ICS2210 Report

By Gabriel Hili

Contents

[Johnson’s Algorithm 2](#_Toc98714931)

[Bibliography 5](#_Toc98714932)

# Johnson’s Algorithm

Johnson’s algorithm was invented by Donald B. Johnson in 1975 [1]. 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()

}

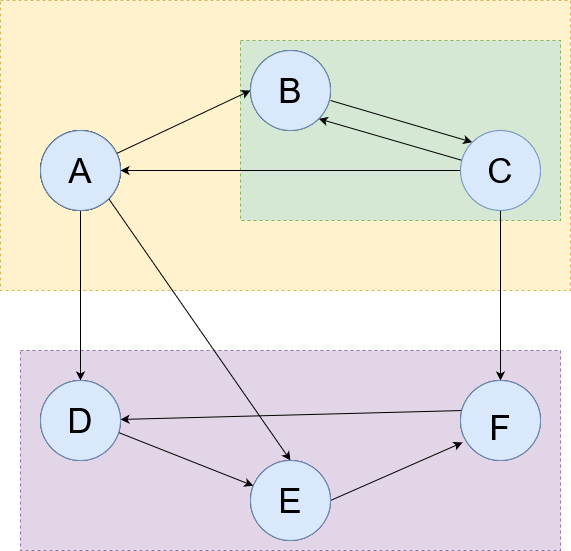
}

}

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

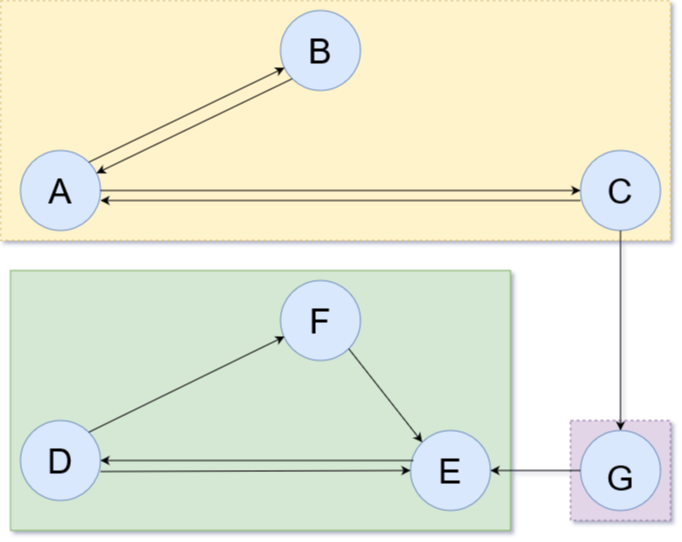
Simple cycles are Hamiltonian Tours of a subset of vertices in the graph such that no vertex appears more than once, except the starting vertex which appears at the end. For example, the following graph contains 3 simple cycles {A →B →C →A}, {B →C →B} and {D →E →F→D}.

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 [2]. 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.

**Case 1**: If the starting vertex is re-encountered then that means a simple cycle was found. The *Stack* and the *Blocked Set* donot allow the DFS to expand into already seen vertices as traversed nodes 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 the *Stack*.

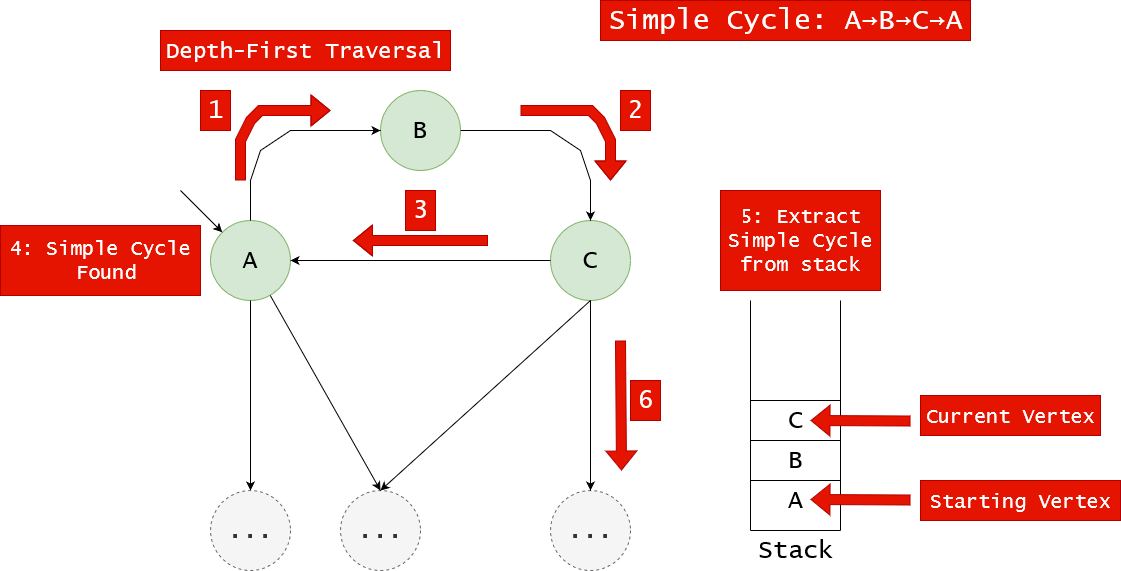
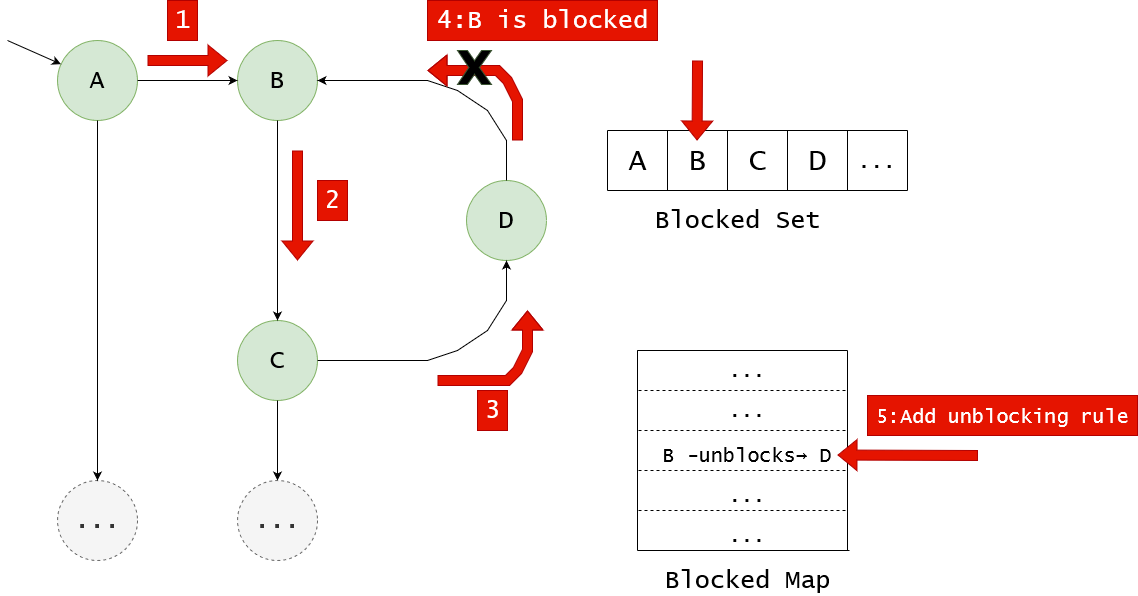


Fig 4. A simple cycle being detected and extracted from the The Stack.

**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 is the starting vertex 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 into *v* and then into the starting vertex (closing off the simple cycle), then there might exist another vertex *w****′*** that leads 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.

# Bibliography

[1] 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.

[2] 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.