

ICS 2210 Assignment Documentation

Contents

Evaluation of implementation	2
Task One - Constructing the deterministic finite state automaton	2
Task Two – Computing the depth	3
Task Three – Minimization using Hopcroft’s algorithm	4
Task Four – Recomputing the depth.....	6
Task Five – Strongly connected components.....	6
Johnson’s algorithm	8
References	10
Statement of Completion	11

Evaluation of implementation

Task One - Constructing the deterministic finite state automaton

Before the random DSA was implemented, two data structures were considered, these are the *Adjacency List* and *Adjacency Matrix*. Comparing the two with regards to time complexity, the adjacency list was chosen based on these accounts: Due to its space complexity $O(|V|+|E|)$, adding a vertex $O(1)$ and finally, removing a vertex $O(|V|+|E|)$ vs the latter's $O(|V|^2)$ for space, insertion and deletion. Thus, an adjacency list was implemented represented as dictionary list (Keys being the vertex's while the values are the edges).

States were created between 16 and 64 inclusive, for each state a coin was flipped to determine if a state is accepting or not, and the accepting states were stored within a List. With this the dictionary list "*myAdjacency*" was populated and with the *add_edges* function, each state received two outgoing edges 'a' and 'b' such that any other random state, including the state itself would receive an inbound edge. The following is sample output of the adjacency created:

```
----- Task 1 -----

Number of nodes in DFA: 21
DFA pre-Minimization:
0 --> [[16, 'a'], [0, 'b']]
1 --> [[18, 'a'], [1, 'b']]
2 --> [[0, 'a'], [4, 'b']]
3 --> [[18, 'a'], [7, 'b']]
4 --> [[7, 'a'], [7, 'b']]
5 --> [[3, 'a'], [6, 'b']]
6 --> [[8, 'a'], [18, 'b']]
7 --> [[15, 'a'], [3, 'b']]
8 --> [[15, 'a'], [3, 'b']]
9 --> [[4, 'a'], [19, 'b']]
10 --> [[12, 'a'], [9, 'b']]
11 --> [[5, 'a'], [8, 'b']]
12 --> [[0, 'a'], [8, 'b']]
13 --> [[16, 'a'], [5, 'b']]
14 --> [[6, 'a'], [0, 'b']]
15 --> [[12, 'a'], [12, 'b']]
16 --> [[7, 'a'], [16, 'b']]
17 --> [[10, 'a'], [7, 'b']]
18 --> [[6, 'a'], [17, 'b']]
19 --> [[11, 'a'], [16, 'b']]
20 --> [[13, 'a'], [1, 'b']]
Accepting states: [0, 1, 3, 5, 9, 10, 12, 20]
```

Figure 1 sample output

```
----- Task 1 -----

Number of nodes in DFA: 36
DFA pre-Minimization:
0 --> [[6, 'a'], [17, 'b']]
1 --> [[12, 'a'], [29, 'b']]
2 --> [[11, 'a'], [8, 'b']]
3 --> [[17, 'a'], [24, 'b']]
4 --> [[8, 'a'], [5, 'b']]
5 --> [[13, 'a'], [26, 'b']]
6 --> [[3, 'a'], [8, 'b']]
7 --> [[8, 'a'], [23, 'b']]
8 --> [[19, 'a'], [26, 'b']]
9 --> [[12, 'a'], [2, 'b']]
10 --> [[33, 'a'], [34, 'b']]
11 --> [[17, 'a'], [29, 'b']]
12 --> [[20, 'a'], [28, 'b']]
13 --> [[32, 'a'], [16, 'b']]
14 --> [[24, 'a'], [12, 'b']]
15 --> [[24, 'a'], [1, 'b']]
16 --> [[16, 'a'], [10, 'b']]
17 --> [[17, 'a'], [16, 'b']]
18 --> [[17, 'a'], [31, 'b']]
19 --> [[3, 'a'], [15, 'b']]
20 --> [[4, 'a'], [2, 'b']]
21 --> [[12, 'a'], [9, 'b']]
22 --> [[13, 'a'], [32, 'b']]
23 --> [[16, 'a'], [14, 'b']]
24 --> [[4, 'a'], [24, 'b']]
25 --> [[20, 'a'], [5, 'b']]
26 --> [[0, 'a'], [13, 'b']]
27 --> [[20, 'a'], [9, 'b']]
28 --> [[14, 'a'], [9, 'b']]
29 --> [[29, 'a'], [23, 'b']]
30 --> [[31, 'a'], [19, 'b']]
31 --> [[20, 'a'], [32, 'b']]
32 --> [[28, 'a'], [29, 'b']]
33 --> [[26, 'a'], [11, 'b']]
34 --> [[23, 'a'], [0, 'b']]
35 --> [[22, 'a'], [11, 'b']]
Accepting states: [0, 1, 7, 9, 13, 14, 21, 23, 25, 26, 28, 31, 33, 34, 35]
```

Figure 2 sample out

Task Two – Computing the depth

Due to there already being a list which stores all the states, simply printing the length of this list shows how many states are in the DFA. The depth however would need to be computed and this was achieved due to the following fact: the depth of a graph can be achieved by returning the deepest level of the graph. Thus, by simply performing a breath first search, an algorithm which traverses a graph level by level and incrementing a counter every time a new level is reached, the depth will then also be achieved. Due to the use of an adjacency list being used the time complexity of calculating the depth is that of $O(V+E)$, ($O(V**2)$ if an adjacency matrix was used).

This algorithm was successfully implemented by using a queue from *collections* that appends each node being traversed to the back of the queue while traversing the nodes at the front, with this it is ensured that the graph will be traversed level by level. To avoid any potential cycles in the graph a visited list is used to ensure that no already visited nodes are re-visited leading to an endless cycle and finally when all the levels are explored, the depth is returned.

The results were tested by setting the DFA generation to fewer states, if for fewer states the depth is accurately returned, then for any size the depth shall be correct. Then by running several instances of the algorithm correctness was verified by manually checking the depth of the smaller DFA.

An example of testing is as follows:

- The image on the left shows a DFA of size 8 with a depth of three, drawing this DFA results in the image to the right which has a matching DFA of depth three thus depth is correct.

```

----- Task 1 -----

Number of nodes in DFA: 8
DFA pre-Minimization:
0 ---> [[1, 'a'], [2, 'b']]
1 ---> [[5, 'a'], [4, 'b']]
2 ---> [[0, 'a'], [2, 'b']]
3 ---> [[0, 'a'], [0, 'b']]
4 ---> [[0, 'a'], [4, 'b']]
5 ---> [[3, 'a'], [6, 'b']]
6 ---> [[0, 'a'], [4, 'b']]
7 ---> [[6, 'a'], [6, 'b']]
Accepting states: [0, 3, 4, 7]

----- Task 2 -----

Number of states in network: 8
Depth of Network: 3
  
```

Figure 3 Correct Depth and size of DFA

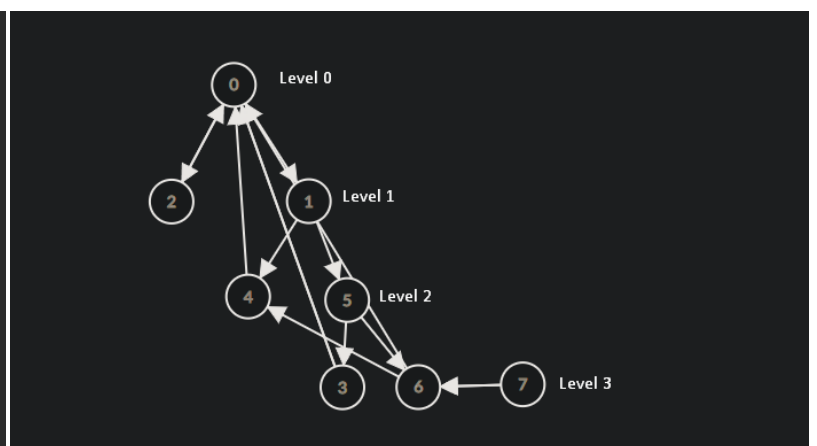


Figure 4 Drawn DFA showcasing fig 3's accuracy

Task Three – Minimization using Hopcroft's algorithm

For a DFA to be minimized, an import factor needs to be considered *the language* of the DFA. If at any point the DFA's language is changed while minimization, then it is not successful. There are two ways a DFA can be minimized:

- Removal of unreachable states.
- Combining non-distinguishable states.

Removal of unreachable states

“A state is *u* reachable by *v* if there is an input string which takes us from state *u* to *v*” [1]. Starting from the root node, any other nodes that cannot be reached from it, then that node is considered unreachable.

Take the right image as an example, all nodes in the graph are reachable except the node labelled 7. Its unreachable cause starting from the root node 0, no traversal can lead us to 7 thus, to minimize the DFA this node would have to be pruned out from the DFA.

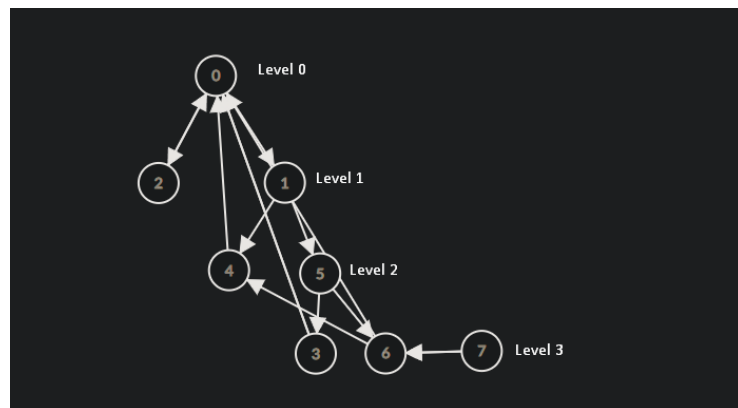


Figure 5 Showcasing node 7 as unreachable

This is quite simple to achieve with what has been done so far, we have already traversed the graph by way of a BFS. Thus, if the visited list is returned, by way of set difference the set of unreachable states *A* can be found by taking the set of all the nodes in the list *B*, and all the nodes present in visited *C* and simply performing set difference such that $A = B \setminus C$. After, since the graph is stored as a dictionary, simply going through the list of unreachable states, and performing deletion on the key of the unreachable node.

```

Number of nodes in DFA: 8
DFA pre-Minimization:
0 ---> [[1, 'a'], [2, 'b']]
1 ---> [[5, 'a'], [4, 'b']]
2 ---> [[0, 'a'], [2, 'b']]
3 ---> [[0, 'a'], [0, 'b']]
4 ---> [[0, 'a'], [4, 'b']]
5 ---> [[3, 'a'], [6, 'b']]
6 ---> [[0, 'a'], [4, 'b']]
7 ---> [[6, 'a'], [6, 'b']]
Accepting states: [0, 3, 4, 7]
  
```

----- Task 2 -----

```

Number of states in network: 8
Depth of Network: 3
  
```

----- Task 3 -----

```

0 ---> [[1, 'a'], [2, 'b']]
1 ---> [[5, 'a'], [4, 'b']]
2 ---> [[0, 'a'], [2, 'b']]
3 ---> [[0, 'a'], [0, 'b']]
4 ---> [[0, 'a'], [4, 'b']]
  
```

Figure 6 Correctly removing node 7

This is correctly implemented as follows:

Pre-Minimization an unreachable node of 7 is present, after minimization, this is successfully pruned out.

Combining non-distinguishable states.

Non-distinguishable states are those that within a DFA cannot be distinguished from one another when outputting a string, thus these strings can be merged. The algorithm I chose for this is Hopcroft's algorithm, this algorithm works by gradually partitioning the list of nodes till we finally end up with an unchanged list of merged lists that for any merged list greater than one, non-distinguishable nodes would have been found and thus they would require to be merged. Therefore, along with the main Hopcroft algorithm, a separate function was written that would be able to take any list of nodes that needed to be merged and merge the nodes themselves, as well as the transition nodes these merged nodes might occur in. This was done by first checking if the node itself is comprised of a merged list, and tupling them accordingly, as well the transition nodes outgoing from these nodes.

After for the nodes that haven't been merged, there out going edges might still need to be fixed thus a list of tuples is maintained and for any transition found which make up one of these tuples, they would be changed to the tuple themselves. A several custom DFA's were created and fed to the main algorithm and the merger to check the correctness. An example of which is as follows:

```
A ---> [['B', 0], ['C', 1]]
B ---> [['B', 0], ['D', 1]]
C ---> [['B', 0], ['C', 0]]
D ---> [['B', 0], ['E', 1]]
E ---> [['B', 0], ['C', 1]]

[['E'], ['D'], ['A', 'C'], ['B']]

('A', 'C') ---> [['B', 0], [('A', 'C'), 1]]
E ---> [['B', 0], [('A', 'C'), 1]]
D ---> [['B', 0], ['E', 1]]
B ---> [['B', 0], ['D', 1]]
```

Figure 7 Combining non-distinguishable A and C

The image on the left shows one such example, first the graph itself is being printed before Hopcroft is called, the following is a list of list where every list greater than one is tuple which would require merging and in this case it is shown that the merging itself happens as expected as the nodes A and C are merged together and every out transition which has A or C is changed to the tuple itself.

When checking the actual validity of the Hopcroft as stated before it is important that

the language itself stays the same, thus if the language generated before and after Hopcroft is the same then the instance is valid. In this case when checking both the languages, the language does not change thus the algorithm partitioned as expected.

Limitations with Hopcroft

The time complexity of this algorithm is that of $O(E(V)^{1/2})$ my implementation of this algorithm is not as efficient and instead has a more inefficient time complexity.

Task Four – Recomputing the depth

When recomputing the depth of the DFA, the minimized DFA was traversed with the same methodology explained in task two. The state count was taken by computing the length of the updated dictionary ‘mergedAdj’.

Task Five – Strongly connected components.

For a graph to be strongly connected, each vertex would have to be reached from every other vertex. For strongly connected components, a graph can be ‘broken’ down into sub graphs, by doing so check we can check within a graph the amount of strongly connected subgraphs are present [2]. Finally, if no cycles are present within the graph, then the size of its SCC is the size of the graph itself because single nodes which do not make up a cycle are also considered strongly connected.

A diagram is given below to better illustrate what has been said:

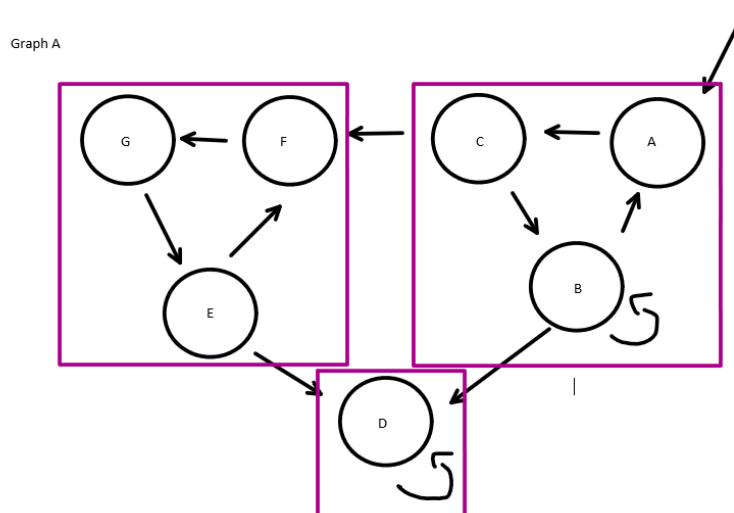


Figure 8 Three strongly connected components

Within graph A, three strongly connected components exist where each vertex is reached from every node in the graph these being $\{\{A, B, C\} \{D\} \{G, F, A\}\}$.

Tarjan algorithm has been implemented to find, the number of SCC present in a graph with a time complexity of $O(|V| + |E|)$ [3].

This is achieved by starting a depth first search from the root node, and for every node visited, assign an index value which translates to the order nodes have been discovered and set its low link to the same counter as index value and finally append the node to a maintained stack. Continue the DFS and if the value is present in low links and is within the stack then set its low link to the minimal between the low link value of the previous node and the index value of the of the current node. If the node isn't present in low links, however, recurs and start the assignment process again. For every call back following these recurses

Compute the minimum of the low link value between the current node and the previous node. Following these two separate minimum computations, check if the current nodes low link

values are equal to their index values. This would mean a new SCC is found so everything in the stack up to the current node would be popped off the stack and be considered as a SCC. Afterwards going over every node present in the graph and check if it has been visited by checking its presence in low link. Once the algorithm finishes return the list of all the strongly connected components.

Like that of Task One, several instances of the algorithm were tested by manually checking the SCC of a smaller DFA.

An example of testing is as follows:

Image on the left shows a DFA with two SCC with the largest being of size six and the smallest of size one, drawing this DFA and checking its SCC shows correctness.

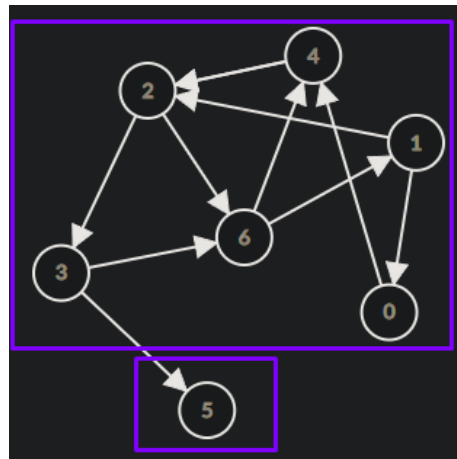
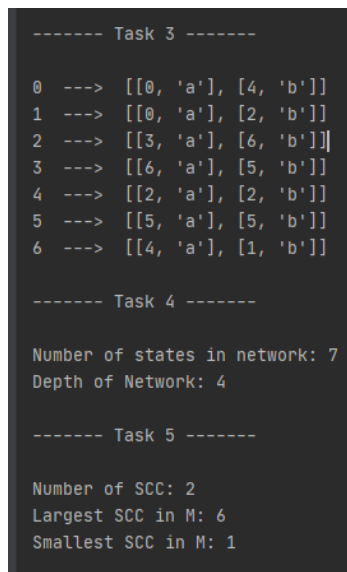


Figure 9 Drawing showcasing Fig 8 SCC's

Similarly, the image on the left shows a DFA with a SCC matching the whole DFA upon drawing the DFA it is shown that there aren't any acyclic nodes present thus showing correctness.

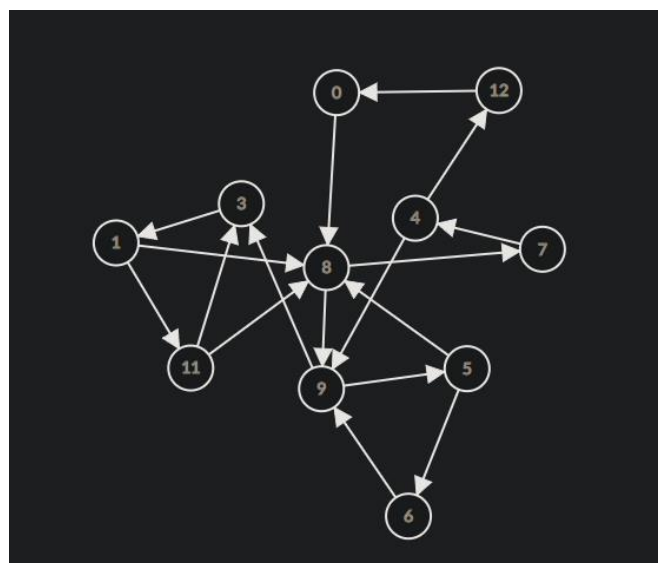
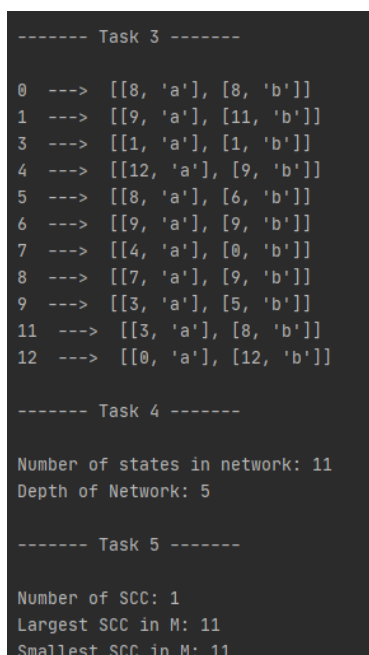


Figure 11 Drawing showcasing fig 10's SCC's

Figure 10 Accurate printing of DFA SCC

Johnson's algorithm

The following shall be based on the paper written by Johnson to find all simple elementary cycles in a directed graph [4].

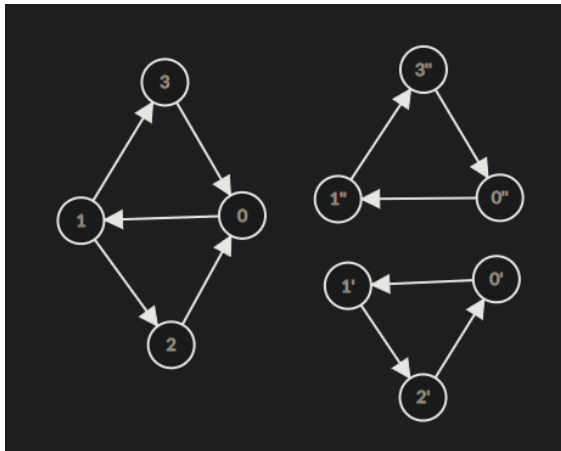


Figure 12 Drawing showcasing an example of a simple cycle breakdown

The first notion which will be explained is what we refer to when we talk about a simple cycle, this is a cycle in a graph with no repeated vertices, if a graph can't be broken down further then it is already a simple cycle [5].

The image on the left shows how a non-simple cycle can be broken down further into two simple cycles starting from 0' and 0'', these cannot be broken down further thus they are both considered to be simple cycles.

We can now reintroduce another notion which has already been talked about during Task 5, *strongly connected components*, to re iterate, for

a graph to be strongly connected, each vertex would have to be reached from every other vertex and if a node is on its own, we can consider that on its own to also be a SCC.

Johnson's algorithms builds on this to achieve simple elementary components, a cycle cannot be present spanning over multiple strongly connected components so the algorithm this to its advantage. To obtain these SCC an algorithm such as Tarjan's algorithm is used as a subroutine which computes the SCC of the graph for every time the original graph is altered by the algorithm.

With the first iteration of the Tarjan subroutine, the SCC have been obtained. By going through each node, check if any of these have a SCC of vertex size greater than one. SCC of size one cannot be simple connected components thus these are skipped by the algorithm.

A simple example shall be constructed which be used to further explain the rest of the algorithm.

Let the graph on the right be the one John's algorithm is performed on, first the subroutine is called on the graph to divide it into its SCC and as one can see this graph gets divided into three SCC. The algorithm then starts to check the nodes, for this example they shall be taken from smallest to largest, thus state 0 is first considered and the rest of the algorithm is performed on the SCC subgraph 0 is found in due to it containing more than 1 vertex.

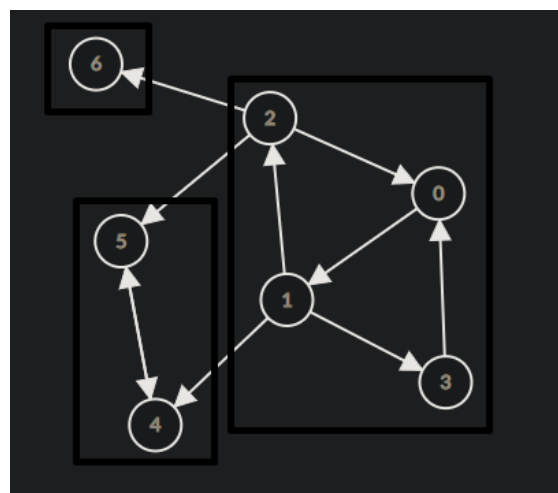


Figure 13 Drawing showcasing example's SCC

Once the subgraph is found, Johnson's algorithm maintains three data structures that are used throughout these steps, these being a *stack*, a *blocked set* and finally a *blocked map*. These three shall be discussed throughout the dry run of the algorithm.

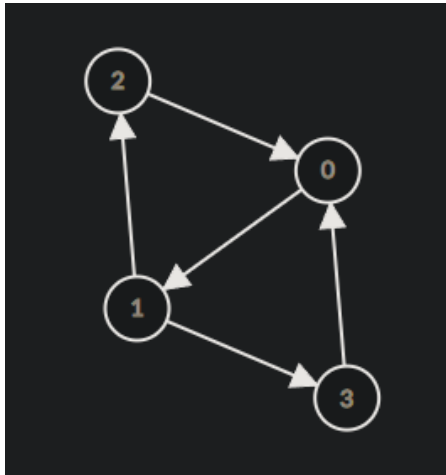


Figure 14 Drawing showcasing Johnson's algorithm

Since we are starting from state 0, for this step we are looking for all simple cycles that start and end with this vertex. This state is pushed onto the stack, as well as the blocked set. States are put into a blocked set similarly to when we traverse a graph with a traversal algorithm and maintain a visited list, however the blocked set differs from the visited list due to the concept of unblocking. The *blocked map* is used to maintain what we should unblock following any other unblocks, for example if a value 'A' within a blocked set is unblocked, then within a blocked map, if A leads to another vertex B, then B is also unblock and removed from the blocked set as well as any other blocked set from B leading to another blocked vertex. This is done recursively.

Continuing with the dry run we are currently at state 0, so 0 is pushed both on the stack and blocked set. The way traversal is done is by using a DFS thus state 1 is now being explored and it is also pushed on the stack and blocked set. Again, since it's a DFS, 2 is now explored and the same happens until we arrive to 2 attempting to travers to state 0. At this point the blocked set and stack both have values {0, 1, 2} within the stack and blocked set but since state 2 leads to the original vertex then a cycle is found and this is the cycle within the stack {0,1,2,0}. This cycle is a simple cycle so it is outputted to the user. Since two is part of the path {0,1,2} which lead to a cycle, it gets unblocked and popped off the stack (we now have {0,1}), this happens since other potential cycles might occur from two. This vertex however does not have any other outgoing vertex's so the DFS recurses back to state 1.

With this we are at state 1 with {0, 1} on the stack and blocked set. This state has another possible traversal to three, so it goes to 3 and pushes it on the stack/blocked set. After three attempts to travers to 0, but again it is the original vertex so the simple cycle {0,1,3,0} is outputted. Following this we recurse to 3 and remove it from the stack and blocked set since its part of a simple path, it doesn't have any other outgoing vertex so we recurse to 1 and the same cycle occurs with 1 so then we recurse back to the original state 0 with the blocked set and stack set to {0}.

This state has no other outgoing transitions thus we are ready from this inner loop. Since node 0 has been traversed from the outer loop of the algorithm, it is now removed from the main graph and the outcome is as on the right. The sub routine for Tarjan's algorithm is called again and since vertex 0 has been eliminated we are only left with one SCC with vertex size greater than one which is 4 leads to 5 which leads back to 4 (This is illustrated on the right). So, the outer loop would go through vertices one, two and three skip them since they make up a SCC of size one and then focus on the SCC which contains vertex 4. The same discussed inner procedure now occurs on this subgraph which outputs the simple elementary cycle {4,5,4} and 4 is removed from main graph. At this point after calling the Tarjan's subroutine, no other SCC with size greater than one exists thus the algorithm goes to the end of the nodes and concludes running.

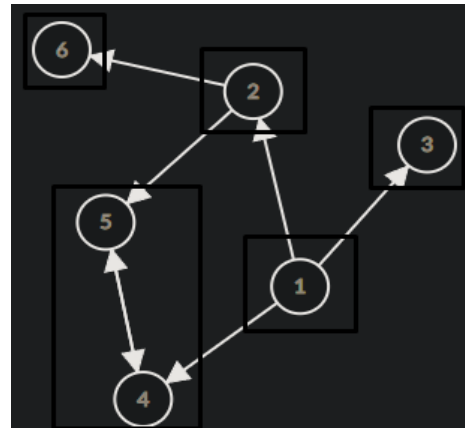



Figure 15 Drawing showcasing example's SCC

To recap from the demonstrated example, Tarjan's algorithm has successfully printed all the elementary cycles {0,1,2,0}, {0,1,3,0}, {4,5,4} found within the example graph. The time complexity of this algorithm is $O(|V| + |E|)(C+1)$ where C is the total amount of cycles in the graph and $O(|V|+|E|)$ occurs from the Tarjan subroutine.

References

- [1] G. Pace, "Removing Unreachable States From Finite State Automata", *Cs.um.edu.mt*, 2022. [Online]. Available: <http://www.cs.um.edu.mt/gordon.pace/Research/Software/Relic/Transformations/FSA/remove-unreachable.html>. [Accessed: 27- May- 2022]
- [2] "Strongly connected component - Wikipedia", *En.wikipedia.org*, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Strongly_connected_component. [Accessed: 27- May- 2022]
- [3] "Tarjan's strongly connected components algorithm - Wikipedia", *En.wikipedia.org*, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm. [Accessed: 27- May- 2022]
- [4] "Find any simple cycle in an undirected unweighted Graph - GeeksforGeeks", *GeeksforGeeks*, 2022. [Online]. Available: <https://www.geeksforgeeks.org/find-any-simple-cycle-in-an-undirected-unweighted-graph/>. [Accessed: 27- May- 2022]
- [5] J. Donald, *Cs.tufts.edu*, 1975. [Online]. Available: <https://www.cs.tufts.edu/comp/150GA/homeworks/hw1/Johnson%2075.PDF>. [Accessed: 27- May- 2022]

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
Printed number and size of SCCs	Yes
Included a good evaluation in your report	Yes
Student signature	

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).

Kian Parnis

Student Name



Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

ICS2210

Course Code

ICS2210 - Assignment Submission - Kian Parnis

Title of work submitted

5/27/2020

Date