ICS2210 - Data Structures and Algorithms 2

Nathan Bonavia Zammit,

Bachelor of Science in

Information Technology (Honours)

(Artificial Intelligence)

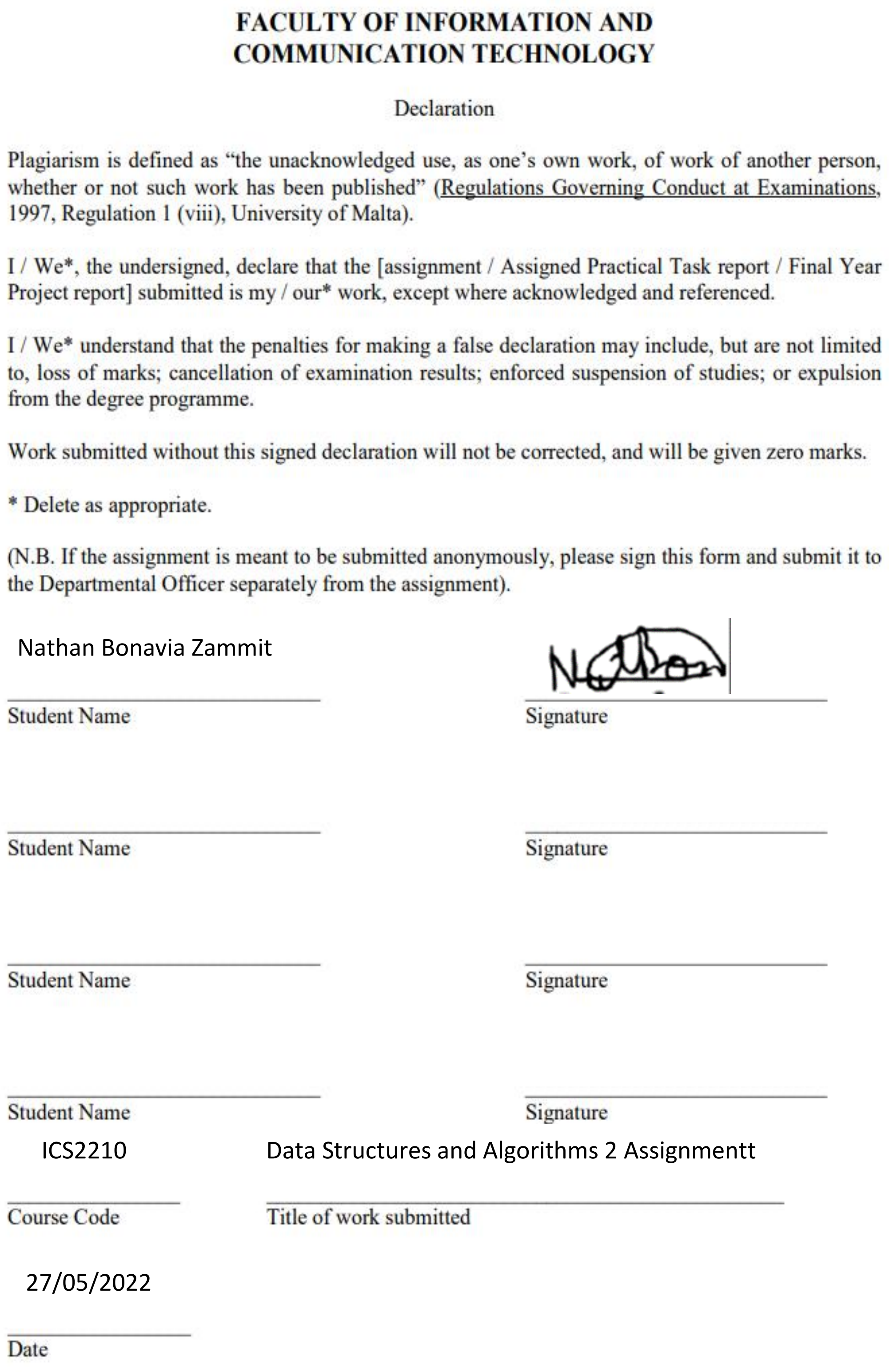
May 2022

Contents

Statement of Completion

Item	**Completed (Yes/No/Partial)**
Created 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
Provided a good discussion on Johnson’s algorithm	Yes
Included a good evaluation in your report	Yes

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 / 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 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.

* 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).

Nathan Bonavia Zammit	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2210	Data Structures and Algorithms 2 Assignmentt
Course Code	Title of work submitted

27/05/2022

Date

Question 1

The classes State and Graph are used in order to construct the required deterministic finite state automata (DFSA) that are used in this assignment.

Starting with the State class, one constructor is present, which takes 3 parameters. These parameters being, 'accepting', 'transitions' and 'stateid'. The 'accepting' parameter is set as a Boolean value which will specify whether or not the state is either an accepting state or a final state. The 'transitions' parameter is expected to be a dictionary which maps the characters in the alphabet of the DFSA, in the following code used this will consist of 'a' and 'b', to the ids of the states which will transition to form 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. The use of dictionaries to map characters in the alphabet to their corresponding transitions is a form of an implementation of an adjacency list. The automaton that was requested is specified to have only 2 edges for each state with a limit of up to 64 states. Therefore, the graph is sparsely connected and as such the use of an adjacency list instead of an adjacency matrix is further more justified.

Besides the 3 mentioned parameters, the State class also contains a Boolean flag 'onstack', and two integers 'order' and 'link'. These will be used for when it comes to implementing Tarjan's algorithm further on.

The Graph class takes up two more parameters, these being 'numofstates' and 'name'. The 'name' parameter is a String parameter which is used to assign a name to the graph in order to make it far easier to differentiate between automata A and M. The 'numofstates' parameter is an integer parameter which is used to generate a number of states with randomized properties via the python library 'random'. A state is determined whether it is accepting or not via the use of the 'random.getrandbits' function to generate either a 0 or a 1 value randomly which will in turn be converted into a Boolean. The transitions are then generated via the use of 'random.randint' in order to have a random integer within the range of states in the graph, this range being between 0 and 'numofstates' -1, for each character in our alphabet. After the generation of 2 random numbers, if both 'a' and 'b' end up leading to the same state, then the number for the transition that is leading from 'b' will be regenerated through the same method, until a different number will be generated. This was done assuming that states with both characters transitioning into the same state were not desired. After a state's properties are generated randomly and it is assigned its index based on the number of states generated before it, it is then added to the graph's 'states' list. If the state is marked as accepting, then it will be also added to the graph's 'finalstates' list. After all the required states are generated, a random number between 0 and numofstates -1 is determined to pick a state to be the starting state. The state whose id is the number generated will have it's start property set to True and therefore, the graph's 'startid' property is then set to said number.

The 'calculatedepth' method which is looked into further detail later on, is called and will return a list of Booleans, with each corresponding to whether the state in the graph's 'states' list is reachable in any possible way from the starting state. Then the state list is iterated through and each each state which is reachable is added to the list 'reachablestates'. The 'depth' property is used to store the depth of the automaton which is calculated by the 'calculatedepth' method. This was required for questions 2 & 4.

The 5 properties 'index', 'stack', 'largestSCC', 'smallestSCC' and 'sccs' are used for Tarjan's algorithm in question 5. When the graph constructor is called, the 'numofstates' parameter is randomly generated between 16 and 64, as requested.

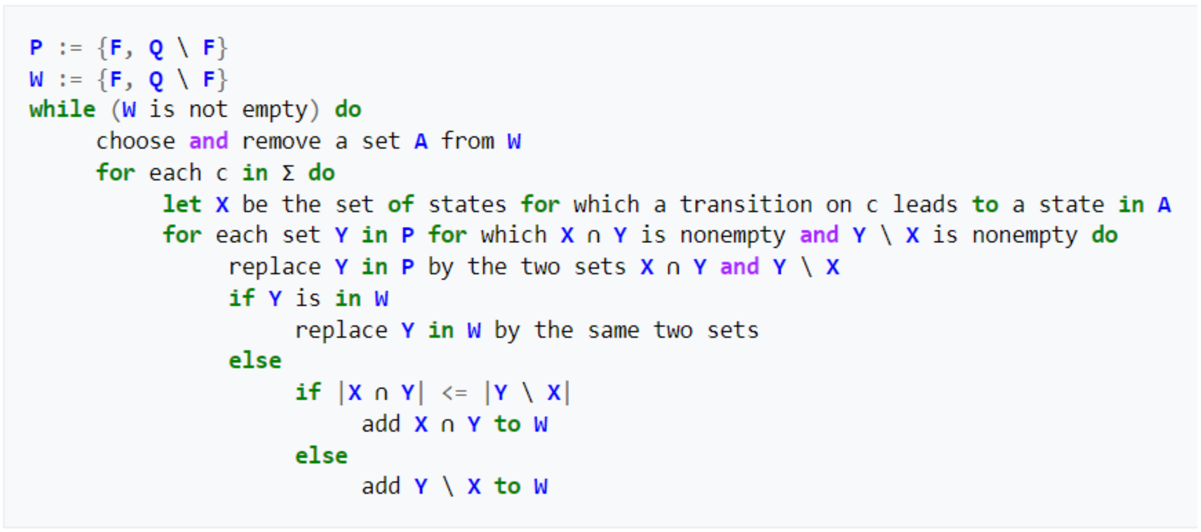
Question 2 & 4

The ‘calculatedepth’ function is defined in the Graph class in order to calculate the depth of the graph as required. This was done via the breadth first search algorithm which is explained down below [1].

1. ‘visitednodes’, a list of Boolean flags, is initialized with a number of elements which are equal to the number of states in the graph, and each index is set to false
2. ‘searchqueue’ is declared which will store the ids of states in queue to be visited. The initial depth of the graph is also 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 of end of the queue. The next element in the queue is then checked and if it is found to be a -1, then the loop will be broken and the list of ‘visitednodes’ is returned. Otherwise, the depth is increased by 1 and the loop will not break.
5. Step 4 will be repeated until the aforementioned condition is reached which will indicate that all reachable nodes have been visited.

Question 3

For the minimization of the automata Hopcroft's algorithm was used instead of Moore's algorithm. This was because even though both Moore's & Hopcroft's algorithms have an average time complexity of O(n log log n), the worst case complexity of Moore's (O(n log n)) is worse than that of Hopcroft's (O(n log log n)) [2]. The code used was based off following pseudocode [3]:



```
P := {F, Q \ F}
W := {F, Q \ 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
```

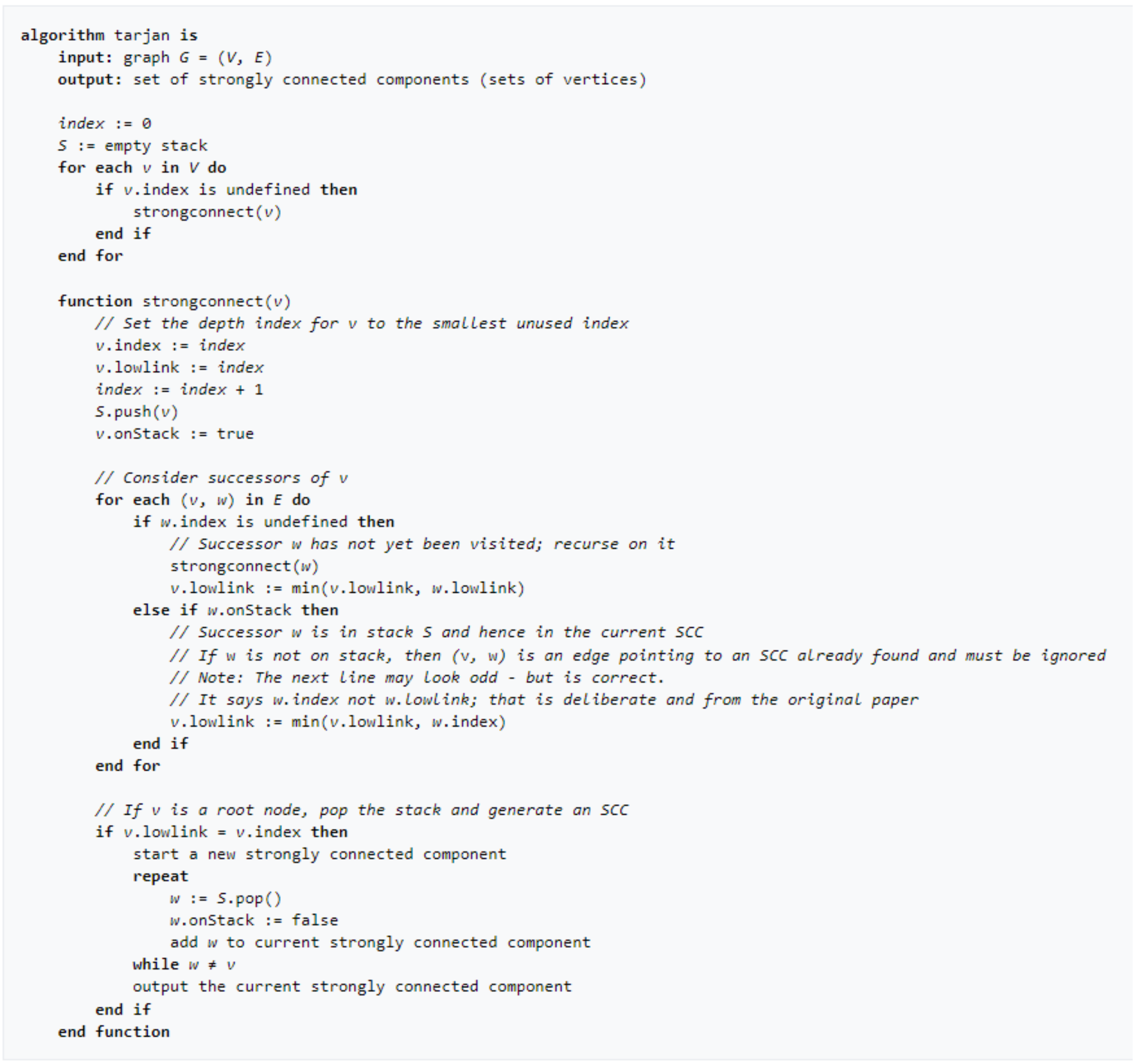
Fig.1. Pseudocode used for Minimization

The new graph, in this case 'M' starts as a deepcopy of the graph which is required to be minimized. This was done as creating a new graph equal to it would simply act as a pointer and therefore lose the properties of the initial graph. The set difference and intersection were done using python's innate set functions. This algorithm returns a list of sets whereby each set represents one or more states which are deemed equivalent to each other in the scope of the automata.

Each of the sets were then iterated through in order to check if the states inside are final states and if the starting state is present as well. It is also determined which of the other sets contains the states which the states inside should transition to. After these properties are checked, the new state corresponding to that set is created with the respective properties and then 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 new properties of the new graph are set based on the 'newstates' list.

Question 5

When it came to implementing Tarjan's algorithm to find the strongly connected components in 'M', the following pseudo code was referenced [4]:



```
algorithm tarjan is
    input: graph G = (V, E)
    output: set of strongly connected components (sets of vertices)

    index := 0
    S := empty stack
    for each v in V do
        if v.index is undefined then
            strongconnect(v)
        end if
    end for

    function strongconnect(v)
        // Set the depth index for v to the smallest unused index
        v.index := index
        v.lowlink := index
        index := index + 1
        S.push(v)
        v.onStack := true

        // Consider successors of v
        for each (v, w) in E do
            if w.index is undefined then
                // Successor w has not yet been visited; recurse on it
                strongconnect(w)
                v.lowlink := min(v.lowlink, w.lowlink)
            else if w.onStack then
                // Successor w is in stack S and hence in the current SCC
                // If w is not on stack, then (v, w) is an edge pointing to an SCC already found and must be ignored
                // Note: The next line may look odd - but is correct.
                // It says w.index not w.lowlink; that is deliberate and from the original paper
                v.lowlink := min(v.lowlink, w.index)
            end if
        end for

        // If v is a root node, pop the stack and generate an SCC
        if v.lowlink = v.index then
            start a new strongly connected component
            repeat
                w := S.pop()
                w.onStack := false
                add w to current strongly connected component
            while w ≠ v
            output the current strongly connected component
        end if
    end function
```

Fig.2. Pseudocode used for Tarjan's algorithm

A strongly connected component (SCC) is the portion of a directed graph in which there is a path from each vertex to another vertex [5]. Kosaraju's algorithm is often used to find these, however in our case we had to look into Tarjan's algorithm. Both algorithms are DFS based algorithms which are used to find the SCC of directed graphs in linear time complexity. The time complexity is represented as O(V+E) where V represents the number of vertices while E represents the number of edges of the graph [6].

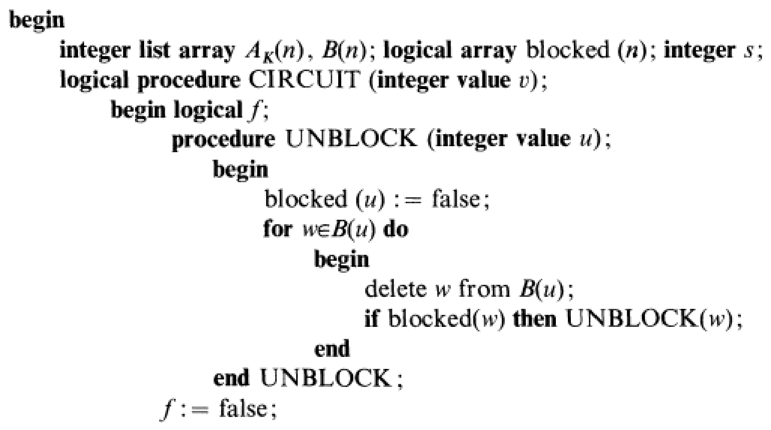
Question 6

In Johnson's algorithm the time consumed between the output of two consecutive circuits as well as before the first and after the last circuits never exceeds the size of the graph, O(n + e). Elementary circuits are constructed from a root vertex s in the sub-graph induced by s and vertices "larger than s" in some ordering of the vertices. Therefore, the output is grouped according to the least vertices of the circuits.

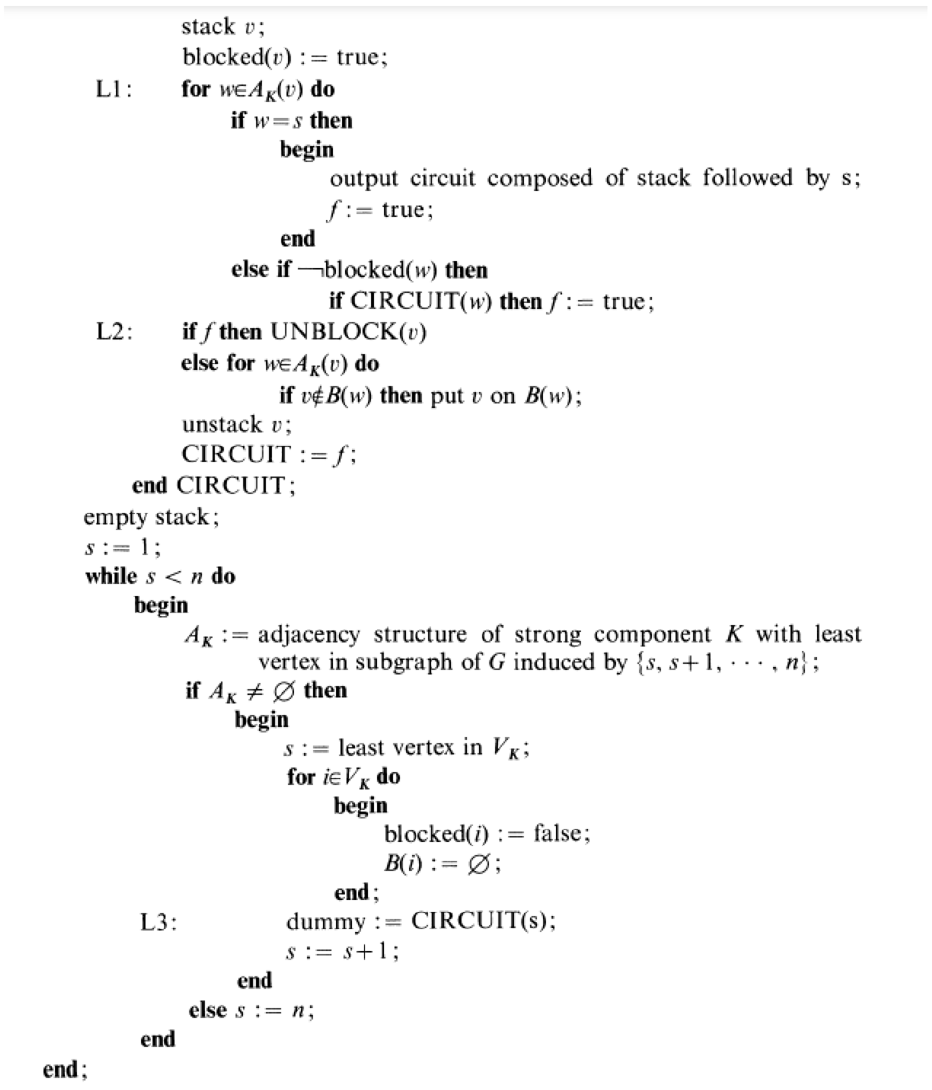
In order to avoid duplicating circuits, a vertex v is 'blocked' when it is added to some elementary path beginning in s. It will stay blocked as long as every path form from v to s intersects the current elementary path at a vertex other than s. Furthermore, a vertex does not become a root vertex for constructing elementary paths unless it is the least vertex in at least one elementary circuit. These two features avoid a lot of the meaningless searching which Tiernan's, Weinblatt's, and Tarjan's, algorithms go through.

The algorithm accepts a graph G represented by an adjacency structure AG composed of an adjacency list AG(v) for each v ∈ V. The list AG(v) contains u if and if edge (v, u) ∈ E. The algorithm assumes that vertices are represented by integers from 1 to n.

The algorithm proceeds by building elementary paths from s. The vertices of the current elementary path are kept on a stack. A vertex is appended to an elementary path by a call to the procedure circuit and is deleted upon return from this call. When a vertex v is appended to a path it is bocked by setting blocked (v) = true, so that v cannot be used twice on the same path. Upon return from the call which blocks v, however, v is not necessarily unblocked. Unblocking is always delayed sufficiently so that any two unblockings of v are separated by either an output of a new circuit or a return to the main procedure.



```
begin
    integer list array $A_K(n)$, $B(n)$; logical array blocked $(n)$; integer $s$;
    logical procedure CIRCUIT (integer value $v$);
        begin logical $f$;
            procedure UNBLOCK (integer value $u$);
                begin
                    blocked $(u) :=$ false;
                    for $w \in B(u)$ do
                        begin
                            delete $w$ from $B(u)$;
                            if blocked$(w)$ then UNBLOCK$(w)$;
                        end
                end UNBLOCK;
            $f :=$ false;
```



```
            stack $v$;
            blocked($v$) := true;
    L1:     for $w \in A_K(v)$ do
                if $w = s$ then
                    begin
                        output circuit composed of stack followed by s;
                        $f$ := true;
                    end
                else if $\neg$blocked($w$) then
                        if CIRCUIT($w$) then $f$ := true;
    L2:     if $f$ then UNBLOCK($v$)
            else for $w \in A_K(v)$ do
                    if $v \notin B(w)$ then put $v$ on $B(w)$;
            unstack $v$;
            CIRCUIT := $f$;
        end CIRCUIT;
    empty stack;
    $s := 1$;
    while $s < n$ do
        begin
            $A_K$ := adjacency structure of strong component $K$ with least
                    vertex in subgraph of $G$ induced by $\{s, s+1, \cdots, n\}$;
            if $A_K \neq \varnothing$ then
                begin
                    $s$ := least vertex in $V_K$;
                    for $i \in V_K$ do
                        begin
                            blocked($i$) := false;
                            $B(i) := \varnothing$;
                        end;
        L3:         dummy := CIRCUIT(s);
                    $s := s+1$;
                end
            else $s := n$;
        end
end;
```

Fig.3. Pseudocode for Johnson’s Algorithm

References:

[1] – Geeks for Geeks, Breadth First Search inspiration that was used in code, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[2] J. David, Average Complexity of Moore's and Hopcroft's Algorithms. 2010. pgs. 22-23. [Accessed: 26/05/2022]

[3] – "DFA Minimization", Pseudocode for Hopcroft's Minimization, https://en.wikipedia.org/wiki/DFA_minimization

[4] – Tarjan's algorithim Pseudocode, https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

[5] – "Strongly Connected Components", https://www.programiz.com/dsa/strongly-connected-components#:~:text=A%20strongly%20connected%20component%20is,only%20on%20a%20directed%20graph

[6] – Comparison of Tarjan's and Kosaraju's algotihms, https://www.geeksforgeeks.org/comparision-between-tarjans-and-kosarajus-algorithm/

[7] Donald B. Johnson, Finding All The Elementary Circuits Of A Directed Graph. 1975. Pgs 4-5. [Accessed: 27/05/2022]