L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Artificial Intelligence

ICS2210 Data Structures and Algorithms 2

Cristina Cutajar* (230802L)

*B.Sc. (Hons) Artificial Intelligence

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

Code: **ICS2210**

Lecturers: **Dr Kristian Guillaumier and Prof. John Abela**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), 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 committing a breach of the regulations include 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).

Cristina Cutajar	
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

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

14/05/2022

Date

Introduction

The aim of this project is to implement a deterministic finite state automata and perform actions such as finding the depth, minimisation and finding the strongly connected components of the automata correctly.

In order to complete this project, at first, I started implementing the code in C. However due to time constraints I decided to switch to Python in order to implement the minimisation algorithm and Tarjan's algorithm in a more time efficient manner. To minimise the DFSA, I chose to implement the Hopcroft minimisation algorithm.

Implementation

DFSA construction

In order to implement the DFSA, I decided to create two classes, one called DFSA and one called State. Since I had initially started to implement the project in C, I made use of [1], [2], [3] and [4] in order to implement the DFSA in C and then I reimplemented the code in Python, keeping the same structure.

The State class contains the initialization function, the set_transition() function, the set_partition_states() function, the flip_coin() function and the print_details() function.

The initialization function sets the state id, sets both transitions to -1 and either sets the accepting variable according to what was given or calls the flip_coin() function to randomly decide whether the state will be accepting or not, if no value is given. The accepting value will only be passed in when performing the minimization algorithm. The initialization function also creates an empty array which will be used during the minimization algorithm to keep track of the states of a particular partition. The set_transitions() function takes in two transitions (state ids) and sets them for the particular state in order to keep track of the state transitions. The set_partition_states() function will be used for the minimization algorithm in order to keep track of the states ids that will afterwards be in a certain partition. The print_details() function prints the state id, both state transitions, whether the state is accepting or not and if eligible, the partition states.

In the DFSA class, I implemented the initialisation function and the print_details() function in order to construct the DFSA. Later on, more functions were added in the DFSA in order to implement the required code which will be covered in the following parts of the report. The initialisation function sets the n number of states according to the given value and sets the alphabet to 'a' and 'b', as required. It also creates an empty array which will later be used to store the visited states during breadth first search. The initialisation function sets the 'scc_state' variable to -1 as well as creates an empty array, both to be used to find the strongly connected components. If 'auto_fill' is passed in as 'True', the initialisation function will randomly choose a state as the initial state, create empty arrays for the final states ids, not_final states ids as well as to store the actual states. It will then loop according to the given

number of states, and it will call the State class in order to create the states one by one with the state ids. Each state is appended to the states array and its id is appended to the final or not_final array depending on whether the state is accepting or not. Then, the final and not_final arrays are sorted to make sure that they are in order. Afterwards, for each state, two random states are chosen and set as the state transitions. If 'auto_fill' is passed in as 'False', then the initial variable, final array, not_final array and states array are all set to the given variables. The print_details() function will print all the details of the DFSA including the number of states, the alphabet, the initial state id, the final states array, the not_final states array as well as calls the print_details() function from the State class for each state in the state array in order to display all the information for each state.

Breadth-first search to compute the depth of DFSA

In order to compute the depth of the deterministic finite state automata, I implemented the breadth-first search function and calculate depth function in the DFSA class. Reference was made to [5], [6], [7] and [8] to correctly calculate the depth of the DFSA.

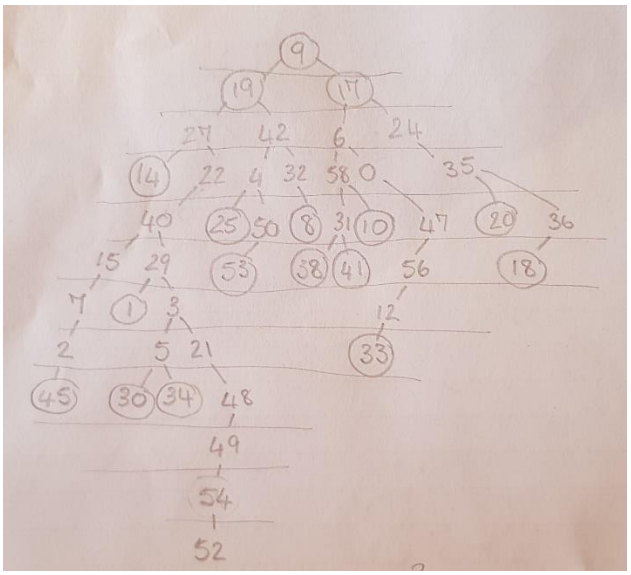
The breadth-first search function makes use of an array named queue which is initially empty, and an array named visited with length of the number of states in the DFSA. All elements in the visited array are initially set to False. This function also makes use of a dictionary in order to store the depth of each node from the initial state by getting the depth of the nodes' parent node and adding 1.

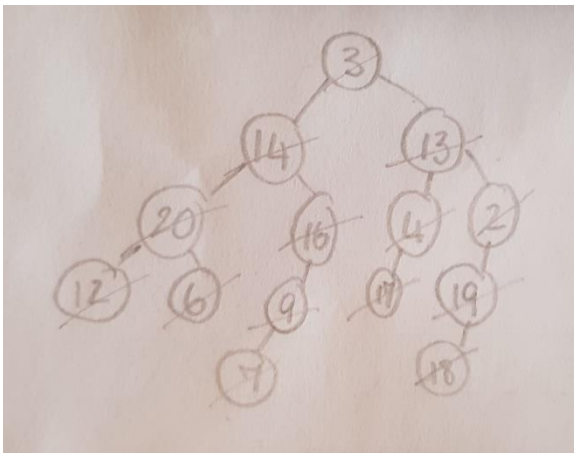
The breadth-first search function starts by appending the initial node to the queue, setting the initial node as True in the visited array and setting the depth of the initial node to be 0. It will then loop the following code until the queue is empty. The first element in the queue is popped and stored as the current node with the use of the current variable. The transition states from the current node are generated and unless they have already been visited, they are appended to the queue and their value in the visited array is set to True. Also, their depth will be set by getting the depth of the current node which would be their parent node and adding 1.

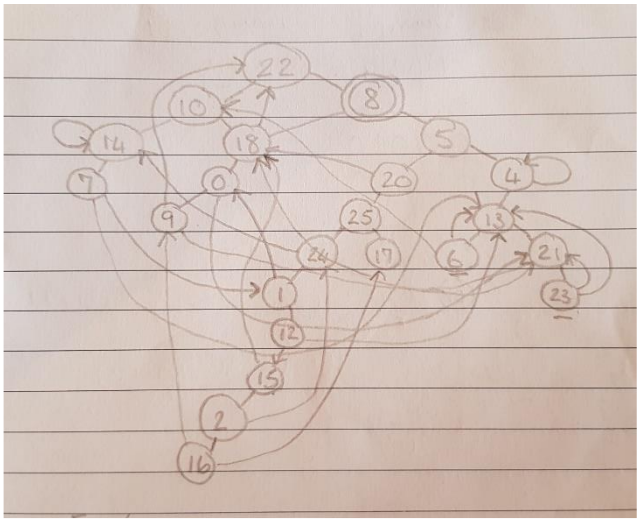
When the queue is empty and the loop terminates, the function will loop through the visited array and if the value at a particular position is set to True, then it will append the corresponding state id to the DFSA visited array. It will then display the DFSA visited array and get the depth by finding the maximum value stored in the depth dictionary. Finally, it will return the maximum depth.

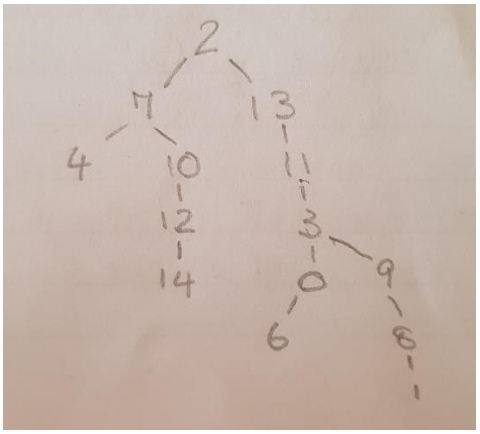
I also implemented the calculate_depth() function which calls the breadth-first search function in order to get the depth of the given DFSA and then it will display the depth of the DFSA.

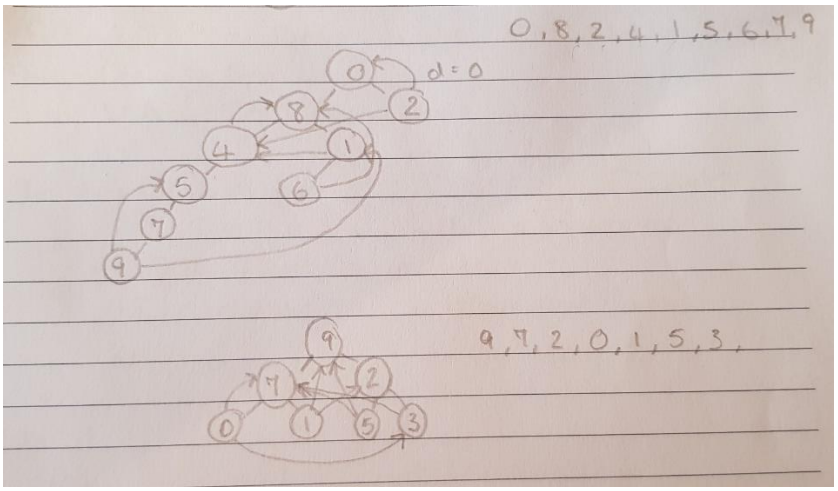
I tested that the given depth was correct by manually drawing the trees from the given initial state and transitions:


9
19
17
27
42
6
24
14
22
4
32
58
0
35
40
25
50
8
31
10
47
20
36
15
29
53
58
41
56
18
7
1
3
12
2
5
21
33
45
30
34
48
49
54
52


3
14
13
20
16
4
2
12
6
9
11
19
7
18


22
10
8
14
18
5
7
0
20
4
9
25
13
24
17
6
21
1
23
12
15
2
16


2
7
13
4
10
12
14
3
0
9
6
8


0, 8, 2, 4, 1, 5, 6, 7, 9
d = 0
0
8
2
4
1
5
6
7
9
9, 7, 2, 0, 1, 5, 3,
9
7
2
0
1
5
3

Minimising DFSA using Hopcroft minimization algorithm

Hopcroft minimisation algorithm sections the states in partitions in order to minimise the DFSA. I implemented this by first looping through the visited states from the breadth_first_search() function and appending the state ids in the visited array to either the final states array or the not final states array. I did this to remove any unreachable states. Afterwards, I initialised the partitions list P and waitlist list W with the final states and not final states.

The algorithm will loop with the following until the waitlist list is empty. The first element in the waitlist list will be popped and stored in variable A. The algorithm will them loop through the alphabet of the DFSA and an empty list named X is created or set to empty if it already exists. For every letter in the alphabet, it will loop through all the state ids stored in the visited array. If the state's transition is in A and not already present in X, then the state is appended to X. Hence, X will contain states that lead to any state in A with a particular transition. Then, the algorithm will loop through the partition array and calculate the intersection and difference of X with Y; Y being the current section of the partition array. If both the intersection and difference lists are not empty, Y will be removed from the partition list and the intersection and difference lists will be appended to the partition list. If Y is present in the waitlist list, it will be removed from the waitlist list and the intersection and difference lists will be appended to the waitlist list. Otherwise, the length of the intersection and difference lists is calculated and the list with the least length is appended to the waitlist list. If the length of the intersection and difference lists are the same, then the intersection list will be appended to the waitlist list. This is done to try to separate each partition section into more partitions to minimise the DFSA. After the waitlist array is empty and the algorithm exits all the loops, the new partition list is returned.

I also implemented the minimise() function which will be called from the main() function. It will call the hopcrofts_algorithm() function to get the partition list and store it in variable P. Then, it will loop through the partitions and for each partition, it will take the first state to check whether the states in the current partition are final states or not. Then the current partition will be set as a state by calling the State class and passing whether the partition is accepting or not. Then, the algorithm will loop through the letters in the alphabet and for each letter, it will get the transition from the first state in the partition corresponding to that letter. It will then loop through all the partition values and if the transition is stored in any of the partitions, then the partition which has the transition in it will be stored as the transition of the current partition with the corresponding letter. This only needs to be done using one state, in this case it's the first state in the partition, as all states in the partition are supposed to lead to the same partition. If the initial state of the DFSA is stored in the current partition, then the current partition will be set as the initial partition. The partition, as a state, will be appended to the new states array and its id will be appended to either the new final states array or to the new not final states array depending on whether the partition is accepting or not. The states in the current partition will be stored in the 'partition_states' array in order to keep track of which states were moved to which partition.

After the above is performed for all partitions, the new n is generated according to the number of partitions and the DFSA class is called to create a new DFSA with the new number of states (partitions), the new initial state, the new final states array, the new not final states array and the new states array. Afterwards, the new DFSA is returned.

To implement Hopcroft's algorithm, reference was mainly made to [14] and to the pseudocode found in [10]:

```
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
```

However, reference was also made to [9], [11], [12] and [13].

Implementing Tarjan's algorithm to find strongly connected components

To implement Tarjan's algorithm to find strongly connected components, reference was made to [15], [16], [17] and [18].

I first created a function called get_SCCs() which starts by resetting the 'scc_state' variable to -1 and the 'scc' array to empty. The it will create an empty array called 'disc_states' in order to keep track of the discovered states. Three arrays with size of the number of states in the DFSA are initialised. Two of these arrays; 'disc_ids' and 'low_values' have their values all set to -1 whilst the remaining array; 'current_stack' has its values all set to False. Then the function will loop through all the state ids of the DFSA and if the state hasn't been discovered yet, then the tarjon_algo() function will be called with the state id and the following arrays; disc_ids, disc_states, low_values and current_stack. After all the states have been discovered, the function will then call the print_SCCs() function.

The print_SCCs() function prints the number of strongly connected components found by printing the length of the 'scc' array. It then prints the size of the largest strongly connected component by finding the element with the maximum length in the 'scc' array and printing the element's length. Similarly, it will then print the size of the smallest strongly connected component by finding the element with the minimum length in the 'scc' array and printing the element's length.

The tarjan_algo() function takes in the state id and the following arrays; disc_ids, disc_states, low_values and current_stack as parameters. This function will start by increasing the

'scc_state' value by 1 and setting it to the 'disc_ids' and 'low_values' arrays at the state id index. It will also set the value of the 'current_stack' array at the state id index to True to show that this particular state is in the stack and it will append the state id to the 'disc_states' array to keep track of the discovered states. Afterwards, the function will loop through all the child nodes of the given state. If the child node is not discovered; meaning that the value in the 'disc_ids' array at the child node's index is still -1, then the function will call the trajan_algo() function with the required values.

After it exits, the values in the 'low_values' array at the state id and child id positions will be compared and the smallest value will be stored in the 'low_values' array at the state id position. If the child node has already been discovered and is on the current stack, then the value in the 'low_values' array at the state id position will be compared to the value in the 'disc_ids' array at the child id position and the smallest value will be stored in the 'low_values' array at the state id position.

After the function loops through all the child states for the given state, a variable named current is set to -1. For the given state, if the value stored in the 'low_values' array at the state id position is equal to the value stored in the 'disc_ids' array at the state id position, then the 'current_SCCs' array is set to empty. While the value stored in current is not equal to the state id, the 'disc_states' array will be popped, and the value will be set to the current variable. The popped state is set to be False in the 'current_stack' array and it is appended to the 'current_SCC' array. When the while loop exits, the 'current_SCCs' array is appended to the 'scc' array. Therefore, finally the 'scc' array will contain arrays of strongly connected components.

Johnson's Algorithm

A cycle is a path in a graph where the first and last vertex are the same. A simple cycle refers to a cycle where only the first and last vertex can be repeated twice, the other vertexes are not repeated. Johnson's algorithm is the fastest at finding simple cycles in a graph as it has time complexity of O(E+V)(C+1); where C is the total number of cycles in the graph and space complexity of O(E+V); where E is the number of edges and V is the number of vertices.

In order to perform Johnson's algorithm to find the simple cycles, the main graph is split into multiple sub graphs of the strongly connected components of the main graph. This is done to facilitate finding cycles as a cycle is never going to span over multiple strongly connected components. We then start from the SCC sub graph which contains the initial node.

We start with the initial node and check for all cycles which start and end at the initial node. In order to do this, a depth first search on the SCC sub graph is performed. Starting from the initial node, for each node, we check the child nodes and if the child nodes lead back to the initial node and so on. If a child node from the initial node leads back to the initial node or has child nodes that lead back to the initial node, then a cycle is found. However, if a path contains the same node more than once, apart from the initial node, then it is not considered a simple cycle. To do this we implement a stack which contains the current visited nodes of a particular path

as well as another stack which contains the path nodes. If a child node is already in the visited stack, then it is counted as blocked, and the algorithm moves on to another child node. To do this, the node which points to the already visited node is removed from the path stack and the child nodes of the topmost node from the path stack are checked. When a simple cycle is found, the path containing the nodes is stored in an array. After gathering all the cycles for the particular node (initial node), the initial node and all outgoing and incoming edges from it from the main graph are removed. The main graph is then split again into multiple sub graphs of SCCs and the initial node will then be the next node from the initial node, which is present in a sub graph that contains at least 2 nodes. The above process is repeated until there are no SCC sub graphs with at least 2 nodes left.

I referred to [19] in order to fully understand how this algorithm works, how it is used for finding all simple cycles and how it can be implemented.

Conclusion

Using the above functions, I completed all the required steps.

In the main function, I first generated a random number between 16 and 64 inclusive and created a DFSA A by calling the DFSA class with that number. I then called the print_details() function to print the details of the DFSA A to check that everything was working correctly. I called the calculate_depth() function to compute the depth of A. Afterwards, in order to minimise DFSA A and obtain a new DFSA M, the minimise() function was called by passing in A as the parameter and allocating the result in variable M. The print_details() function was called on DFSA M to check the result of the minimisation. The calculate_depth() function was later on called on DFSA M to compute the depth of this automata. Lastly, the get_SCCs() function was called on DFSA M in order to find the strongly connected components of M and print the requested information.

I wanted to implement Johnson's algorithm but unfortunately due to time constraints I didn't.

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

References

[1] "Deterministic Finite Automaton" tutorialspoint.com. https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm (Accessed Mar. 18, 2022)

[2] M. Mohsin. "C code to implement DFA(Deterministic Finite Automata)" coders-hub.com. https://www.coders-hub.com/2013/05/c-code-for-dfadeterministic-finite.html (Accessed Mar. 18, 2022)

[3] A. Bhatia. "DFA.c" gist.github.com. https://gist.github.com/bhatiaabhinav/d28e037dcc3d94b42076 (Accessed Mar. 18, 2022)

[4] I. Bondin. (2021). Formal Languages and Automata [PowerPoint Slides]. Available: https://www.um.edu.mt/vle/pluginfile.php/937526/mod_resource/content/2/Slide%20Deck%20%2825112021%29%20-%20Formalisation%20of%20DFA.pdf (Accessed Mar. 18, 2022)

[5] C. Maklin. "Breadth First Search In Python" medium.com. https://medium.com/geekculture/breadth-first-search-in-python-822fb97e0775 (Accessed Apr. 20, 2022)

[6] "Breadth First Search or BFS for a Graph" geeksforgeeks.org. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ (Accessed Apr. 23, 2022)

[7] S. Bhadaniya. "Breadth First Search in Python (with Code) | BFS Algorithm" favtutor.com. https://favtutor.com/blogs/breadth-first-search-python (Accessed Apr. 23, 2022)

[8] "Program to calculate Height and Depth of a node in a Binary Tree" geeksforgeeks.org. https://www.geeksforgeeks.org/program-to-calculate-height-and-depth-of-a-node-in-a-binary-tree/ (Accessed Apr. 23, 2022)

[9] "DFA Minimization by Hopcroft algorithm [closed]" stackoverflow.com. https://stackoverflow.com/questions/13549662/dfa-minimization-by-hopcroft-algorithm (Accessed Apr. 28, 2022)

[10] "DFA minimization" en.wikipedia.org. https://en.wikipedia.org/wiki/DFA_minimization#Hopcroft.27s_algorithm (Accessed Apr. 28, 2022)

[11] “Hopcroft's algorithm - DFA Minimization” stackoverflow.com. https://stackoverflow.com/questions/26727766/hopcrofts-algorithm-dfa-minimization (Accessed Apr. 28, 2022)

[12] “DFA Minimization” swaminathanj.github.io. https://swaminathanj.github.io/fsm/dfaminimization.html (Accessed Apr. 29, 2022)

[13] P. Garcia, D. Lopez. and M. Vazquez de Parga. “DFA minimization: from Brzozowski to Hopcroft” Dept. Information Systems and Computing, Univ. Polytechnic Valencia. [Online]. Available: https://m.riunet.upv.es/bitstream/handle/10251/27623/partial%20rev%20determ.pdf

[14] “Minimization of DFA” geeksforgeeks.org. https://www.geeksforgeeks.org/minimization-of-dfa/ (Accessed May. 03, 2022)

[15] “Tarjan’s Algorithm to find Strongly Connected Components” geeksforgeeks.org. https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/ (Accessed May. 12, 2022)

[16] “Tarjan's strongly connected components algorithm” en.wikipedia.org. https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm (Accessed May. 12, 2022)

[17] N. Kumar Singh. “TARJAN’S ALGORITHM FOR STRONGLY CONNECTED COMPONENTS” topcoder.com. https://www.topcoder.com/thrive/articles/tarjans-algorithm-for-strongly-connected-components (Accessed May. 12, 2022)

[18] A. Dey. “Tarjan's Algorithm for finding Strongly Connected Components in Directed Graph” thealgorists.com. https://www.thealgorists.com/Algo/GraphTheory/Tarjan/SCC (Accessed May. 12, 2022)

[19] Tushar Roy - Coding Made Simple, Johnson's Algorithm - All simple cycles in directed graph. (Nov. 23, 2015). Accessed: May 15, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=johyrWospv0