CSC 225 - Summer 2019

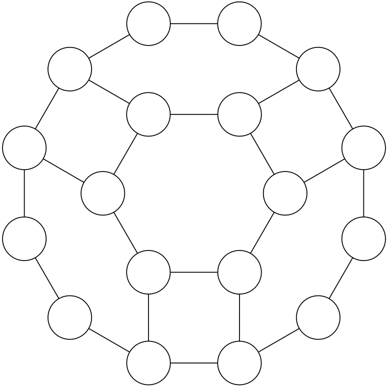
Connectivity

Bill Bird

Department of Computer Science
University of Victoria

July 30, 2019

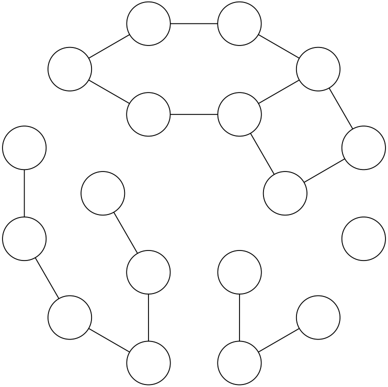
Connectivity (1)



A graph G is **connected** if, for every pair of vertices u and w, there exists at least one uw-path.

The graph above is connected.

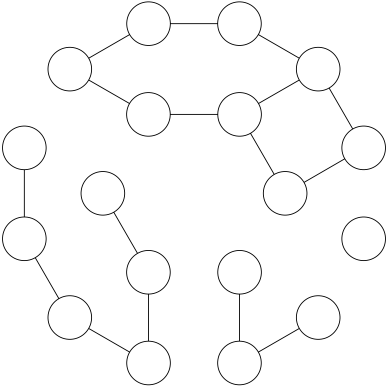
Connectivity (2)



The graph above is not connected.

A disconnected graph consists of at least two **connected components** (the above graph has four).

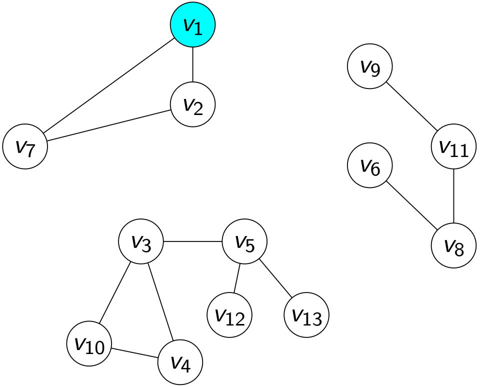
Connectivity (3)



To test whether an undirected graph is connected, a single traversal (from an arbitrary starting point) is sufficient.

If the traversal visits all n vertices, then the graph is connected.

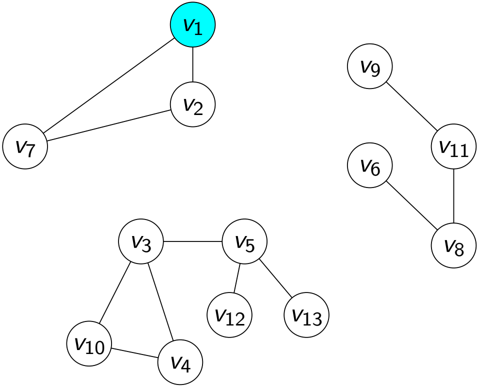
Finding Connected Components (1)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Exercise: Design an algorithm to partition the vertices of a graph G into connected components.

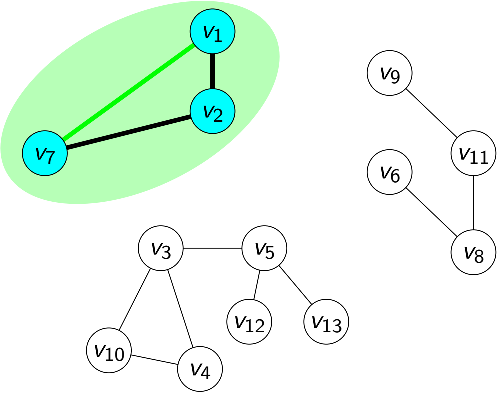
Finding Connected Components (2)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Starting a DFS or BFS traversal at any vertex will visit an entire connected component.

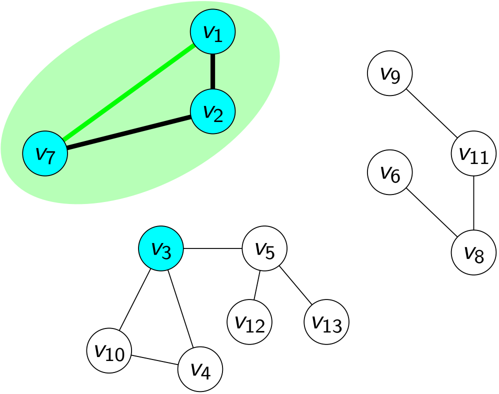
Finding Connected Components (3)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	1	1	×	×	×	×	1	×	×	×	×	×	×

To identify the first component, run a traversal and add all visited vertices to component 1.

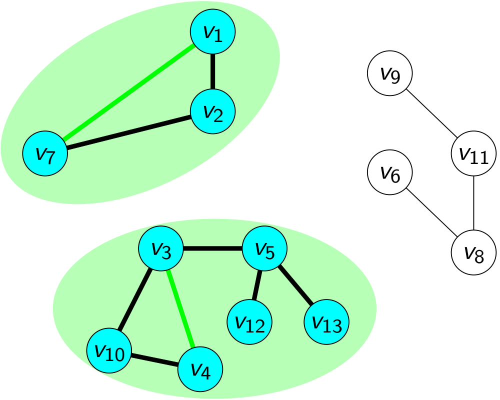
Finding Connected Components (4)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	1	1	×	×	×	×	1	×	×	×	×	×	×

If any unvisited vertices remain, choose one and run another traversal.

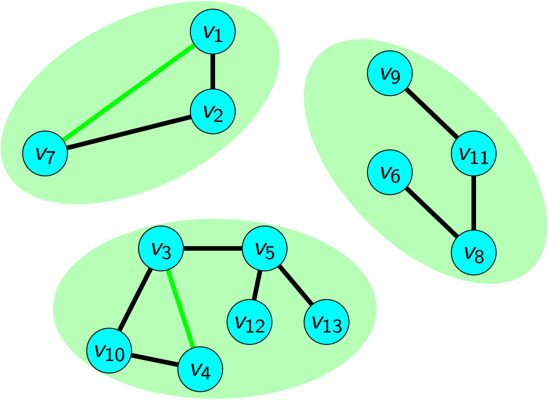
Finding Connected Components (5)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	1	1	2	2	2	$\times$	1	$\times$	$\times$	2	$\times$	2	2

Add all visited vertices to component 2.

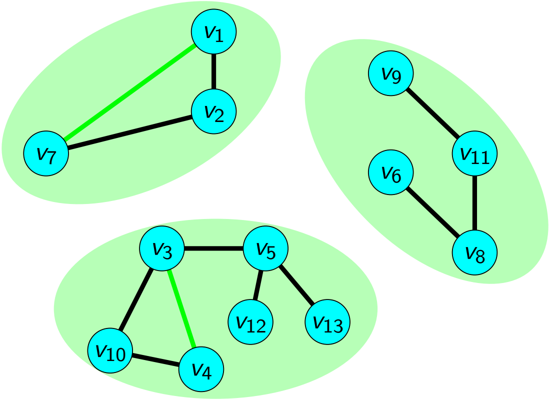
Finding Connected Components (6)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	1	1	2	2	2	3	1	3	3	2	3	2	2

Continue running traversals until all vertices are visited.

Finding Connected Components (7)



Vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
Component	1	1	2	2	2	3	1	3	3	2	3	2	2

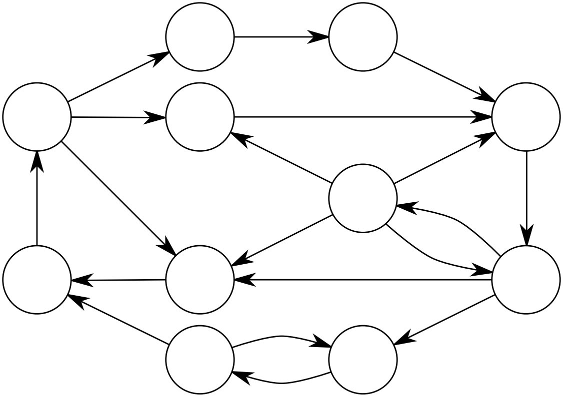
Although multiple executions of the traversal algorithm are required, each vertex is visited exactly once, so this process requires $\Theta(n + m)$ time.

Finding Connected Components (8)

```
procedure RecursiveDFS(G, components, component_num, v)
    components[v] ← component_num
    for each neighbour w of v do
        if components[w] = −1 then
            RecursiveDFS(G, components, component_num, w)
        end if
    end for
end procedure
procedure ConnectedComponentsDFS(G)
    components ← Array of size n, initialized to −1
    component_num ← 1
    for each vertex v in G do
        if components[v] = −1 then
            RecursiveDFS(G, components, component_num, v)
            component_num ← component_num + 1
        end if
    end for
    return components
end procedure
```

The pseudocode above computes components using DFS and the value -1 to represent an invalid component.

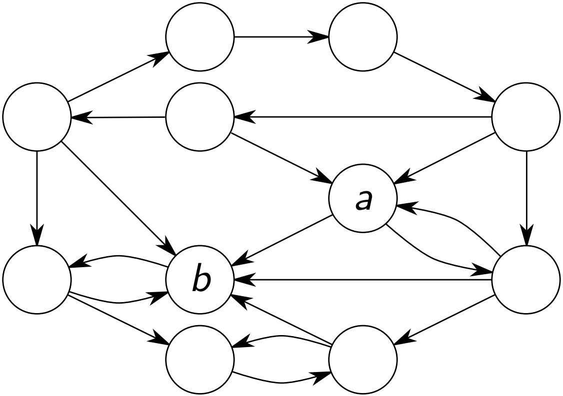
Strong Connectivity (1)



A directed graph G is **strongly connected** if, for any two distinct vertices u and v, there is a directed path from u to v.

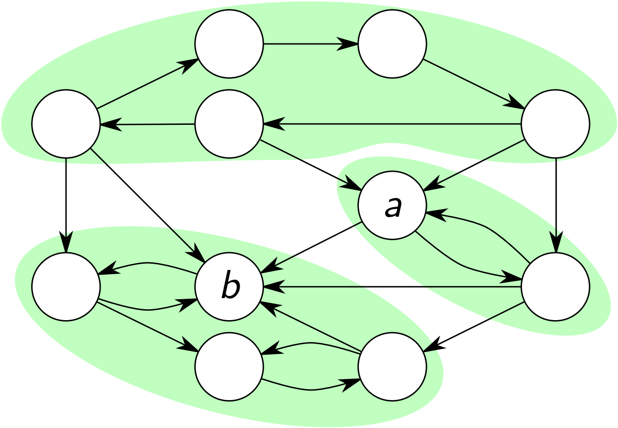
The graph above is strongly connected.

Strong Connectivity (2)



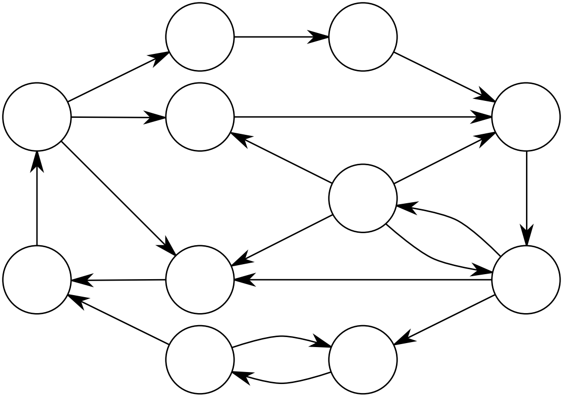
The graph above is not strongly connected: A path from a to b exists, but there is no path from b to a.

Strong Connectivity (3)



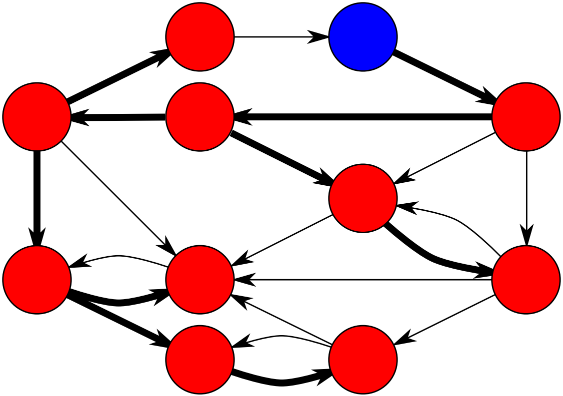
A graph which is not strongly connected can be partitioned into two or more **strongly connected components**.

Strong Connectivity (4)



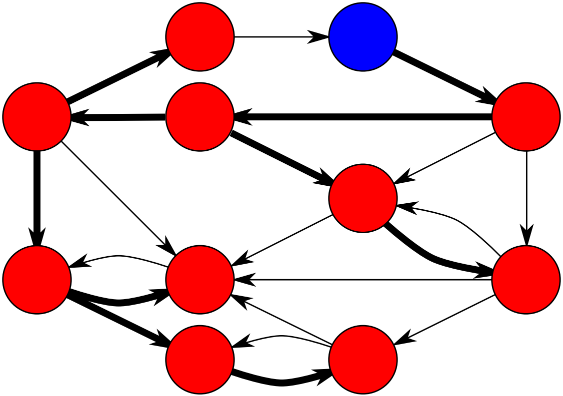
Problem: Design an algorithm to test if a graph is strongly connected.

Strong Connectivity (5)



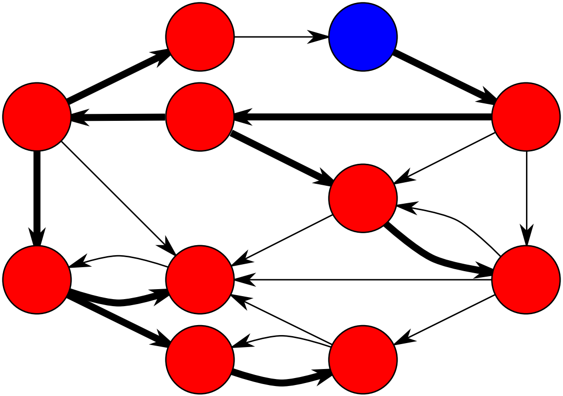
A single traversal is not sufficient: Every vertex is reachable from the blue vertex, but the graph is not strongly connected.

Strong Connectivity (6)



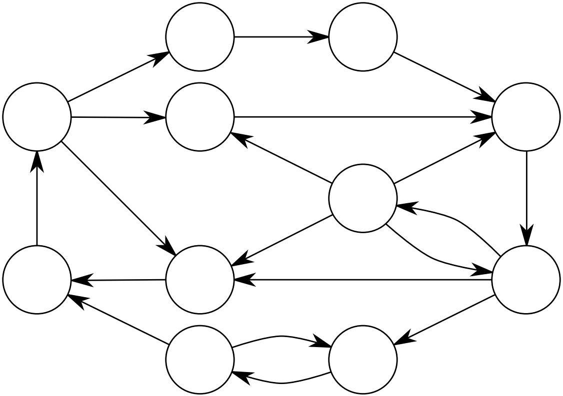
A sequence of n traversals would work. If, for every vertex v, a traversal rooted at v visits every vertex, then the graph is strongly connected.

Strong Connectivity (7)



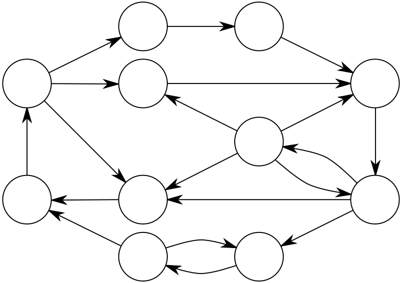
Running n traversals requires $\Theta(n(n+m))$ time in the worst case.

Strong Connectivity (8)

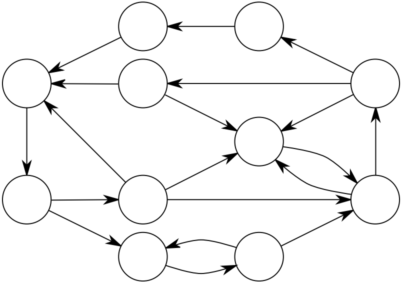


A faster strong connectivity test can be designed by traversing edges in the wrong direction.

Strong Connectivity (9)



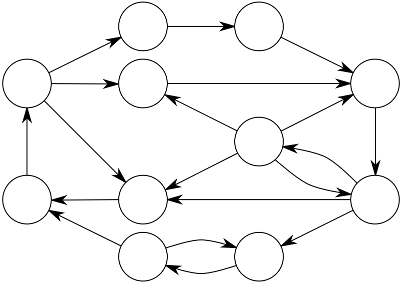
G



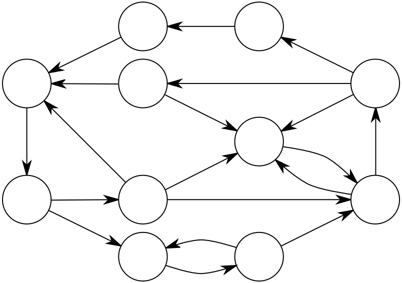
G^T

If G is a directed graph, then the **transpose graph** of G, denoted G^T, is a copy of G with all edge directions reversed.

Strong Connectivity (10)



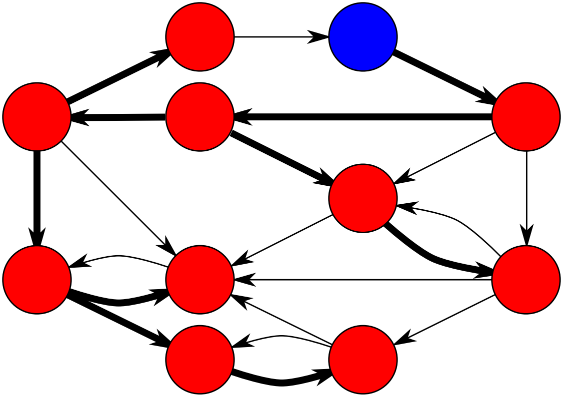
G



G^T

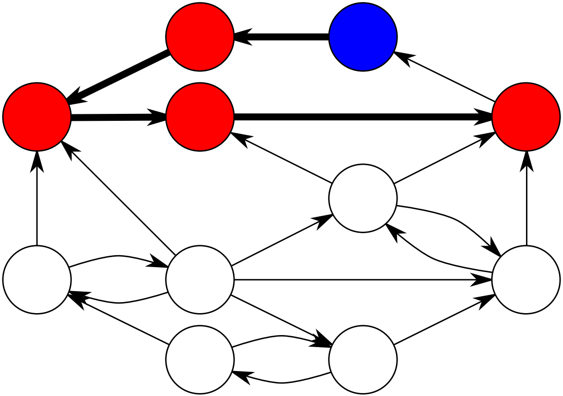
The adjacency matrix for G^T is the transpose of the adjacency matrix for G.

Strong Connectivity (11)



A traversal rooted at v in the original graph visits all vertices reachable from v.

Strong Connectivity (12)

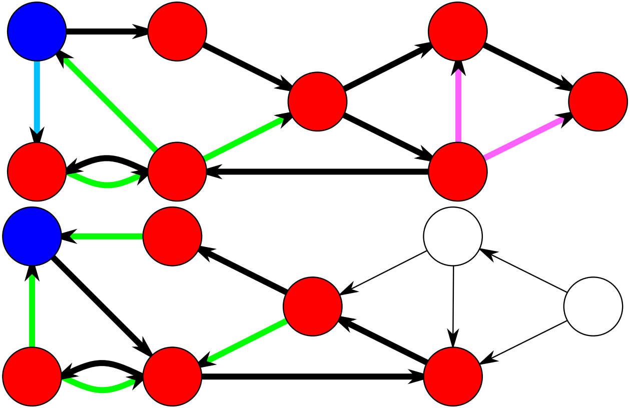


A traversal rooted at v in the transpose graph visits all vertices from which v is reachable in the original graph.

Strong Connectivity (13)

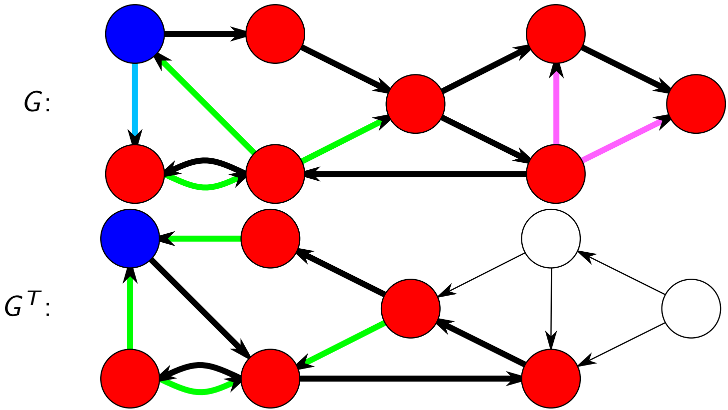
G:

G^T:



To test strong connectivity, choose an arbitrary vertex v and run a traversal rooted at v in both the original and transpose graphs.

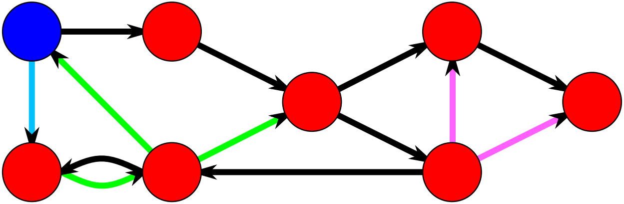
Strong Connectivity (14)



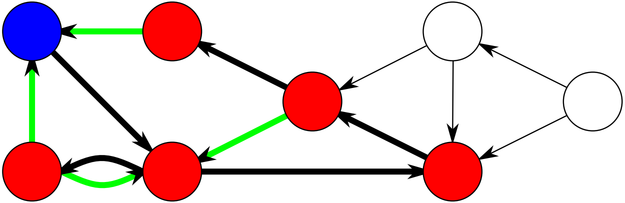
If both traversals visit all vertices, then the graph is strongly connected.

Strong Connectivity (15)

G:



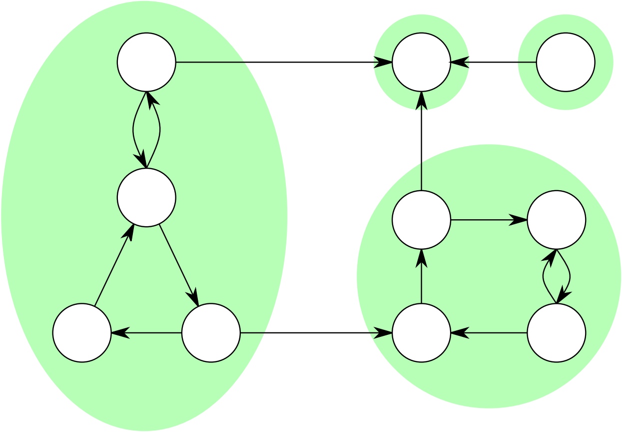
G^T:



The graph above is not strongly connected.

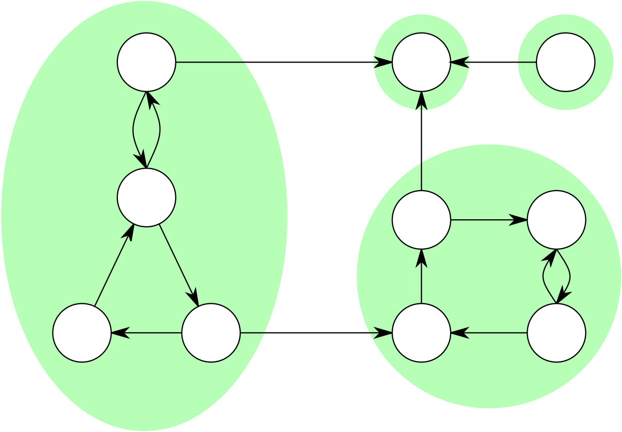
The set of vertices visited by both traversals corresponds to the strongly connected component containing the root vertex.

Strongly Connected Components (1)



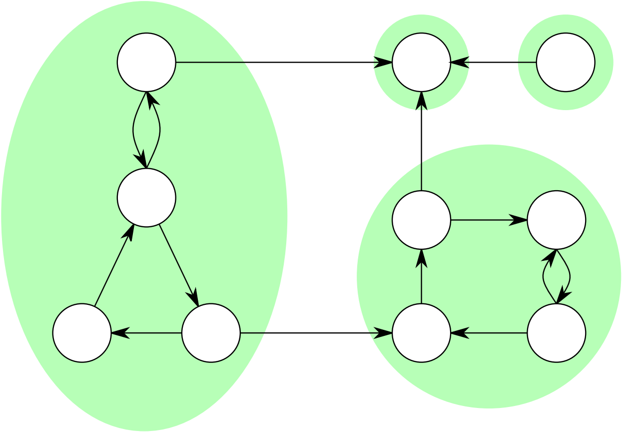
Problem: Design an algorithm to find all strongly connected components of a directed graph.

Strongly Connected Components (2)



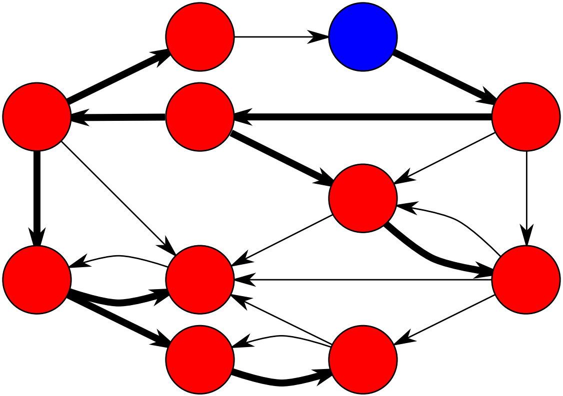
The obvious algorithm uses the strong connectivity test repeatedly to find one component at a time.

Strongly Connected Components (3)



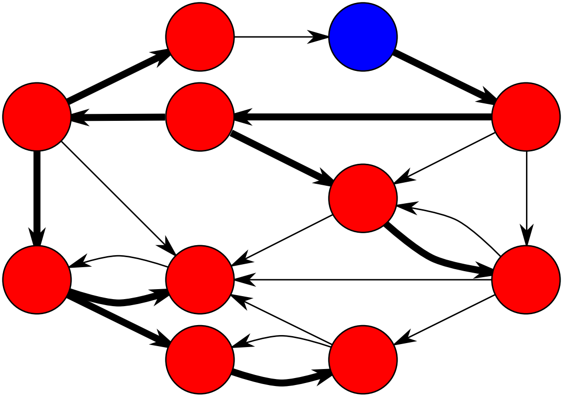
However, each strong connectivity test may require $\Theta(n+m)$ time, so n repetitions of the strong connectivity test requires $\Theta(n(n+m))$ time.

Strongly Connected Components (4)



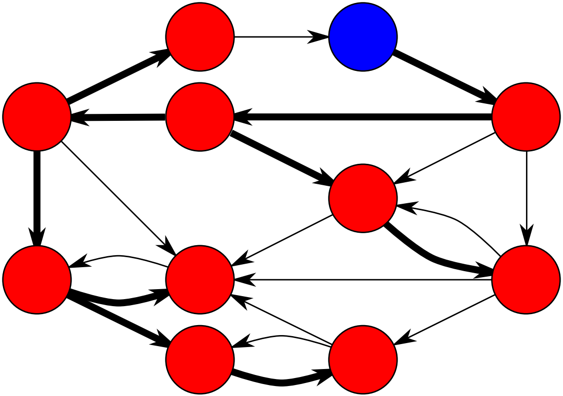
Some of the traversal information can be reused. For example, subtrees of the DFS tree above correspond to DFS trees for other visited vertices.

Strongly Connected Components (5)



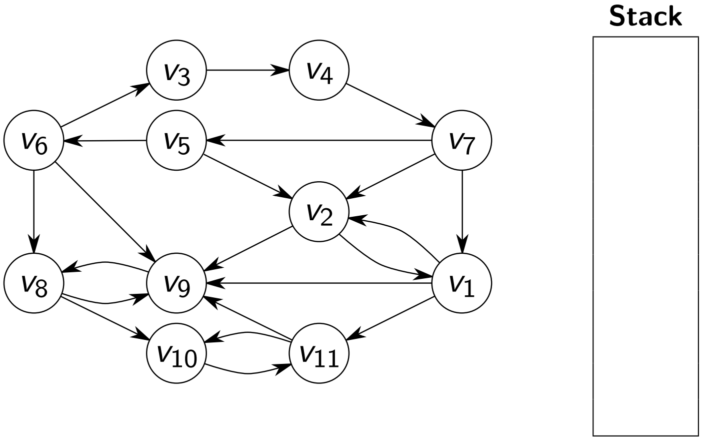
There are several algorithms which leverage this fact to find strongly connected components in $\Theta(n + m)$ time.

Strongly Connected Components (6)



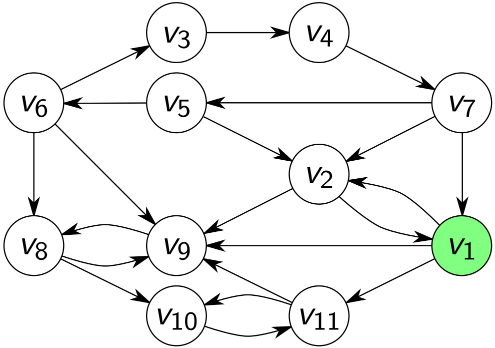
One example is Kosaraju's algorithm, which uses a variant of post-order DFS.

Kosaraju's Algorithm (1)

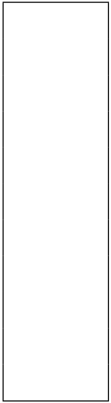


Kosaraju's algorithm finds strongly connected components with DFS and a stack.

Kosaraju's Algorithm (2)

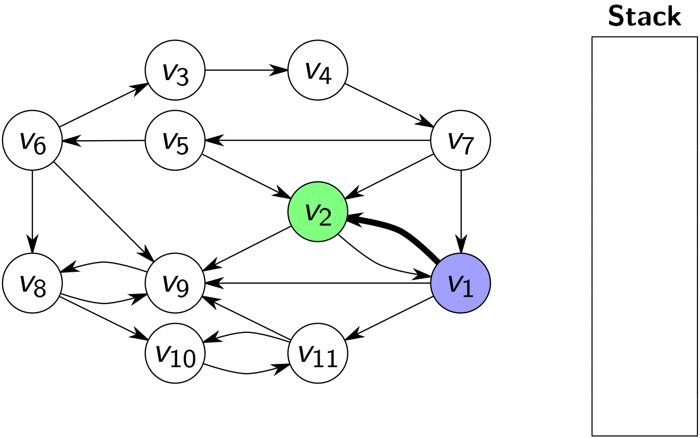


Stack



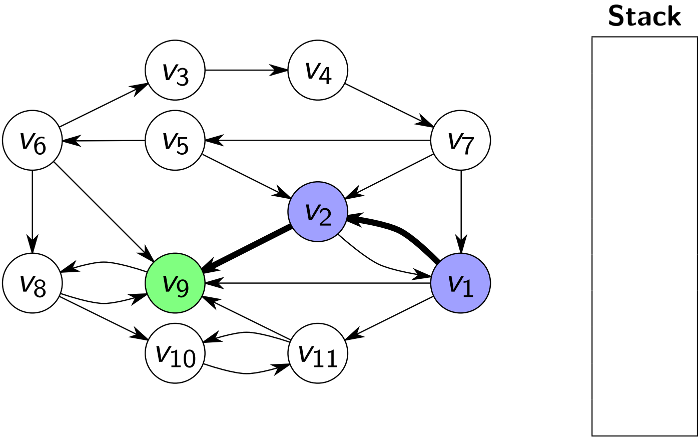
Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (3)



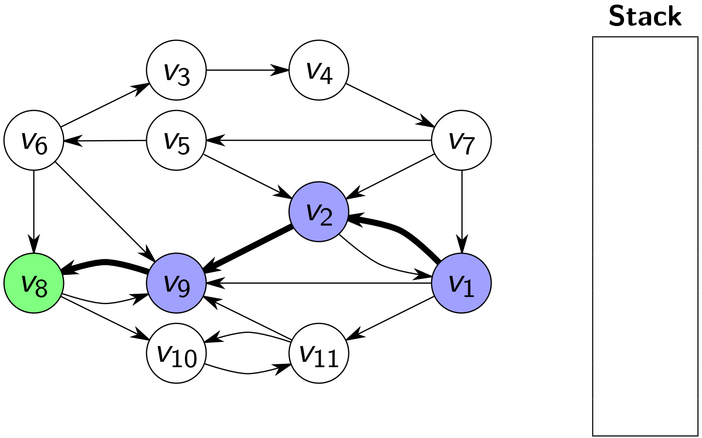
Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (4)



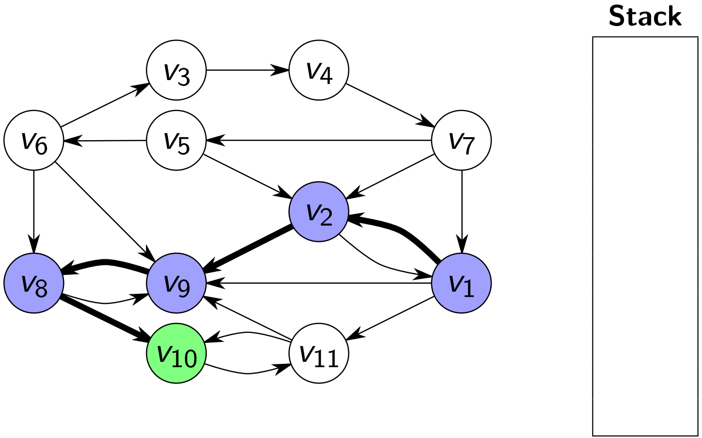
Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (5)



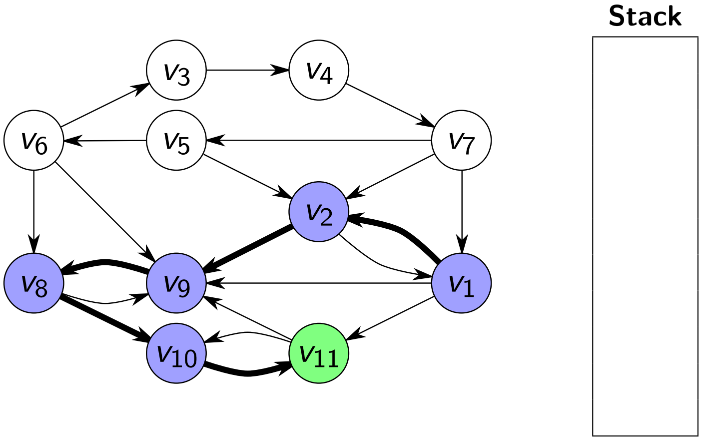
Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (6)



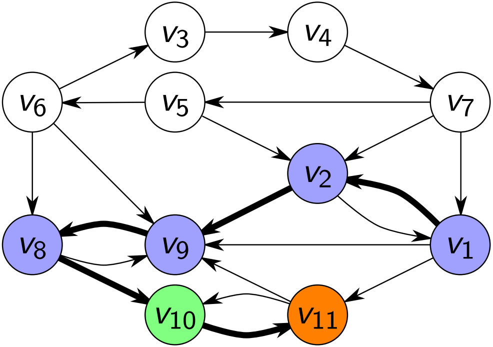
Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (7)

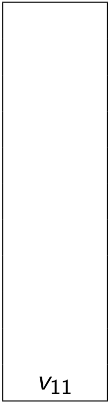


Start by choosing an arbitrary vertex and running DFS.

Kosaraju's Algorithm (8)

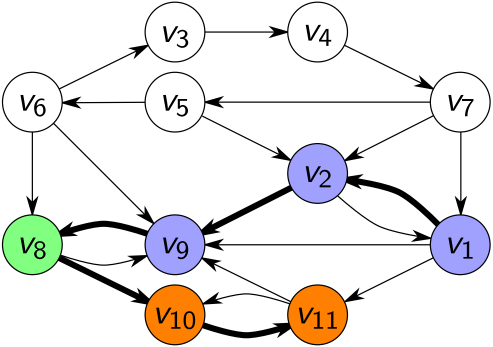


Stack



When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (9)

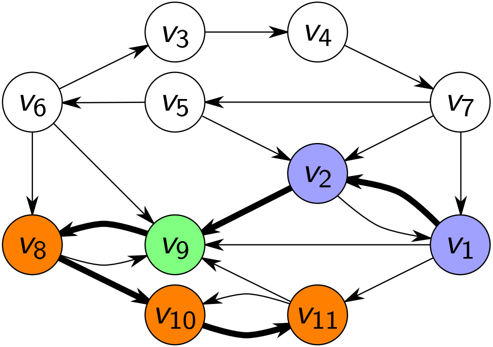


Stack

v_{10}
v_{11}

When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (10)

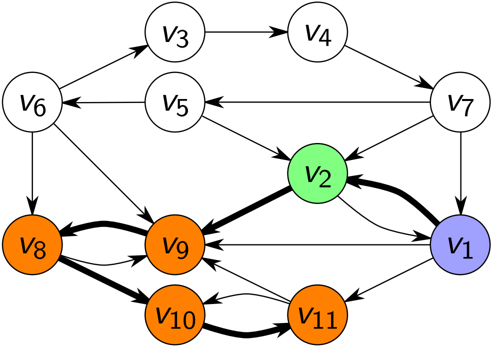


Stack

v_8
v_{10}
v_{11}

When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (11)

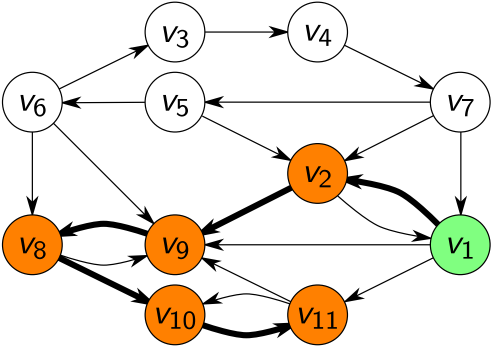


Stack

v_9
v_8
v_{10}
v_{11}

When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (12)

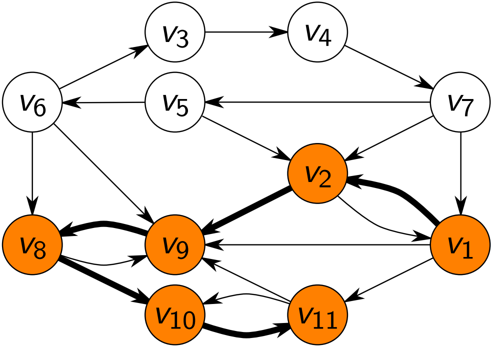


Stack

v_2
v_9
v_8
v_{10}
v_{11}

When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (13)

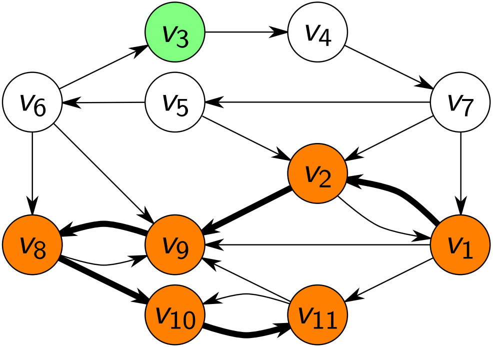


Stack

v_1
v_2
v_9
v_8
v_{10}
v_{11}

When DFS unwinds from a vertex, push it onto the stack.

Kosaraju's Algorithm (14)

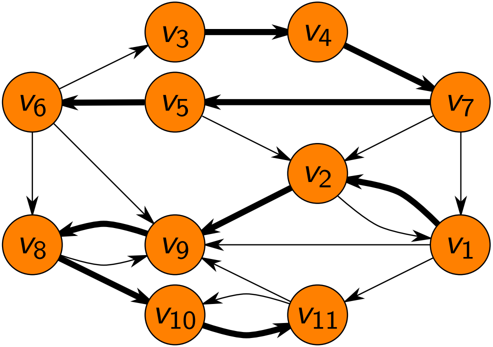


Stack

v_1
v_2
v_9
v_8
v_{10}
v_{11}

If unvisited vertices remain after the traversal finishes, choose an unvisited vertex and start another traversal.

Kosaraju's Algorithm (15)

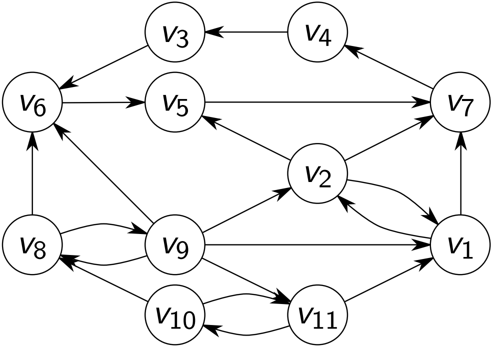


Stack

v_3
v_4
v_7
v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

Eventually, all vertices will be on the stack.

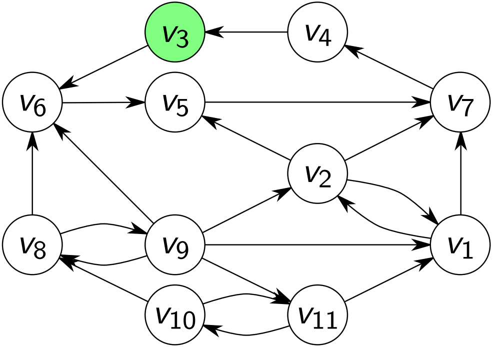
Kosaraju's Algorithm (16)



Stack

v_3
v_4
v_7
v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

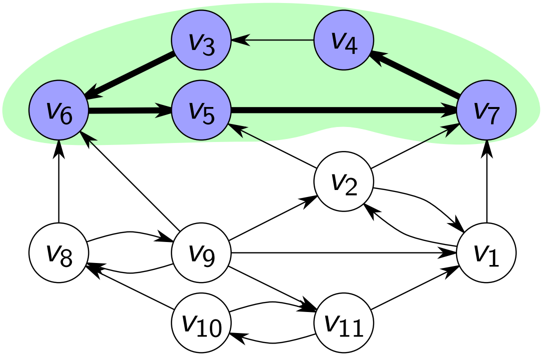
Once every vertex is on the stack, switch to the transpose graph and mark every vertex as unvisited.



Stack

v_4
v_7
v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

Pop the top vertex off the stack and run a DFS traversal in the transpose graph.

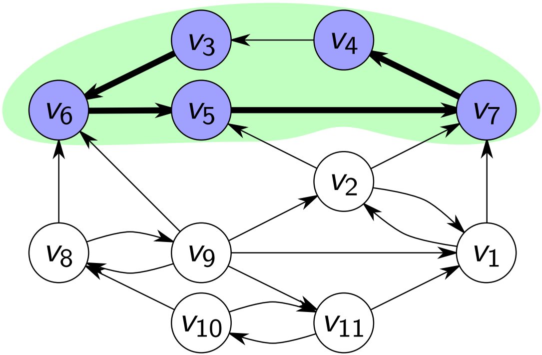


Stack

v_4
v_7
v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

The set of vertices discovered by the traversal will be a strongly connected component of the graph.

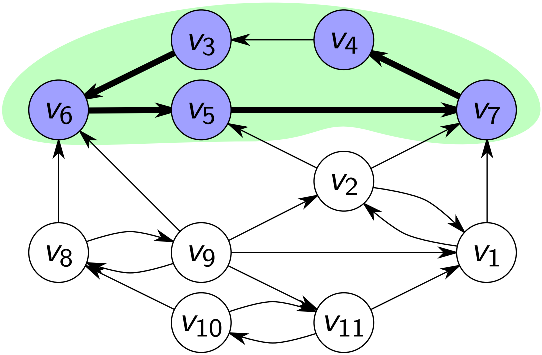
Kosaraju's Algorithm (19)



Stack

v_7
v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

While unvisited vertices remain, continue popping the stack until an unvisited vertex is found.

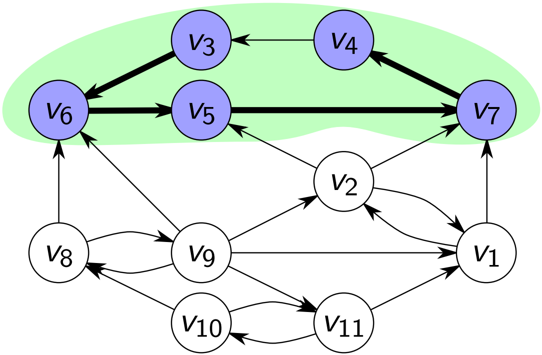


Stack

v_5
v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

While unvisited vertices remain, continue popping the stack until an unvisited vertex is found.

Kosaraju's Algorithm (21)

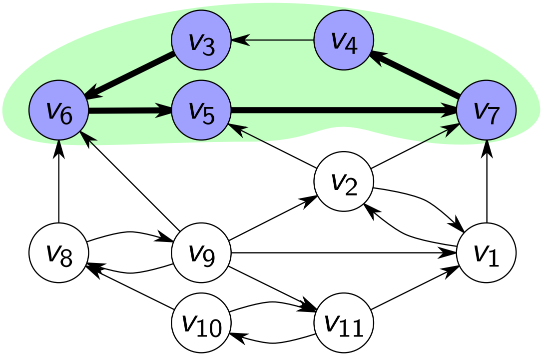


Stack

v_6
v_1
v_2
v_9
v_8
v_{10}
v_{11}

While unvisited vertices remain, continue popping the stack until an unvisited vertex is found.

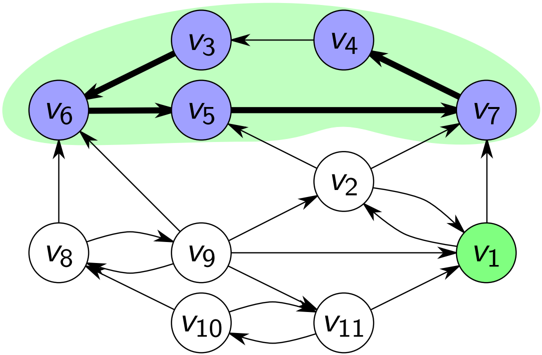
Kosaraju's Algorithm (22)



Stack

v_1
v_2
v_9
v_8
v_{10}
v_{11}

While unvisited vertices remain, continue popping the stack until an unvisited vertex is found.

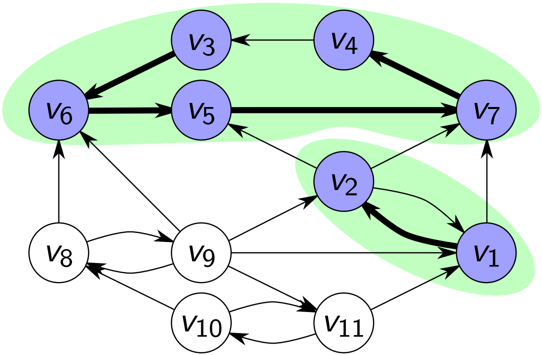


Stack

v_2
v_9
v_8
v_{10}
v_{11}

When an unvisited vertex is found in the stack, run a traversal starting at that vertex.

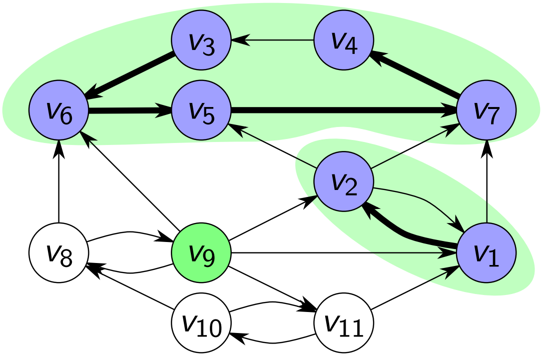
Kosaraju's Algorithm (24)

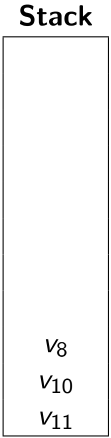


Stack

v_2
v_9
v_8
v_{10}
v_{11}

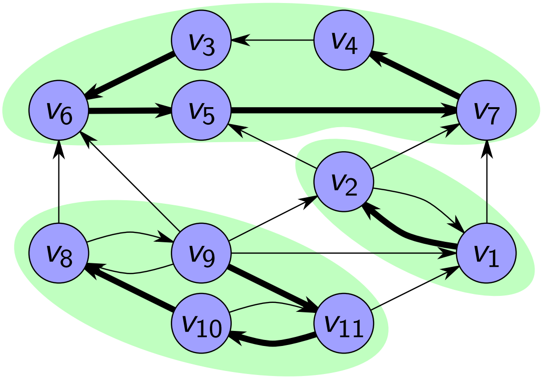
The set of vertices discovered will yield another strongly connected component.

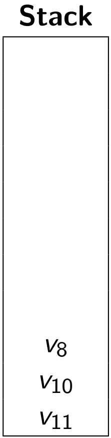




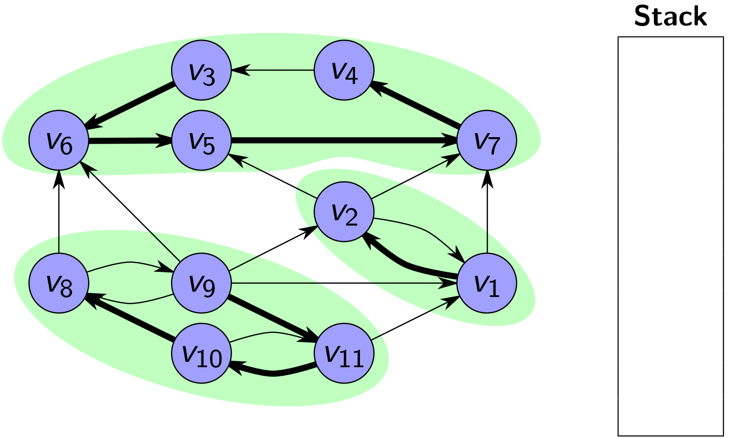
Continue running traversals and collecting components until the stack is empty.

Kosaraju's Algorithm (26)





Continue running traversals and collecting components until the stack is empty.



Continue running traversals and collecting components until the stack is empty.

Kosaraju's Algorithm (28)

```
procedure PhaseOneDFS(G, S, v)
    Mark v as visited
    for each neighbour w of v do
        if w is unvisited then
            PhaseOneDFS(G, S, w)
        end if
    end for
    Push v onto S
end procedure
procedure KosarajuPhaseOne(G)
    S ← Empty stack
    Mark all vertices unvisited
    for each vertex v in G do
        if v is unvisited then
            PhaseOneDFS(G, S, v)
        end if
    end for
    return S
end procedure
```

The pseudocode above corresponds to the first part of Kosaraju's algorithm, which builds the stack.

```
procedure PhaseOneDFS(G, S, v)
    Mark v as visited
    for each neighbour w of v do
        if w is unvisited then
            PhaseOneDFS(G, S, w)
        end if
    end for
    Push v onto S
end procedure
procedure KosarajuPhaseOne(G)
    S ← Empty stack
    Mark all vertices unvisited
    for each vertex v in G do
        if v is unvisited then
            PhaseOneDFS(G, S, v)
        end if
    end for
    return S
end procedure
```

The total running time of phase one is $\Theta(n + m)$ in the worst case.

Kosaraju's Algorithm (30)

```
procedure PhaseTwoDFS(G^T, components, component_num, v)
    components[v] ← component_num
    for each neighbour w of v do
        if components[w] = −1 then
            PhaseTwoDFS(G^T, components, component_num, w)
        end if
    end for
end procedure
procedure KosarajuPhaseTwo(G^T, S)
    components ← Array of size n, initialized to −1
    component_num ← 1
    while S is non-empty do
        v ← Pop(S)
        if components[v] = −1 then
            PhaseTwoDFS(G^T, components, component_num, v)
            component_num ← component_num + 1
        end if
    end while
    return components
end procedure
```

The pseudocode above corresponds to the second part of Kosaraju's algorithm.

Kosaraju's Algorithm (31)

```
procedure PhaseTwoDFS(G^T, components, component_num, v)
    components[v] ← component_num
    for each neighbour w of v do
        if components[w] = −1 then
            PhaseTwoDFS(G^T, components, component_num, w)
        end if
    end for
end procedure
procedure KosarajuPhaseTwo(G^T, S)
    components ← Array of size n, initialized to −1
    component_num ← 1
    while S is non-empty do
        v ← Pop(S)
        if components[v] = −1 then
            PhaseTwoDFS(G^T, components, component_num, v)
            component_num ← component_num + 1
        end if
    end while
    return components
end procedure
```

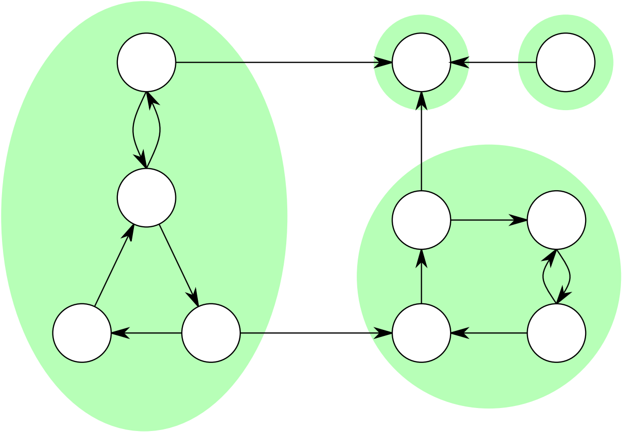
KosarajuPhaseTwo takes the stack constructed in phase one and outputs the set of components.

Kosaraju's Algorithm (32)

```
procedure PhaseTwoDFS(G^T, components, component_num, v)
    components[v] ← component_num
    for each neighbour w of v do
        if components[w] = −1 then
            PhaseTwoDFS(G^T, components, component_num, w)
        end if
    end for
end procedure
procedure KosarajuPhaseTwo(G^T, S)
    components ← Array of size n, initialized to −1
    component_num ← 1
    while S is non-empty do
        v ← Pop(S)
        if components[v] = −1 then
            PhaseTwoDFS(G^T, components, component_num, v)
            component_num ← component_num + 1
        end if
    end while
    return components
end procedure
```

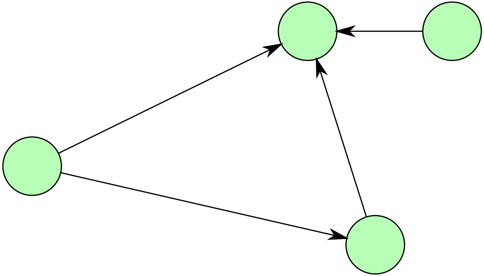
Phase two is also $\Theta(n+m)$, so the total running time of Kosaraju's algorithm is $\Theta(n+m)$.

Reduced Graphs (1)



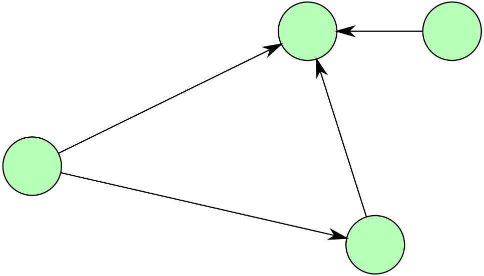
The strongly connected components of a directed graph can be collapsed into single vertices.

Reduced Graphs (2)



The result is the **reduced graph** or **condensed graph**.

Reduced Graphs (3)



Exercise: Prove that the reduced graph of any directed graph is acyclic.