University of Malta

Faculty of Information and Communication Technology

Course Project 2025

Data Structures and Algorithms 2

Name: Ben Taliana

Student ID: [REDACTED]

Contents

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).

Ben Taliana	BenT
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS 2210 — Course Code

Data Structures & Algorithms 2 Course Project 2025 — Title of work submitted

21/05/2025 — Date

Statement of Completion

Item	**Completed (Yes/No/Partial)**
Create random DFAs	Yes
Computing the depth of DFAs	Yes
Minimised DFAs	Yes
Finding SCCs in A	Yes
Finding SCCs in M	Yes
Empirically show that DFA minimisation works properly	Yes
Empirically show that SCC implementation works properly	Yes

1 Overview and Setup

The implementation of this assignment was carried out using Python for its readability and support for algorithmic problem solving. Initially, the implementations were developed in separate Python scripts but were later combined into a single Jupyter Notebook (DFA.ipynb) to improve the overall organization, maintain modularity, and facilitate easier correction, evaluation, and demonstration.

In the notebook, each section was annotated using Markdown headers and Markdown text to indicate the purpose of the subsequent code blocks. The main headers of the notebook are as follows.

- Question 1: Construction of A
- Question 2: Computing depth of d_A of A using BFS
- Question 3: Minimization of A using Hopcroft's Algorithm
- Question 4: Computing the depth d_M of M using BFS
- Question 5: Tarjan's Algorithm and SCCs
- Empirical Testing and Edge Cases

To support and consolidate the correctness of the implementation, Graphviz was used to visualize the DFA outputs and compare them with credible examples.

2 Implementation

To encapsulate the implementation's functions, a class DFA was created. Moreover, because of the notebook cell structure, monkey patching was used to dynamically assign each function to the DFA class in their respective cells.

2.1 Question 1

Because the assignment involves randomness, particularly in this section, a 'seed' parameter is introduced so that the outputs can be reproduced and verified. Additionally a boolean 'debug' parameter was added, which calls a helper 'print_summary' that prints out the DFA's attributes in the terminal. The snippet below shows the creation of the DFA class and the '__init__' function, which initializes the DFA to randomly create n states, where

- $n \in [16, 64]$
- Each n is randomly designated to an *accepting* or *rejecting* label.
- Every n is randomly assigned an outgoing transition for a and b to another state.
- One n is randomly selected as a start state.

2.1.1 Adjacency List vs. Adjacency Matrix

When implementing the DFA, we chose to represent its structure using an adjacency list implemented as a nested Python dictionary. This choice is ideal because each state has only one transition per symbol, which results in a sparse structure. An adjacency matrix requires $O(n^2)$ space regardless of how the states are connected, leading to high memory usage. Nevertheless, an adjacency list requires only $O(n|\Sigma|)$ where Σ is the alphabet. This approach only stores the transitions that actually exist. Additionally, Python dictionaries provide $O(1)$ access and align well with DFA logic. This makes the code more readable and easier to maintain.

```python
class DFA:
    # Question 1a
    def __init__(self, min=16, max=64, debug=False, seed=None): # Min = 16 and Max = 64
        self._rng = random.Random(seed)

        self.n = self._rng.randint(min, max) # Question 1a
        self.states = list(range(self.n)) # Question 1a

        # Question 1b
        self.accepting = {state: self._rng.choice([True, False]) for state in self.states} # Random accepting

        # Question 1c
        self.alphabet = ['a', 'b'] # a and b
        # Random transitions for each state and symbol
        self.transition = {
            state: {
            symbol: self._rng.randint(0, self.n - 1)
            for symbol in self.alphabet
            } for state in self.states
        }

        # Question 1d
        self.start_state = self._rng.choice(self.states) # Random start state

        # Helper function for debugging
        if debug:
            self.print_summary()
```

The snippet below instantiates A with debug = True and seed = 2025 for reproducibility.

```python
A = DFA(debug=True, seed=2025)
```

2.2 Question 2

This question required the implementation of a BFS algorithm to compute the shortest path to a reachable state, where the maximum of these lengths is returned as the graph's depth. The BFS algorithm is a very efficient way to find the shortest path in an unweighted graph. In my implementation below, the 'bfs_depth' function uses the concept of infinity to represent undiscovered states for mathematical precision and Python's double ended queue (deque) for optimal queue operations.

```python
def bfs_depth(self):
    # Set all distances to inf for now (explanation in report)
    distances = {state: float('inf') for state in self.states}
    distances[self.start_state] = 0  # Set the start state's distance to 0

    # Using a double ended queue
    queue = deque([self.start_state])

    while queue:
        current = queue.popleft()
        # Every transition from current state
        for symbol in self.alphabet:
            next_state = self.transition[current][symbol]
            # If next isn't visited update distance and enqueue
            if distances[next_state] == float('inf'):
                distances[next_state] = distances[current] + 1
                queue.append(next_state)

    # Use only reachable states for depth
    reachable_distances = [d for d in distances.values() if d != float('inf')]

    return max(reachable_distances) if reachable_distances else 0

DFA.bfs_depth = bfs_depth # Monkey Patching to attach function to DFA
```

The code below prints the number of states in A and d_A in the terminal:

```python
print(f"Number of states in A: {A.n}")
d_A = A.bfs_depth()
print(f"Depth d_A of A: {d_A}")
```

As shown in the notebook, the output for seed = 2025 is:

```
Number of states in A: 51
Depth d_A of A: 11
```

2.3 Question 3

This question requires a comparison of two minimization algorithms, Hopcroft's Algorithm [1] and Moore's Algorithm [2], both of which minimize, but contrast significantly:

Feature	**Hopcroft's Algorithm**	**Moore's Algorithm**
Time Complexity	$O(n \log n)$	$O(n^2)$
Mechanism	Worklist refinement (only splits affected partitions)	Full partition refinement in each iteration
Efficiency	Faster on sparse/large DFAs	Slower on large or random DFAs
Implementation Complexity	Moderate	Simple and intuitive
Practical Suitability	Best for repeated runs and experimental setups	Good for small DFA examples and teaching

Table 1: Comparison of Hopcroft's and Moore's DFA minimisation algorithms

2.3.1 Justification for Choice (Hopcroft's Algorithm)

Hopcroft's algorithm is asymptotically faster than Moore's algorithm, $O(n \log n)$ vs. $O(n^2)$, and is a better choice for this assignment because it scales better to the random nature of the DFAs this assignment generates (up to 64 states). Hopcroft's worklist ensures fewer unnecessary refinements, which reduces overhead compared to Moore's, which is ideal for repeated testing and minimization. Hopcroft's $O(n \log n)$ complexity makes it ideal for compilers, network protocol verification, and text processing tools, because it performs well on large DFAs. This reduced execution time makes it ideal for performance-sensitive applications.

2.3.2 Implementation Details

Before proceeding with Hopcroft's algorithm, it is important to note that A must be pruned to remove the unreachable [6] and useless [7] states. In order to do this, I implemented a 'prune' function and helpers 'get_reachable_states' and 'get_useful states_states'. In combination, these functions prune the unwanted states and then reconstruct the new DFA while using a trap state to ensure that any state mapping to the removed states is remapped correctly. The 'prune' function can be found in Appendix A

My implementation of Hopcroft's algorithm differs from traditional implementations, such as that in [13]. The version I used pre-computes reverse transitions and avoids scanning all states when splitting partitions, which is ideal for large graphs and repeated minimization cases. With this small optimization, the lookup cost changes from $O(n)$ to $O(1)$ or $O(k)$ where k is the number of predecessors. The 'hopcroft_minimization' algorithm also uses a helper 'empty_dfa' that initializes an empty DFA for reconstruction.

```
def hopcroft_minimization(self):
    states = set(self.states)

    # Split states into accepting and non-accepting
    accepting_states = frozenset(state for state in states if
    ↪   self.accepting[state])
    non_accepting_states = frozenset(states - accepting_states)

    P = [] # Create the partition list
    if accepting_states:
        P.append(accepting_states)
    if non_accepting_states:
        P.append(non_accepting_states)

    # Create the worklist with the smaller partition
    W = set()
    if accepting_states and non_accepting_states:
        W.add(accepting_states if len(accepting_states) <=
        ↪   len(non_accepting_states) else non_accepting_states)
    elif accepting_states:
        W.add(accepting_states)
    elif non_accepting_states:
        W.add(non_accepting_states)

    # Precompute reverse transitions for each symbol for efficiency
    reverse_transition = {
        'a': {s: set() for s in states},
        'b': {s: set() for s in states}
    }
    for s in states:
        for symbol in self.alphabet:
            target = self.transition[s][symbol]
            reverse_transition[symbol][target].add(s)

    # Refinement loop
    while W:
        A = W.pop()
        for symbol in self.alphabet:
            X = set()
            for target in A:
                X.update(reverse_transition[symbol][target])
            X = frozenset(X)

            partition_changes = []
```

```
                for i, Y in enumerate(P):
                    intersection = Y & X
                    difference = Y - X
                    if intersection and difference:
                        partition_changes.append((i, Y, intersection, difference))

                # Refine the partitions and update the worklist
                for i, Y, intersection, difference in reversed(partition_changes):
                    P.pop(i)
                    P.append(intersection)
                    P.append(difference)

                    if Y in W:
                        W.discard(Y)
                        W.add(intersection)
                        W.add(difference)
                    else:
                        W.add(intersection if len(intersection) <= len(difference) else difference)

        # Mapping the old states to new states
        state_to_new_state = {
            state: new_state
            for new_state, partition in enumerate(P)
            for state in partition
        }

        # Building the minimized DFA
        minimized_dfa = DFA.empty_dfa()
        minimized_dfa.n = len(P)
        minimized_dfa.states = list(range(len(P)))

        # Creating the accepting states in minimized DFA
        minimized_dfa.accepting = {
            new_state: any(self.accepting[state] for state in partition)
            for new_state, partition in enumerate(P)
        }

        # Creating  the transitions in minimized DFA
        minimized_dfa.transition = {
            new_state: {
                symbol: state_to_new_state[self.transition[next(iter(partition))][symbol]]
                for symbol in self.alphabet
            } for new_state, partition in enumerate(P)
        }
```

```

    # Set start state
    minimized_dfa.start_state = state_to_new_state[self.start_state]
    return minimized_dfa

# Adding to the DFA class
DFA.hopcroft_minimization = hopcroft_minimization

```

Here, we're applying the 'prune' function on A and creating M by deploying Hopcroft's minimization algorithm on the pruned version of A.

```
A.prune()
M = A.hopcroft_minimization()
```

2.4 Question 4

2.4.1 Number of states in M and d_M

The snippet below prints the number of states n at M and M's depth d_M.

```
print(f"Number of states in M: {M.n}")
d_M = M.bfs_depth()
print(f"Depth d_M of M: {d_M}")
```

As shown in the notebook, the output for is:

```
Number of states in M: 38
Depth d_M of M: 11
```

This output shows a significant decrease in the number of states from A to M. Further inspection of the result showed that the main reason for such a large reduction is largely due to pruning, and this aligns with the fact that the random nature of A can create many unreachable or useless states. The depth remains constant, which is expected because BFS disregards undiscovered and unreachable states.

2.5 Question 5

2.5.1 What are SCCs?

Strongly connected components are subsets of states within a directed graph, where every state is reachable from every other state in the subset. SCCs can reveal the internal cycles, loops, and structural properties of DFA. Tarjan's algorithm was implemented to[3] identify SCCs.

2.5.2 Tarjan's Algorithm Implementation

Tarjan's algorithm has a time complexity of $O(V + E)$ where V is the number of vertices (states in the DFA) and E is the number of edges (transitions). This implies that the algorithm scales linearly with the size of the graph, making it efficient for sparse and dense graphs. The linear runtime ensures that Tarjan's algorithm performs consistently, even with randomly generated DFA instances, similar to ours. The algorithm is critical for detecting loops and SCCs, and its linear time complexity makes it useful, even for large graphs, such as those in software analysis or biological networks. Its efficiency ensures that it can run frequently as a part of real-time tools without causing bottlenecks. In the snippet below, the 'compute_sccs' function implements Tarjan's algorithm for identifying SCCs.

```python
def compute_sccs(self):
    index = 0
    index_map = {}  # State indices
    lowlink_map = {}  # Dict for lowest index
    stack = []  # Stack for current path
    on_stack = set()  # Set for looking up states
    sccs = [] # List for SCCs

    # Recursive DFS function
    def strongconnect(v):
        nonlocal index # Allows modification of index from strongconnect
        index_map[v] = index
        lowlink_map[v] = index
        index += 1
        stack.append(v)
        on_stack.add(v)

        for symbol in self.alphabet:
            w = self.transition[v][symbol]
            if w not in index_map: # If w is not visited
                strongconnect(w)
                lowlink_map[v] = min(lowlink_map[v], lowlink_map[w])
            elif w in on_stack:
               lowlink_map[v] = min(lowlink_map[v], index_map[w]) # Update lowlink

        if lowlink_map[v] == index_map[v]: # If v lowlink = index then v is a root node
            scc = []
            while True:
                w = stack.pop()
                on_stack.remove(w)
                scc.append(w)
                if w == v: # If at the root
                    break
```

```python
            sccs.append(scc)

    for state in self.states:
        # If state hasn't been visited
        if state not in index_map:
            strongconnect(state)

    return sccs

DFA.compute_sccs = compute_sccs
```

The following two snippets print the number of SCCs and the size of the largest and smallest SCCs in A and M, respectively.

2.5.3 SCCs in DFA A

```
sccs_A = A.compute_sccs()
sizes_A = [len(scc) for scc in sccs_A]
print(f"Number of SCCs in A: {len(sccs_A)}")
print(f"Size of largest SCC in A: {max(sizes_A)}")
print(f"Size of smallest SCC in A: {min(sizes_A)}")
```

Output:

```
Number of SCCs in A: 1
Size of largest SCC in A: 39
Size of smallest SCC in A: 39
```

The number of SCCs in A is one, meaning that all reachable states in A are mutually reachable and that every state in A is part of the cycle, indicating that there are no isolated states, dead ends, or acyclic chains.

2.5.4 SCCs in Minimized DFA M

```
sccs_M = M.compute_sccs()
sizes_M = [len(scc) for scc in sccs_M]
print(f"Number of SCCs in M: {len(sccs_M)}")
print(f"Size of largest SCC in M: {max(sizes_M)}")
print(f"Size of smallest SCC in M: {min(sizes_M)}")
```

Output:

```
Number of SCCs in M: 1
Size of largest SCC in M: 38
Size of smallest SCC in M: 38
```

The same conclusions for A apply to M because it has only one SCC. However, their size decreased from 39 to 38. This behavior is predictable and can be explained by the minimization of merged equivalent states, eliminating redundancy but preserving the cycle structure.

3 Experiments and Evaluation

Two visualization functions were created: one for regular DFAs, and the other for DFAs with SCCs. For modularity and encapsulation, a DFATest class was created, which contained all of the functions used for testing.

3.1 Empirical Verification of DFA Minimisation

When minimizing A to M, the language of the automate must be preserved: $L(A) = L(M)$. The same string sequence that is accepted for A should also be accepted for M, and when A rejects, so does M. To verify this property, a function 'validate_equivalence' was implemented, this creates a diverse set characters to properly test whether the language is preserved. The 'validate_equivalence' can be found in Appendix A. Some of the test cases include the following.

- Randomly generated strings.
- An empty string.
- Strings of length equal to the DFA's depth.
- Strings with common patterns to mimic cycles .

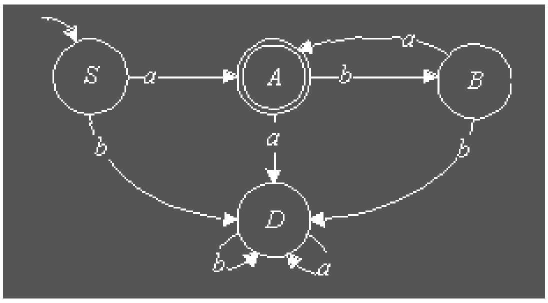
Output:

```
--- Testing if A and M are functionally equivalent ---
Input: '' -> A: True, M: True
Input: 'bab' -> A: False, M: False
Input: 'bbbbbbbbbbb' -> A: True, M: True
Input: 'aabababaaa' -> A: True, M: True
Input: 'aabbaabbaabb' -> A: False, M: False
Input: 'abbbaab' -> A: True, M: True
Input: 'bbaabbaabb' -> A: True, M: True
Input: 'bababababa' -> A: True, M: True
Input: 'b' -> A: True, M: True
Input: 'baa' -> A: True, M: True
Input: 'aaaaaaaaaa' -> A: True, M: True
Input: 'bbbbbbbaa' -> A: True, M: True
Input: 'aab' -> A: False, M: False
Input: 'ababababab' -> A: True, M: True
Input: 'a' -> A: True, M: True
Input: 'aaabb' -> A: True, M: True
Input: 'bbbbbbbbbb' -> A: False, M: False
Input: 'bbababb' -> A: True, M: True
Input: 'aaaaaaaaaaa' -> A: True, M: True
Test Passed: L(A) = L(M), all strings were accepted or rejected identically.
```

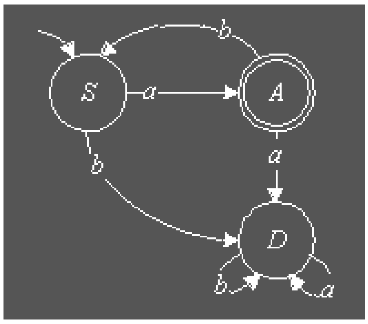
The test shows that minimization preserves the language of A, so this property holds.

3.1.1 Examples and Edge Cases

In order to have some ground truth examples of minimization to compare my algorithm's output to, I designed three functions 'example_1', 'example_2' and 'example_3'. Example 1 uses a credible example from UM's RELIC portal, which is Interactive Learning of Formal Languages and Automata [5]. This source explains how minimization works and provides graphs before (a) and after minimization (b) for comparison.



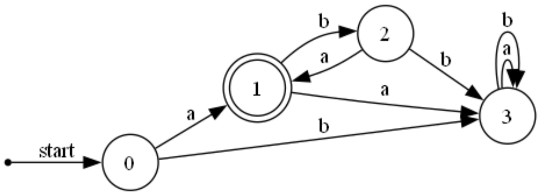
(a) RELIC Example Before Minimization



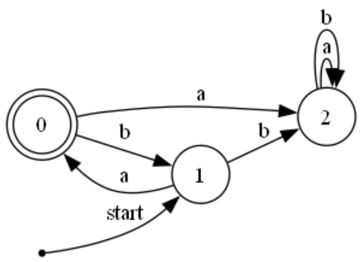
(b) RELIC Example After Minimization

Figure 1: RELIC comparison of DFA structure before and after minimization

The 'example_1' function recreates the DFA from the example and visualizes it. After minimization and pruning are applied, the new minimized graph is visualized and compared to the example from the source. The equivalence validation function was also used to verify the results. Appendix A provides a quick reference to the 'example_1' function.



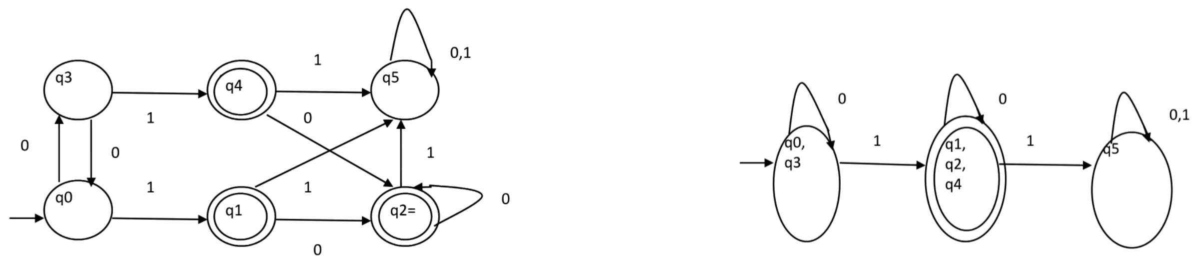
(a) Implementation Example Before Minimization



(b) Implementation Example After Minimization

Figure 2: Example 1 comparison of DFA structure before and after minimization

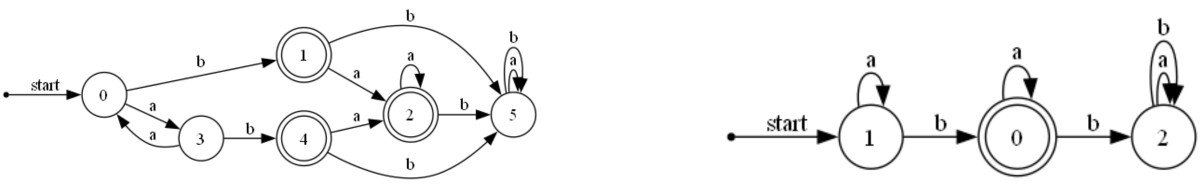
Inspection of the visual result shows that the minimized function outputs correctly, and the equivalence test outputs that $L(A) = L(M)$, meaning that when minimizing the example, the language is preserved. It must be noted, however, that the accepting state in the RELIC example remains the same, whereas in the output of the algorithm, although the semantics of the DFA are preserved. This is because State IDs are arbitrary and they're renumbered for efficiency during minimization.The 'example_2' function does the same with a different example from GeeksforGeeks [10]



(a) Implementation Example Before Minimization

(b) Implementation Example After Minimization

Figure 3: GeeksforGeeks comparison of DFA structure before and after minimization



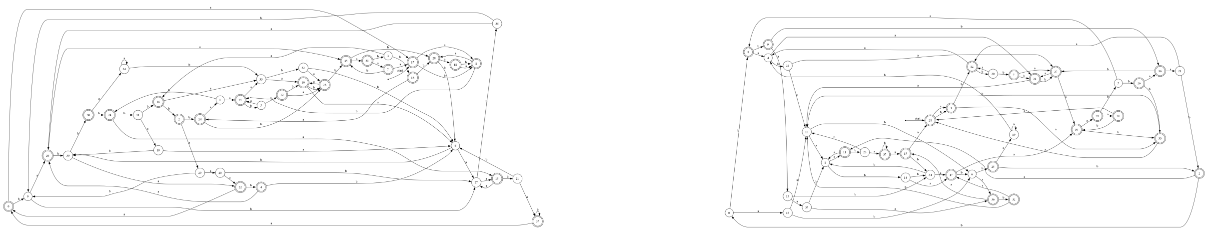
(a) Implementation Example Before Minimization

(b) Implementation Example After Minimization

Figure 4: Example 2 comparison of DFA structure before and after minimization

The output for 'example_2' matches the example's output, semantics are preserved and this example passed the functional equivalence test.

The 'example_3' function simply displays A and M using the visualization function for comparison purposes.



(a) Implementation Example Before Minimization

(b) Implementation Example After Minimization

Figure 5: Example 3 comparison of DFA structure before and after minimization

Since the code for Examples 1-3 is repetitive, only 'example_1' code is shown in this report, the rest can be found in the notebook.

To wrap up the testing for my minimization algorithm, I have designed a function 'hopcroft_test_cases', that has a variety of test cases with asserts and expected outputs which include:

- All-Accepting DFA
- Already minimal DFA
- One-State DFA

Output:

```
--- Hopcroft Test Cases ---
Test Case 1: All-Accepting DFA
Expected: 1 state, Actual: 1 state
Test Case 2: Already Minimal DFA
Expected: 2 states, Actual: 2 states
Test Case 3: One-State DFA
Expected: 1 states, Actual: 1 states
Expected Depth: 0, Actual Depth: 0
```

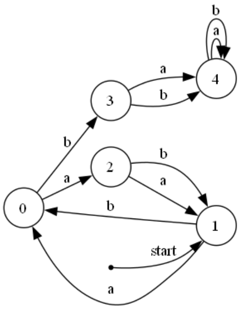
The output shows that my algorithm handles trivial edge cases properly.The code for this function can be found in Appendix A.

3.2 Empirical Verification of SCC Detection

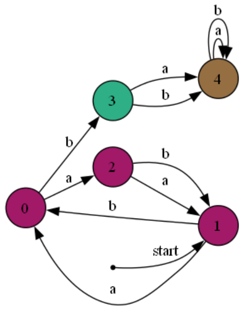
Similarly to the previous section, I created a function 'test_tarjan_properties' that manually checks if all the returned nodes from the algorithm are SCCs. Additionally, I found five examples from GeeksforGeeks that had a text-based example. I also found a graphical example from Stanford lecturing material[?]. I compared the output of my program with examples from sources and verified their output. The visuals for this section used a different visualization function that shows the SCC groups in different colors. The 'test_tarjan_properties' function was run for each example and can be found in Appendix A

Example 10 displays A, M, and their SCCs for comparison.

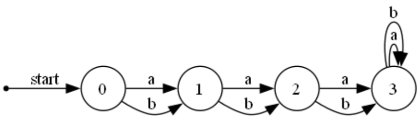
The outputs of each 'test_tarjan_properties' function was valid and these examples are verified against the examples from the sources shown in Appendix A.



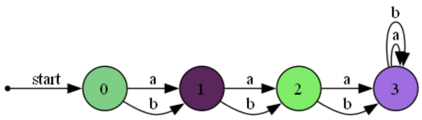
(a) Example 4 - Original DFA



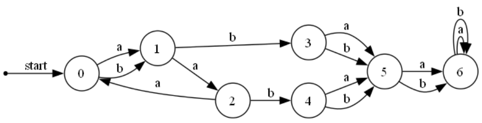
(b) Example 4 - SCC Highlighted



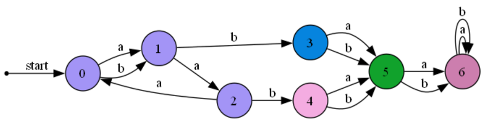
(c) Example 5 - Original DFA



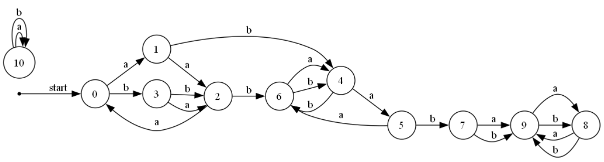
(d) Example 5 - SCC Highlighted



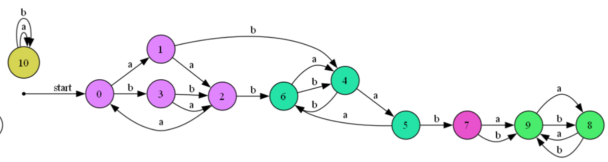
(e) Example 6 - Original DFA



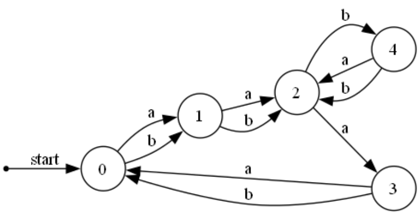
(f) Example 6 - SCC Highlighted



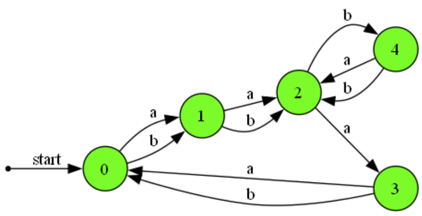
(g) Example 7 - Original DFA



(h) Example 7 - SCC Highlighted

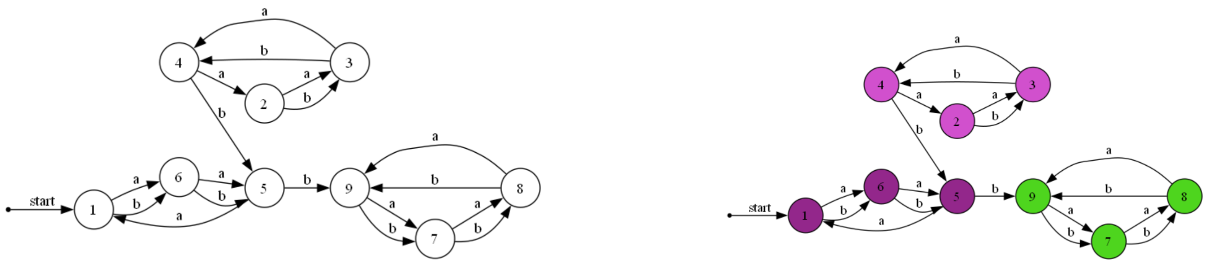


(i) Example 8 - Original DFA



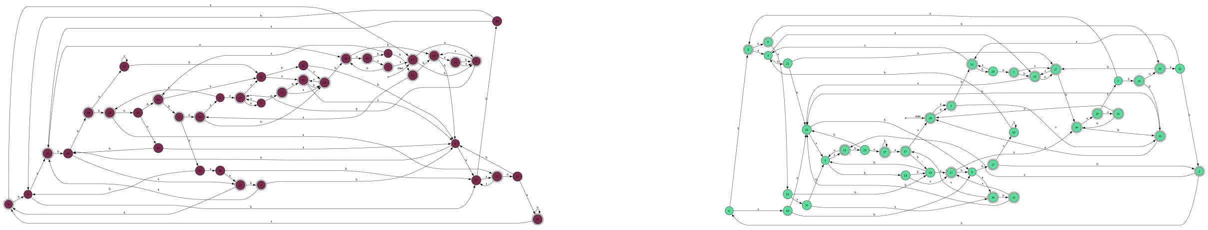
(j) Example 8 - SCC Highlighted

Figure 6: Side-by-side comparison of DFA visualisations and their SCC-highlighted counterparts for Examples 4 to 8.



(a) Example 9 (b) Example 9 with SCCs

Figure 7: Example 9 Comparison



(a) Example 10 (b) Example 10 with SCCs

Figure 8: Example 10 Comparison

4 Conclusion

In this report, I present the full implementation and extensive testing of ICS2210 Course Project 2025. This included the construction of a DFA, calculating its depth using BFS and the minimization of the DFA using Hopcroft's algorithm, as well as the detection of strongly connected components (SCCs) using Tarjan's algorithm. Using a Python-based approach, each component was verified visually and programmatically. Minimization was shown to preserve the structural and semantic properties of the original DFA, and SCC analysis provided insights into the connectivity of the DFA. Design choices, such as using an adjacency list, Hopcroft's algorithm over Moore's, and the precomputing of reverse transitions, were justified for their efficiency. Overall, the implementation meets all the requirements of the assignment and demonstrates correctness and clarity across a range of test cases.

A Appendix A: Source Code Snippets

A.1 Prune Function

```
def prune(self):
    # Using helpers get_reachable_states and get_useful_states
    reachable = self.get_reachable_states()
    useful = self.get_useful_states()

    # Valid states are both reachable and useful
    valid = reachable & useful

    # If valid states remain, reset to an empty DFA
    if not valid:
        self.states = []
        self.n = 0
        self.accepting = {}
        self.transition = {}
        self.start_state = None
        return


    state_mapping = {old: new for new, old in enumerate(valid)}

    # Creating a trap state for possible invalid transitions
    trap_state_id = len(state_mapping)

    # Update state list and count
    self.states = list(state_mapping.values())
    self.n = len(self.states)

    # Remap accepting states
    self.accepting = {
        state_mapping[s]: self.accepting[s]
        for s in valid
    }

    # Remap transitions
    new_transitions = {}
    trap_needed = False # default

    for s in valid:
        new_s = state_mapping[s]
        new_transitions[new_s] = {}
        for symbol in self.alphabet:
            t = self.transition[s][symbol]
```

```
            if t not in valid: # If transition is invalid, map to the trap
                trap_needed = True
                new_transitions[new_s][symbol] = trap_state_id
            else:
                new_transitions[new_s][symbol] = state_mapping[t] # Else map to new state

    # If needed, define the trap state
    if trap_needed:
        self.states.append(trap_state_id)
        self.n += 1
        self.accepting[trap_state_id] = False
        new_transitions[trap_state_id] = {'a': trap_state_id, 'b': trap_state_id}

    # Update transitions and start
    self.transition = new_transitions
    self.start_state = state_mapping[self.start_state]

DFA.prune = prune
```

A.2 Validate Equivalence Function

```
class DFATest:
    @staticmethod
    def generate_random_strings(count=10, max_length=10):
        # Generate a list of random strings
        strings = []
        for _ in range(count):
            length = random.randint(1, max_length)  # Random length
            s = ''.join(random.choice(['a', 'b']) for _ in range(length))  # Random string
            strings.append(s)
        return strings

    @staticmethod
    def validate_equivalence(dfa, minimized_dfa, num_tests=15, max_len=10):
        print("\n--- Testing if A and M are functionally equivalent ---")
        test_strings = DFATest.generate_random_strings(num_tests, max_len)

        # Add empty string for edge case
        test_strings.append("")

        # Addding strings that are the same length as the DFA depth
        if hasattr(dfa, 'bfs_depth'):
```

```
            try:
                depth_a = dfa.bfs_depth()
            except Exception:
                depth_a = max_len
        else:
            depth_a = max_len
        if hasattr(minimized_dfa, 'bfs_depth'):
            try:
                depth_m = minimized_dfa.bfs_depth()
            except Exception:
                depth_m = max_len
        else:
            depth_m = max_len

        # Generate strings at DFA depth
        test_strings.append('a' * depth_a)
        test_strings.append('b' * depth_a)
        test_strings.append('ab' * (depth_a // 2))
        test_strings.append('ba' * (depth_a // 2))
        test_strings.append('a' * depth_m)
        test_strings.append('b' * depth_m)
        test_strings.append('ab' * (depth_m // 2))
        test_strings.append('ba' * (depth_m // 2))

        # Add some repetitive pattern strings
        test_strings.append('ababababab')
        test_strings.append('aaaaaaaaaa')
        test_strings.append('bbbbbbbbbb')
        test_strings.append('aabbaabbaabb')
        test_strings.append('bbaabbaabb')

        # Remove duplicates
        test_strings = list(set(test_strings))
        all_match = True
        mismatched_strings = []

        # Test each string on both DFAs
        for s in test_strings:
            result_a = dfa.accepts_string(s)
            result_m = minimized_dfa.accepts_string(s)
            print(f"Input: '{s}' -> A: {result_a}, M: {result_m}")
            if result_a != result_m:
                print("Mismatch found!")
                all_match = False
                mismatched_strings.append(s)

```

```python
        if all_match:
            print("Test Passed: L(A) = L(M), all strings were accepted or rejected
             ↪  identically.")
        else:
            print("A functional mismatch was found between A and M.")
            print(f"Mismatched strings: {mismatched_strings}")

        # Assert for automated testing
        assert all_match, f"DFAs are not equivalent. Mismatched strings:
         ↪  {mismatched_strings}"

        return all_match
```

A.2.1 Example 1 Function

```python
@staticmethod
def example_1():

    dfa = DFA.empty_dfa()
    dfa.states = [0,1,2,3]
    dfa.n = 4
    dfa.accepting = {0: False, 1: True, 2: False, 3: False}

    # Starting from 0 for clarity
    dfa.transition = {
        0: {'a': 1, 'b': 3}, # State S or State 1
        1: {'a': 3, 'b': 2}, # State A or State 2
        2: {'a': 1, 'b': 3}, # State B or State 3
        3: {'a': 3, 'b': 3}, # State D or State 4
        }

    dfa.start_state = 0

    print("\n--- Example 1 ---")

    print_summary(dfa)

    # Visualize Example 1 before minimization
    dot_example = visualize_dfa(dfa, title="DFA Example 1 Before Minimization")
    dot_example.render('example_1', format='png', cleanup=True)
    display(Image('example_1.png'))

    # Prune and Minimize the DFA
    dfa.prune()
```

```python
    minimized_dfa = dfa.hopcroft_minimization()

    print("\n--- Example 1 After Pruning and Minimization ---")

    print_summary(minimized_dfa)
    # Visualize Example 1 after minimization
    dot_example_minimized = visualize_dfa(minimized_dfa, title="DFA Example 1 After Minimization")
    dot_example_minimized.render('example_1_minimized', format='png', cleanup=True)
    display(Image('example_1_minimized.png'))

    # Checking for equivalence
    DFATest.validate_equivalence(dfa, minimized_dfa)


DFATest.example_1 = example_1
DFATest.example_1()
```

A.3 Hopcroft Test Cases Function

```python
@staticmethod
def hopcroft_test_cases():
    print("--- Hopcroft Test Cases ---")

    # All-accepting DFA
    print("Test Case 1: All-Accepting DFA")
    dfa = DFA()
    dfa.n = 3
    dfa.states = [0, 1, 2]
    dfa.accepting = {0: True, 1: True, 2: True}
    dfa.transition = {
        0: {'a': 1, 'b': 2},
        1: {'a': 0, 'b': 2},
        2: {'a': 2, 'b': 1}
    }
    dfa.start_state = 0
    minimized = dfa.hopcroft_minimization()
    expected_states = 1
    print(f"Expected: {expected_states} state, Actual: {minimized.n} state")
    assert minimized.n == expected_states, "All-accepting DFA should minimize to 1 state"

    # Already minimal DFA
```

```
    print("Test Case 2: Already Minimal DFA")
    dfa = DFA()
    dfa.n = 2
    dfa.states = [0, 1]
    dfa.accepting = {0: False, 1: True}
    dfa.transition = {
        0: {'a': 1, 'b': 0},
        1: {'a': 1, 'b': 1}
    }
    dfa.start_state = 0
    minimized = dfa.hopcroft_minimization()
    expected_states = 2
    print(f"Expected: {expected_states} states, Actual: {minimized.n} states")
    assert minimized.n == expected_states, "Minimal DFA should remain unchanged"

    # One-State DFA
    print("Test Case 3: One-State DFA")
    dfa = DFA()
    dfa.n = 1
    dfa.states = [0]
    dfa.accepting = {0: True}
    dfa.transition = {
        0: {'a': 0, 'b': 0}
    }
    dfa.start_state = 0
    minimized = dfa.hopcroft_minimization()
    expected_states = 1
    expected_depth = 0  # Start is accepting, no depth needed
    print(f"Expected: {expected_states} states, Actual: {minimized.n} states")
    print(f"Expected Depth: {expected_depth}, Actual Depth: {dfa.bfs_depth()}")
    assert minimized.n == expected_states, "One-state DFA should remain as one
    ↪ state"
    assert dfa.bfs_depth() == expected_depth, "Depth of one-state DFA should be 0"

DFATest.hopcroft_test_cases = hopcroft_test_cases

```

A.4 Test Tarjan Properties Function

```
@staticmethod
def test_tarjan_properties(dfa):
    sccs = dfa.compute_sccs()
    all_nodes = set(dfa.states)
    recovered_nodes = set(node for scc in sccs for node in scc)
```

```python
    # Ensure every node is in some SCC
    assert all_nodes == recovered_nodes, "Not all nodes are in SCCs!"

    # Check strong connectivity within each SCC
    for scc in sccs:
        if len(scc) > 1:
            visited = set()
            stack = [scc[0]]
            while stack:
                node = stack.pop()
                if node not in visited:
                    visited.add(node)
                    for sym in dfa.alphabet:
                        target = dfa.transition[node][sym]
                        # Only traverse within the SCC
                        if target in scc and target not in visited:
                            stack.append(target)
            # All nodes in the SCC should be reachable from any node
            assert visited == set(scc), f"Nodes in SCC {scc} are not strongly connected"
    print("Tarjan's SCC implementation validated successfully.")

DFATest.test_tarjan_properties = test_tarjan_properties
```

A.4.1 Example 4 Function

```python
@staticmethod
def example_4():
    print("--- Example 4 ---")
    dfa = DFA.empty_dfa()
    dfa.states = [0, 1, 2, 3, 4]
    dfa.n = 5
    dfa.accepting = {s: False for s in dfa.states}

    dfa.transition = {
        0: {'a': 2, 'b': 3},
        1: {'a': 0, 'b': 0},
        2: {'a': 1, 'b': 1},
        3: {'a': 4, 'b': 4},
        4: {'a': 4, 'b': 4},
    }
    dfa.start_state = 1

    dot = visualize_dfa(dfa, title="Example 4 DFA")
```

```
    dot.render('example_4', format='png', cleanup=True)
    display(Image('example_4.png'))

    sccs = dfa.compute_sccs()
    print_scc(sccs)
    dot = visualize_dfa_scc(dfa, sccs, title="SCCs of Example 4")
    dot.render('example_4_scc', format='png', cleanup=True)
    display(Image('example_4_scc.png'))
    DFATest.test_tarjan_properties(dfa)

DFATest.example_4 = example_4
DFATest.example_4()
```

A.4.2 Tarjan's Algorithm SCC GeeksforGeeks Source

```
SCCs in first graph
4
3
1 2 0

SCCs in second graph
3
2
1
0

SCCs in third graph
5
3
4
6
2 1 0

SCCs in fourth graph
8 9
7
5 4 6
3 2 1 0
10

SCCs in fifth graph
4 3 2 1 0
```

A.4.3 Tarjan's Algorithm SCC Stanford Source

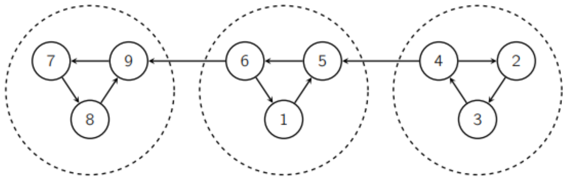


Figure 9: Stanford Example

References

[1] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* 3rd ed., Pearson, 2006.

[2] Moore, E. F. "Gedanken-experiments on Sequential Machines." *Automata Studies*, 1956.

[3] Tarjan, R. E. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing*, vol. 1, no. 2, 1972, pp. 146–160.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms.* 3rd ed., MIT Press, 2009.

[5] University of Malta - RELIC DFA Minimisation Tool. http://www.cs.um.edu.mt/gordon.pace/Research/Software/Relic/Transformations/FSA/minimise.html

[6] RELIC Tool – Remove Unreachable States. http://www.cs.um.edu.mt/gordon.pace/Research/Software/Relic/Transformations/FSA/remove-unreachable.html

[7] RELIC Tool – Remove Useless States. http://www.cs.um.edu.mt/gordon.pace/Research/Software/Relic/Transformations/FSA/remove-useless.html

[8] CS161 Winter 2023 Lecture 10 Notes – Stanford University. https://stanford-cs161.github.io/winter2023/assets/files/lecture10-notes.pdf

[9] GeeksforGeeks – Breadth-First Search (BFS) for a Graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[10] GeeksforGeeks – Minimization of DFA. https://www.geeksforgeeks.org/minimization-of-dfa/

[11] GeeksforGeeks – Tarjan's Algorithm for Strongly Connected Components. https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

[12] Wikipedia – Breadth-First Search. https://en.wikipedia.org/wiki/Breadth-first_search

[13] Wikipedia – DFA Minimization. https://en.wikipedia.org/wiki/DFA_minimization