ICS2210 Report

By Gabriel Hili

Contents

How to run

Navigate to the root folder (ics2210) and run *make*. Then execute the resulting file. Alternatively you can execute the shell script *run* in order to compile and run the executable. The program was developed in C++ 20 on WSL (Windows Subsystem for Linux)

Evaluation & Discussion

Underlying Data Structure

Since I knew that a state would always have 2 transitions, I represented my DFA in a state-transition table. The state-transition table has two arrays, one for each transition. The transitions of some state *i* are the indices in position *i* in the arrays of the transition table. I chose to use a state-transition table because; it has O(1) look-up time like an adjacency matrix and O(*v***t*) space complexity, like an adjacency list, where *v* is the number of vertices and *t* is the length of the alphabet. Since a given state can only have 2 transitions using an adjacency matrix would be wasteful, as the connections would be sparse.

To represent the final states, a Boolean array was used. A final state *f* means that the value at position *f* in the array is *True* and for non-final states it would be *False*.

Algorithms

Depth of DFA

The depth of the DFA is calculated by performing a Breadth-First Search from the starting state. The length of the path of the furthest vertex encountered from the starting state is the depth of the DFA.

The BFS is implemented using a queue. Initially the staring vertex and a dummy marker vertex are pushed onto the queue. The front-most vertex is popped from the queue and its children are pushed into the queue. If the front-most element is the marker vertex, the current depth is incremented by 1 and the dummy vertex is re-pushed in the queue. Vertices that have been in the queue will not be pushed if re-traversed. When the queue contains one element (the dummy marker vertex), the algorithm returns the value of the current depth, which is the depth of the DFA.

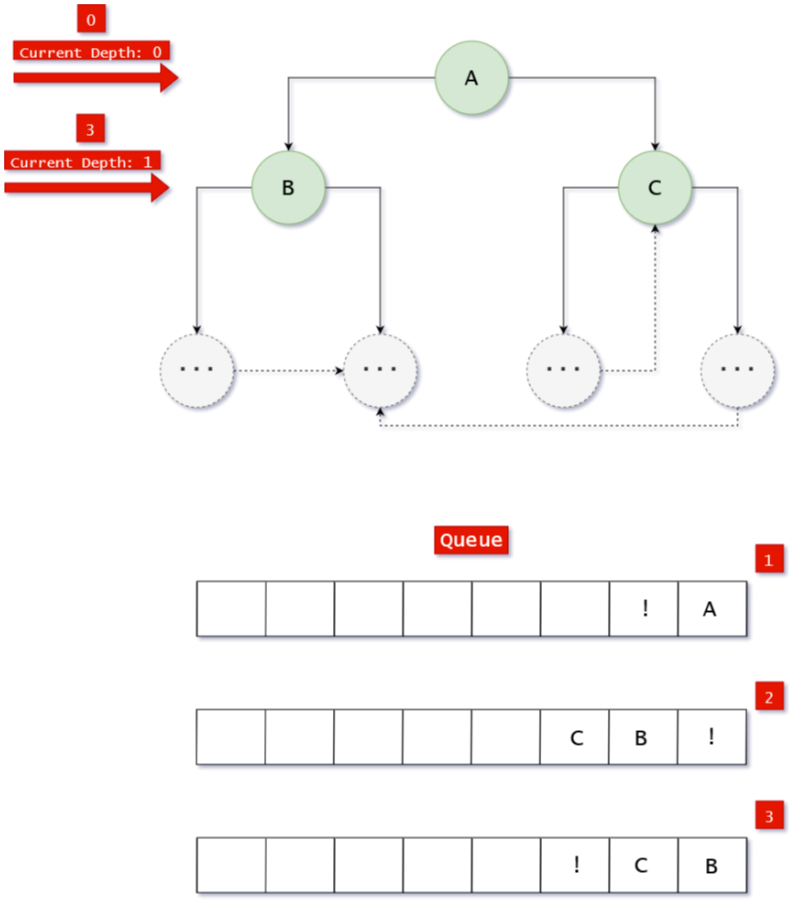


Fig 1. Incrementing the current depth of the graph using a dummy vertex in the queue

The algorithms time complexity is O(*v*) where *v* is the number of vertices. This is because BFS must traverse every vertex only once, and will terminate once this condition has been met. BFS could also have been implemented using recursion. However storing the count and seen vertices would require global variables and as a result I opted for the iterative implementation.

Hopcroft's Minimisation Algorithm

The Algorithm uses the Myhill and Nerode [1] theorem in order to partition the sets into *k*-equivalence classes. Once the k^{th} computed equivalence class is identical to the $k\text{-}1^{th}$ equivalence class, then all states within a partition are merged into a singular state. The algorithm does this in O(*n.log n*) time.

Hopcroft's minimisation algorithm can be evaluated by checking if the automata still recognises the same language after minimisation. The moment the DFA is generated 100,000 strings of length 64 are randomly generated and passed through the DFA. These strings are then passed through the optimised DFA. If the automatons reject and accept the same strings, then we can conclude with high probability that the automatons are equivalent.

The DFA class stores strings as 64-bit numbers, where *a* and *b* are represented by 0 and 1 respectively. If the string was accepted in the original graph, but the state it ended up in was not a final state, then we can conclude that the language recognised is not the same.

```cpp
bool Dfa::minimiseTest() {

    //Pass previously generated testing strings to the DFA.
    int length = sizeof(word[0])*8;

    for (int i = 0; i < STRING_NUM; i++) {

        int current_state = start;
        unsigned long long current_transition = word[i];

        for (int j = 0; j < length; j++) {
            current_state = mat[(current_transition) & 1][current_state];
            current_transition >>= 1;
        }

        //Strings was not supposed to be accepted/rejected.
        if (accepted[i] != final[current_state]) {
            cout << "Error on string " << i << " :";

            for (int j = 0; j < length; j++) {
                cout << "ab"[(word[i]>>j) & 1];
            }
            cout << '\n';

            cout << "Expected: " << accepted[i] << " Actual: " << final[current_state] << '\n';
            return false;
        }
    }

    //All strings were accepted.
    return true;

}
```

Fig 1. Function that tests Hopcroft's algorithm

Before Hopcroft's minimisation takes place, all unreachable and dead states are removed. The unreachable states are selected by performing a BFS from the starting vertex. Any vertex not discovered is hence unreachable. The dead states are discovered by performing BFS from every vertex. If a final state was not discovered then that state is dead. Upon removing the unreachable and dead states, the language recognised by the automata should remain the same. This is tested by passing the strings generated in the beginning and evaluating their output.

Tarjan's Strongly Connected Components Algorithm

Strongly Connected Components are a subset of vertices in the graph such that any vertex is reachable from every other vertex. Tarjan's algorithm uses a Depth-First Search in order to traverse the graph. Each node is assigned a *traversal number*, *low link value* and a *visited value*. The traversal number is the order in which the vertices were seen by the DFS, the low link value of a vertex is the smallest low link value of any adjacent vertices and *visited* dictates whether a vertex has been traversed or not,

To evaluate the algorithm, one can check if all vertices are reachable from every other vertex in a given Strongly Connected Component. This can naïvely be done by a brute-force check for all vertices. Given a vertex v and a strongly connected component C, the test algorithm attempts to reach all vertices $v' \in V_C$ from v.

Johnson's Algorithm

Johnson's algorithm was invented by Donald B. Johnson in 1975 [2]. It is used to find all simple cycles in a directed graph. The time complexity of this algorithm is O((E+V)(c+1)) where E is the number of edges, V is the number of vertices and c is the number of simple cycles in the graph.

In a nutshell, the algorithm first finds the Strongly Connected Components (SCC) of a graph. Then, for every vertex in a SCC, all simple cycles starting and ending in that vertex are found, by traversing

the SCC in a Depth-First Search (DFS) fashion. After all simple cycles starting from that vertex are found, the vertex is removed from the graph, the SCCs are recomputed, and a new starting vertex is chosen. The algorithm repeats until the graph is empty.

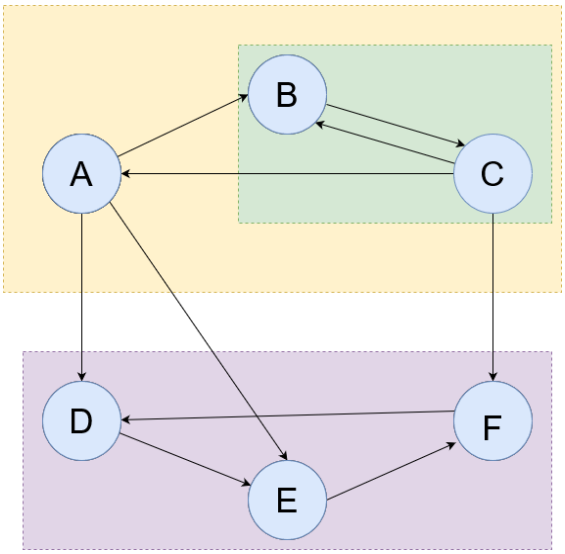
```
while (!empty(G)) {
      SCCs = G.getSCC()

      for (scc in SCCs) {
            for (v in scc) {
                  cycles.add(G.getSimpleCycles(v,scc))
                  G.remove(v)
                  SCCs = G.getSCC()
            }
      }
}
```

Fig 1. Pseudo-code for Johnson's Algorithm.

Simple cycles are Hamiltonian Tours of a subset of vertices in the graph such that no vertex appears more than once, except the starting vertex which appears at the end. For example, the following graph contains 3 simple cycles {A →B →C →A}, {B →C →B} and {D →E →F→D}.

Fig 2. A Directed Graph with 3 simple cycles.



Strongly Connected Components are a subset of vertices in the graph such that any vertex is reachable from every other vertex. SCCs can be computed using Tarjan's Strongly Connected Component Algorithm [3]. For every SCC, there cannot exist a path such that it leaves the SCC and comes back. For example the following graph has 3 SCCs.

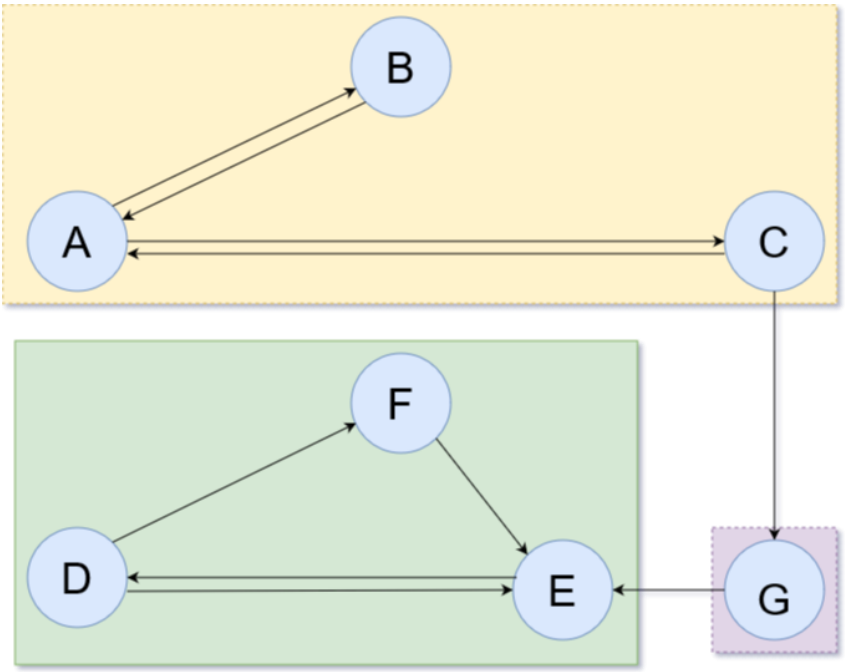


Fig 3. A Directed Graph with 3 Strongly Connected Components.

Given a strongly connected component and a starting vertex, Johnson's algorithm will find all simple cycles starting from the starting vertex, within that strongly connected component. To do this, it makes use of 3 data structures. The *Stack, Blocked Set and Blocked Map*. The *Stack* is used to keep track of vertices traversed by the Depth-First Search. The *Blocked Set* is used to close off vertices because a valid cycle cannot be found through those vertices. This saves the algorithm time because it does not have to traverse a path which won't lead to a simple cycle. Any vertex placed in the *Stack* or *Blocked Set* cannot be traversed by the DFS. The *Blocked Map* creates rules such that if a vertex in the *Blocked Set* becomes unblocked (it is removed from the *Blocked Set*) then another vertex must also be unblocked. Vertices unblocked by this rule may also fire additional rules which will unblock more consequent vertices. Once a rule fires, the rule is removed from the *Blocked Map*.

The algorithm starts from the starting vertex and performs DFS. Anytime an untraversed (not in the *Stack*) and unblocked (not in *Blocked Set*) vertex is encountered, it is added to the *Stack* and the *Blocked Set*. It is added to the *Blocked Set* to prevent the DFS from traversing a path which will not lead it to a simple cycle. If that vertex does lead to a simple cycle later on, then it is unblocked.

Given a vertex *v* traversed by the DFS, the algorithm can react in 3 different ways.

Case 1: If the starting vertex is re-encountered then that means a simple cycle was found. The *Stack* and the *Blocked Set* do not allow the DFS to expand into already seen vertices as untraversed vertices must be unseen and unblocked, so we know that the starting vertex was re-encountered through a simple cycle. The simple cycle found is composed of all the values between the top most value and the starting vertex in the *Stack*. The algorithm then recurses back to the parent, pops *v* from the *Stack* and continues traversing any other children, if any.

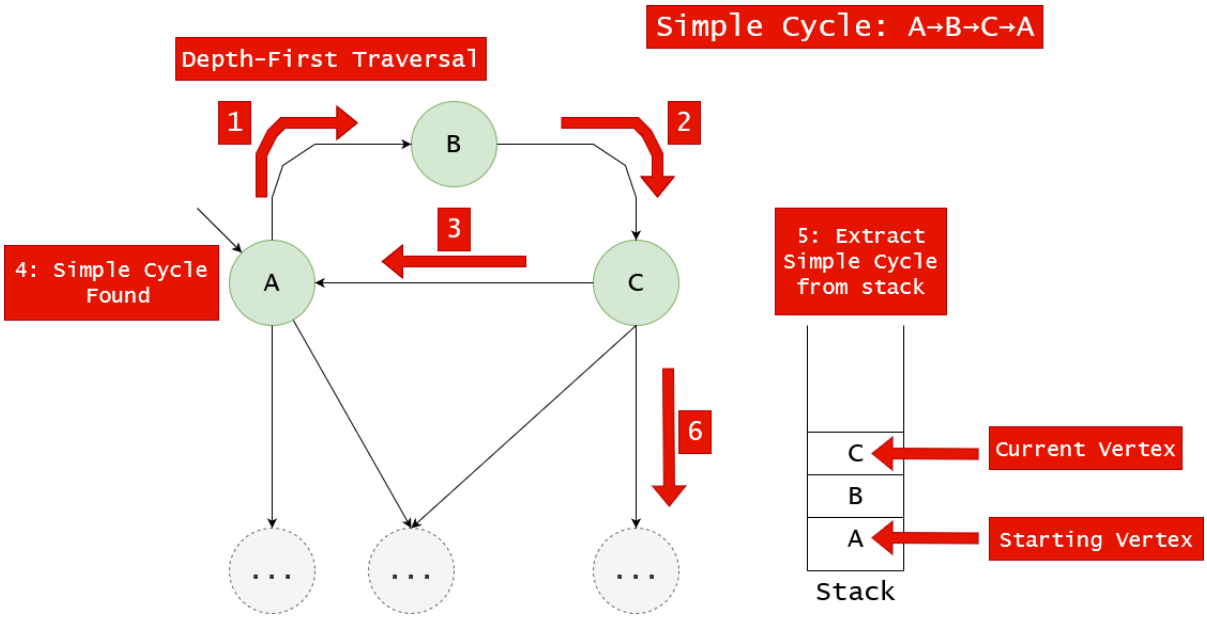


Fig 4. A simple cycle being detected and extracted from the Stack

Case 2: If a vertex *v* cannot be expanded further because all child vertices are in the *Blocked Set*, then the DFS recurses back to the parent of *v* and is popped from the *Stack*. We note that *v* cannot be expanded only because the children are blocked. So we add a rule in the *Blocked Map* that says if any child of *v* ever becomes unblocked, then we will unblock *v* to facilitate any future cycles passing through *v* and its child.

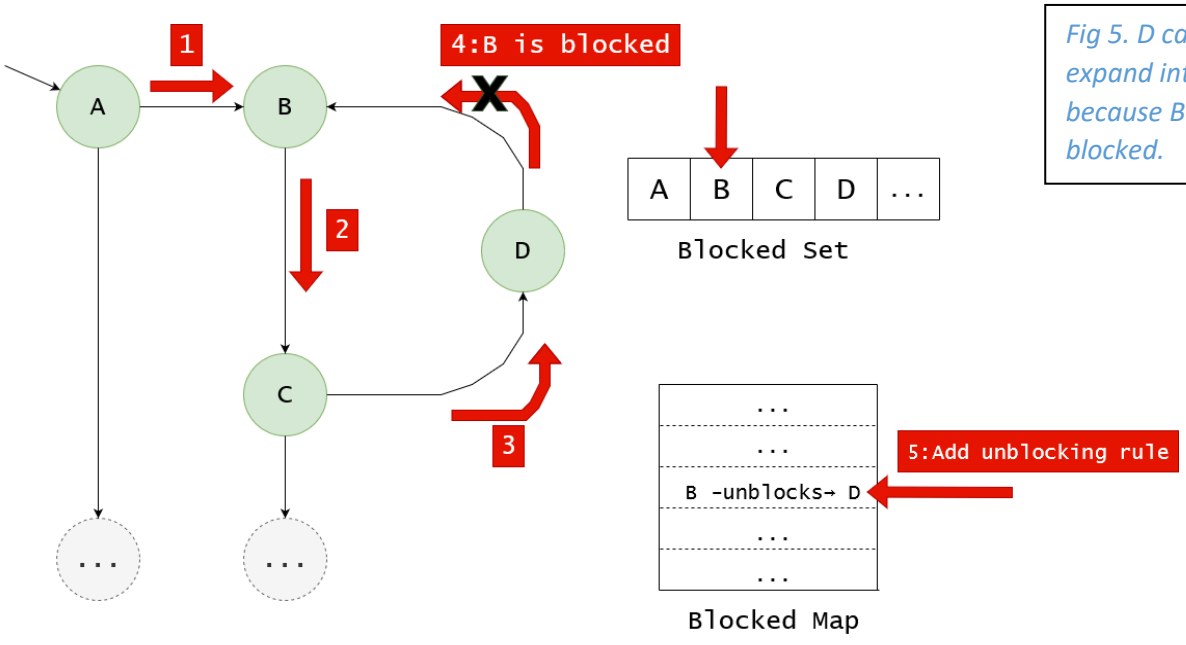


Fig 5. D cannot expand into B, because B is blocked.

Case 3: If a vertex *v* cannot be expanded further because all child vertices are in *Blocked Set*, but one of *v's* children was part of a found cycle, then: *v* is removed from the *Blocked Set*, DFS recurses back to the parent of *v* and *v* is popped from the *Stack*. We remove *v* from the *Blocked Set* because if we know there exists a vertex *w* that leads directly into *v* and then eventually into the starting vertex, then there might exist another vertex *w'* that is reachable from the starting vertex, which leads directly into *v*, hence finding another simple cycle. When *v* is unblocked, the algorithm consults the *Blocked Map* to see if any other vertices must also be unblocked.

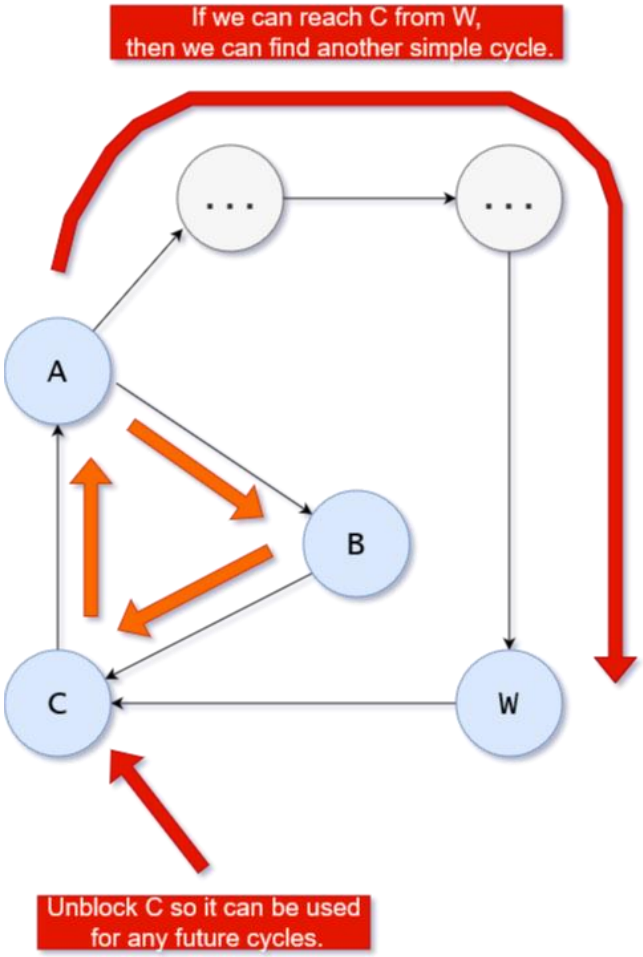
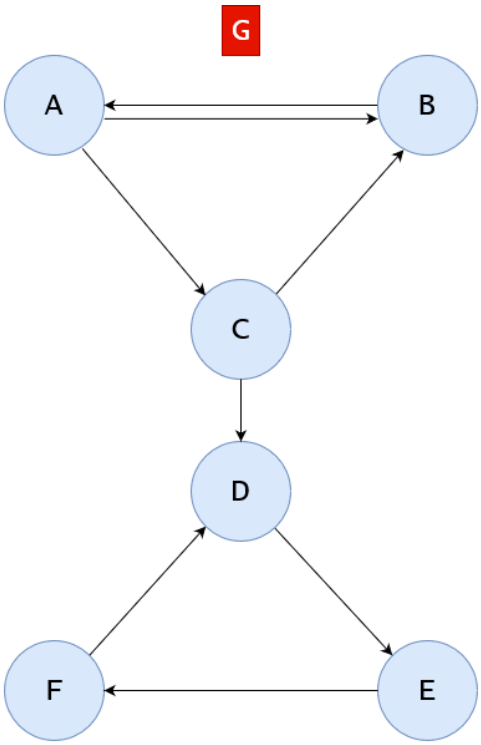


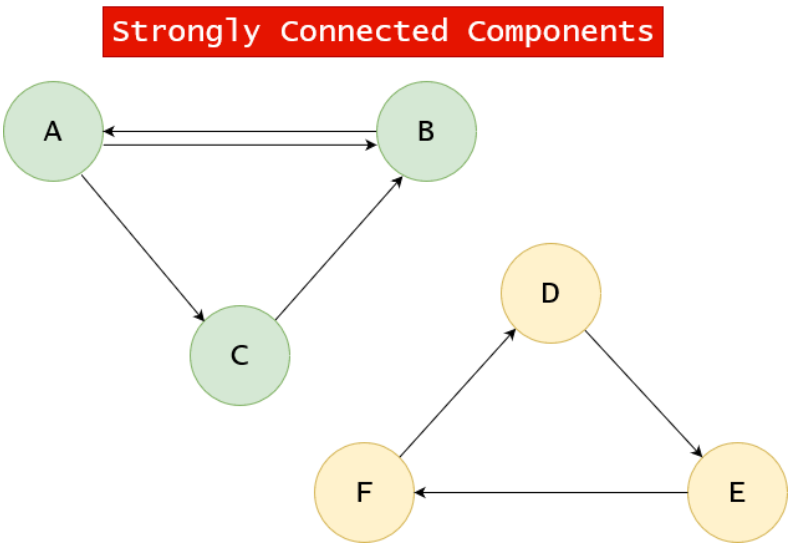
Fig 6. Unblocking C so the algorithm could attempt to re-enter C through W.

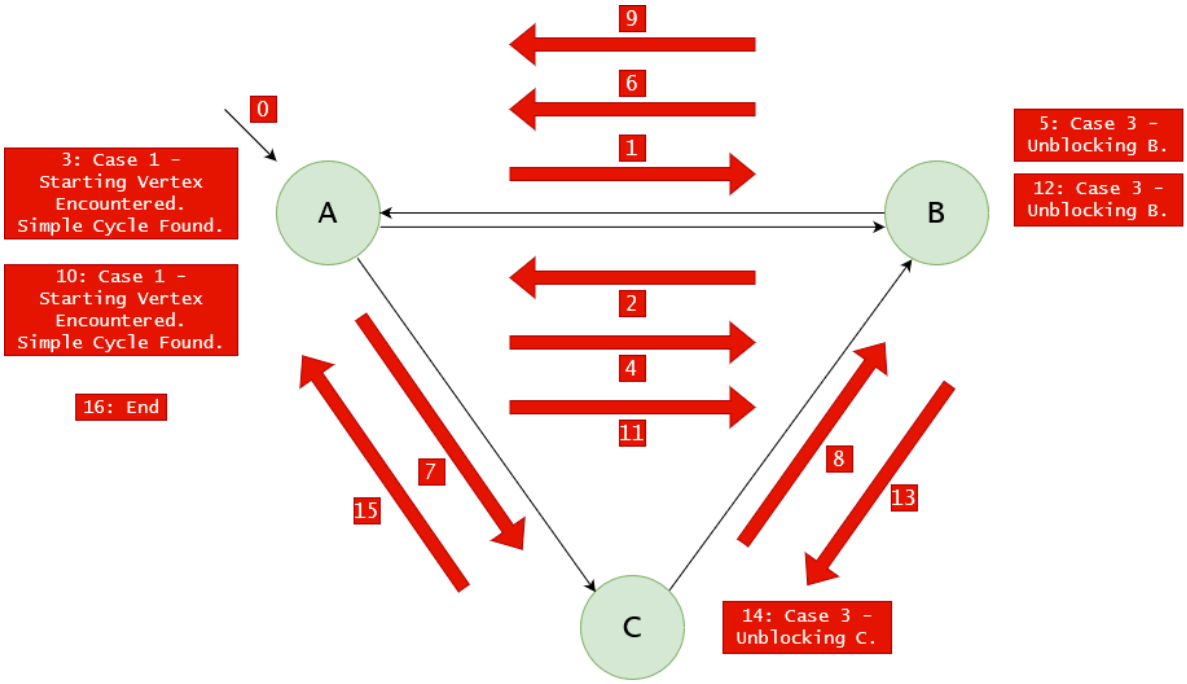
The algorithm traverses the graph using DFS updating the *Stack, Blocked Set* and *Blocked Map,* based on the above 3 cases. When the DFS finishes and returns to the starting vertex, as mentioned before, the algorithm will remove that vertex from the graph, re-compute the SCCs and start over. This repeats until all vertices are popped from the Graph.

Dry Run

The following is a dry run of finding all simple cycles from the starting vertex A. We first decompose the graph G into its Strongly Connected Components. Then we pick the SCC containing A and perform the algorithm.


G
A
B
C
D
F
E


Strongly Connected Components
A
B
C
D
F
E



Time Step	Stack	Blocked Set	Blocked Map	Found Cycles
0	[A]	[A]	[]	[]
1	[A,B]	[A,B]	[]	[]
2	[A,B]	[A,B]	[]	[]
3	[A,B]	[A,B]	[]	[A→B→A]
4	[A,B]	[A,B]	[]	[A→B→A]
5	[A,B]	[A]	[]	[A→B→A]
6	[A]	[A]	[]	[A→B→A]
7	[A,C]	[A,C]	[]	[A→B→A]
8	[A,C,B]	[A,C,B]	[]	[A→B→A]
9	[A,C,B]	[A,C,B]	[]	[A→B→A]
10	[A,C,B]	[A,C,B]	[]	[A→B→A,A→C→B→A]
11	[A,C,B]	[A,C,B]	[]	[A→B→A,A→C→B→A]
12	[A,C,B]	[A,C]	[]	[A→B→A,A→C→B→A]
13	[A,C]	[A,C]	[]	[A→B→A,A→C→B→A]
14	[A,C]	[A]	[]	[A→B→A,A→C→B→A]
15	[A]	[A]	[]	[A→B→A,A→C→B→A]
16	[]	[]	[]	[A→B→A,A→C→B→A]

Bibliography

[1]A. Nerode, “Linear automaton transformations,” Proceedings of the American Mathematical Society, vol. 9, no. 4. American Mathematical Society (AMS), pp. 541–544, 1958. doi: 10.1090/s0002-9939-1958-0135681-9.

[2] D. B. Johnson, “Finding All the Elementary Circuits of a Directed Graph,” SIAM Journal on Computing, vol. 4, no. 1, pp. 77–84, Mar. 1975, doi: 10.1137/0204007.

[3] R. Tarjan, “Depth-First Search and Linear Graph Algorithms,” SIAM Journal on Computing, vol. 1, no. 2, pp. 146–160, Jun. 1972, doi: 10.1137/0201010.