Graphs

Last updated: November 26th 2018, at 1.05pm

Contents

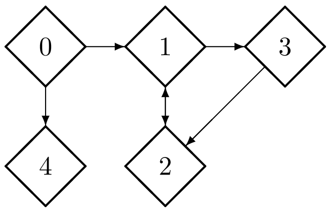
1 Non-Cyclic Graphs

1.1 Cycles in Graphs

1.1.1 Paths through Graphs

Definition: A *path* is a list of edges, each of which follows the other, in a given graph.

Example: In the graph:



the following...

...are paths in this graph:	...are not paths in this graph:
$0 \longrightarrow 4$	$0 \longrightarrow 3 \longrightarrow 2$
$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 1 \longrightarrow 2$	$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 4$
$2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 2$	$0 \longrightarrow 5$

1.1.2 Cycles in Graphs

Definition: A *cycle* is a (non-empty) path that starts and ends at the same node or vertex.

Example: Using the graph in the previous example, the following are all cycles:

- $2 \longrightarrow 1 \longrightarrow 2$
- $1 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1$
- $2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 2$

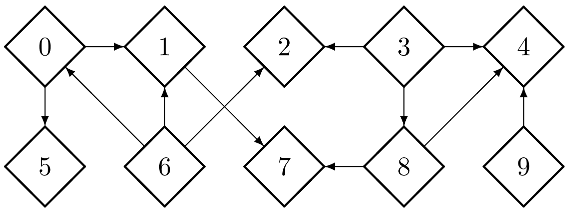
1.2 Ayclic Graphs

Definition: A *non-cyclic* graph is simply a graph that does not contain any cycles.

Trees are an obvious source of examples of non-cyclic graphs.

2 Topological Sorting

Consider the following non-cyclic graph:

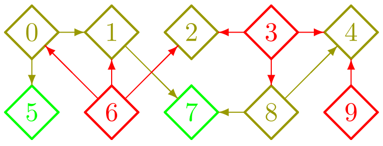


Clearly, since this graph is acyclic, if there is a path from a node n to another node m there cannot be a path from m to n (otherwise there would be a cycle: n to m to n). Is it possible to list the nodes of this graph in such a way that:

> **if** n and m are nodes in the graph
> **and** there is a path from n to m
> **then** n occurs *before* m in our listing?

Definition: Any such listing of the nodes of a non-cyclic graph is called a *topological sorting.*

Example:



So, for the previous graph, we need a listing of the nodes that satisfies the following constraints:

- node 0 occurs before nodes 1 and 5 (and before node 7)
- node 1 occurs before node 7
- node 3 occurs before nodes 2, 4 and 8 (and before node 7)
- node 6 occurs before nodes 0, 1 and 2 (and before nodes 5 and 7)
- node 8 occurs before nodes 4 and 7
- node 9 occurs before node 4

Note that the conditions between brackets do not need to be stated explicitly, as they follow from the other constraints. For example, node 7 must come after node 0 because node 7 must come after node 1 and node 1 must come after node 0.

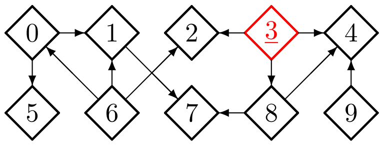
2.1 Attempt 1

Will depth-first traversal yield a topological sort for non-cyclic graphs?

Example:

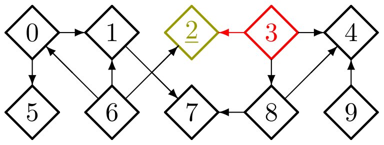
0	$\{1,5\}$
1	$\{7\}$
2	$\emptyset$
3	$\{2,4,8\}$
4	$\emptyset$
5	$\emptyset$
6	$\{0,1,2\}$
7	$\emptyset$
8	$\{4,7\}$
9	$\{4\}$

If we do a depth-first traversal starting at a root node (e.g. node 3):



0	X	$\{1,5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	✓	$\{2,4,8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	X	$\{0,1,2\}$
7	X	$\emptyset$
8	X	$\{4,7\}$
9	X	$\{4\}$

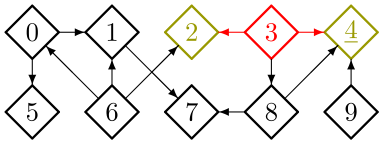
[3]



0	X	$\{1,5\}$
1	X	$\{7\}$
2	✓	$\emptyset$
3	✓	$\{2,4,8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	X	$\{0,1,2\}$
7	X	$\emptyset$
8	X	$\{4,7\}$
9	X	$\{4\}$

[3,2]

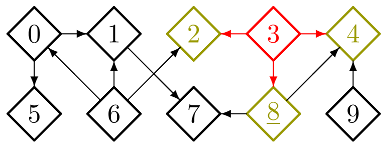
Backtrack to node 3, and then descend to node 4:



0	X	$\{1,5\}$
1	X	$\{7\}$
2	✓	$\emptyset$
3	✓	$\{2,4,8\}$
4	✓	$\emptyset$
5	X	$\emptyset$
6	X	$\{0,1,2\}$
7	X	$\emptyset$
8	X	$\{4,7\}$
9	X	$\{4\}$

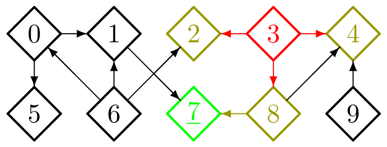
[3,2,4]

Backtrack to node 3, and then descend to node 8:



0	X	$\{1, 5\}$
1	X	$\{7\}$
2	✓	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	✓	$\emptyset$
5	X	$\emptyset$
6	X	$\{0, 1, 2\}$
7	X	$\emptyset$
8	✓	$\{4, 7\}$
9	X	$\{4\}$

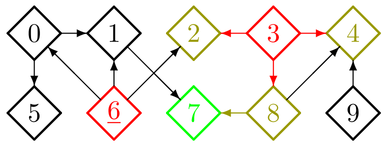
[3,2,4,8]



0	X	$\{1, 5\}$
1	X	$\{7\}$
2	✓	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	✓	$\emptyset$
5	X	$\emptyset$
6	X	$\{0, 1, 2\}$
7	✓	$\emptyset$
8	✓	$\{8, 7\}$
9	X	$\{4\}$

[3,2,4,8,7]

Backtrack all the way back to node 3. All its descendants have been visited, so select the next unvisited (root level) node: node 6.



0	X	$\{1, 5\}$
1	X	$\{7\}$
2	✓	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	✓	$\emptyset$
5	X	$\emptyset$
6	✓	$\{0, 1, 2\}$
7	✓	$\emptyset$
8	✓	$\{4, 7\}$
9	X	$\{4\}$

[3,2,4,8,7,6]

Depth-first traversal gives a traversal that starts with

$$[3, 2, 4, 8, 7, 6, \ldots]$$

but this is not a topological sort since:

- node 2 is before node 6 in this list
- $6 \to 2$ is an edge!

Reversing the list to give

$$[\ldots, 6, 7, 8, 4, 2, 3]$$

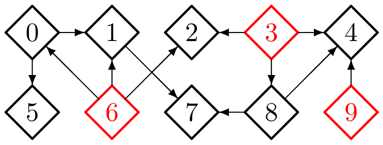
also fails to give a topological sort since 4 is before 3 in the list, but there is an edge $3 \to 4$.

2.2 Attempt 2

Will breadth-first traversal yield a topological sort for non-cyclic graphs?

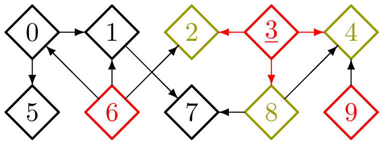
No, for similar reasons to the depth-first traversal.

We will start by adding all the root level nodes to the "to do" list.



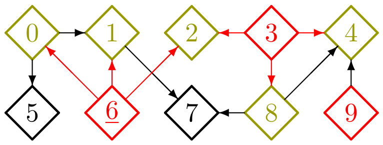
visited: []
to do: [3,6,9]

0	✓	{1, 5}
1	X	{7}
2	X	∅
3	X	{2, 4, 8}
4	X	∅
5	X	∅
6	X	{0, 1, 2}
7	X	∅
8	X	{4, 7}
9	X	{4}



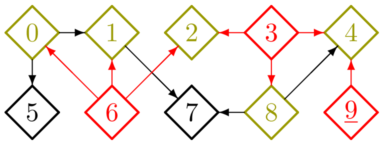
visited: [3]
to do: [6,9,2,4,8]

0	X	{1, 5}
1	X	{7}
2	X	∅
3	✓	{2, 4, 8}
4	X	∅
5	X	∅
6	X	{0, 1, 2}
7	X	∅
8	X	{4, 7}
9	X	{4}



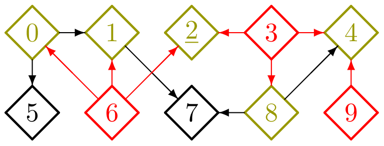
visited: [3,6]
to do: [9,2,4,8,0,1,2]

0	X	{1, 5}
1	X	{7}
2	X	∅
3	✓	{2, 4, 8}
4	X	∅
5	X	∅
6	✓	{0, 1, 2}
7	X	∅
8	X	{4, 7}
9	X	{4}



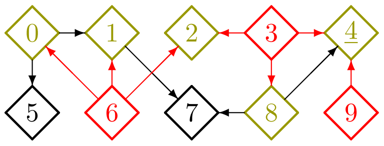
visited: [3,6,9]
to do: [2,4,8,0,1,2,4]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	✓	$\{0, 1, 2\}$
7	X	$\emptyset$
8	X	$\{4, 7\}$
9	✓	$\{4\}$



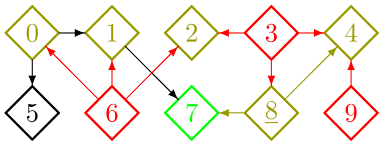
visited: [3,6,9,2]
to do: [4,8,0,1,2,4]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	✓	$\{0, 1, 2\}$
7	X	$\emptyset$
8	X	$\{4, 7\}$
9	✓	$\{4\}$



visited: [3,6,9,2,4]
to do: [8,0,1,2,4]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	✓	$\{0, 1, 2\}$
7	X	$\emptyset$
8	X	$\{4, 7\}$
9	✓	$\{4\}$



visited: [3,6,9,2,4,8]
to do: [0,1,2,4,7]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	✓	$\{2, 4, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	✓	$\{0, 1, 2\}$
7	X	$\emptyset$
8	X	$\{4, 7\}$
9	✓	$\{4\}$

Breadth-first traversal gives a traversal that starts:

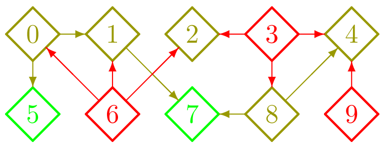
$$[3, 6, 9, 2, 4, 8, \ldots]$$

but

- node 4 is before node 8
- $8 \to 4$ is an edge

The reversed list $[\ldots, 8, 4, 2, 9, 6, 3]$ is even worse, since all the root level nodes come at the end of the list.

Do topological sorts exist? For our example:



a topological sort requires:

- node 0 occurs before nodes 1 and 5
- node 1 occurs before node 7
- node 3 occurs before nodes 2, 4 and 8
- node 6 occurs before nodes 0, 1 and 2
- node 8 occurs before nodes 4 and 7
- node 9 occurs before node 4

A topological sort:

$$[9, 6, 3, 8, 4, 2, 0, 5, 1, 7]$$

It is easy to check that this matches the requirements.

It can be shown that *all* non-cyclic graphs have at least *one* topological sort.

2.3 Existence of Topological Sorts

- Find a node with no (unsorted) successors. Such a node *must* exist since we are considering acyclic graphs. If every node has at least one successor we can continue along a path in the graph for as long as we wish by simply following one of these links. In particular we must be able to find a path that is longer than the number of nodes in the graph. On such a path at least one node must appear more than once — i.e. the path must contain a cycle. But then the graph is not acyclic.
- This node can safely be placed in the sort after any remaining unsorted nodes in the graph (and before any previously sorted ones).
- "Remove" this node and any links to it from the graph to give a simpler graph — one with fewer nodes.
- Repeat the process

This will give a topological sort.

2.4 Depth-first Topological Sort

```
the sort is empty
while (there are nodes left in the graph) {
   find a node with no successors;
   add it to the sort in front of any previously sorted nodes;
   remove it from the graph;
}
```

- `find a node with no successors`

  Use a depth-first traversal — when it backtracks the node has no (unvisited) successors.

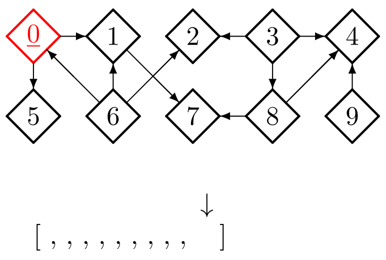
- `add it in front of any previously sorted nodes`

  Keep an index, starting with the last element of the sort, and decrease it each time a node is added, or use, e.g., a stack, or other list structure that adds elements at the head of the list.

- `remove it from the graph`

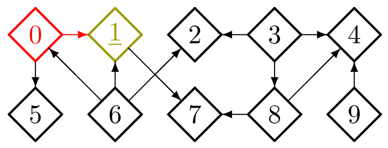
  Mark the node as visited.

We start at node 0 — it doesn't matter where we start. Note, however, that we do not yet mark it as visited. Nodes are only marked as visited once they are added to the topological sort.



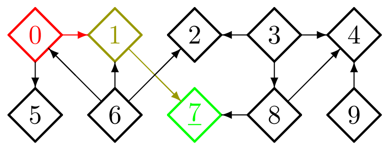
0	X	{1, 5}
1	X	{7}
2	X	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	X	∅
6	X	{0, 1, 2}
7	X	∅
8	X	{4, 7}
9	X	{4}

This shows the array in which the topological sort will be stored and the pointer to the end of the unused part of this array.



↓
[, , , , , , , , ,]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	X	$\{2, 4, 7, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	X	$\{0, 1, 2\}$
7	X	$\emptyset$
8	X	$\{4, 7\}$
9	X	$\{4\}$

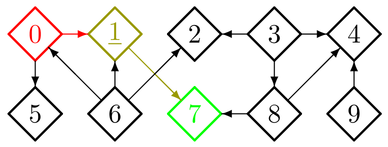


↓
[, , , , , , , , ,7]

0	X	$\{1, 5\}$
1	X	$\{7\}$
2	X	$\emptyset$
3	X	$\{2, 4, 7, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	X	$\{0, 1, 2\}$
7	✓	$\emptyset$
8	X	$\{4, 7\}$
9	X	$\{4\}$

7 has no (univisited) successors, so it is marked as visited and added to the topological sort.

Backtrack to node 1.

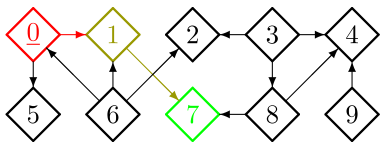


↓
[, , , , , , , ,1,7]

0	X	$\{1, 5\}$
1	✓	$\{7\}$
2	X	$\emptyset$
3	X	$\{2, 4, 7, 8\}$
4	X	$\emptyset$
5	X	$\emptyset$
6	X	$\{0, 1, 2\}$
7	✓	$\emptyset$
8	X	$\{4, 7\}$
9	X	$\{4\}$

1 has no (univisited) successors, so it is marked as visited and added to the topological sort.

Backtrack to node 0.

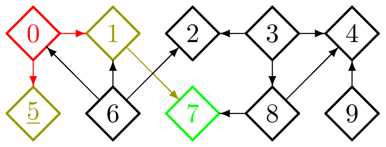


↓

[, , , , , , , ,1,7]

0	X	{1, 5}
1	✓	{7}
2	X	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	X	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Descend to node 5.



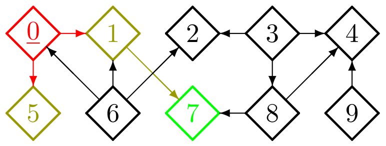
↓

[, , , , , , ,5,1,7]

0	X	{1, 5}
1	✓	{7}
2	X	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Node 5 has no unvisited successors.

Backtrack to node 0.



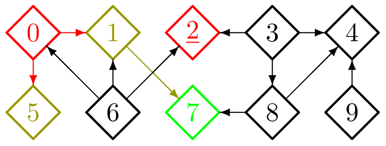
↓

[, , , , , ,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	X	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Node 0 has no unvisited successors.

No more nodes accessible from node 0 so visit next unvisited node — node 2.

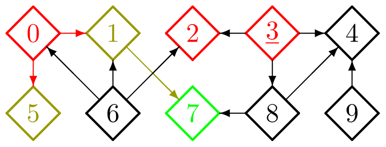


↓
[, , , , ,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Node 2 has no unvisited successors.

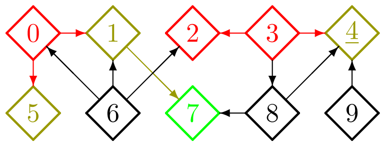
No more nodes accessible from node 0 so visit next unvisited node — node 3.



↓
[, , , , ,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	X	{2, 4, 7, 8}
4	X	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Node 2 has already been visited — descend to node 4.

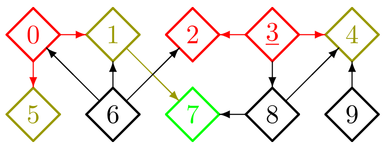


↓
[, , , ,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	X	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

Node 4 has no (unvisited) descendants.

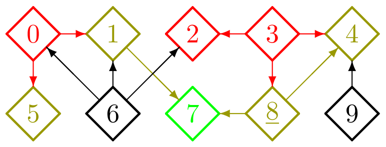
Backtrack to node 3.



$\downarrow$
[, , , ,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	X	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	X	{4, 7}
9	X	{4}

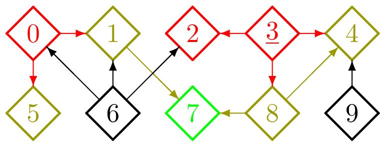
Descend to node 8.



$\downarrow$
[, , ,8,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	X	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	✓	{4, 7}
9	X	{4}

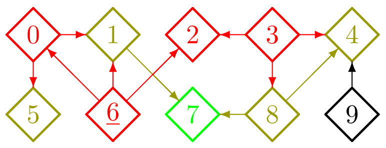
Node 8 has no unvisited successors:
Backtrack to node 3.



$\downarrow$
[, ,3,8,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	✓	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	X	{0, 1, 2}
7	✓	∅
8	✓	{4, 7}
9	X	{4}

Node 3 has no unvisited successors.
Node 3 has no unvisited successors — go to next unvisited node (node 6).



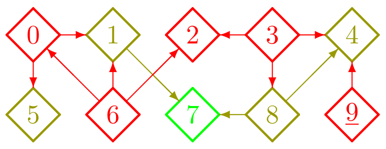
↓

[,6,3,8,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	✓	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	✓	{0, 1, 2}
7	✓	∅
8	✓	{4, 7}
9	X	{4}

Node 6 has no unvisited successors.

Node 6 has no unvisited successors — go to next unvisited node (node 9).



[9,6,3,8,4,2,0,5,1,7]

0	✓	{1, 5}
1	✓	{7}
2	✓	∅
3	✓	{2, 4, 7, 8}
4	✓	∅
5	✓	∅
6	✓	{0, 1, 2}
7	✓	∅
8	✓	{4, 7}
9	X	{4}

Node 9 has no unvisited successors.

The depth-first topological sort is

$$[9, 6, 3, 8, 4, 2, 0, 5, 1, 7]$$

It is easy to verify that this satisfies:

- node 0 occurs before nodes 1 and 5
- node 1 occurs before node 7
- node 3 occurs before nodes 2, 4 and 8
- node 6 occurs before nodes 0, 1 and 2
- node 8 occurs before nodes 4 and 7
- node 9 occurs before node 4

2.5 Reference-counting Topological Sort

In the depth-first topological sort we looked for nodes with no successors and placed them at the end of the topological sort. An alternative is to look for nodes with no predecessors and place them at the start of the topological sort.

- Find a node with no predecessors. Such a node must exist in an acyclic graph for the same reason that nodes with no successors must exist. Use the same reasoning, but construct the paths "backwards".
- Add it to the end of the current sort — i.e. after any previously sorted nodes, and before any of the currently unsorted nodes.
- "Remove" it and any links leaving it from the graph.
- Repeat the process until no nodes are left.

This process will terminate with a topological sort.

```
the sort is empty
while (there are nodes left in the graph) {
   find a node with no predecessors;
   add it after any previously sorted nodes;
   remove it from the graph;
}
```

- `find a node with no predecessors`

  Maintain a "reference count" count for each node. The reference count is the number of edges leading *into* this node. Look for nodes with a zero reference count.

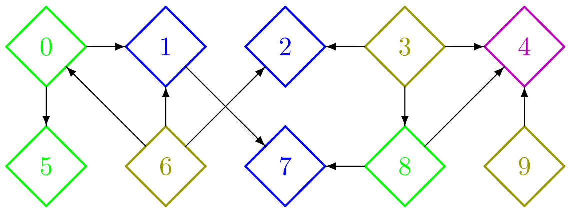
- `add it after any previously sorted nodes`

  Use an incrementing pointer, or use a list structure in which elements are added at the tail of the list (e.g. `add(element)` in `Lists`).

- `remove it from the graph`

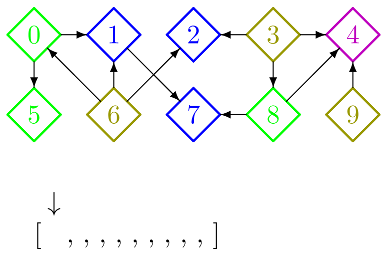
  Mark the node as visited, and decrease the reference count of any successors

The reference count is simply the number of edges leading in to a given node.



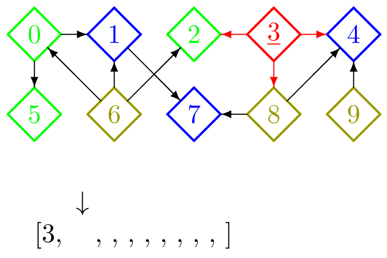
Node	0	1	2	3	4	5	6	7	8	9
Reference count	1	2	2	0	3	1	0	2	1	0

Start by calculating the reference counts for all the nodes.



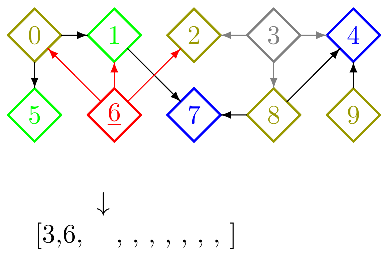
0	X	1	$\{1, 5\}$
1	X	2	$\{7\}$
2	X	2	$\emptyset$
3	X	0	$\{2, 4, 8\}$
4	X	3	$\emptyset$
5	X	1	$\emptyset$
6	X	0	$\{0, 1, 2\}$
7	X	2	$\emptyset$
8	X	1	$\{4, 7\}$
9	X	0	$\{4\}$

The first node with a reference count of zero is node 3. Mark it as visited and reduce the reference count of its successors: nodes 2, 4 and 8. Also add node 3 to the topological sort.



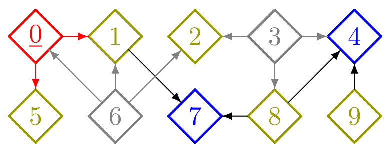
0	X	1	$\{1, 5\}$
1	X	2	$\{7\}$
2	X	1	$\emptyset$
3	✓	0	$\{2, 4, 8\}$
4	X	2	$\emptyset$
5	X	1	$\emptyset$
6	X	0	$\{0, 1, 2\}$
7	X	2	$\emptyset$
8	X	0	$\{4, 7\}$
9	X	0	$\{4\}$

The first unvisited node with a reference count of zero is node 6.



0	X	0	$\{1, 5\}$
1	X	1	$\{7\}$
2	X	0	$\emptyset$
3	✓	0	$\{2, 4, 8\}$
4	X	2	$\emptyset$
5	X	1	$\emptyset$
6	✓	0	$\{0, 1, 2\}$
7	X	2	$\emptyset$
8	X	0	$\{4, 7\}$
9	X	0	$\{4\}$

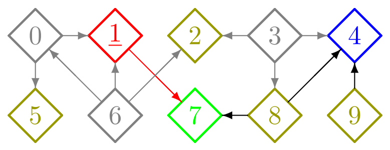
Now visit node 0 and reduce the reference count of nodes 1 and 5.



↓
[3,6,0, , , , , , ,]

0	✓	0	$\{1,5\}$
1	X	0	$\{7\}$
2	X	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	2	$\emptyset$
5	X	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	X	2	$\emptyset$
8	X	0	$\{4,7\}$
9	X	0	$\{4\}$

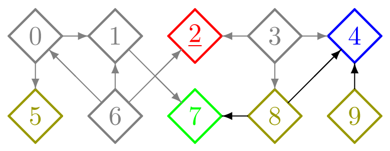
Now visit node 1, and reduce reference count for node 7.



↓
[3,6,0,1, , , , , ,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	X	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	2	$\emptyset$
5	X	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	X	1	$\emptyset$
8	X	0	$\{4,7\}$
9	X	0	$\{4\}$

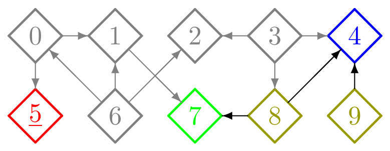
Now visit node 2.



↓
[3,6,0,1,2, , , , ,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	2	$\emptyset$
5	X	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	X	1	$\emptyset$
8	X	0	$\{4,7\}$
9	X	0	$\{4\}$

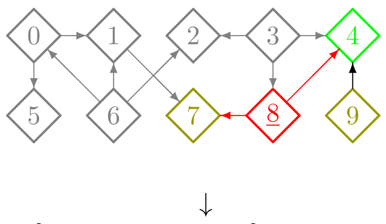
The next unvisited node with reference count zero is node 5.



↓
[3,6,0,1,2,5, , , ,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	2	$\emptyset$
5	✓	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	X	1	$\emptyset$
8	X	0	$\{4,7\}$
9	X	0	$\{4\}$

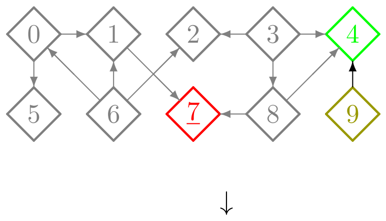
Now visit node 8 and reduce the reference count for 4 and 7.



[3,6,0,1,2,5,8, , ,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	1	$\emptyset$
5	✓	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	X	0	$\emptyset$
8	✓	0	$\{4,7\}$
9	X	0	$\{4\}$

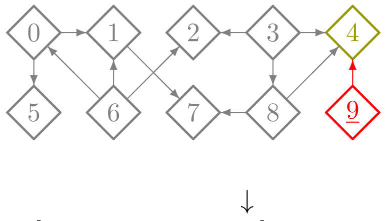
Visit node 7.



[3,6,0,1,2,5,8,7, ,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	1	$\emptyset$
5	✓	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	✓	0	$\emptyset$
8	✓	0	$\{4,7\}$
9	X	0	$\{4\}$

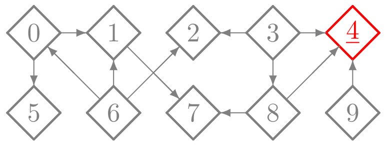
Next node to visit is node 9 — reduce count for 4.



[3,6,0,1,2,5,8,7,9,]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	X	0	$\emptyset$
5	✓	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	✓	0	$\emptyset$
8	✓	0	$\{4,7\}$
9	✓	0	$\{4\}$

Finally we visit node 4.



[3,6,0,1,2,5,8,7,9,4]

0	✓	0	$\{1,5\}$
1	✓	0	$\{7\}$
2	✓	0	$\emptyset$
3	✓	0	$\{2,4,8\}$
4	✓	0	$\emptyset$
5	✓	0	$\emptyset$
6	✓	0	$\{0,1,2\}$
7	✓	0	$\emptyset$
8	✓	0	$\{4,7\}$
9	✓	0	$\{4\}$

The reference-counting topological sort is

$$[3, 6, 0, 1, 2, 5, 8, 7, 9, 4]$$

It is easy to verify that this satisfies:

- node 0 occurs before nodes 1 and 5
- node 1 occurs before node 7
- node 3 occurs before nodes 2, 4 and 8
- node 6 occurs before nodes 0, 1 and 2
- node 8 occurs before nodes 4 and 7
- node 9 occurs before node 4

2.6 Notes

2.6.1 Topological Sorts are not Unique

Notice that topological sorts are not unique. In general, a non-cyclic graph will have many topological sorts. As we have seen in this example — the depth-first sort gave a different result to the reference-counting sort. Both were valid topological sorts.

2.6.2 Some Uses for Topological Sorts

- Compiler garbage collection — The data heap contains data being used, including pointers to data. Data may be valid (in use) or invalid. If we do a topological sort of the data heap any data items on the heap that occur before the known valid data items on the heap, must be garbage (since they are not referenced by any valid heap data items).
- Determining a course of study — modules can only be studied when their prerequisites have been passed.

3 Implementation

3.1 Depth First

3.1.1 Visiting a Node

On visiting a node:

- check if it has already been visited;
- if not:
  - note that it is being visited
  - visit its children;
  - add it to the sort;

Has the node been sorted?

```
if (visited.contains(node)) return;
```

Note that the node is being visited

```
visited.add(node);
```

Sort the node's children

```
for (T neighbour:  getNeighbours(node)) {
   visitNode(neighbour);
}
```

Add the node to the sort

```
sort.push(node); // use e.g. Stack
```

The visitNode method

```
private void visitNode(T node) {
   if (visited.contains(node)) return;
   visited.add(node);
   for (T neighbour:  getNeighbours(node)) {
      visitNode(neighbour);
   }
   sort.push(node);
}
```

3.1.2 The Full Sort

```
private Stack<T> sort = new Stack<T>();
private Set<T> visited = new HashSet<T>();

private List<T> getSort() {
   for (T node:  nodes()) {
      if (!visited.contains(node)) {
         visitNode(node);
      }
   }
   Collections.reverse(sort);
   return sort;
}
```

3.2 Reference Count

We need to:

- set up the reference counts;
- initialise the sort results;
- perform the sort.

3.2.1 Setting Up the Reference Counts

```
private void setUpReferenceCounts() {
   for (T node:  nodes()) {
      for (T successor:  successors(node)) {
         successor.increaseReferenceCount();
      }
   }
}
```

3.2.2 Initialise the Sort Results

```
private List<T> sort = new ArrayList<T>();
```

3.2.3 Visit a Node

On visiting a node we need to:

- add the node to the sort;
- reduce the reference count of its children.

Add the node to the sort

```
sort.add(node);
```

Reduce the reference count of the children

```
for (T successor:  successors(node)) {
   successor.decreaseReferenceCount();
}
```

The visitNode method

```
private void visitNode(T node) {
   sort.add(node);
   for (T successor:  successors(node)) {
      successor.decreaseReferenceCount();
   }
}
```

3.2.4 The Full Sorting

```
private void doSort() {
   T node;
   while ((node = nextReferenceZeroNode()) != null) {
      visitNode(node);
   }
}
```

End of topological sort lecture