L-Università ta' Malta

ICS2210 - Assignment
Data Structures and Algorithms 2

Daniel Magro

Bachelor of Science in
Information Technology (Honours)
(Artificial Intelligence)

May 2018

Contents

1 Representing the DFSA 3

2 Time Complexities of Algorithms 6

2.1 Minimisation Algorithm - Hopcroft's Algorithm 6

2.2 Tarjan's Algorithm for Strongly Connected Components . . . 10

2.3 Johnson's Algorithm for Finding Simple Cycles 12

3 Evaluation 16

3.1 Task 1 16

3.2 Task 2 17

3.3 Task 3 18

3.4 Task 4 19

3.5 Task 5 20

3.6 Task 6 21

4 How to Run the Program 22

5 References 23

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 Tarjan's algorithm	Yes
Implemented Johnson's algorithm	Yes
Evaluation	Yes

1

Representing the DFSA

DFSAs were represented as follows:
A class *State* was created. This class defined that every instance of it (i.e. every State) will be made up of

- An integer *stateID*, which stores the 'ID' number of that state. IDs range from 0 to n-1; where n is the number of states in the DFSA.
- A boolean *finalState*, which stores whether a state is final/accepting (represented by 1/true) or if it is non-final/rejecting (represented by 0/false).

A class *DFSA* was created, which defines the structure of the DFSA. This class defines that a DFSA is made up of an

- integer *startStateID*, which stores the stateID of the start state of the DFSA. The state's ID is stored, and not all the information relating to the state, because throughout this project, states are only stored in memory once, in the *states* array list (as explained in the next point).
- An ArrayList (i.e. a 'resizeable' array) *states*, which stores instances of the class *State*. This arraylist stores the only instance of every state, this, paired with the fact that states are always referred to by their stateID (which is the same as their index inside the arraylist),
  - allows states to only exist in one place in memory, and thus eliminate duplicate and wasted memory.
  - allows retrieval of states in constant time, since they can be accessed by their index in the arraylist.

		State IDs			
		0	1	2	3
inputs	a	1	3	2	1
	b	3	0	1	2

Table 1.1: Graphical Representation of a Transition Table

- A 2D array of integers *adjacencyMatrix* (Please note that despite the name, the edges of the DFSA are stored in a Transition Table, and not an adjacency Matrix, this is explained further below). This 2D array stores the Transition Table for the DFSA. This is a 2D array of stateIDs against inputs(transitions), where every cell represents a stateID. The columns represent all the state IDs inside the DFSA, whereas the rows represent all the transitions inside the DFSA. Table 1.1 shows an example of how the edges are stored.
  To compute what state the DFSA will go to next, algorithms simply need to look up the contents of cell [current StateID][transition].
  Using the example matrix in Table 1.1, if the current state is 2, and the next input is b, then the next state will be state 1.

As mentioned above, the Transition Table is named *adjacencyMatrix*, however the data structure used is in fact **not** a traditional adjacency matrix. Adjacency Matrices very often result in a lot of wasted memory since graphs are generally sparse, and this would definitely be the case for this project, as every state will always only have 2 outgoing transitions, an *a* transition and a *b* transition. Thus to mitigate this problem, but at the same time retaining the advantages that an adjacency matrix offers that are relevant to this project, a Transition Table as explained above is used.
The Transition Table allows for

- finding the outgoing transitions of a state in O(number of different transitions) time (by looking for all cells under a state ID's column). In this case O(n) is negligible as there are always only 2 transitions and thus it can be virtually done in O(1) time.
- finding all the incoming 'x' transitions of a state in O(number of states) time. Finding all incoming transitions to a state for all inputs can be done in O(number of states $\times$ number of different transitions) time. This basically involves looking at all current state x transition pairs. While this is can be in fact considered a long operation, this is not performed by any of the algorithms in this project and thus was not considered a problem.

- finding the next state of the DFSA given its current state ID and transition in O(1) time. Since the data structure used is a 2D array, look up can be done in O(1) time as the current state id is already known and inputs are enumerated into their respective row (i.e. as in row 0 and bs in row 1). Thus, as mentioned earlier, computing what state ID should be next, algorithms simply need to query transitionTable[currentStateID][transitionRow].

2

Time Complexities of Algorithms

2.1 Minimisation Algorithm - Hopcroft's Algorithm

The first step of DFSA Minimisation algorithms is removing all unreachable states.
The chosen algorithm, Hopcroft's Algorithm, works by then dividing the reachable accepting states and reachable rejecting states into two different sets, as these two sets contain states which are by definition different, and a rejecting state cannot possibly accomplish the same task as a rejecting state, and vice-versa. The algorithm continues by further sub-dividing the sets into smaller sets of states, until they can no longer be sub-divided. At this point, every set contains a group of non-distinguishable (identical) states, where each set of non-distinguishable states can be replaced by just one state.
The time complexity was computed as follows:

- Dividing all the states into two sets of reachable accepting states and reachable rejecting states was accomplished as follows. The algorithm iterates through all the states. If a state is reachable, it then checks whether it is final, if so, it is added to the set of accepting states, if not it is added to the set of rejecting states.
  Both checking whether a set is reachable and whether it is final or not can be done in O(1) time. Iterating through every state is done in **O(n)** time, where n is the number of states in the DFSA.

- The next step involves adding the set of accepting states and the set of rejecting states to the Set of Sets of States (i.e. integers since states are stored by their IDs), P. Adding elements to a set is done in $\mathbf{O(1)}$ time.
- Next, the set of rejecting states is added to the Set of Sets of States(integers) W. This again is done in $\mathbf{O(1)}$ time.
- The algorithm then iterates over the next instructions for as long as the set of sets, W is not empty. For calculating the time complexity, the worst case will be considered where the DFSA cannot be minimised and this loop will run for every state, thus *(number of states - unreachable states)* time. Thus $\mathbf{O(n)}$ $\times$ **the complexity of the contents of this loop**
  - Set the set A to be equal to any other set inside W. This is a deep copy and thus takes $\mathbf{O(n)}$ time.
  - Remove the set which is now inside A from W. This is done in constant $\mathbf{O(1)}$ time.
  - The algorithm now iterates over all the different transitions of the DFSA. In the case of this project this is 2 $\times$ the complexity of the contents of the loop. In the general case, it will be considered **O(number of different transitions,** $\boldsymbol{c}$**)** $\times$ **the complexity of the contents of this loop**.
    - Declare set of states X and fill it with all the states that given the current transition will result in a state which is contained in set A. In this step all reachable states are considered, thus this step takes $\mathbf{O(n)}$ time.
    - Create a copy of set P, named *Pcopy*. This is a deep copy and thus takes at most $\mathbf{O(n)}$ time (since there can only ever be as many sets in P as there are reachable states).
    - The algorithm now iterates over all the sets in Pcopy, which as mentioned above can only ever be as large as the number of reachable states. Thus its time complexity is said to be $\mathbf{O(n)}$ $\times$ **the complexity of the contents of this loop**.
      - A set Y is declared, and set to be a copy of the contents of the current subset being considered from Pcopy. Assuming the worst case where setInP is the initial set of all

reachable accepting or rejecting states, the time complexity is considered to be $\mathbf{O(n)}$.

· A set *intersection* is declared and set to be a deep copy of set X. This takes, at worst, $\mathbf{O(n)}$ time.

· *retainAll* is performed on *intersection* and *Y*, which is the equivalent of computing their intersection (X∩Y). This takes $\mathbf{O(n)}$ time.

· Similarly, a set *difference* is declared, and set to be a deep copy of set Y. This takes, at worst, $\mathbf{O(n)}$ time.

· *removeAll* is performed on *difference*, removing all states present in *X*, which is the equivalent of computing their difference (Y\X). This takes $\mathbf{O(n)}$ time.

· Next, if both X∩Y and Y\X are not empty, then the following occurs. For computing time complexity, the worst case will be assumed, where this will always be the case and the next steps are always executed.

· Set *Y* is removed from the set of sets *P*. The sets *intersection* and *difference* are added to *P*. Each of these operations is done in $\mathbf{O(1)}$ time.

· If set *W* contains set *Y*, then *Y* is removed from W and *intersection* and *difference* are added to *W*. Each of these operations is done in $\mathbf{O(1)}$ time.

· If set *W* does not contain set *Y*, then if the set *intersection* contains less elements than the set *difference*, *intersection* is added to *W*, else, *difference* is added to *W*.

- Finally the algorithm rebuilds the now minimised DFSA. The Set P is converted into an ArrayList, so that it now has an order and index for each sub-set. This is done in $\mathbf{O(n)}$ time.

- A loop iterates over every sub set in the ArrayList P, so that each sub-set is converted into a state and added to the new DFSA Data Structure. Thus the time complexity of this loop is $\mathbf{O(n)}$ × **the complexity of the contents of this loop**.

  – The statements in the loop can all be completed in constant time, however there are 2 for-loops which iterate over every state in the DFSA which takes $\mathbf{O(n)}$ time.

Combining all the time complexities, the equation will look something like: (Please note that large blocks of consecutive operations in the same scope with identical time complexities are combined into one since they do not affect the final result).

- O(n)
- O(1)
- O(1)
- O(n) *
  - O(n)
  - O(1)
  - O(c) *
    - O(n)
    - O(n)
    - O(n) *
      - O(n)
      - ⋮
      - O(n)
      - O(1)
      - O(1)
- O(n)
- O(n) *
  - O(n)

Combining all of the above time complexities, i.e. adding when complexities are in the same level, and multiplying when a complexity is being looped, the total time complexity of Hopcroft's Algorithm is $\underline{\mathbf{O}(c.n^3)}$; where n is the number of states inside the DFSA and c is the number of unique transitions (size of the alphabet).

2.2 Tarjan's Algorithm for Strongly Connected Components

Tarjan's Algorithm finds all the Strongly Connected Components within a directed graph. The DFSA's states are divided into different components, where each state can only be in one component, and can reach any other state in the same component via the directed transitions. The main concept behind Tarjan's algorithm is the Depth First search, however and *index* and a *lowlink* is kept for each state.

Time complexity of the **getSCCs** method.

- The variables utilised by this algorithm are initialised. That means, *SCCs* is set to an empty ArrayList - O(1), *lastIndex* is set to 0 - O(1), the arrays *index* and *lowlink* are initialised and all elements are filled with '-1' - O(n), *stateIDStack* is initialised as an empty stack - O(1) and *stateIDStack* is initialised as an empty array - O(1). This setup takes **O(n)** time; where n is the number of states in the DFSA.
- A loop iterates over every state inside the DFSA. For every state it checks whether the index of that state is -1 (i.e. not yet calculated), if so, the function *strongConnect* is called on that state. Thus, the time complexity of this loop is **O(n) × time complexity of *strongConnect***.

Time complexity of the **strongConnect** method.

- The *index* and *lowlink* for the current state ID are set to *lastIndex*. *lastIndex* is incremented. The current stateID is pushed to the *stateIDStack*, and *onStack* of the current stateID is set to true. All of these operations can be done in **O(1)** time.
- A for-loop iterates over every successor of the current state. For this project, this will be at most 2 iterations for each state since this is a total DFSA with 2 different types of transitions (a and b). Thus, its time complexity is said to be **O(size of alphabet × the complexity of the contents of this loop)**.
  - The state ID of the successor of the current state given the current transition is retrieved from the Transition Table. This can always be done in **O(1)** time.
  - If the index of the successor state has not yet been computed

        * *strongConnect* is called on it. *strongConnect* is only run once per state. Since the for-loop in *getSCCs* is already said to have a O(n) time complexity, and if a strongConnect call occurs from within strongConnect, that state's index will no longer be *-1* (i.e. uninitialised), and thus strongConnect will not be called again from getSCCs. Thus in total, strongConnect will always run exactly **n** times, and this call does not add to it.
        * Once strongConnect returns, the lowlink of the current state is set to either the lowlink of the successor state, or the lowlink of the current state, whichever is lower. This can be done in **O(1)** time.

    If the index of the successor state has been computed, and the successor is on the stack, then the lowlink of the current state is set to either the lowlink of the current state, or the index of the successor state, whichever is lower. This can be done in **O(1)** time.

- If the current node is a root node (i.e. its lowlink is the same as its index), then a SCC has been found.
    - A new ArrayList of StateIDs, *SCC*, is declared, which will store the Strongly Connected Component. This is done in **O(1)** time.
    - A do-while loop pops states from the *stateIDStack*, marks that state as no longer on the stack, and adds it to the ArrayList *SCC* until the current state has been reached. Each iteration is done in constant time. Since it has been proven that a State can only belong to one SCC, this process will only be run once for each state, thus, since the for-loop in *getSCCs* already accounts for the one iteration per state by O(n), this step is considered as **O(1)**.

The time complexity of the strongConnect method is thus O(1) + O(size of alphabet $\times$ complexity of loop) + O(1). Let c represent the size of the alphabet, i.e. the number of unique transition types (in this case a and b, so c=2). The complexity of the loop was concluded to be O(1). Thus the time complexity of strongConnect is O(1) + O(c)$\times$O(1) + O(1), which equals $\underline{\mathbf{O}(c)}$.
The time complexity of getSCCs is thus O(n) + O(n)$\times$complexity of strongConnect. Substituting the time complexity of strongConnect from what was discussed above, the time complexity of getSCCs is O(n)+O(n)$\times$O(c), which equals $\underline{\mathbf{O}(n \times c)}$
Thus, in conclusion, the time complexity of Tarjan's Algorithm can be said

to be **O**($n \times c$). The time complexity can be rewritten to be in the same form as that quoted in literature, that is, $\underline{\mathbf{O}(n + t)}$; where n is the number of states in the DFSA, and t is the total number of transitions, or edges in the graph (Since this is a total DFSA, and each state has 2 outgoing transitions, $t = 2 \times n$).

2.3 Johnson's Algorithm for Finding Simple Cycles

This implementation of Johnon's Algorithm for Finding Simple Cycles assumes that the SCCs have already been found. In this case they have been found by Tarjan's algorithm and are stored in the ArrayList *SCCs*. SCCs are relevant to Johnson's Algorithm because a simple cycle can only exist over nodes which are members of the same SCC, thus this can be used as an optimisation to limit the search space.
In short, Johnson's algorithm works by first starting from a node in a SCC, and from that node visiting a neighbour, and then that neighbour's neighbour ..., until the node that the algorithm is on, is the state of the SCC that it left off from.

Time complexity of the **findSimpleCycles** method.

- The variables utilised by this algorithm are initialised. This is done in **O(1)** time.
- A for loop iterates over every SCC that was found in the DFSA. This loop takes **O(s $\times$ the complexity of the contents of this loop)** time; where s is the number of Strongly Connected Compontents (SCCs) in the DFSA.
  - Another nested for-loop iterates over every states inside the current SCC. This loop takes **O(nInS $\times$ the complexity of the contents of this loop)** time; where *nInS* is the number of states inside the current Strongly Connected Compontents (SCC).
    - In each iteration, the *startNode* is set to the current state of the SCC - O(1), the *blockedSet* and *blockedMap* are cleared - O(size of Set and Map), in the worst case O(n) and *findSimpleCyclesInSCC* is called. Thus the time complexity of

this block is O(1) + O(n) + time complexity of *findSimpleCyclesInSCC*.

Time complexity of the **findSimpleCyclesInSCC** method.

- First a boolean value *foundCycle* is declared and initialised to false, the currentNode (or rather, its stateID), is pushed to the *cycleStack*, and added to the *blockedSet*. These statements are all executed in **O(1)** time.
- Next, a for-loop iterates over every neighbour of the currentState, thus the time complexity of this loop can be said to be **O(c $\times$ the complexity of the contents of the loop)**; where c is the number of unique transitions (size of the alphabet).
  - The next state reached from the current state given the current transition is retrieved from the transition table and stored in *nextState*. This is done in **O(1)**.
  - If the nextState is the same as the startState that was passed to *findSimpleCyclesInSCC*, then a cycles has been found.
    * A new ArrayList *cycle* is created which stores the order of states that create a cycle. The startState is pushed again to the end of the *cycleStack* so that the cycle starts and ends with the start state of that cycle. Then, the states inside of *cycleStack* are all added to *cycle*. The state at the top of the *cycleStack* is popped. And finally, the *cycle* is added to *simpleCycles*, and *foundCycle* is marked true. The time complexity of *addAll* in java is most times completed in constant time, however to consider the worst case, the time complexity of these commands will be considered **O(n)**.

    Else

    * Else, if *nextNode* is not the same as the *startNode*, i.e. a cycle has not yet been found, then if the *nextNode* is not in the *blockedSet*, *foundCycle* is set to the boolean OR of the current value of *foundCycle*, and the boolean return of *findSimpleCyclesInSCC*, and in this call updating *currentNode* to the value of *nextNode*. The time complexity of this recursive call is *nInS*-1, the minus one is important to mention since with each call, the algorithm advances one state which will not be checked again since it is now in the blockedSet. Thus

the time complexity of *findSimpleCyclesInSCC* is O(n), and approaches O(1) with every call.

- If a cycles has been found following the for-loop described above,
  - Then the *unblock* is called on the *currentNode*. The *unblock* method sometimes calls itself recursively if there is an entry for *currentNode* in the *blockedMap*. The total number of unblocks that take place cannot be any larger than *nInS*, thus this function will be said to take, in the worst case, **O(nInS)** time.

  Else (i.e. if a cycle was not found),
  - An entry is added to the blockedMap to show that from the current state, following the transitions to the adjacent states will not result in any cycles, so the blockedMap makes this clear. So an entry is added to the *blockedMap* for the current state to all adjacent states. Since at most, there can only be as many adjacent states to be blocked as there are outgoing transitions from the current state, this step takes **O(c)** time.
- Finally, the state at the top of the *cycleStack* is popped, and *foundCycle* is returned. This takes **O(1)** time.

Summarising all of the complexities, and removing constant time complexities to reduce clutter, the time complexity of the algorithm looks as follows:

- O(s) ×
  - O(nInS) ×
    - O(n) +
    - time complexity of *findSimpleCyclesInSCC*

For findSimpleCyclesInSCC, the time complexity is:

- O(c) ×
  - O(n)
  - OR (i.e. the worst case is considered)
  - A recursive call to this method
- O(nInS)
- OR (i.e. the worst case is considered)

- O(c)

To compute the complexity of the recursive function *findSimpleCyclesInSCC*, represent the algorithm as
$f(n) = c{*}f(n{-}1) = c{*}(c{*}f(n{-}2)) = c{*}(c{*}(c{*}f(n{-}3))) = c{\times}c{\times}c{\times}\ldots{\times}O(1)$
Thus the complexity of the function can be said to be $\underline{c^n \times n}$. The O(nInS) or O(c) are 'not considered' since they are added to the time complexity, and their significance is *negligible* compared to the complexity of the recursive calls and loops.
Thus, the complexity of Johnson's algorithm can then be said to be
$O(s \times nInS \times n \times c^n)$. In the worst case, $nInS$ can only be as large as n, thus they will be considered the same. Therfore, the time complexity of Johnson's Algorithm is
$\underline{O(s \times n^2 \times c^n)}$; where s is the number of SCCs in the DFSA, n is the total number of states in the DFSA, and c is the size of the alphabet.

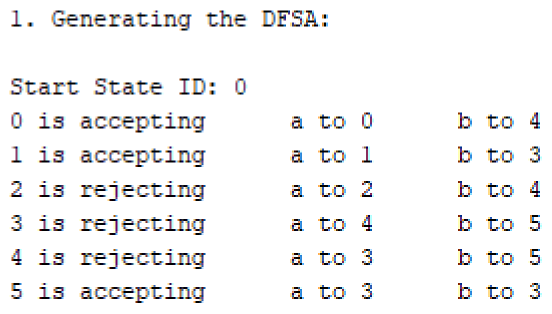
3

Evaluation

In order to show that the implemented algorithms do, in fact, work properly, the entire program was run with a random DFSA of only 6 states. Functions such as *printDFSA*, *printSCCs* and *printCycles* were implemented, which output the result of their respective algorithms in a human readable format, so that the results can be compared and evaluated.

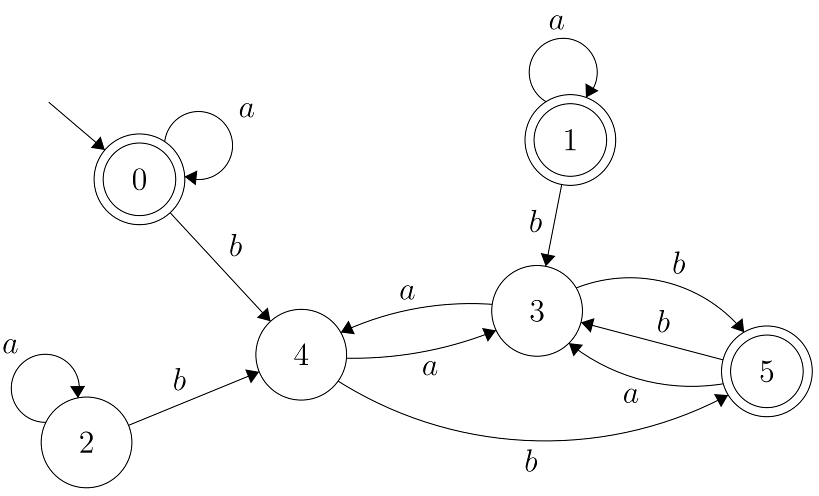
3.1 Task 1

The first task involved the generation of a random DFSA. For evaluation, the amount of states was set to 6, so that the DFSA would be much more manageable.
The DFSA produced by the program is the following:



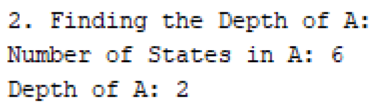
```
1. Generating the DFSA:

Start State ID: 0
0 is accepting      a to 0      b to 4
1 is accepting      a to 1      b to 3
2 is rejecting      a to 2      b to 4
3 is rejecting      a to 4      b to 5
4 is rejecting      a to 3      b to 5
5 is accepting      a to 3      b to 3
```

If the DFSA given by the program is converted to a graphical DFSA, it looks as follows:



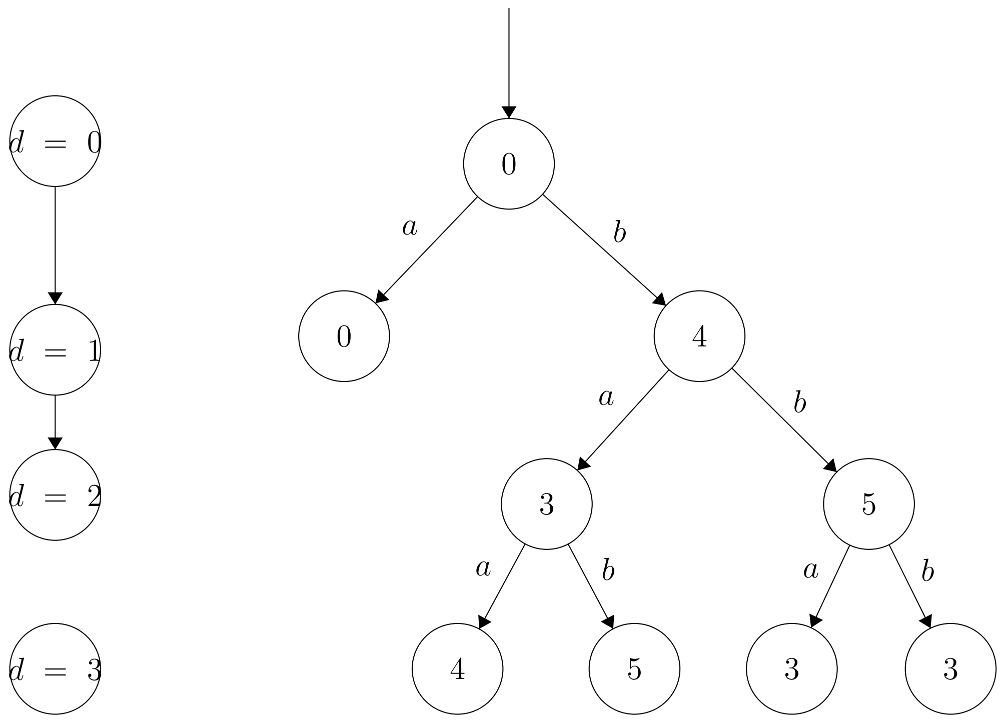
3.2 Task 2

The Second algorithm calculates the depth of the DFSA using a Breadth First Traversal. The program outputs the following with regards to the depth:



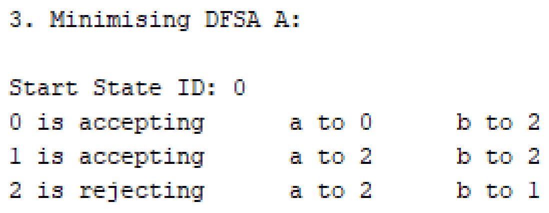
```
2. Finding the Depth of A:
Number of States in A: 6
Depth of A: 2
```

When computed by hand, the depth is worked out as follows:
When the depth reaches 2, after the next 'level' is computed, it is evident that all states have already been visited, thus all reachable states can be reached with a string of length 2. This shows that the algorithm's output was correct.



3.3 Task 3

In task 3, a Minimisation Algorithm was implemented, which removes all unreachable states from the DFSA, as well as combines any non-distinguishable states.
The program outputted a DFSA as follows:

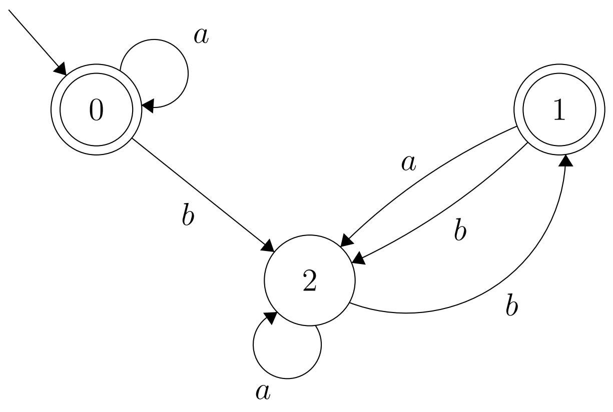


```
3. Minimising DFSA A:

Start State ID: 0
0 is accepting      a to 0      b to 2
1 is accepting      a to 2      b to 2
2 is rejecting      a to 2      b to 1
```

When looking at DFSA A, it is immediately evident that States 1 and 2 are unreachable, and thus serve no purpose for the DFSA. This can also be seen from the fact that they never appear in the BFS done in Task 2 when computing the depth.

The algorithm further minimises the DFSA by grouping together non-distinguishable states.

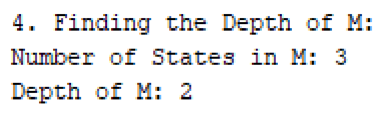
The minimised DFSA M, produced by Hopcroft's Algorithm, shown graphically looks as follows:



Using FSM2Regex[7], DFSA A was converted into a Regular Expression
`(b+(a+$)a*b)(a+b(a+b))*b+a+$+(a+$)a*(a+$)`
The resulting, minimised DFSA M was also converted into a regex, which resulted in
`(b+(a+$)a*b)a*(b+b((a+b)a*b)*($+(a+b)a*b))+a+$+(a+$)a*(a+$)`
While the regexes do not look identical, they in fact are, and produce the same language.
To show this in a more tangible manner, a number of strings can be run through the DFSAs to show that they both accept and reject the same strings. For example "aaabaabab" is in fact accepted by both DFSAs.

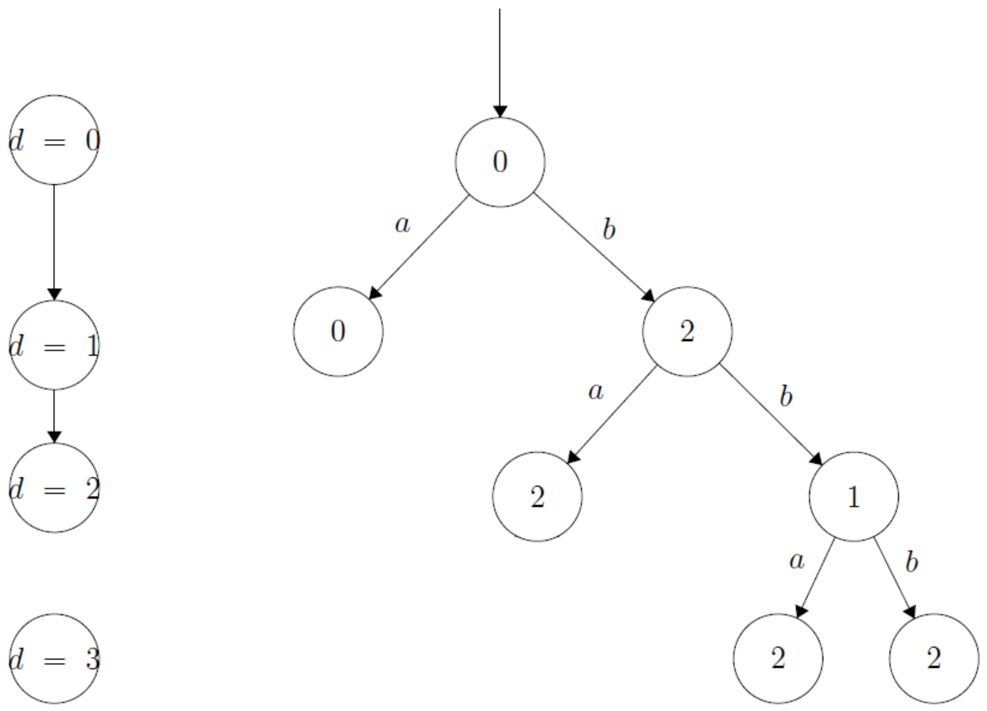
3.4 Task 4

In task 4, the depth of the Minimised DFSA M is computed. This should of course be the same as for DFSA A, since they accept the same language.



```
4. Finding the Depth of M:
Number of States in M: 3
Depth of M: 2
```

To verify, the depth is also computed by hand, similarly to what was done in Task 2. The answers match, which suggests that the algorithm was successful and worked properly.

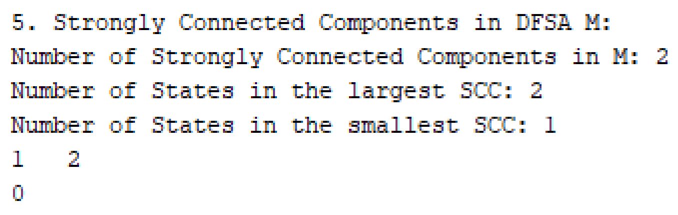


3.5 Task 5

Tarjan's Algorithm, which was implemented for Task 5, finds all the Strongly Connected Components in the minimised DFSA. In an SCC, from any one of the states, any other state in that same SCC can be reached by following the direct edges. Every state can only be a member of one SCC.

The output of the program is shown on the next page.

Clearly, just by looking at the graphical representation of DFSA M, the output of the program is correct.
From state 1, a transition exists to state 2, and from state 2, a transition exists to state 1. State 0 is clearly not part of this component as neither state 2 nor 1 have a transition to it.
State 0 is in a SCC made up only of itself.

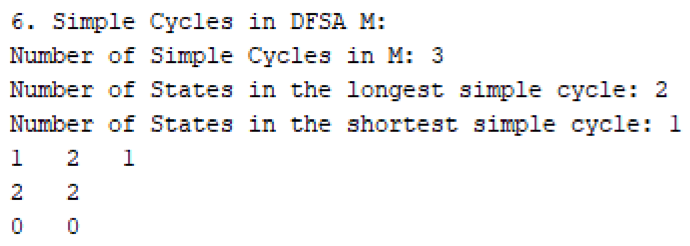


```
5. Strongly Connected Components in DFSA M:
Number of Strongly Connected Components in M: 2
Number of States in the largest SCC: 2
Number of States in the smallest SCC: 1
1   2
0
```

3.6 Task 6

Johnson's Algorithm, implemented for Task 6, finds all the Simple Cycles in the Minimised DFSA, M. Since a simple cycle can only exist within a SCC, the algorithm only looks for nodes within the SCCs found from Task 5.

The output of the program was as follows:



```
6. Simple Cycles in DFSA M:
Number of Simple Cycles in M: 3
Number of States in the longest simple cycle: 2
Number of States in the shortest simple cycle: 1
1   2   1
2   2
0   0
```

Similarly to the case of Task 5, just by looking at the graphical representation of DFSA M, it can easily be determined that the output of the program is correct.

From state 1, a transition exists to state 2, and from state 2, a transition exists back to state 1.
From state 2, a transition exists to state 2.
From state 0, a transition exists to state 0.

4

How to Run the Program

To run the program, simply go to the *Executable* folder, and double click on *run.bat*, which will run the *DFSA* jar file. No input is required from the user since DFSA A is generated randomly.

5

References

Hopcroft's Minimisation Algorithm was obtained from:
[1] - "Hopcroft's algorithm", Wikipedia (April 2018),
`https://en.wikipedia.org/wiki/DFA_minimization#Hopcroft's_algorithm`
[2] - P. Linz, "An Introduction to Formal Languages and Automata", Chapter 2.1, Fifth Edition

Tarjan's Algorithm for finding Strongly Connected Components was obtained from:
[3] - 'Michael Mroczka', "Strongly Connected Components Tutorial", YouTube,
`https://youtu.be/ju9Yk7OOEb8`
[4] - "Tarjan's strongly connected components algorithm", Wikipedia (April 2018),
`https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`

Johnson's Algorithm for finding Simple Cycles was obtained from:
[5] - 'Tushar Roy', "Johnson's Algorithm - All simple cycles in directed graph", YouTube, `https://youtu.be/johyrWospv0`
[6] - D. Johnson, "Finding All the Elementary Circuits of a Directed Graph", SIAM Journal on Computing, 1975

[7] - FSM2Regex, Online tool which converts FSMs to Regular Expressions,
`http://ivanzuzak.info/noam/webapps/fsm2regex/`