Data Structures and Algorithms 2 Course Project 2022

Constructing a random DFSA, computing its depth, optimising it and finding its SCCs.

Andre' Vella
32601L
andre.vella.19@um.edu.mt

University of Malta

May 2022

An assignment submitted in partial fulfilment of the requirements for the unit of Data Structures and Algorithms 2 in 2022.

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, the undersigned, declare that the Assigned Practical Task report submitted is my work, except where acknowledged and referenced.

Andre’ Vella

[illegible]

May 20, 2022

Statement of completion

Created a random dfsa.	**Completed**
Correctly computed the depth of the dfsa.	**Completed**
Correctly implemented dfsa minimisation.	**Completed**
Correctly computed the depth of the minimised dfsa.	**Completed**
Correctly implemented Tarjan's algorithm.	**Completed**
Printed number and size of strongly connected components.	**Completed**
Provided a good discussion on Johnson's algorithm.	**Completed**
Included a good evaluation in your report.	**Completed**

Table 1: Statement of completion.

Programming language used for implementation : `Python3`
Extras: Some *unit tests* are written to test and evaluate the work and an *implementation* of Johnson's algorithm is given at the end.

Contents

0

Introduction

We start by formally defining the object we want to explore.

Definition 1 (dfsa). *A deterministic finite state automaton (dfsa) is a 5-tuple* $< K, T, t, k_1, F >$ *where:*

- *K is a finite set of states*
- *T is the alphabet of the input strings*
- *$k_1 \in K$ is the initial state*
- *$F \subseteq S$ is the set of final states*
- *t is the transition function of the automaton where t is of the type $S \times T \to S$*

We can represent a dfsa as a **labelled digraph** as follows:
Consider the following example of dfsa $M =< K, T, k_1, F, t >$ where:
$K = \{1, 2, 3, 4\}$, $T = \{a, b\}$, $k_1 = 2$, $F = \{2, 4\}$,
$t = \{((1, b), 4), ((2, a), 3), ((2, b), 1), ((3, b), 4), ((4, a), 2), ((4, b), 1)\}$
then the automaton M can be represented as a labelled digraph as shown in figure 0.1.

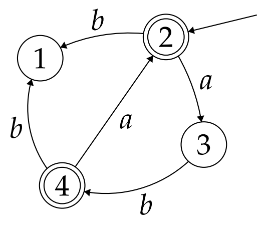


Figure 0.1: representation of M as a labelled digraph

0.1 | Implementing a dfsa in Python

Implementation described in this subsection
can be found in file `dfsa.py` under class `DFSA`.

We represent a dfsa object with attributes described in formal definition 1.

```python
def __init__(self):
    """
    instance attributes
    as given by formal definition of Dfsa
    """
    self.states=set()
    self.alphabet=set()
    self.transitions=set()
    self.accepting_states=set()
    self.start_state=None
```

Listing 1: initialising a dfsa

0.1.1 | Instance methods

To read/modify the state of a dfsa the following instance methods where implemented:

- setters/getters:
  - `def set_states(self,states)` -> where *states* is some iterable
  - `def get_states(self)`
  - `def set_start_state(self,start_state)` -> *start _state* is some state in the set of states
  - `def get_start_state(self)`
  - `def set_accepting_states(self,accepting_states)` -> where *accepting_states* is some iterable whose elements are contained in the set of states of the same dfsa object
  - `def get_accepting_states(self)`
  - `def set_alphabet(self,alphabet)`, where `alphabet` is some iterable containing symbols
  - `def get_alphabet(self)`

  - `def set_transitions(self,transitions)` -> where `transitions` is some iterable contaning tuples of the type $((\mathbf{S_1}, \mathbf{x}), \mathbf{S_2})$ where $S_1,\ S_2$ is in the set of states and x is in the alphabet set of the same instance
  - `def get_transitions(self)`

- methods to add element to a set :
  - `def add_state(self,state)` -> add state to set of states
  - `def add_acccepting_state(self,accepting_state)` -> add accepting state to set of accepting states
  - `def add_transition(self,transition)` -> add transition to set of transitions

- other utilities:
  - `remove_state(self)` -> removes state from dfsa and set some other state as starting state
  - `def display_dfsa(self)` -> displays information about the dfsa such as set of states ,set of accepting states ,starting state ,set of transitions ,alphabet ,number of states ,number of accepting states ,number of transitions.
  - `def plot_dfsa_as_labelled_digraph(self)` -> plot dfsa as a labelled digraph
  - `get_next_state(self,at,letter)` -> returns x where $((at, letter), x) \in transitions$
  - `def get_next_states(self,state)` -> returns set $\{to_s tate : ((state, q), to_s tate)$ for all $q \in alphabet\}$
  - `def get_unreachable_states(self)` -> returns list of states that are unreachable from the starting stat.
  - `def get_shortest_path(self,at_state,to_state)` -> returns the shortest path from two states, `at_state` and `to_state`.
  - `def get_depth(self)` -> returns the maximum out all states $i \in S$ of the length of the shortest string which leads to that state i. Described in Section 2.
  - `init_random(self)` -> randomly generates an automaton based on a recipe described in section 1.1 .

1

Task 1 : Constructing a random dfsa

1.1 | Implementation

Implementation described in this subsection
can be found in file `dfsa.py` under class `DFSA`.

A randomly dfsa is constructed by calling instance method `init_random(self)` which generates a random dfsa based on the following recipe:

- Create n states, where n is a random number between 16 and 64 inclusive:

```
# generate random number n from 16-64
n = random.randint(16, 64)
# generate set of vertices {1,2,...n}
self.set_states([i for i in range(1,n)])
```

Listing 1.1: generating $states = \{1, 2 \ldots, n\}$

- For each state flip a coin ($\equiv random.randint(0,1) == 1$) to determine whether the state is accepting or rejecting

```
# for each state in states  add state if random.randint(0,1)==1 (coin
    flip)
self.set_accepting_states([i for i in range(1,n+1) if random.randint
   (0,1)==1])
```

Listing 1.2: randomly decide if state is accepting or rejecting

- Set the alphabet of the random dfsa to $\{a, b\}$. Define set of transitions as $\forall S_1 \in States$ & $\forall x \in \{a, b\}, \exists S_2 \in States$ such that $((S_1, x), S_2) \in transitions$ where S_2 is randomly assigned:

```python
#the alphabet consists only of symbols a and b
self.set_alphabet(['a','b'])
# traverse all the states
for s1 in self.states:
  # traverse all the symbols
  for x in self.alphabet:
      # the transition leads to a random state
      s2 = random.randint(1,n)
      self.transitions.add(((s1,x),s2))

```

Listing 1.3: every one of the n states must have two outgoing transitions leading to two other random states

- Choose any random state as the starting state of the dfsa:

```python
# randomly choose starting vertex from set of states
self.set_start_state(random.randint(1,n))
```

1.2 | Testing

Unit tests where written to check the validity of the `init_random(self)` function

> The unit tests described in this subsection
> can be found in `tests.py` under the class
> `TestRandomDfsaInit(unittest.TestCase)`

- Initialising a random dfsa to run unit tests on:

```python
random_dfsa_test = Dfsa()
# initialise the dfsa randomly
random_dfsa_test.init_random()
```

Listing 1.4: setup

- Unit Tests:

```python
def test_number_of_states_is_between_16_and_64(self):
    # n is number of states
    n = len(self.random_dfsa_test.get_states())
    check = 16<=n and n<=64
    self.assertTrue(check)
```

Listing 1.5: number of states n is between 16 and 64 inclusive

```python
def test_number_of_states_is_between_16_and_64(self):
    # n is number of states
    n = len(self.random_dfsa_test.get_states())
    check = 16<=n and n<=64
    self.assertTrue(check)
```

Listing 1.6: set of accepting states is a subset of set of states

```python
def test_alphabet_consists_only_of_a_and_b(self):
    self.assertEqual(self.random_dfsa_test.get_alphabet(), set(['a','
   b']))
```

Listing 1.7: alphabet == $\{a, b\}$

```python
def test_dfsa_complete(self):
    isComplete = True
    """
    traverse each state and letter
    if no next state can be found
    the DFSA is not complete
    """
    for state in self.random_dfsa_test.get_states():
        for letter in self.random_dfsa_test.get_alphabet():
            if self.random_dfsa_test.get_next_state(state, letter)==
   None:
                # not complete
                isComplete = False
                break
    self.assertTrue(isComplete)
```

Listing 1.8: dfsa is complete (testing transition function generation correctness)

```python
def test_start_state_in_state(self):
    self.assertTrue(self.random_dfsa_test.get_states().__contains__(
   self.random_dfsa_test.get_start_state()))
```

Listing 1.9: starting state is in the set of states

All unit tests pass ✓ (See section 7)

1.3 | Generating a random dfsa A

Code described this subsection can be found in `main.py`

We are going to generate a dfsa A. For purpose of testing the algorithms described in later sections we are going to fix a seed value of 18. The randomly generated sequence produces a clear example when it comes to evaluating Hopcroft's algorithm and Tarjan's algorithm.

- Generating A:

```python
# fix seed
random.seed(18)
# create empty Dfsa
A = Dfsa();
# init random Dfsa
A.init_random()
# display random Dfsa
print('--------------------------------')
print('DFSA A')
print('--------------------------------')
A.display_dfsa()
print('--------------------------------')
# plot dfsa
A.plot_dfsa_as_labelled_digraph()
```

Listing 1.10: generating random dfsa

1.3.0.1 | Outputs

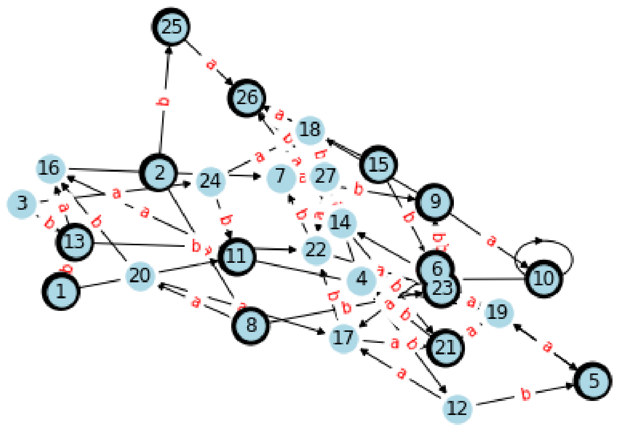
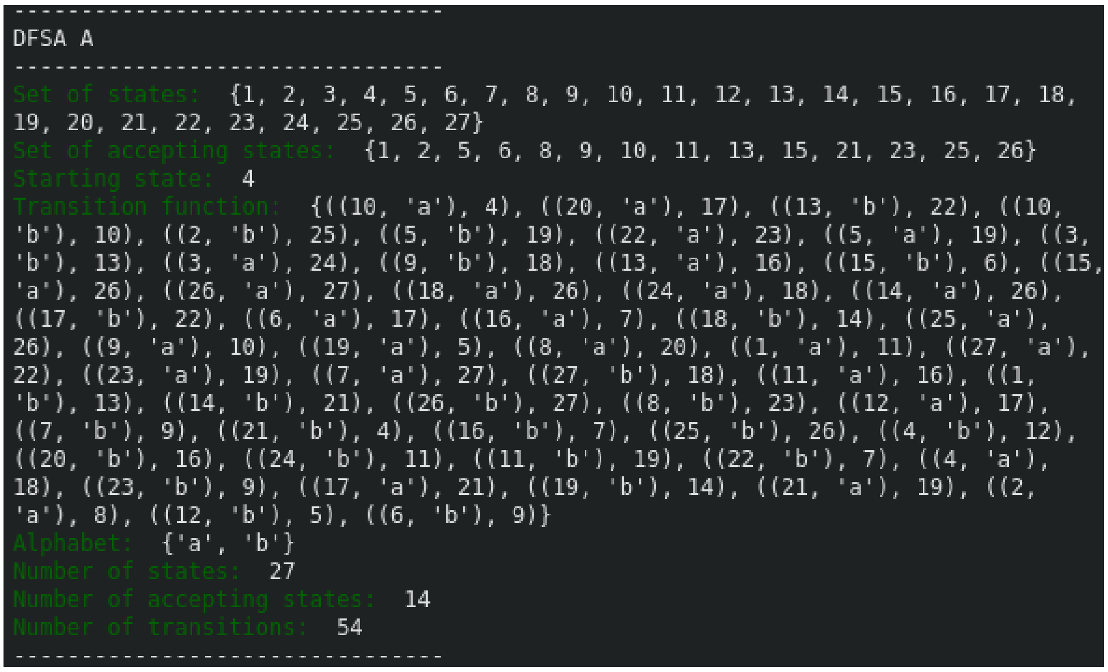


Figure 1.1: plot of A using $A.plot\_dfsa\_as\_labelled\_digraph()$



```
-------------------------------
DFSA A
-------------------------------
Set of states:  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27}
Set of accepting states:  {1, 2, 5, 6, 8, 9, 10, 11, 13, 15, 21, 23, 25, 26}
Starting state:  4
Transition function:  {((10, 'a'), 4), ((20, 'a'), 17), ((13, 'b'), 22), ((10,
'b'), 10), ((2, 'b'), 25), ((5, 'b'), 19), ((22, 'a'), 23), ((5, 'a'), 19), ((3,
'b'), 13), ((3, 'a'), 24), ((9, 'b'), 18), ((13, 'a'), 16), ((15, 'b'), 6), ((15,
'a'), 26), ((26, 'a'), 27), ((18, 'a'), 26), ((24, 'a'), 18), ((14, 'a'), 26),
((17, 'b'), 22), ((6, 'a'), 17), ((16, 'a'), 7), ((18, 'b'), 14), ((25, 'a'),
26), ((9, 'a'), 10), ((19, 'a'), 5), ((8, 'a'), 20), ((1, 'a'), 11), ((27, 'a'),
22), ((23, 'a'), 19), ((7, 'a'), 27), ((27, 'b'), 18), ((11, 'a'), 16), ((1,
'b'), 13), ((14, 'b'), 21), ((26, 'b'), 27), ((8, 'b'), 23), ((12, 'a'), 17),
((7, 'b'), 9), ((21, 'b'), 4), ((16, 'b'), 7), ((25, 'b'), 26), ((4, 'b'), 12),
((20, 'b'), 16), ((24, 'b'), 11), ((11, 'b'), 19), ((22, 'b'), 7), ((4, 'a'),
18), ((23, 'b'), 9), ((17, 'a'), 21), ((19, 'b'), 14), ((21, 'a'), 19), ((2,
'a'), 8), ((12, 'b'), 5), ((6, 'b'), 9)}
Alphabet:  {'a', 'b'}
Number of states:  27
Number of accepting states:  14
Number of transitions:  54
-------------------------------
```

Figure 1.2: output information about A

From the output we deduce that $A =< K, T, t, k1, F >$ where:

- $K = \{1, 2, 3, \ldots, 27\}$
- $T = \{a, b\}$

- $t = \{(6,a) \mapsto 17, (11,b) \mapsto 19, \ldots, (9,a) \mapsto 10\}$
- $k_1 = 4$
- $F = \{1,2,5,6,8,9,10,11,13,15,21,23,25,26\}$

2

Task 2 : Computing depth of a dfsa

2.1 | Implementation

Implementation described in this subsection
can be found in file `dfsa.py` under class `DFSA`.

We define depth of dfsa as follows :

Definition 2 (depth of dfsa)**.** *The depth of any dfsa A is defined to be $max\{len(shortest\ path\ from\ starting\ state\ to\ S | S \in States\ of\ A\}$.*

The shortest path can be obtained using dfsa as follows:

```
def get_depth(self):
    max_ = 0
    for to_state in self.get_states():
        # obtain shortest path by bfs algorithm
        # path -> a list containing vertices in order of path
        path = self.get_shortest_path(self.get_start_state(), to_state)
        # length of path = #vertices - 1
        len_path = len(path)-1
        if max_<len_path:
            max_ = len_path
    return max_
```

Listing 2.1: finding the depth of the dfsa

The main essence of the algorithm is the method
`def get_shortest_path(self,at_state,to_state)`.
To find the shortest path between any 2 states we need to make use of breadth first search which explores nodes on a depth level as a result this allows us to obtain a shortest path.
We show how the algorithm works by giving an example:

- Consider the following dfsa:

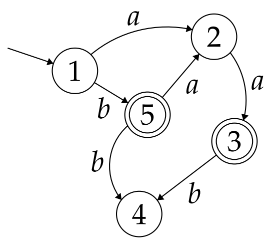


Figure 2.1: dfsa example

  Suppose we want to find the shortest path from 1 to 4.

  The shortest path is evidently of size 2 obtained by traversing string *bb*. We can obtain this by using bfs as follows.

- Initialise a queue : `queue = []`
- Append the state from which the path starts to the queue (in this case 1) : `queue.append(at_state)`
- Construct a list ,`visited` ,of length the number of states with *False* values,`prev = [False]*len(self.get_states())`

False	*False*	*False*	*False*	*False*
State 1	State 2	State 3	State 4	State 5

Table 2.1: initial visited list for this example

- Construct similar list, `prev`, but instead False it has null values, `prev = [None]*len(self.get_states())`

None	*None*	*None*	*None*	*None*
State 1	State 2	State 3	State 4	State 5

Table 2.2: initial prev list for this example

- To go through all the elements we pop of the element from the front of the queue, get its next_states, if the states has not been visited append them to the queue, mark them as visited, and set the prev value of next state to the element popped of out of the queue. Repeat until queue is empty i.e. when all states have been visited.

```
while (len(queue)!=0):
    at_state=queue.pop(0)
    next_states = self.get_next_states(at_state)
    for next in next_states:
        # if not visited
        if visited[next-1]==False:
        # append to queue
        queue.append(next)
        visited[next-1]=True
        prev[next-1]=at_state
    return prev
```

- Running the while loop on the example we get the following iterations:
  - Initial State

1

Table 2.3: queue

False	*False*	*False*	*False*	*False*
State 1	State 2	State 3	State 4	State 5

Table 2.4: visited list

None	*None*	*None*	*None*	*None*
State 1	State 2	State 3	State 4	State 5

Table 2.5: prev list

  - `Iteration 1`

2	5

Table 2.6: queue

False	*True*	*False*	*False*	*True*
State 1	State 2	State 3	State 4	State 5

Table 2.7: visited list

None	*1*	*None*	*None*	*1*
State 1	State 2	State 3	State 4	State 5

Table 2.8: prev list

- `Iteration 2`

5	3

Table 2.9: queue

False	*True*	*True*	*False*	*True*
State 1	State 2	State 3	State 4	State 5

Table 2.10: visited list

None	*1*	*2*	*None*	*1*
State 1	State 2	State 3	State 4	State 5

Table 2.11: prev list

- `Iteration 3`

3	4

Table 2.12: queue

False	*True*	*True*	*True*	*True*
State 1	State 2	State 3	State 4	State 5

Table 2.13: visited list

None	*1*	*2*	*5*	*1*
State 1	State 2	State 3	State 4	State 5

Table 2.14: prev list

- `Last iteration` (queue is empty)

Table 2.15: queue

False	*True*	*True*	*True*	*True*
State 1	State 2	State 3	State 4	State 5

Table 2.16: visited list

None	*1*	*2*	*5*	*1*
State 1	State 2	State 3	State 4	State 5

Table 2.17: prev list

The prev list is important since with it we can construct the shortest path from a given state (usually `start state`) to another given state.

To reconstruct the path we call a method `reconstruct_path(at_state,to_state,from_state)`

Which creates an empty list path traverses the prev list starting from the state the transition leads to at_state, append that state to the list, obtain the previous state from a one-less level depth, append that value to path. Update the at_state to prev value, repeat the process until there is no previous value (i.e. at_state = None).

The list will contain elements of the shortest path from the end state of the path to the states in previous depth leading to start state of the path. If the last element is not the start state then we conclude that the end state is not reachable from the first state. If a path is found then we return the reverse of list. If no path is found then return the empty list.

```python
def reconstruct_path(self,at_state,to_state,prev):
    # set up empty list
    path = []
    # state the previous state to to_state
    prev_state = to_state
    # traverse prev list until Null value is found
    while (prev_state != None):
        # append the previous depth elements
        path.append(prev_state)
        prev_state = prev[prev_state-1]
    #  reverse the elements of the list
    path.reverse()
    # if the first element of the path is the intended
    # starting state then the to state is reachable
    # else return empty list
    if path[0]==at_state:
        return path
    return []
```

Listing 2.2: implementing reconstruction of path from prev list in Python.

NB: We do not specify the state which the transition leads to in the prev list.

2.2 | Testing the DFSA method.

The unit tests can be found in tests.py under the class TestComputeDepthOfDfsa(unittest.TestCase)

For these unit test consider the example shown in figure 2.1 .

The following unit test are written to ensure that the depth of the dfsa algorithm is correctly implemented.

- Initialising example dfsa:

```python
example_dfsa = Dfsa()
example_dfsa.set_states([1,2,3,4,5])
example_dfsa.set_accepting_states([3,5])
example_dfsa.set_start_state(1)
example_dfsa.set_alphabet(['a','b'])
example_dfsa.set_transitions([((1,'a'),2),((1,'b'),5),((2,'a'),3)
    ,((3,'b'),4),((5,'b'),4)])
example_dfsa.display_dfsa()
```

Listing 2.3: set up

```python
def test_expected_path_is_empty(self):
    self.assertEqual(self.example_dfsa.get_shortest_path(2,5) , [])
```

Listing 2.4: state 5 in unreachable from state 2 so we expect an empty path

```python
def test_expected_path(self):
    self.assertEqual(self.example_dfsa.get_shortest_path(1,4) ,
    [1,5,4])
```

Listing 2.5: shortest path from 1 to 4 is $1, 5, 4$ obtained by traversing string bb

```python
def test_expected_depth(self):
    self.assertEqual(self.example_dfsa.get_depth(), 2)
```

Listing 2.6: testing for the expected depth

All unit tests pass ✓ (See section 7)

2.3 | Computing the depth of the random dfsa A.

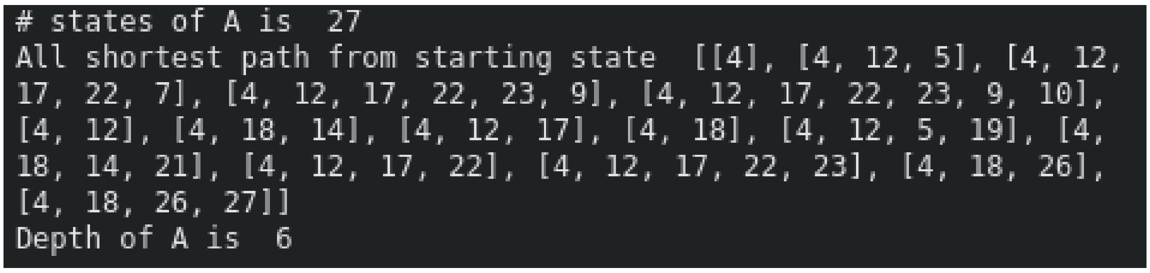
Code described in this subsection can be found in file `main.py`

Consider the randomly generated dfsa A described in section 1.3.
We are going to print the number states of A, all shortest path from the starting state to any other state, and its depth.

```
# get # states of A
print('number states of A is ',len(A.get_states()))
# shortest path ,excluding unreachable states i.e. empty paths
shortest_paths = [A.get_shortest_path(A.get_start_state(), to_state) for
    to_state in A.get_states() if len(A.get_shortest_path(A.
    get_start_state(),to_state))!=0]
print('All shortest path from starting state ',shortest_paths)
# get depth of A
print('Depth of A is ',A.get_depth())
```

Listing 2.7: printing depth of A

2.3.1 | Output.



```
# states of A is  27
All shortest path from starting state  [[4], [4, 12, 5], [4, 12,
17, 22, 7], [4, 12, 17, 22, 23, 9], [4, 12, 17, 22, 23, 9, 10],
[4, 12], [4, 18, 14], [4, 12, 17], [4, 18], [4, 12, 5, 19], [4,
18, 14, 21], [4, 12, 17, 22], [4, 12, 17, 22, 23], [4, 18, 26],
[4, 18, 26, 27]]
Depth of A is  6
```

Figure 2.2: output of Listing 2.7

3 Task 3 : Minimising a dfsa using Hopcroft's Algorithm.

3.1 | Implementation

Implementation described this subsection can be found in file `dfsa.py` under class `HopcroftsAlgorithm` .

Two dfsa are said to be equivalent if they generate the same regular language.
In this section we describe an algorithm used to `optimise/minimise` a dfsa by obtaining an `equivalent` dfsa with less states.
Hopcroft' algorithm is based on partition refinement. For example consider a set of states $\{1, 2, \ldots, n\}$.,we group states together if they possess equivalent behaviour in some form of relation. We make use equivalence class partitioning where the relation is an equivalence relation and we end up with disjoint subsets of equivalent states.
We implement this algorithm by creating a class with 2 instance attributes `old_dfsa` which is passed as a constructor parameter and a `optimised_dfsa` of type `dfsa`. The `optimised_dfsa` is obtained by running hopcroft's algorithm:

```python
def __init__(self,old_dfsa):
    self.old_dfsa = old_dfsa
    self.optimised_dfsa = self.hopcroft_algorithm()
```

Listing 3.1: initialising class

The magic happens in method `def hopcroft_algorihtm()` where we return an `optimised_dfsa` which is equivalent to the old `old_dfsa`.
Before actually merging the `non distinguishable states` we can first remove all those states which are not reachable from the initial state.
Inside method `hopcroft_algorithm(self)` the unreachable states are removed in the following way:

- Create a new `dfsa`:

```python
#initialise empty DFSA
old_dfsa_without_unreachables = Dfsa()
```

Listing 3.2: create an empty dfsa

- Set the same alphabet and the same starting state of the new `dfsa` as the `old_dfsa`:

```python
#set the same alphabet as old
old_dfsa_without_unreachables.set_alphabet(self.old_dfsa.get_alphabet
    ())
```

```
# set the same start state as old
 old_dfsa_without_unreachables.set_start_state(self.old_dfsa.
    get_start_state())
```

Listing 3.3: setting the same alphabet and starting state as *old_dfsa*

- The states of `new_dfsa` are all the states of `old_states` without `unreachable states.`

  The accepting states of the `new_dfsa` are the reachable states of `old_dsa` that are also accepting.

  The transition of the `new_dfsa` are all the transitions in the `old_dfsa` but whose corresponding states are in the newly obtained states that is they are reachable from the starting vertex i.e. $\forall((A_1, k), A_2) \in transitions(new\_dfsa) \implies ((A_1, k), A_2) \in transitions(old\_dfsa)$ and $A_1, A_2 \in states(new\_dfsa)$.

  The `unreachable states` of `old_dfsa` are obtained using method `unreachable_states(self)` as described in section 0.1.1.

```
# get unreachables states of old
unreachables_of_old_dfsa = self.old_dfsa.get_unreachable_states()

# getting reachables from start state in  old_dfsa
reachables_of_old_dfsa = [state for state in self.old_dfsa.get_states
    () if state not in unreachables_of_old_dfsa]

# set new states reachables
old_dfsa_without_unreachables.set_states(reachables_of_old_dfsa)

# getting the accepting reachables from start state in old_dfsa
accepting_reachables_of_old_dfsa = [state for state in self.old_dfsa.
    get_states() if state not in unreachables_of_old_dfsa and state
    in old_dfsa_without_unreachables.get_accepting_states()]

# set accepting states
old_dfsa_without_unreachables.set_accepting_states(
    accepting_reachables_of_old_dfsa)

# getting transitions whose states are in new dfsa
new_transitions = [transition for transition in self.old_dfsa.
    get_transitions() if transition[0][0] not in
    unreachables_of_old_dfsa and transition[1] not in
    unreachables_of_old_dfsa]

# set new transitions
```

```
old_dfsa_without_unreachables.set_transitions(new_transitions)
```

Listing 3.4: setting states & transitions of *new dfsa*

We have obtained a new `dfsa` without the unreachable states. To optimise further one may also remove `dead states` i.e. those states from which no final state is reachable. There are certain scenarios where one would not remove dead states (e.g. for completeness). The definition of a deterministic finite state machine is a machine that accepts/rejects finite strings of symbols, so if a particular dead state is not defined and we cannot define a particular move (in this case we cannot reject).(3)
For this reason and for the fact that it does not really affect Hopcroft's algorithm I did not remove dead states .

Now we move to explain how we can use `Hopcroft's algorithm` to merge (i.e. put in one partition) `nondistinguishable` states i.e. those states that cannot distinguished from one another for any input string (1),(2).
NB: having removed the unreachable states will accelerate this process of merging non distinguishable states. (2)
The algorithm for merging non-distinguishable states is based on partition refinement which may seem counter intuitive since we want to find the largest partitions possible. But by refining we are partition states intro groups where the states in some groups have equivalent behaviour on all input sequences. Two states are equal (i.e. are in the same equivalence class/partition) if they related by the Myhill-Nerode equivalence relation. (2).
2 states are said to be Myhill Nerode Equivalent if there is no string over the respective alphabet which is a distinguishing extension i.e. in the dfsa we are considering $\forall w \in \{a, b\}^*$ (this set of input sequences) and $\forall A_1, A_2 \in Q$ where Q is some equivalence class in set of partitions. The transitions determined by w should take A_1 and A_2 to the same state.
The main idea is to start by partitioning the accepting and non-accepting states since these are clearly different types of states. And for each partition we ask is there some transition which takes the state to some transition. If this is so for each partition then we end up with 4 different sets. We keep on repeating this process until we cannot

refine the partitions anymore. Then we conclude that each element in each set is Myhill Nerode equivalent.
The logic of this algorithm is implemented in Python as follows:

- Start by partitioning the set of states as 2 subsets containing accept and reject states respectively:

```python
# initialise current partition
current = set()

# add accepting states to current partition
current.add(frozenset(old_dfsa_without_unreachables.
    get_accepting_states()))

# add non accepting states to current partition
current.add(frozenset(rejecting_states))

# define an empty set
partitions = set()
```

Listing 3.5: initial partition

- We want to continue refining the partitions until we cannot partition any more. `partitions` keep track of the previous partition:

```python
while current != partitions:
```

Listing 3.6: loop definition

- We refine the `current` partitions (inside `while loop`) by partitioning the sets inside the current partition into equivalent states until we cannot anymore.

```python
# for each of the current partition
# return a possible partition of that partition
for partition in partitions:
     current = current | self.split(partition, partitions,
    old_dfsa_without_unreachables)
# if the new partition is the same as previous partition
# all states in each partition are Myhill Nerode Equivalent
# stop the algorithm
```

Listing 3.7: body of while loop

The power of partitioning each partition relies on method
`split(partition, partitions, old_dfsa_without_unreachables)`

Method `split` refines a partition as follows:

- Transform the set of partition in a list:

```
# transform set of partitions into a list
partition_list = list(partitions)
```

Listing 3.8: list of partitions

  Initially `partition_list` will just contain 2 element one being the `frozenset` containing the **accepting states** and another `frozenset` containing **rejecting states**.

- We are going to create a list to store the refinements that contains n empty sets where n is the numbers of partitions (initially $n = 2$,the accepting and rejecting states).

  We are going to get the index of the partition in consideration from the `partition_list` which will be used in the logic yet to be described:

```
#a new list to store refinments
refinement_list = [frozenset() for i in range(len(partition_list))]

# get index where the partition to iterate over lies in in
    partition_list
index_current_partition = partition_list.index(partition)
```

Listing 3.9: initialising list with empty sets and obtaining index of partition in consideration from the *partition_list*

- For each input and state in partition that we are trying to refine ,we ask; does the states in lead to states that are in different partitions when given some input? (the leading states are obtained using `get_next_state(self,state,letter)` as described in section 0.1.1)

  We refine the partition based on the partition they correspond to. Now if for a input we manage to re-partition a partition then we just return that partition else we try to re-partition a partition for other inputs.

  The partition is not refined if for all inputs the states leads to the some unique partition in the partition list.

```
# traverse over the letters
for letter in dfsa.get_alphabet():
    # traverse over the state of the partition we want to refine
    for state in partition:
        # get next state
        leading_state = dfsa.get_next_state(state, letter)

        '''
        if next state is none
        than the state remains in the same
        partition
        '''
        if leading_state is None:
            # next state does not exist
            # current state remains in same partition
            refinement_list[index_current_partition]=refinement_list[index_current_partition] | frozenset([state])

        # next states exists
        else:
        # if next state exists
        # find out in which partition leadingstate lies
            for partition in partition_list:
                # if next state lies in paritiion
                if leading_state in partition:
                    index = partition_list.index(partition)
                    #put the element in the set corresponding to that partition
                refinement_list[index] = refinement_list[index] | {state}

    # find  a length of some non empty set in the refinement list
    len_of_some_non_empty_refinment = None
    for refinement in refinement_list:
        if len(refinement)!=0:
            len_of_some_non_empty_refinment =len(refinement)
            break

    # if the len is not equal to the amount of elements of the partition
    # we have successfully refined the partition return the refinement
    if len_of_some_non_empty_refinment<len(partition_list[index_current_partition]):
        return {i for i in refinement_list if i!=frozenset()}
        # reset refinement list and try again with next input
        refinement_list = [frozenset() for i in range(len(partition_list))]
# no refinement occurs
return {partition}
```

Listing 3.10: refining a partition

- Going back inside `hopcrofts_algorithm(self)` method we construct the new `optimised` dfsa as follows:
  - Obtain a `partition` list containing of `frozensets` partitions of equivalent states:

```python
# get the refinement consisting of equivalent states
partitions = list(current)
```

Listing 3.11: refinements

  - Create a new dfsa. Set its alphabet as previous. Each partition will be merged into one state ∴ the new dfsa has states $1, 2, \ldots, n$ where n is the number of partitions:

```python
# create new dfsa
optimised_dfsa = Dfsa()
# set alphabet as previous
optimised_dfsa.set_alphabet(old_dfsa_without_unreachables.
    get_alphabet())
# set states
optimised_dfsa.set_states([i for i in range(1,len([partitions])
    +1)])
```

Listing 3.12: new states

The accepting states, starting state and transitions are obtained by matching the index of the partition in `partition_list` to the states generated:

```python
# KEEP TRACK OF STATES (INDEX IN PARTITION LIST)
state = 1

for partition in partitions:
    # check if an element in partition is accepting
    if list(partition)[0] in old_dfsa_without_unreachables.
    get_accepting_states() :
        # add the state corresponding to the partition
        optimised_dfsa.add_accepting_state(state)

    if old_dfsa_without_unreachables.get_start_state() in
    partition:
        optimised_dfsa.set_start_state(state)


    # build the new set of transitions
    # based on the partitions in lists
    for letter in  old_dfsa_without_unreachables.get_alphabet():
        next_state_old = old_dfsa_without_unreachables.get_next_state(list(partition)[0],letter)
        # find in which partition next state lies
        next_state_new = 1
        for partition in partitions:
            if next_state_old in partition:
                optimised_dfsa.add_transition(((state,letter),next_state_new))
                break
            next_state_new+=1

    # move in to the next partition
    # -> move in to the next state representing that partition
    state+=1

# constructing the dfsa from the refinements
return optimised_dfsa
```

Listing 3.13: setting accepting states , starting state, and transitions of the new optimised dfsa

3.2 | Testing the refinements

The unit tests described in this subsection can be found in `test.py` under class `TestHopcroftAlgorithm` .

To test the algorithm we are going to test how a partition is refined based on some given partitions.
We are going to test 2 cases one where the *partition is actually refined* and one where the *partition is not refined.* Consider the following example dfsa K (1)

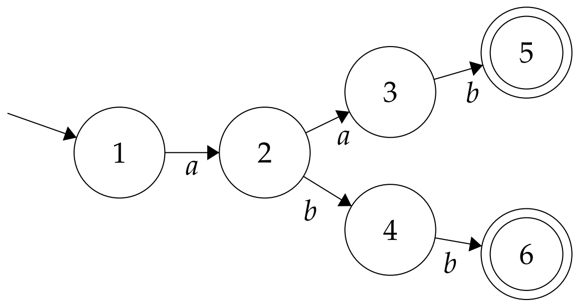


Figure 3.1: dfsa K

The initial partitions of K that we would like to refine are $\{\{1,2,3,4\},\{5,6\}\}$ (i.e. the accept and non accept states).
If we try to refine $\{1,2,3,4\}$ we have that on input a:

- State 1 goes to State 2 which is in the same partition i.e. State 1 remains in its own partition.
- State 2 goes to State 3 which is in the same partition i.e. State 2 remains in its own partition.
- State 3 goes nowhere i.e. State 3 remains in its own partition.
- State 4 goes nowhere i.e. State 3 remains in its own partition.

On input a no refinement occurs.
Let us try to see what happens on input b.

- State $2,3$ goes in the second partition (i.e. $5,6$) and $1,4$ do not.

Hence partition $\{1,2,3,4\}$ should be refined to $\{\{1,2\},\{3,4\}\}$
This is written as a unit test as follows:

- Setup $DFSA$ for testing.

```
example_dfsa = Dfsa()
example_dfsa.set_states([1,2,3,4,5,6])
example_dfsa.set_accepting_states([5,6])
example_dfsa.set_alphabet(['a','b'])
example_dfsa.set_start_state(1)
example_dfsa.set_transitions([((1,'a'),2),((2,'a'),3),((2,'b'),4)
    ,((3,'b'),5),((4,'b'),6)])
```

Listing 3.14: test dfsa

- Unit test for the refinement explained above

```
def test_split_refinement_success(self):
    refinement = HopcroftsAlgorithm(self.example_dfsa).split(
    frozenset([1,2,3,4]),set([frozenset([1,2,3,4]),frozenset([5,6])])
    ,self.example_dfsa)
    self.assertEqual(refinement,set([frozenset([1,2]),frozenset
    ([3,4])]))
```

Listing 3.15: testing for expected refinement

- Now trying refine partition $\{5,6\}$ in partition list $\{\{1,2,3,4\},\{5,6\}\}$ we get that on input a and b they lead to nowhere so they remain in the same partition i.e. no refinement occurs for $\{5,6\}$ so we expect that `split` returns $\{\{5,6\}\}$
- Unit test for when no refinement occurs.

```python
def test_split_no_refinement_occurs(self):
    refinement = HopcroftsAlgorithm(self.example_dfsa).split(
   frozenset([5,6]),set([frozenset([1,2,3,4]),frozenset([5,6])]),
   self.example_dfsa)
    self.assertEqual(refinement,set([frozenset([5,6])]))
```

Listing 3.16: unit test for a partition the is not refined further

All unit tests pass ✓ (See section 7)

3.3 | Obtaining a minimised dfsa M from random dfsa A

The code of this subsection can be found in file `main.py`

Consider the random A obtained in section 1.3.
To optimise A, initialise *HopcroftsAlgorithm* and obtain the optimised the optimised dfsa by calling the getter method `get_optimised_dfsa(self)`

Listing 3.17: obtaining optimised dfsa

3.3.1 | Comparing the old dfsa to the optimised dfsa

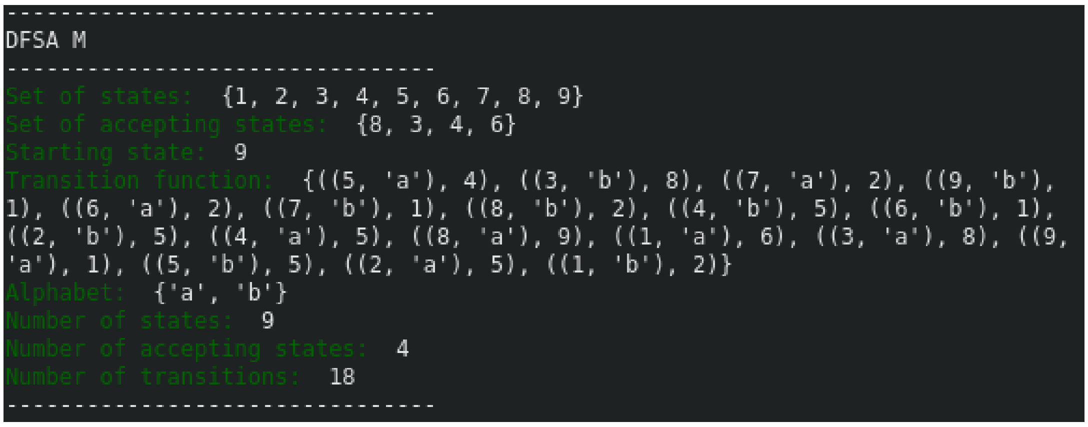
- The new dfsa M is obtained by initialising the `HopCroftsAlgorithm` instance with dfsa A. Information and a plot of M is obtained using methods `display_dfsa` and `plot_dfsa_as_labelled_digraph` (see section 0.1.1) respectively.

```python
print('---------------------------------')
print('DFSA M')
print('---------------------------------')
M = HopcroftsAlgorithm(A).get_optimised_dfsa()
# obtaining info about optimised DFSA
M.display_dfsa()
print('---------------------------------')
```

```
# plotting DFSA
M.plot_dfsa_as_labelled_digraph()
```

Listing 3.18: obtaining information about M

3.3.2 | Output



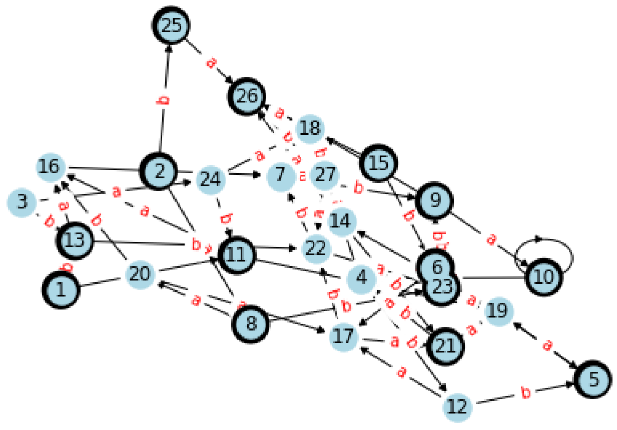
```
--------------------------------
DFSA M
--------------------------------
Set of states:  {1, 2, 3, 4, 5, 6, 7, 8, 9}
Set of accepting states:  {8, 3, 4, 6}
Starting state:  9
Transition function:  {((5, 'a'), 4), ((3, 'b'), 8), ((7, 'a'), 2), ((9, 'b'),
1), ((6, 'a'), 2), ((7, 'b'), 1), ((8, 'b'), 2), ((4, 'b'), 5), ((6, 'b'), 1),
((2, 'b'), 5), ((4, 'a'), 5), ((8, 'a'), 9), ((1, 'a'), 6), ((3, 'a'), 8), ((9,
'a'), 1), ((5, 'b'), 5), ((2, 'a'), 5), ((1, 'b'), 2)}
Alphabet:  {'a', 'b'}
Number of states:  9
Number of accepting states:  4
Number of transitions:  18
--------------------------------
```

Figure 3.2: information about optimised dfsa M

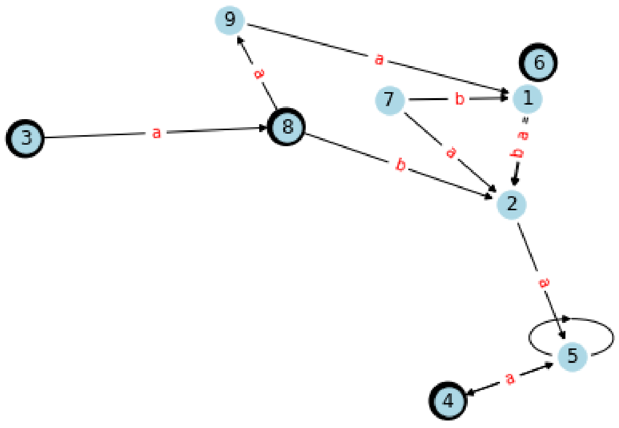
From the output we deduce that for the M the optimisation of A we have that $M =< K', T', t', k1', F' >$ where:

- $K' = \{1, 2, 3, \ldots, 9\}$
- $T' = \{a, b\}$
- $t' = \{(5, a) \mapsto 4, (3, b) \mapsto 8, \ldots, (1, b) \mapsto 2\}$
- $k1' = 4$
- $F' = \{3, 4, 6, 8\}$

The optimised dfsa M has 9 states whereas A has 27 states.
To picture how the complexity has reduced we can draw both dfsa using `plot_dfsa_as_labelled_digraph`



((a)) plot of A



((b)) plot of optimisation of M

4

Task 4 : Obtaining the depth of M

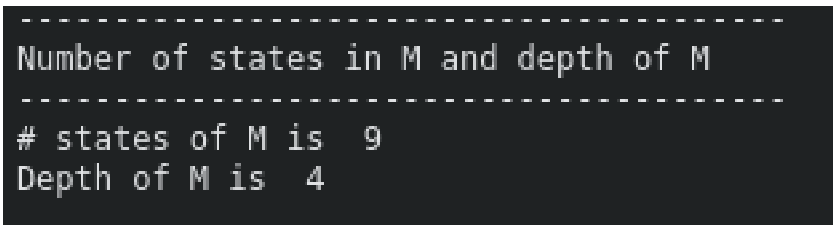
The code of described in this sub-section can be found in `main.py`.

The depth of M can be obtained equivalently as obtained for A in section **??**

```
print('---------------------------------------')
print('Number of states in M and depth of M')
print('---------------------------------------')
# get # states of M
print('# states of M is ',len(M.get_states()))
# get depth of M
print('Depth of M is ',M.get_depth())
```

Listing 4.1: obtaining depth of M

4.1 | Output



```
-----------------------------------------
Number of states in M and depth of M
-----------------------------------------
# states of M is  9
Depth of M is  4
```

Figure 4.1: depth of M

NB: The depth of A was 6. It is expected for the depth to get smaller since when reducing the number of states the possibility of having bigger pat from the starting state is reduced.

5

Task 5 : Finding SCCs using *Tarjan's Algorithm*

We need to find out the number of strongly connected components (SCCs) in M together with the size of the smallest and largest SCCs in M.
SCCs are defined as follows.

Definition 3 (strongly connected component)**.** *A directed is **strongly connected** if there is a path in each direction between each pair of vertices of the graph. Equivalently, a strongly connected component of a directed graph G is a maximal subgraph that is strongly connected that is no additional edges or vertices from G can be included in the subgraph without breaking its property of being strongly connected.*

The class `TarjansAlgorithm` is implemented with the following instance attributes:

- `self.strongly_connected_components` -> a list of lists of vertices in SCCs
- `self.number_of_sccs` -> the number of SCCs in a dfsa
- `self.largest_scc` -> a list of vertices of some largest strongly connected components (not necessarily unique).
- `self.number_of_states_in_largest_scc` -> number of states in a largest strongly connected component
- `self.smallest_scc` -> a list of vertices of some smallest strongly connected component (not necessarily unique)

- `self.number_of_states_in_smallest_scc` -> number of states in a smallest strongly connected component
- Some other variables useful for the algorithm :
  - `self.undefined = -1` -> using -1 to indicate undefined
  - `self.list_of_states = list(self.dfsa.get_states())` -> obtain an ordered list of states so we can use it to relate low link and index value of the states
  - `self.indices = [self.undefined]*len(self.list_of_states)` -> used to store indices of state where the indices number the states in order that they are discovered.
  - `self.low_link = [self.undefined]*len(self.list_of_states)` -> list of low link values of states where low link of a state *S* represents the smallest index of any state on the stack known to be reachable from the state *S*.

5.1 | Implementation

In the `init` method of the class we need to run the method `strongconnect` on each state and report any strongly connected component of that subgraph.

```
for state in self.list_of_states:
    if self.indices[self.list_of_states.index(state)]==self.undefined:
        self.strongconnect(state)
```

Listing 5.1: reporting back any SCC for each state

```
def strongconnect(self,state):
    index_at = self.list_of_states.index(state)
    # Set the depth index for v to the smallest unused index
    self.indices[index_at] = self.index
    self.low_link[index_at] = self.index
    self.index = self.index + 1
    self.S.append(state)
    self.onStack[index_at] = True
    for next_state in self.dfsa.get_next_states(state):
        index_to = self.list_of_states.index(next_state)
        if self.indices[index_to]==self.undefined:
    # Successor w has not yet been visited;
    # recurse on it
            self.strongconnect(next_state)
            self.low_link[index_at] = min(self.low_link[index_to],self.low_link[index_at])
        elif self.onStack[index_to]:
            self.low_link[index_at] = min(self.low_link[index_to],self.low_link[index_at])

    # If v is a root state, pop the stack and generate an SCC

    if self.low_link[index_at] ==  self.indices[index_at]:
        scc = set()
        # start a new strongly connected component

        while True:
            w = self.S.pop()
            index_w = self.list_of_states.index(w)
            self.onStack[index_w] = False
            scc.add(w)
            if w == state:
                break
        self.strongly_connected_components.append(scc)
```

Listing 5.2: strongconnect method

5.2 | Testing

The unit tests described in this subsection can be found in `tests.py` under the class `TestTarjanAlgorithm` .

Consider the follwing example (4):

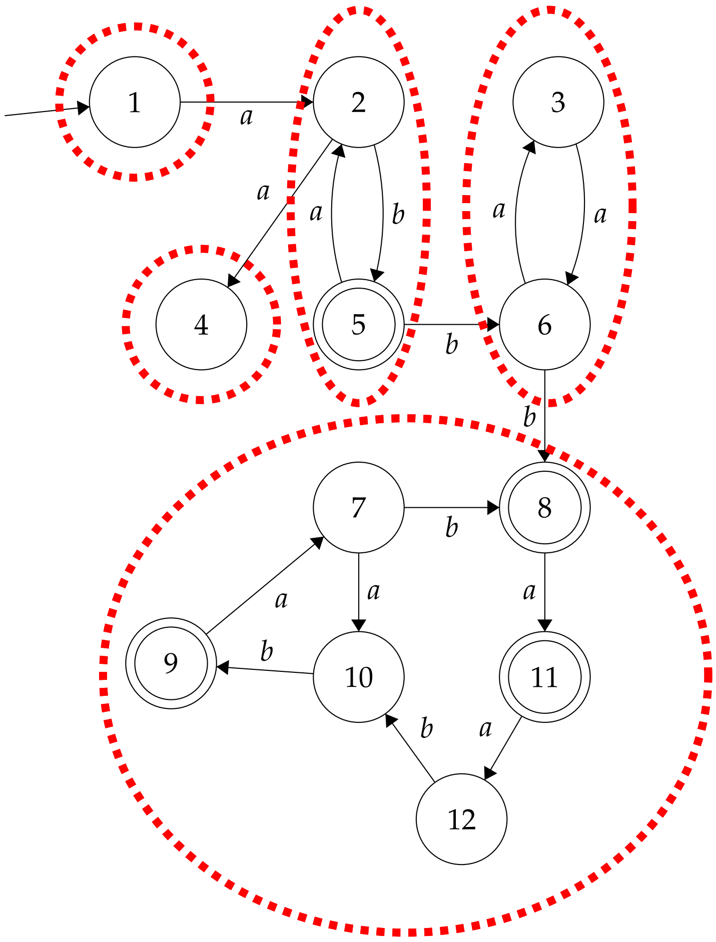


Figure 5.1: dfsa and its strongly connected components

The dfsa shown in figure 5.1. We have 5 SCCs i.e. if we add another state to an SCC it looses is Strongly Connected characteristic.
Unit testing Tarjan's Algorithm returns the expected result :

- Setting up the example dfsa as shown in figure 5.1

```
example_dfsa = Dfsa()
example_dfsa.set_states([1,2,3,4,5,6,7,8,9,10,11,12])
example_dfsa.set_accepting_states([5,8,9,11])
example_dfsa.set_alphabet(['a','b'])
example_dfsa.set_start_state(1)
example_dfsa.set_transitions([((1,'a'),2),((2,'a'),4),((2,'b'),5)
    ,((3,'a'),6),((5,'a'),2),((5,'b'),6),((6,'a'),3),((6,'b'),8),((7,
    'a'),10),((7,'b'),8),((8,'a'),11),((9,'a'),7),((10,'b'),9),((11,'
    a'),12),((12,'b'),10)])
```

```
TarjanResults = TarjansAlgorithm(self.example_dfsa)
```

Listing 5.3: setup example dfsa

- Thee expected SCCs in this example are $\{1\}, \{4\}, \{2,5\}, \{3,6\}, \{7,8,9,10,11,12\}$,

```
def test_expected_sccs(self):
    sccs = self.TarjanResults.get_sccs()
    check = True
    for scc in [set([1]),set([4]),set([2,5]),set([3,6]),set
    ([7,8,9,10,11,12])]:
        if scc not in sccs:
            check = False
            break
    self.assertTrue(check)
```

Listing 5.4: unit tests for expected SCCs

- The expected largest SCC in this example is given by states $\{7,8,9,10,11,12\}$.

```
 def test_get_largest_scc(self):
    self.assertEqual(set([7,8,9,10,11,12]),TarjansAlgorithm(self.
   example_dfsa).get_largest_scc())
```

Listing 5.5: expected largest SCCs

- The expected smallest SCC in this example is given by states $\{1\}$ or $\{4\}$:

```
def test_get_smallest_scc(self):
    self.assertTrue(TarjansAlgorithm(self.example_dfsa).
   get_smallest_scc() in [set([1]),set([4])])
```

Listing 5.6: expected smallest SCCs

- Expected size of a largest SCC is 6:

```
def test_size_largest_scc(self):
    self.assertEqual(self.TarjanResults.
   get_number_of_states_in_largest_scc() , 6)
```

Listing 5.7: expected length of a largest SCC

- Expected size of a smallest SCC is 1

```
def test_size_smallest_scc(self):
        self.assertEqual(self.TarjanResults.
   get_number_of_states_in_smallest_scc(), 1)
```

Listing 5.8: Unit test for size of smallest SCC

All unit tests pass ✓ (See section 7)

5.3 | Obtaining SCCs of optimised dfsa M

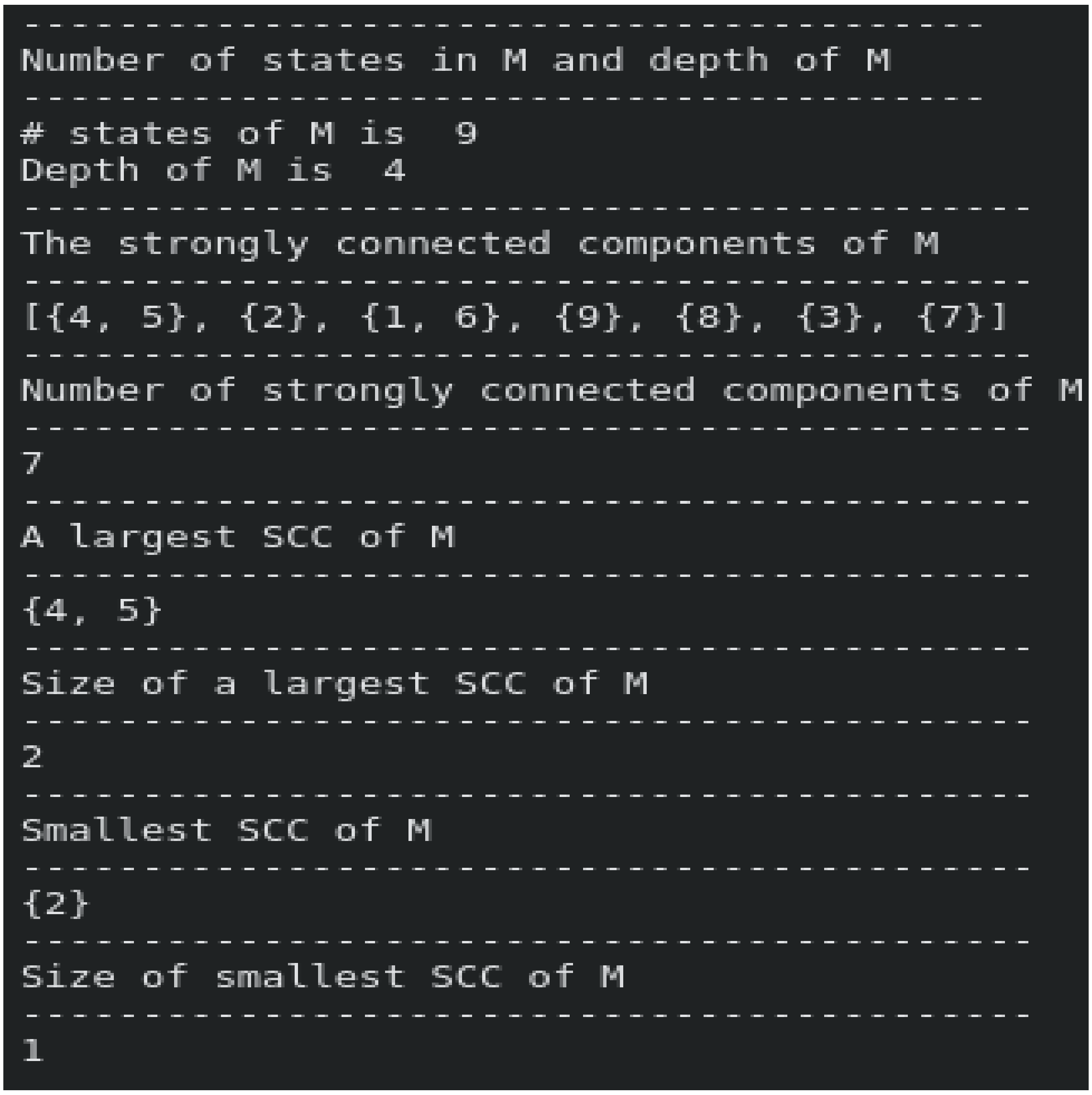
The code of this subsection can be found in `main.py`

Consider the dfsa M generated in section 3.3.
We are going to obtain: The list of sets of vertices in each SCC in M, the number of SCCs in M, a largest SCC in M, the number of states in a largest SCC in M, a smallest SCC in M, the number of states in a smallest SCC in M.

```python
print('------------------------------------------')
print('The strongly connected components of M')
print('------------------------------------------')
print(TarjansAlgorithm(M).get_sccs())
print('------------------------------------------')
print('Number of strongly connected components of M')
print('------------------------------------------')
print(len(TarjansAlgorithm(M).get_sccs()))
print('------------------------------------------')
print('A largest SCC of M')
print('------------------------------------------')
print(TarjansAlgorithm(M).get_largest_scc())
print('------------------------------------------')
print('Size of a largest SCC of M')
print('------------------------------------------')
print(len(TarjansAlgorithm(M).get_largest_scc()))
print('------------------------------------------')
print('A smallest SCC of M')
print('------------------------------------------')
print(TarjansAlgorithm(M).get_smallest_scc())
print('------------------------------------------')
print('Size of a smallest SCC of M')
print('------------------------------------------')
print(len(TarjansAlgorithm(M).get_smallest_scc()))
```

Listing 5.9: info about SCCs of M

5.3.1 | Output



```
----------------------------------------
Number of states in M and depth of M
----------------------------------------
# states of M is  9
Depth of M is  4
------------------------------------------
The strongly connected components of M
------------------------------------------
[{4, 5}, {2}, {1, 6}, {9}, {8}, {3}, {7}]
------------------------------------------
Number of strongly connected components of M
------------------------------------------
7
------------------------------------------
A largest SCC of M
------------------------------------------
{4, 5}
------------------------------------------
Size of a largest SCC of M
------------------------------------------
2
------------------------------------------
Smallest SCC of M
------------------------------------------
{2}
------------------------------------------
Size of smallest SCC of M
------------------------------------------
1
```

Figure 5.2: the output of Listing 5.9

5.3.2 | A plot of M and its SCCs.

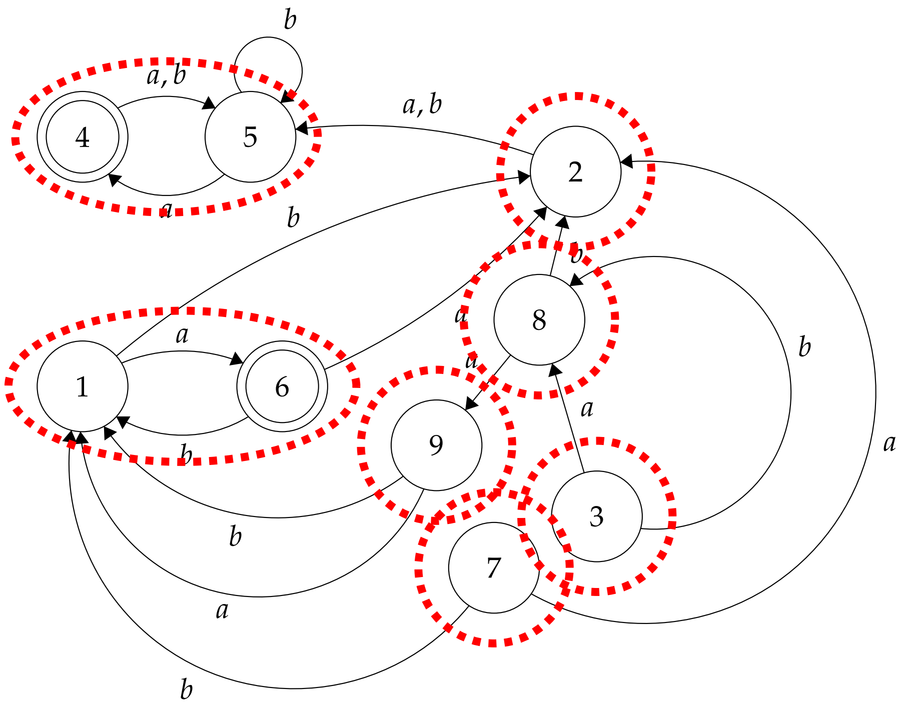


Figure 5.3: M and its strongly connected components

6

Task 6 : Discussion and implementation of Johnson's Algorithm

Johnson's algorithm is used to find all the elementary/simple cycles of any directed graph (5).

A simple cycle in a directed graph is defined as follows:

Definition 4 (Simple cycle in a directed graph)**.** *A cycle is a path in the graph such that the first and last vertex are the same. A simple cycle is a path in the graph such that no vertex is the same except for the first and last vertices.*

In a simple cycle there is a path between any vertex this implies that a simple cycle is strictly contained in one and only one SCC.
This can be proved using by means of contradiction.

Proof. Consider a simple cycle C in G.
Suppose for contradiction some 2 vertices X, Y in a simple cycle in some digraph G lies in different SCC S_1 and S_2 in G respectively. If we add the vertex Y to S_1 then we have from path going to X to Y and from Y to X. We have a larger SCC containing all the vertices in S_1 i.e. S_1 is not maximal hence it is not a SCC.
$\therefore$ Any vertex in a simple cycle must lie in exactly one strongly connected component. $\square$

Since cycles only exists in SCCs we can find them first. The strongly connected components can be obtained using some algorithm such as Tarjan's as explained in section 5.
In order to obtain the simple cycles we will make use of 3 data structures:

- Stack
- Blocked Set
- Blocked Map

From each SCC we pick a starting vertex (we generally start with the vertex with the least id) and then we try to find all simple cycles that start and end with the starting vertex by means of a DFS (i.e. using the idea of back tracking)
We start exploring neighbors of vertices in the following way:

- If a neighbour is not in the blocked set then we can explore that neighbour by putting it in the stack and blocked set and exploring its neighbours.
- If neighbour is the same as starting vertex then we have found a cycle.
- If a vertex is in the block set then we cannot explore that vertex since it has already been visited and what we are looking for are simple cycles i.e. cycles without repeated vertices.

- If a vertex S has all its neighbours in the blocked set we recurse back from that S and remove it from the stack but not from the blocked set since the way that the blocked vertices are set up is that if the current explored cycle has the neighbours of S already visited then we cannot go back to S since S will go back to its neighbours and the we cannot obtain a simple cycle. We keep a mapping $S_{neighbour} \mapsto S$ to indicate that when its neighbours can get freed (i.e. unblocked) then we can go back to S. We keep the mappings in a blocked map data structure.
- If a state S is done exploring all its neighbors and it is in a cycle that has been found then this opens up the possibility that it is present in other cycle hence we remove it from the blocked set and remove the corresponding vertices using maps inside blocked map.
- Finally set the starting vertex to another vertex to obtain cycles with different start/end vertex.

Let us see how the logic of the algorithm can be used to obtain all simple cycles in a SCC of a directed graph starting and ending with some vertex through an example.

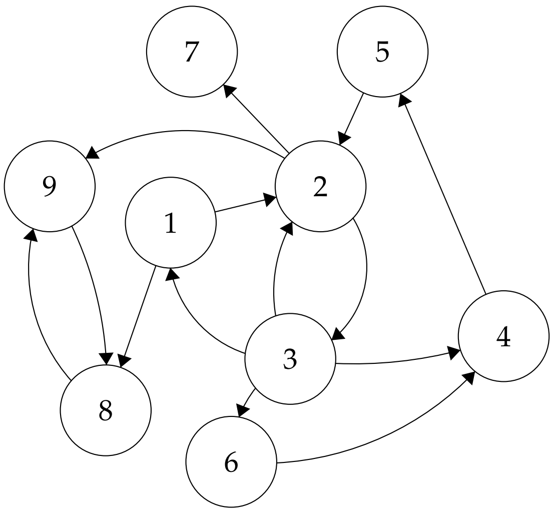


Figure 6.1: example directed graph

Consider the directed graph in 6.1. In order to find the simple cycles we need to first apply Tarjans Algorithms to find the SCCs of the digraph. Applying Tarjan's algorithm we find that the SCCs of this digraph are given by set of states $\{1, 2, 3, 4, 5, 6\}$, $\{7\}$ and $\{8, 9\}$.
Consider the SCC given by $\{1, 2, 3, 4, 5, 6\}$ and let us start Johnson's algorithm by finding all simple cycles starting and ending with 1.

Initially we have that $stack = [], blocked - set = \{\}, blocked - map = \{\}$. We start our exploration with 1 so we put 1 on the stack i.e. $stack = \{1\}$ and we mark 1 as visited $blocked - set = \{1\}$.
We then obtain the neighbours of 1 and start exploring them one by one. We start exploring the neighbour 2. Since $2 \notin blocked - set$ then we mark the $stack$ and the $blocked - set$ as $\{1, 2\}$.
Then we explore the only neighbour 3 of 2. Since $3 \notin blocked - set$ then we mark the $stack$ and the $blocked - set$ as $\{1, 2, 3\}$. Explore neighbours of 3. We start from 1. We have that $1 \in blocked - states$. Since $1 == starting - vertex$ we have found a cycle. Then we go back and explore other neighbors of 3. So we try to explore 2 but $2 \in blocked - set$. So we try to explore another neighbour of 3. Another neighbor that we can explore is 4. Since $3 \notin blocked - set$ then we mark the $stack$ and the $blocked - set$ as $\{1, 2, 3, 4\}$. We try to explore neighbours of 4. We explore the only neighbour of 4 is 5. Since $5 \notin blocked - set$ then we mark the $stack$ and the $blocked - set$ as $\{1, 2, 3, 4, 5\}$. We try to explore neighbours of 5.
The only neighbour of 5 is 2. Since $2 \in blocked - set$. We are now stuck since $2! = starting - vertex$ and 5 has no other neighbour unblocked.
So we need to recurse back from 5 i.e. remove 5 from the stack and now we have that the stack becomes $\{1, 2, 3, 4\}$.
But as of now we cannot remove 5 from the $blocked - set$ since the path that we are considering still goes from 2 and we cannot have that a path goes to 2 and then goes to 5 and then we go back to 2 i.e. we have repeated vertices which is not what we want. But we need to keep in store that if transition 2 ever gets unblocked then we can unblock 5 therefore we add the mapping $2 \mapsto 5$ to $blocked - map$ i.e. $blocked - map = [(2, 5)]$.
So from 5 we go back to 4. The only neighbour of 4 is 5. $5 \in blocked - set$. Since we have that all neighbours of 4 are blocked we remove 4 from states that is $states = \{1, 2, 3\}$ and add the required mappings to $blocked - map$ which now becomes $\{(2, 5), (5, 4)\}$.
Now we go back to 3. 3 has one more neighbour which is 6. Since $6 \notin blocked - set$ we now have that $states = \{1, 2, 3, 6\}$ and $blocked - set = \{1, 2, 3, 4, 5, 6\}$.
All the neighbours of 6 are $blocked$ so we add the required mappings to $blocked - set$ that is$blocked - set = \{(2, 5), (5, 4), (4, 6)\}$ and we remove 6 from $states$.
Since neighbours of 3 has been explored and $3 \in$ some cycle. So we can remove 3 from the stack and unblock it and we also check if we can unblock other states by looking at the mappings in $blocked - states$. The data structures now becomes $states = \{1, 2, 6\}$ and $blocked - set = \{1, 2, 4, 5, 6\}$ and $blocked - set = \{(2, 5), (5, 4), (4, 6)\}$.

Since 2 has no more neighbours that we can check and $2 \in$ some cycle. Then we remove 2 from the stack and unblock it and we also check if we can unblock other states by looking at the mappings in $blocked - states$. The data structures now becomes $states = \{1,6\}$ and $blocked - set = \{1\}$ and $blocked - set = \{(4)\}$.
Now we repeat the process with next neighbours of 1 until 1 has no more neighbours to be explored.

6.1 | Analysing time complexity of the algorithm.

The worst possible time to find one cycle is given by $O(V + E)$. If an SCC has exponential number of cycles then the time complexity of the whole algorithm can get very big.
The space complexity of the algorithm is given by $O(V + E)$.

6.2 | Implementation of Johnson's Algorithm to obtain simple cycles of M

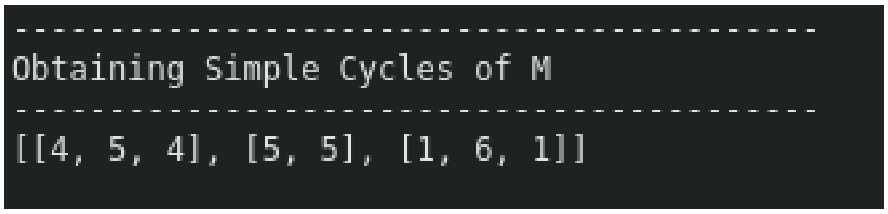
```python
class JohnsonsAlgorithm:
    def __init__(self,dfsa):
        self.dfsa = dfsa
        self.simple_cycles = []
        self.stack = []

        self.blocked_set = set([])
        self.blocked_map = set([])
        self.johnsons_algorithm()


    def get_simple_cycles(self):
        return self.simple_cycles

    def johnsons_algorithm(self):

        temp_dfsa = copy.deepcopy(self.dfsa)
        sccs = TarjansAlgorithm(self.dfsa).get_sccs()
        # going through all states 1,2,3,4,5,...,n
        for scc in sccs:
            for state in scc:

                # find cycles in scc  with starting-ending state  = state
                self.find_cycles_in_scc(scc,state,state)
                # clear contents of stack for next iteration
                self.stack = []
                # clear contents of blocked_set for next iteration
                self.blocked_set = set([])
                # keeps a map of states that can be freed if some state is freed
                self.blocked_map = set([])
                # remove state frome dfsa so it would not be included in next cycle
                self.dfsa.remove_state(state)

        self.dfsa = temp_dfsa



    def find_cycles_in_scc(self,scc,start_state,current_state):
        found_cycle = False
        self.stack.append(current_state)
        self.blocked_set.add(current_state)
        for neighbour in self.dfsa.get_next_states(current_state):
            if neighbour == start_state:
                self.stack.append(start_state)
                cycle =  [state for state in self.stack]
                cycle.reverse()
                if  cycle not in self.simple_cycles:
                    self.simple_cycles.append(cycle)
                self.stack.pop()
                found_cycle = True

            # else if neighbour is not start state and not in block_set
            elif  neighbour not in self.blocked_set:
                # got_cycle is true if neighbour find cycle in its path
                got_cycle = self.find_cycles_in_scc(scc,start_state,neighbour)
                #  if found cycle is true it will be true for current vertex
                found_cycle = found_cycle or got_cycle

        # if found cycle is true we unblock the current vertex
        if found_cycle:
            self.unblock(current_state)
        # no cycle is not found in the path
        # add all the neighbours of current vertex to blocked-map
        else:
            for neighbour in self.dfsa.get_next_states(current_state):
                self.blocked_map.add((neighbour,current_state))

        self.stack.pop()
        return found_cycle


    def unblock(self,state):
       self.blocked_set.remove(state)
       list_block_map = [s[1] for s in self.blocked_map if s[0]==state]
       # if list not empty
       if list_block_map:
           # unblock all states that needs to be unblocked recursively
           for state_to_unblock in list_block_map:
               if state_to_unblock in self.blocked_set:
                   self.unblock(self,state_to_unblock)
           for i in self.blocked_map:
               if i[0]==state:
                   self.blocked_map.remove(i)
```

Listing 6.1: implementation of Johnson Algorithm

```
print('------------------------------------------')
print('Obtaining Simple Cycles of M')
print('------------------------------------------')
johnson_algorithm_M = JohnsonsAlgorithm(M)
print(johnson_algorithm_M.get_simple_cycles())
```

Listing 6.2: finding simple cycles in M

6.2.1 | Output



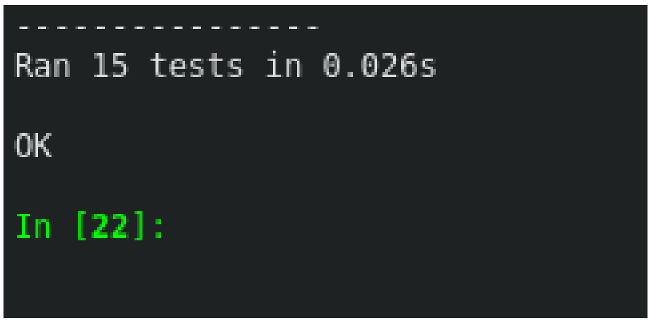
```
------------------------------------------
Obtaining Simple Cycles of M
------------------------------------------
[[4, 5, 4], [5, 5], [1, 6, 1]]
```

Figure 6.2: the output of Listing 6.2

7

Test Results.

Note all unit test written in sections 1.2, 2.2 3.2, 5.2 pass.



```
-----------------
Ran 15 tests in 0.026s

OK

In [22]:
```

Figure 7.1: all unit tests pass

References

[1] DavidWesselsVIU. Compilers: Hopcroft's algorithm. *YouTube*, 2021. URL: https://www.youtube.com/watch?v=D0107TKCQX8.

[2] Wikipedia Editors. Dfa minimization. *Wikipedia*. URL: https://en.wikipedia.org/wiki/DFA_minimization.

[3] Prashant Bhardwaj. Is dead state included in the minimized dfa or not? *stackoverflow*, 2012. URL: https://stackoverflow.com/questions/9141293/is-dead-state-is-included-in-the-minimized-dfa-or-not.

[4] Rosalind. Algo: Strongly connected component. URL: https://rosalind.info/glossary/algo-strongly-connected-component/.

[5] Tushar Roy Coding Made Simple. Johnson's algorithm - all simple cycles in directed graph. *YouTube*, 2015. URL: https://www.youtube.com/watch?v=johyrWospv0.