CS 310 - Advanced Data Structures and Algorithms

Chapter 14 Graphs and Paths

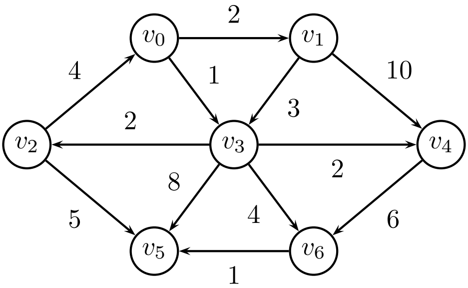
April 4, 2017

Graph – Definitions

- Graph – a mathematical construction that describes objects and relations between them
- A graph consists of a set of *vertices* and a set of *edges* that connect the vertices
- $G = (V, E)$ where V is the set of vertices (nodes) and E is the set of edges (arcs)
- In a *directed graph*, each edge is an ordered pair (u, v) where $u, v \in V$
- In an *undirected graph*, each edge is a set $\{u, v\}$
- For *weighted* graphs (directed or undirected), each edge is associated with a weight W
- Vertex v is *adjacent* to vertex u if and only if $(u, v) \in E$ for a directed graph, or $\{u, v\} \in E$ for an undirected graph

A Directed Graph Example

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$
$$E = \{(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10),$$
$$(V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2),$$
$$(V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1)\}$$

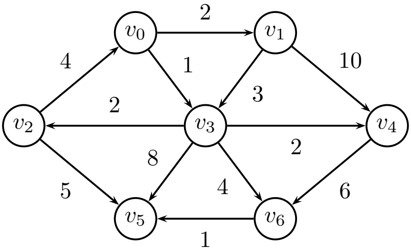


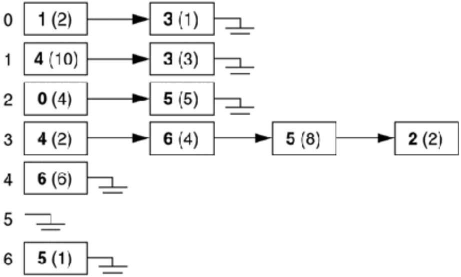
Definitions

- Path: a sequence of vertices $v_1, \ldots, v_n$ connected by edges such that $\{v_i, v_{i+1}\} \in E$ for each $i = 1, \ldots, n$
- Number of vertices: n
- Number of edges: m
- Path length: the number of edges on the path
- Weighted path length: in a weighted graph, the sum of the costs of the edges on the path
- Cycle: a path that begins and ends at the same vertex and contains at least one edge

Graph Representation

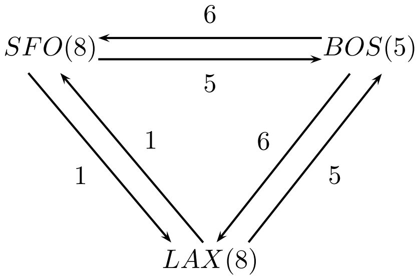
- Use a 2-dimensional array called *adjacency matrix*, `a[u][v]` = edge cost
- Nonexistent edges initialized to ∞
- For *sparse* graphs, use an *adjacency list* that contains a list of adjacent indices and weights



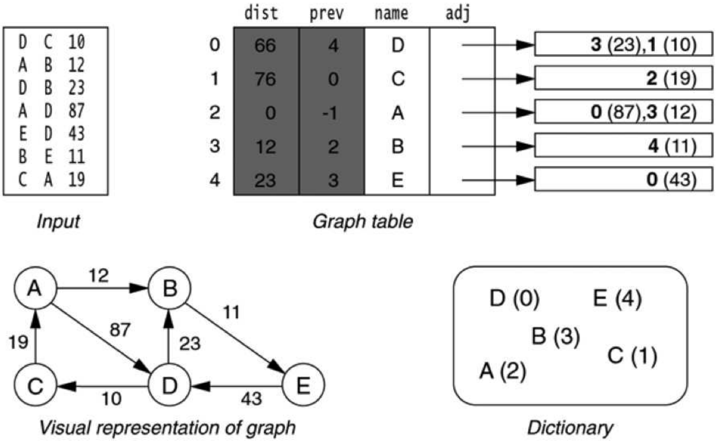


Example – Traveling Between Cities

- Consider airports LAX, SFO, and BOS with edges with integer weights that are hours of flight time: (LAX, SFO, 1), (SFO, LAX, 1), (LAX, BOS, 5), (BOS, LAX, 6), (SFO, BOS, 5), (BOS, SFO, 6)
- This is a directed graph



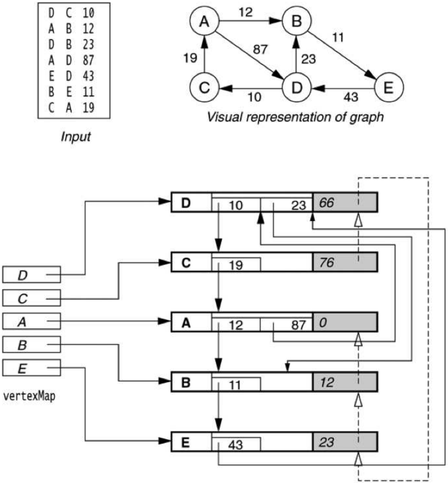
Representation and Shortest Weighted Path



The shortest weighted path from A to C is A to B to E to D to C

Figure 14.5

- Vertices
  - The previous slide uses 0, 1, 2, 3, 4 as internal vertex numbers
  - This picture uses object *references* as internal numbers
- Adjacency lists are used
- Edge object contains an internal vertex number or a Vertex reference, and an edge cost
- Shaded items are computed by the shortest path algorithms



Adjacency Matrix or Adjacency List

Comparison	Winner
Test if (x, y) is in graph?	adjacency matrix
Find the degree of a vertex?	adjacency list
Less memory on small graphs?	adjacency list $\Theta(m+n)$ vs. $\Theta(n^2)$
Less memory on big graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency list $\Theta(m+n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency list

Graph Traversal

- The most fundamental graph problem is to visit every edge and vertex in a graph in a systematic way
- Key idea: Mark each vertex when we first visit it, and keep track of what we have not completely explored
- Each vertex is in one of three states:
  1. Undiscovered
  2. Discovered: The vertex has been found, but we have not yet checked out all its incident edges
  3. Processed: We have visited all its incident edges
- A data structure is maintained to hold the vertices that we have discovered but not yet completely processed
  - A queue for BFS
  - A stack for DFS

Outline of Graph Traversal

- Initially, only the start vertex s is considered to be discovered
  - Put s in the data structure
- Remove a vertex u from the data structure of discovered vertices
- Inspect every edge incident upon u
- If an edge leads to an undiscovered vertex v, mark v as discovered and add it to the data structure
- If an edge lead to a processed vertex, ignore this edge
- If an edge leads to a discovered but not processed vertex, ignore this edge

Outline of Graph Traversal

- Assume the graph is connected
- For an undirected graph, each edge will be considered exactly twice
  - For edge $\{u, v\}$, going from u to v, and from v to u
- For a directed graph, each edge will be considered only once
- Every edge and vertex must eventually be visited

Breadth-First Search

- BFS
- When searching an undirected graph by breadth-first, we assign a direction to each edge, from the discoverer u to the discovered v
- Vertex u is the *parent* of vertex v
- The start vertex is the root of the search tree
- All other vertices have exactly one parent

BFS of Undirected Graphs

- The BFS tree defines a shortest path from the root to every other vertex in the tree
- Let the tree be drawn with the root at the top
- Non-tree edges of the graph can point only to vertices on the same level (horizontally), or to vertices on the next level (immediately below)
- For *directed* graphs, BFS tree may have back-pointing edges

Implementation of BFS

- Associate with a `Vertex` object
  - A `status` data member, with possible values of `undiscovered`, `discovered`, and `processed`
  - A `parent` data member
- Initialize a queue to hold only the start vertex
- Mark the start vertex as `discovered`, with no parent
- Loop
  - Dequeue a vertex u, mark it as `processed`
  - Loop through the adjacency list of u for each of the edges (u, v)
    - If v is `undiscovered`, mark v as `discovered` and enqueue, and make u the parent of v
- $O(n + m)$ running time
- Question: In the inner loop, is it possible that vertex v is already `discovered` or `processed`?

Shortest Paths in Unweighted Graphs

- The BFS tree gives the shortest paths from the *root* to all other vertices in unweighted graphs
- To print the shortest path from root to x
    - Start from x
    - Follow the parent link by recursion, and print the vertex itself after recursion comes back
    - Stop recursion when there is no parent (that is, the root)

BFS Application: Connected Components

- A graph is *connected* if there is a path between any two vertices
- A *connected component* of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices
- The question of whether a puzzle such as Rubik's cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected
- To find connected components, do BFS and obtain one component, and then repeat with the remaining vertices until all vertices appear in a component
- For directed graphs, there are weakly connected and strongly connected components, to be considered later

Graph Coloring

- The graph coloring problem seeks to assign a color to each vertex of a graph in such a way that no edge links two vertices of the same color
- Trivial solution: Assign a unique color to each vertex
- Serious solution: Use as few colors as possible
- The smallest number of colors that must be used is called the *chromatic number* of the graph, $\chi(G)$
- The chromatic number problem, whether a graph is k-colorable (a decision problem), is NP-complete
- Finding a k-coloring of a grpah is NP-hard
- Vertex coloring arises in scheduling applications, such as register allocation in compilers

Register Allocation in Compilers

- A compiler first translates source code to intermediate code that uses many temporary variables
- Later it assigns temporary variables to registers
- Source code

```
e = (a + b) * c - d;
```

Intermediate code

```
t1 = a + b
t2 = t1 * c
e = t2 - d
```

Register assignment

```
R1 = R2 + R3
R1 = R1 * R4
R1 = R1 - R5
```

Register Assignment

- A temporary variable becomes dead when its value is no longer needed
- Two temporary variables cannot be allocated to the same register if they are live at the same time
- Construct an undirected graph
- Use a node for each temporary variable
- There is an edge between two nodes if they are live simultaneously somewhere in the program
- This is the *register interference graph*
- Let k be the number of registers
- The problem becomes how to k-color the graph

Graph Coloring Heuristic

- To k-color a graph
- Repeat till the graph becomes empty
  1. Pick a node u with fewer than k neighbors
  2. Remove u and all edges incident on u from the graph
  3. Push u into a stack
- If the smaller graph is k-colorable, so is the original one
- Reason: After we obtain a k-coloring of the smaller graph, we can assign u a color that is not used by its neighbors
- Therefore, repeat till the stack becomes empty
  1. Pop a node u from the stack
  2. Assign u a color not used by its neighbors

Bipartite Graphs

- A graph is bipartite if it is 2-colorable
- How to determine if a graph is bipartite?
- BFS, and color a child vertex the opposite of its parent
- If a *nondiscovery edge* connects two vertices of the same color, the graph cannot be two-colored
  - $O(n + m)$ runtime
  - Computationally easy to determine whether a graph is 2-colorable
- The problem becomes NP-hard when we want to determine whether a graph is k-colorable for $k \geq 3$

Depth-First Search

- DFS
- Replace the queue (FIFO) in BFS by a stack (LIFO)
- Instead of using a real stack, we can just use the recursive calls where the runtime system maintains the call stacks
- An edge of the graph is either a DFS *tree edge* or a *back edge* linking to an *ancestor* vertex
- That is, an edge is either a forward edge or a backward edge
- No edges exist between siblings
- $O(n + m)$ runtime

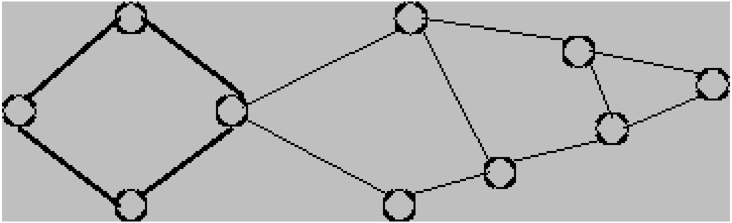
DFS Traversal Timestamp of a Vertex

- Let's keep a clock during DFS
- The clock ticks each time we enter or exit a vertex
- Each vertex has two additional data members: entry time and exit time
- If u is an ancestor of v, the entry and exit times of u properly encompass those of v
  - Entering u earlier than v
  - Exiting u later than v
- Half of the difference between the entry and exit times is the number of decendents in the DFS subtree

DFS Application: Finding Cycles in Undirected Graphs

- DFS of an undirected graph visits each edge (u, v) twice
- When going from u to v
  1. If v is undiscovered, the edge (u, v) becomes a tree edge
     - We make a recursive DFS call
  2. If v is discovered and the parent of u, we would be following the tree edge backwards to where we just came from
     - So don't go there – just move on to the next edge in the adjacency list
  3. If v is discovered but not the parent of u, we have found a cycle
     - We can print the cycle using the `parent` links
- The second case is a spurious two-vertex cycle (v, u, v)

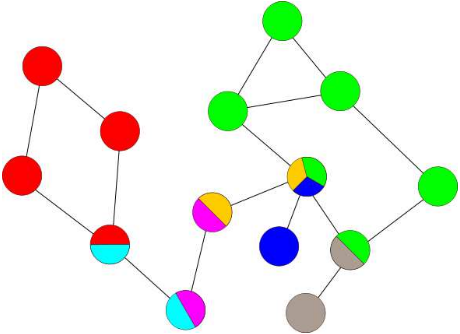
Articulation Vertices



- If you are allowed to remove only one vertex from the network, which vertex would you remove to cause the largest disruption to the network?
- A vertex is called an *articulation vertex* or a *cut-node* if its deletion disconnects a connected component of an undirected graph

Connectivity

- We can use BFS to find connected components of an undirected graph
- The *connectivity* of a graph is the smallest number of vertices whose deletion will disconnect the graph
- The connectivity is one if the graph has an articulation vertex
- A graph is *biconnected* if it remains connected after deletion of any single vertex



Wikipedia

Finding an Articulation Vertex

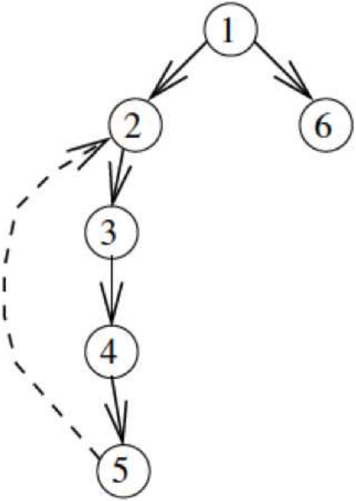
- Brute force
  - Temporarily delete each vertex v
  - Do BFS or DFS to see if the modified graph is still connected
  - $O(n(n+m))$ runtime
- Faster algorithm: Use DFS in $O(n+m)$ time

Articulation Vertices in a Tree

- Assume the graph is a *tree*
- Let's do a DFS from a vertex
- Observations:
- All leaf nodes of the DFS tree are *not* articulation vertices
  - When removed, a leaf node disconnects only itself
- The root of the search tree is an articulation vertex if it has two or more children
  - If the root of the search tree has only one child, it is like a leaf node, hence, not an articulation vertex
- All other internal nodes of the search tree are articulation vertices

Back Edges of DFS as Safety Cables

- DFS of an *undirected graph*
- Vertices 1 and 2 are cut-nodes
- The back edge $(5, 2)$ acts like a safety cable that keeps vertices 3 and 4 from being cut-nodes
- To capture the idea of safety cables, we need to keep track of the most ancient ancestor a vertex can reach via a back edge during DFS
- We also keep track of the number of children in the DFS tree for each vertex

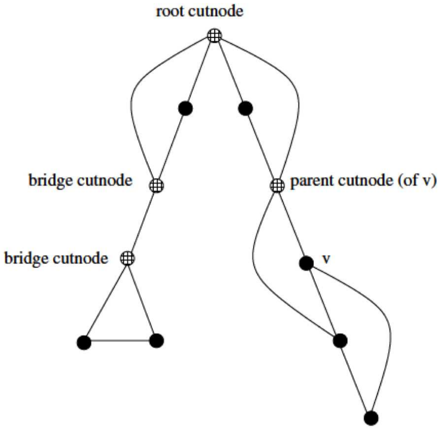


Reachable Ancestor When Entering a Vertex in DFS

- Remember: during DFS of an undirected graph, an edge is either a forward edge or a backward edge
- During DFS, when we first enter vertex u
  1. Set `reachableAncestor[u]` to u
  2. Set `numChildren[u]` to 0
  3. Set entry time
- When we go through the adjacency list of u, for each edge (u, v)
  1. If v is being discovered – that is, (u, v) is a tree edge – increment `numChildren[u]` by 1
  2. Otherwise, if v is not the parent of u
     - If entry time of v is earlier than the entry time of `reachableAncestor[u]`, set `reachableAncestor[u]` to v

Reachable Ancestors Help Find Articulation Vertices

- Root cut-nodes: If the root of the DFS tree has two or more children, it is an AV
- Bridge cut-nodes: If the earliest reachable ancestor of u is u, then deleting the edge (parent[u], u) disconnects the graph
  - Parent of u is an AV
  - u is an AV if it is not a leaf
- Parent cut-nodes: If the earliest reachable ancestor of v is the parent of v, and if the parent of v is not the root, then the parent of v is an AV



Reachable Ancestor When Exiting a Vertex in DFS

- When exiting a vertex u during DFS

1. If u does not have a parent, it is the root
   1. If the root has two or more children, it is a root cut-node
   2. Return
2. If the parent of u is not the root
   1. If `reachableAncestor[u]` is parent of u, then parent of u is a parent cut-node
   2. If `reachableAncestor[u]` is u itself, then parent of u is a bridge cut-node, and if u is not a leaf (no child), u itself is also a bridge cut-node
3. Housekeeping: propragate the reachable ancestor upwards
   - If the entry time of `reachableAncestor[u]` is earlier than the entry time of `reachableAncestor[parent[u]]`, set `reachableAncestor[parent[u]]` to `reachableAncestor[u]`

Vertex Connectivity and Edge Connectivity

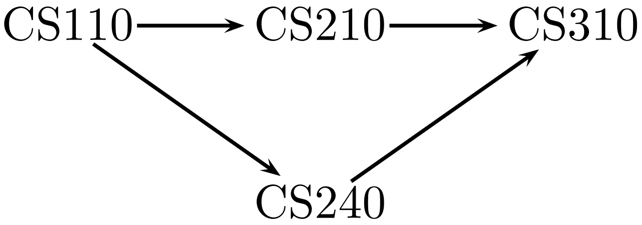
- Up to this point we are concerned with vertex connectivity only: whether the graph becomes disconnected after deleting a vertex
- We can look at edge connectivity as well: whether the graph becomes disconnected after deleting an edge
- This can be achieved in $O(n + m)$ time with slight modification of the algorithm in the proceeding slides

DFS on Directed Graphs

- Recall that during DFS of an *undirected* graph, every edge in the graph is either a tree edge or a back edge to an ancestor in the DFS tree
  - Each edge is encountered twice
- During DFS on a *directed* graph, an edge (u, v) is
  1. Tree edge: `parent[v]` is u
  2. Back edge: `discovered[v] && !processed[v]`
  3. Forward edge: `processed[v] && entryTime[v] > entryTime[u]`
  4. Cross edge: `processed[v] && entryTime[v] < entryTime[u]`
- The same DFS algorithm works for both undirected and directed graphs
- For directed graphs, each edge is encountered once

Acyclic Directed Graph (DAG)

- A cycle in a directed graph is a path that returns to its starting vertex
- An acyclic directed graph is also called a DAG
- These graphs show up in lots of applications
    - Example, the graph of course prerequisites
- It is a DAG, since a cycle in prerequisites would be ridiculous



DAGs

- A DAG induces a *partial order* on the nodes
  - Not all element pairs have an order, but some do, and the ones that do must be consistent
- So CS110 $<$ CS210 $<$ CS310, and so CS110 $<$ CS310, but CS210 and CS240 have no order between them
- Suppose a student takes only one course per term in CS
- A sequence that satisfies the partial order requirements, for example, is CS110, CS210, CS240, CS310
- Another possible sequence is CS110, CS240, CS210, CS310
- One of these fully ordered sequences that satisfy a partial order (DAG) is called a *topological sort* of the DAG

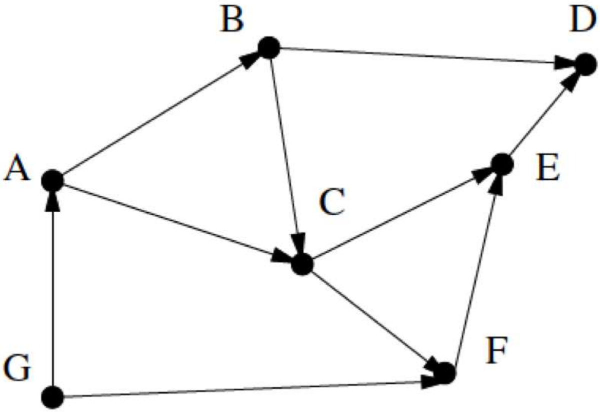
Topological Sort

- A topological sort orders the nodes such that if there is a path from u to v, then u will appear before v
- The vertices of the graph are ordered on a straight line such that all edges point from left to right
- Each DAG has at least one topological sort
- A topological sort gives us an ordering to process each vertex before any of its successors
- Example: An assembly line
- Example: A compiler that optimizes for instruction-level parallelism (ILP) may shuffle instructions but must respect the partial order (time-dependency) among the instructions

Use DFS to Find a Topological Sort

- Start DFS from a vertex with *in-degree* 0
- A back edge during DFS indicates that there is a loop
  - The graph is not DAG
- Ordering the vertices in the *reverse order* that they are marked *processed* is a topological sort
- Consider the edge (u, v) when we process the vertex u
  1. If v is *undiscovered*, we will start a recursive DFS from v. So v will be completely *processed* before u. So u will appear before v in the listing, as it must.
  2. If v is *processed*, and because u is yet completely processed, u will appear before v in the listing, as it must.
  3. If v is *discovered* but not yet completely *processed*, then (u, v) is a back edge, which is forbidden in a DAG.
- How to print this topological sort in linear time?

DFS Example



There is only one topological sort of this graph: (G, A, B, C, F, E, D)

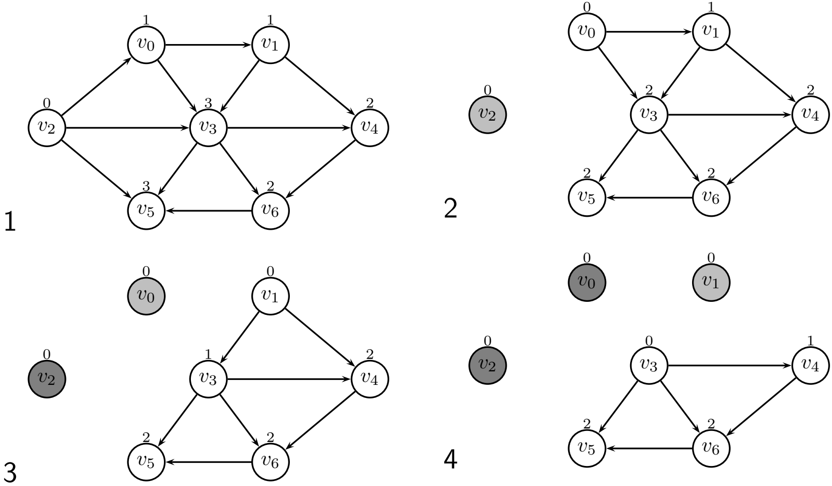
Another Method to Find a Topological Sort

- The textbook presents a non-recursive algorithm for finding a topological sort of a DAG, checking that it really has no cycles
- The first step of this algorithm is to determine the in-degree of all vertices in the graph
- The *in-degree* of a vertex is the number of edges in the graph with this vertex as the destination-vertex
- Once we have all the in-degrees for the vertices, we look for a vertex with in-degree 0
- Because it has no incoming edges, it can be the vertex at the start of a topological sort, like CS110

Finding a Topological Sort

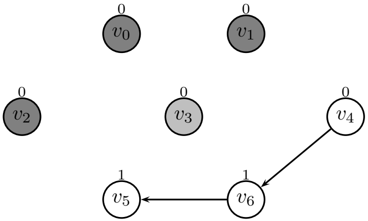
- Note that there must be a node with in-degree 0
  - If there weren't, then we could start a path anywhere, extend backwards along some in-edge from another vertex and from there to another, etc
  - Eventually we would have to start repeating vertices
- For example, if we have managed to avoid repeating vertices and have visited all the vertices, then the last vertex still has an in-edge not yet used, and it goes to another vertex, completing a cycle
- Thus the lack of an in-degree-0 vertex is a sure sign of a cycle and a DAG doesn't have any cycles
- Now we have the very first vertex, but what about the rest? Think recursively!

Topological Sort Example

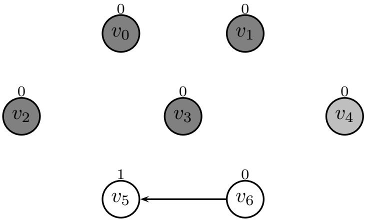


Topological Sort Example

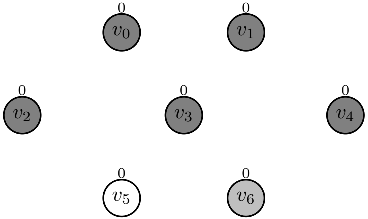
5



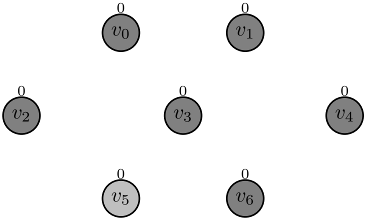
6



7



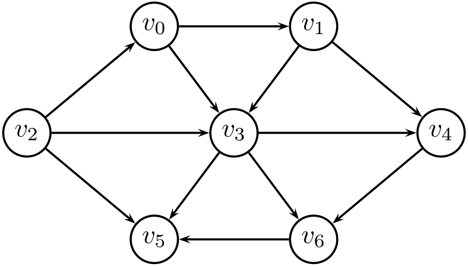
8



The topological order is: $V_2, V_0, V_1, V_3, V_4, V_6, V_5$

Topological Sorting Using an Array of In-degree Values

V_0	V_1	V_2	V_3	V_4	V_5	V_6	output
1	1	0	3	2	3	2	V2
0	1	0	2	2	2	2	V0
0	0	0	1	2	2	2	V1
0	0	0	0	1	2	2	V3
0	0	0	0	0	1	1	V4
0	0	0	0	0	1	0	V6
0	0	0	0	0	0	0	V5



Pseudocode for Topological Sort

1. Create a queue q and enqueue all vertices of in-degree 0
2. Create an empty list t for topologically-sorted vertices
3. Loop while q is not empty
    1. Dequeue from q a vertex u and append it to list t
    2. Loop over vertices v adjacent to u
    3. Decrement in-degree of v
    4. If v's in-degree becomes 0, enqueue v in q
4. Return topologically-sorted list t

What Happens If There is a Cycle?

- The presence of a loop means in-degree is at least 1 for all nodes of the cycle, so the cycle protects the whole group from being put on the queue
- For example, consider $A \rightarrow B \rightarrow C \rightarrow A$ and see in-degree $= 1$ for all nodes
- Each pass of the loop dequeues an element from the queue, so eventually the queue goes empty with the cycle members still un-enqueued
- Use a counting trick to add cycle-detection to this algorithm

DFS – Some Comments

- We can do DFS of any directed (or undirected) graph, and if it's acyclic, DFS yields a topological sort
- If there is a cycle in the graph, it doesn't cause an infinite loop because DFS doesn't revisit a vertex
- In general DFS works in phases, finding trees, so the whole thing finds a forest
- The trees in the forest may have inter-tree edges back to trees that are finished earlier, so they need to be ordered last-first in topological-sort order
- The ability of DFS to turn a graph into a forest of trees is useful in many algorithms
- Trees are a lot easier to work with than general graphs
- Note that a graph does not have to be acyclic to do a DFS – it's just that you can only get a topological sort out of it if it's acyclic

DFS versus BFS

- DFS is like preorder tree traversal, plunging further and away from the source node until we can't go any further, then back
    - Can detect cycles
    - Can do topological sort of an acyclic graph
- BFS: explore nodes adjacent to the source node, then nodes adjacent to those (that haven't been visited yet), and so on
    - Good for finding all neighbors, all neighbors of neighbors, etc. – hop counts

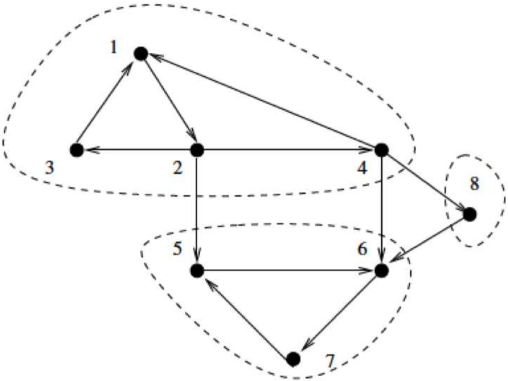
Strongly Connected Components

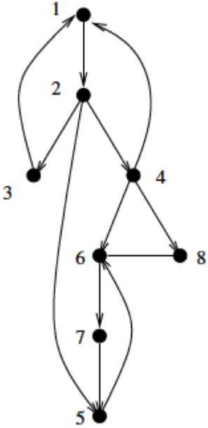
- A *directed graph* is:
  - *weakly connected* if replacing all directed edges with undirected edges leads to a connected undirected graph
  - *connected* if there is a directed path from u to v *or* a directed path from v to u for every pair of vertices u and v
  - *strongly connected* if there is a directed path from u to v *and* a directed path from v to u for every pair of vertices u and v
- Often we are interested in *strongly connected components* of a directed graph, that is, partitioning the graph into components that are strongly connected

How to Determine If a Graph is Strongly Connected

- Arbitrarily choose a starting vertex u from a graph $G = (V, E)$
- Do a traversal (BFS or DFS) from u
  - Every vertex had better be reachable from u, otherwise G is not strongly connected
- Construct a graph $G' = (V, E')$ such that the edges in E' are reversed of the edges in E
  - $(u, v) \in E$ if and only if $(v, u) \in E'$
- Do a DFS from u in G'
  - A path from u to v in G' corresponds to a path from v to u in G
  - By doing DFS from u in G', we find all vertices with paths to u in G

Example of Strongly Connected Components





Find Strongly Connected Components

- The idea is similar to finding articulation vertices in an undirected graph
- Do DFS
- Modify the notion of oldest reachable ancestor, considering both back edges and cross edges
  - Forward edges have no impact on reachability over the DFS tree edges
- Two data members of u
- Let `low[u]` be the oldest (entry time) vertex known to be in the same strongly connected component as u
  - `low[u]` may be an ancestor or a distant cousin due to cross edges
- Let `scc[u]` be the number of the strongly connected component of u, initialized to `-1`

Find Strongly Connected Components

- During DFS, when we encounter a non-tree edge (u, v)
- If it is a forward edge, do nothing
- If it is a back edge
  - If `entryTime[v] < entryTime[ low[u] ]`, set `low[u] = v`
- If it is a cross edge
  - If `scc[v] != -1`, v belongs to a SCC that has already been found, and thus u cannot join them
  - If `scc[v] == -1` and `entryTime[v] < entryTime[ low[u] ]`, set `low[u] = v`

Find Strongly Connected Components

- During DFS, on entering a vertex u, push u into a stack
- On leaving a vertex u
  - We need to propogate `low[u]` back up the DFS tree – if `entryTime[ low[u] ] < entryTime[ low[ parent[u] ] ]`, set `low[ parent[u] ] = low[u]`
  - If `low[u] = u`, we have found a SCC (the vertices from the top of the stack down to u)
    - To output the SCC
    - Increment `numSCC` by 1
    - Pop the stack and get a vertex, v
    - Set `scc[v] = numSCC`
    - Repeat until `v == u`