is 0 and b being appended if the value is 1. This is repeated for however many characters the string should have according to the number generated initially. Therefore, if the initial number is 0 the string would simply be empty. Since strings in python are immutable in python the characters are appended by creating a new string consisting of the old string concatenated with the character to be appended.

Evaluatestring

The evaluatestring function is a function for the graph class which starts at the starting state of a graph, takes a string as input and iterates through the string, transitioning to the appropriate state after each character. Finally, after the whole string has been processed, “Accepting” or “Rejecting” is returned depending on whether or not the resulting state at the end of the string is a final state. Since the states in our graph data structure’s state list are ordered according to their index unless manually changed in the driver program, they are directly accessed in the list and not searched for, thus giving this process a linear time complexity with the affecting factor being the number of characters in the string. It was manually evaluated by using the IDE’s debugger to go through the process step by step.

Question 6

My implementation for Tarjan's algorithm was based off the following pseudocode [3]:

Algorithm 1 Tarjan

```
function StrongConnect(vertex u)
    num ← num + 1                                   ▷ increment num
    order(u) ← num                                  ▷ set order(u) to smallest unused number
    link(u) ← order(u)                              ▷ least order(v) accessible is u itself
    push u on S
    for all neighbors v of u do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
            link(u) ← min(link(u), link(v))
        else if v is on stack S then                ▷ v is in current component
            link(u) ← min(link(u), order(v))
    if link(u) = order(u) then                      ▷ u is root of component, create SCC
        create new strongly connected component
        repeat
            v ← top of S
            add v to strongly connected component
            pop top from S
        until u = v
function Tarjan(G(V, E))
    num ← 0
    initialize new empty stack S
    for all vertices v ∈ V do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
```

The main adjustment made was that when checking whether the stack contains the current node being checked, instead of searching through the stack for the node, each node has a Boolean flag 'onstack' mentioned previously which is set accordingly, so as to make this check occur in constant time. Having this check occur in constant time (O(1)) allows for the whole algorithm overall to run in linear time [4] based on the number of states in the automaton. The function for this algorithm was tested using the IDE's debugger on a number of smaller automata.

Bibliography

[1] J. David, Average Complexity of Moore's and Hopcroft's Algorithms. 2010, pp. 22-23. [Accessed: 29- May- 2019].

[2] "DFA minimization", en.wikipedia.org, 2019. [Online]. Available: https://en.wikipedia.org/wiki/DFA_minimization. [Accessed: 29- May- 2019].

[3] C. Abela, "Analysing Graphs", University of Malta, 2019.

[4]"Tarjan's strongly connected components algorithm", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm. [Accessed: 29- May- 2019].