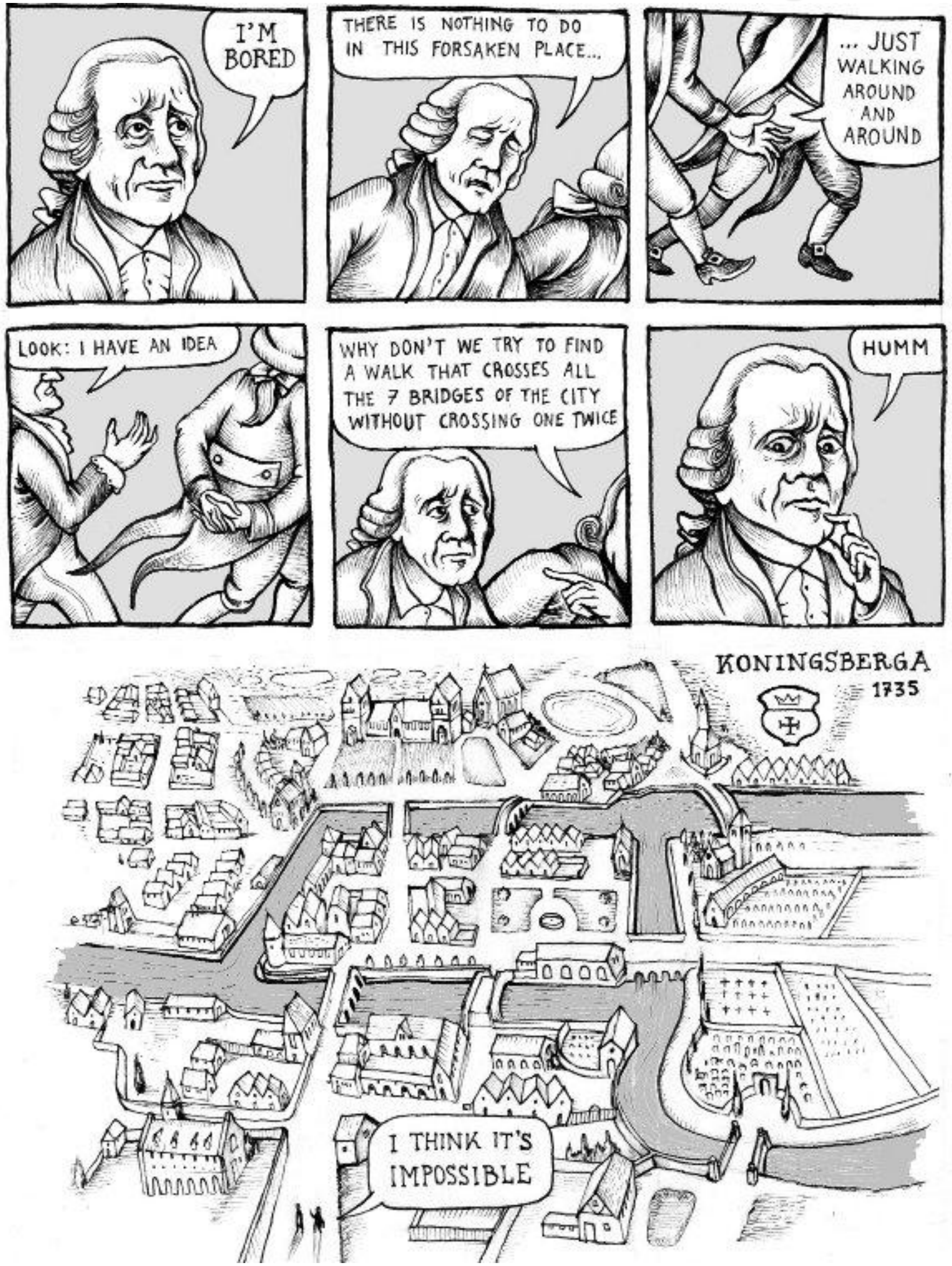
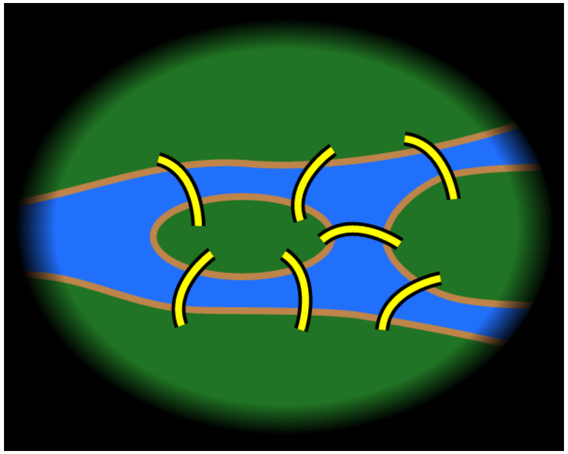
Graphs
(chapter 6)

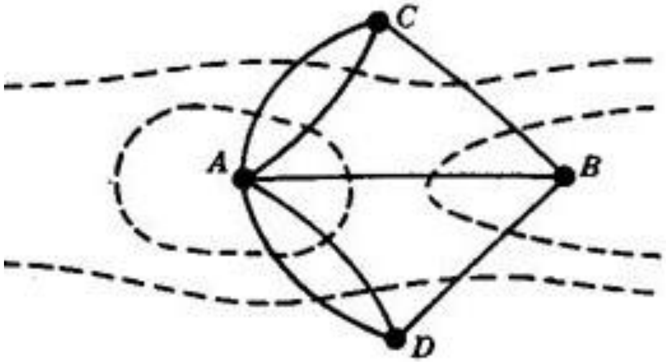
- Graph definition and terminology
- Graph representation
- Search
- Connected components
- Minimum cost spanning trees
- Shortest paths and transitive closures
- Activity networks

A Graph Example

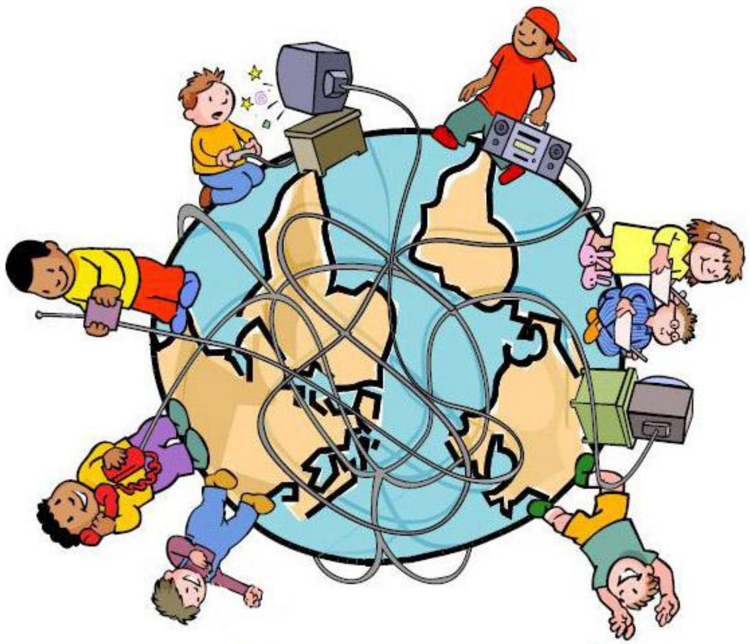


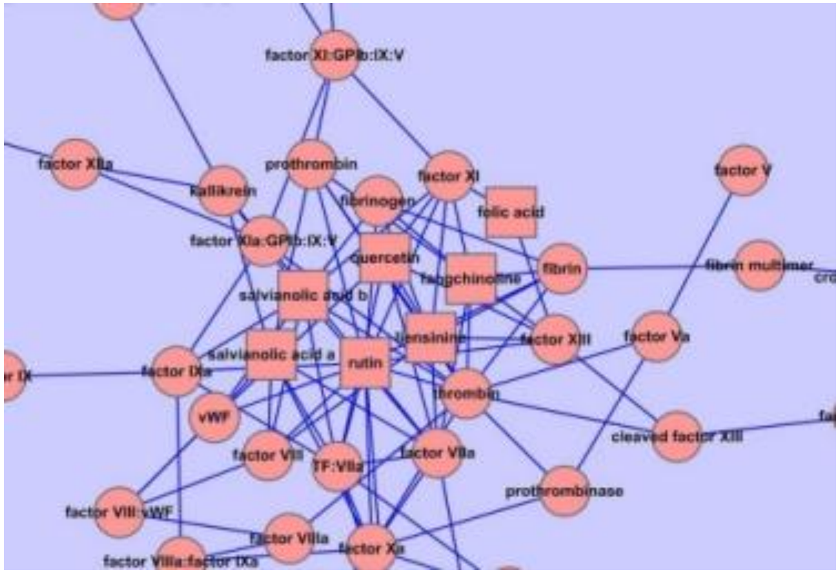
This is the first recorded evidence of the use of graphs (1736): The Konigsberg bridge problem by Euler.

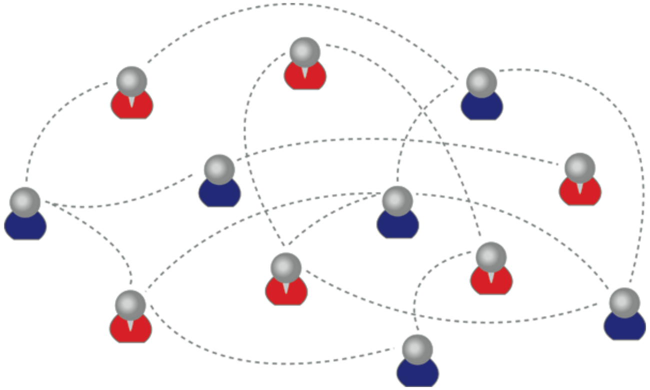




More Example Graphs





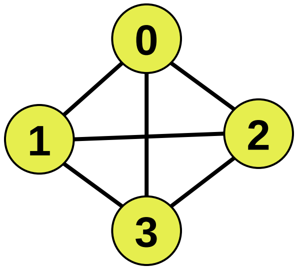


Definition of Graphs

- A **graph** G consists of two sets:
  - V or V(G): A nonempty set of **vertices**.
  - E or E(G): A set of "pairs of vertices" called **edges**.
  - We usually use the expression G(V,E) to represent a graph with vertices V and edges E.
- **Undirected graph**: The edges do not have particular directions; for vertices u and v, (u,v) and (v,u) represent the same edge.
- **Directed graph**: Edges have particular directions; for vertices u and v, <u,v> represents the edge u→v and <v,u> represents the edge v→u.

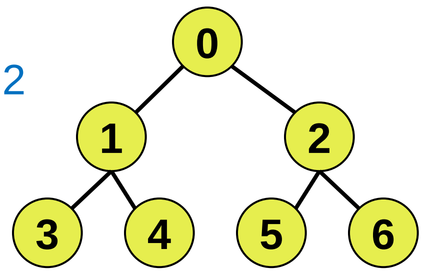
Vertices and Edges

G_1



$V(G_1)$: {0, 1, 2, 3}
$E(G_1)$: {(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}

G_2



$V(G_2)$: {0, 1, 2, 3, 4, 5, 6}
$E(G_2)$: {(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}

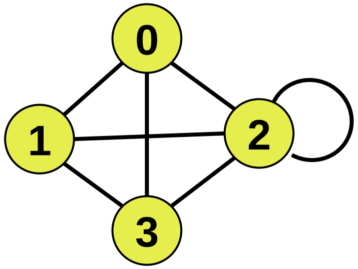
G_3

$V(G_3)$: {0, 1, 2}
$E(G_3)$: {<0,1>, <1,0>, <1,2>}

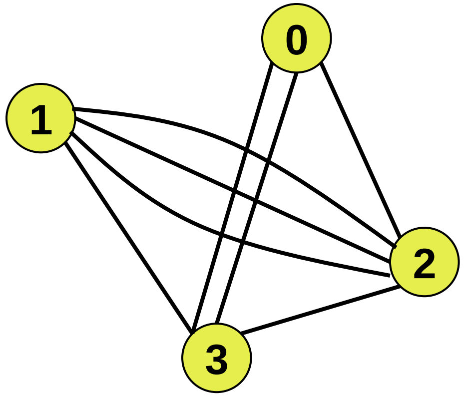
More Restrictions on Edges

We will only consider graphs with these properties:

- There is no edge that points from a vertex back to itself; i.e., edges like (v,v) or <v,v> are not allowed.
  - Such edges, if allowed, are called self edges or self loops.



- There can not be multiple occurrences of the same edge.
  - Otherwise, the graph is called a multigraph.



Terminologies and Properties

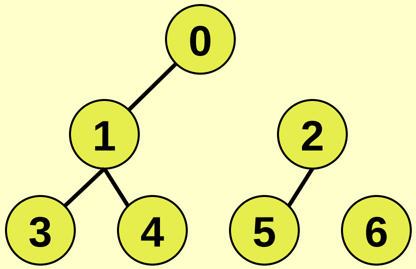
- For a <u>undirected graph</u> with n vertices, there are at most $n(n-1)/2$ edges.
- For a <u>directed graph</u> with n vertices, there are at most $n(n-1)$ edges.
- A complete graph is a graph that contains an edge from every vertex u to every vertex v ($u \neq v$).
- $G'(V',E')$ is a subgraph of $G(V,E)$ **iff** $V' \subseteq V$ and $E' \subseteq E$.

Terminologies and Properties

- If there is an edge between two vertices u and v, then u and v are adjacent. The edge is said to be incident on u and v.
- A path from vertex w_1 to vertex w_k is a series of vertices w_1, w_2, …, w_k, such that all $(w_i, w_{i+1}) \in E$. The path length is the number of edges in the path.
- A simple path is a path whose vertices (other than the first and last vertices) are distinct.
- A cycle is a path whose first and last vertices are the same.
- An acyclic graph is a graph containing no cycles.

Terminologies and Properties

- An undirected graph is connected if there exists a path between every pair of distinct vertices u and v.
- A directed graph is strongly connected if there exists a path from u to v for every pair of distinct vertices u and v. (Paths exist from u to v and from v to u.)
- A connected component of a graph is a maximally connected subgraph.

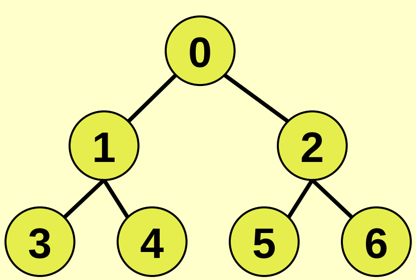


This graph has three connected components.

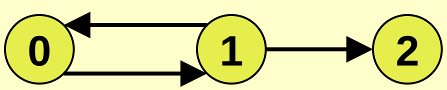
Terminologies and Properties

- The degree of a vertex is the number of edges incident on that vertex.
- For a directed graph:
  - The in-degree of vertex u is the number of edges pointing to u (i.e., in the form <w,u>)
  - The out-degree of vertex u is the number of edges pointing from u (i.e., in the form <u,w>).
  - degree = in-degree + out-degree

Q: Give the degrees of the vertices in this graph:

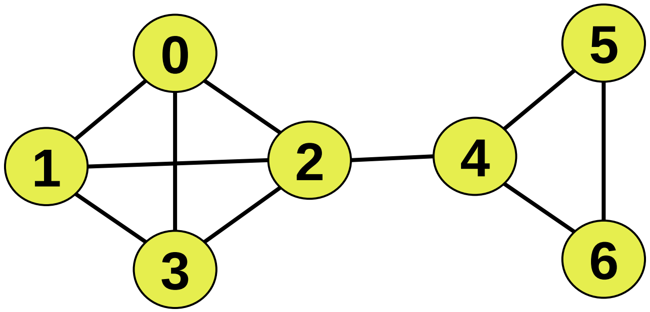


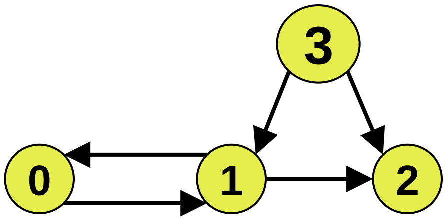
Q: Give the in-degrees and out-degrees of the vertices in this graph:



Graph Representation: Adjacency Matrix

- A square matrix of **0**'s and **1**'s. Element (a,b) being **1** indicates an edge from a to b.
- An adjacency matrix is symmetric for undirected graphs.





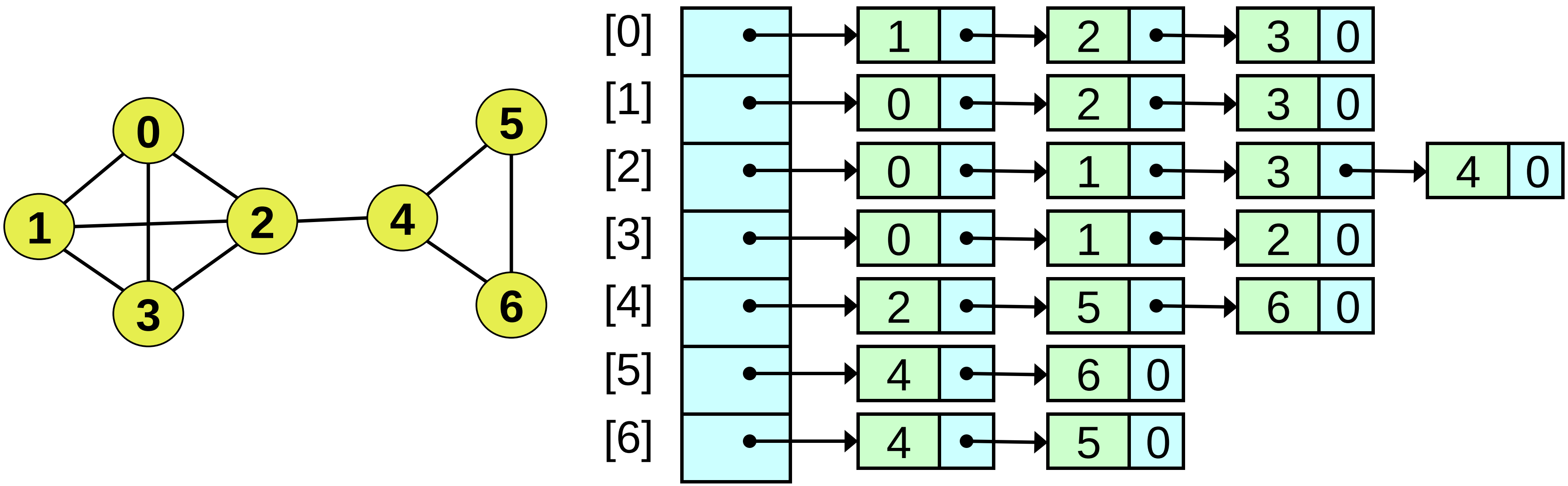
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

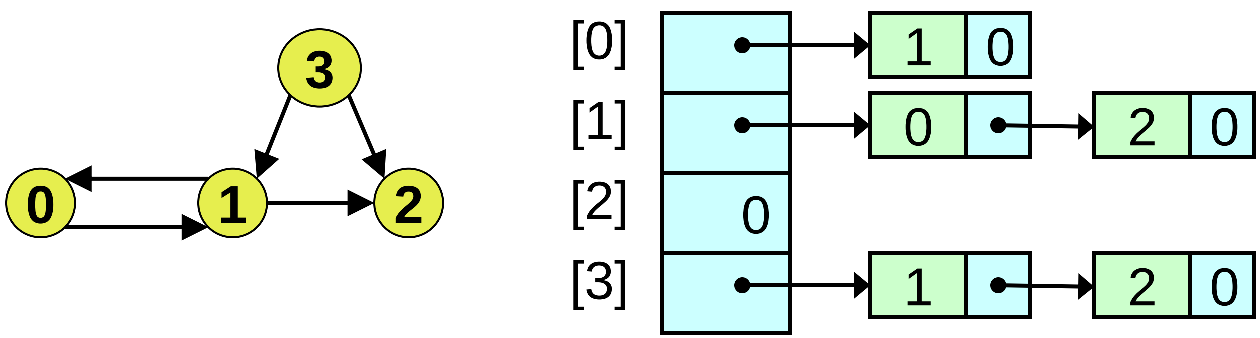
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Graph Representation: Adjacency List

- Adjacency matrices have high space and time complexities for sparse graphs.
  - Remember sparse matrices?
- A solution is to use an array of linked lists, each one containing the neighbors (directly connected vertices) of a vertex.
  - This is what we did in the equivalent class problem back in chapter 4.

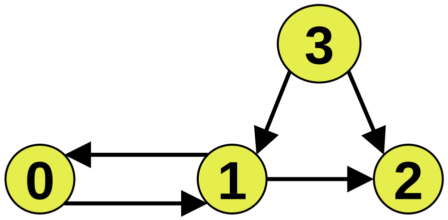
Graph Representation: Adjacency List

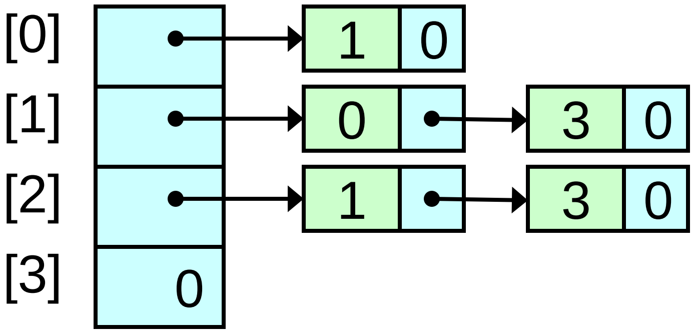




Inverse Adjacency List

For directed graphs, sometimes it is useful to remember the <u>incoming</u> edges.





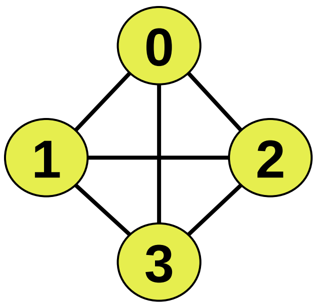
Adjacency Multilists

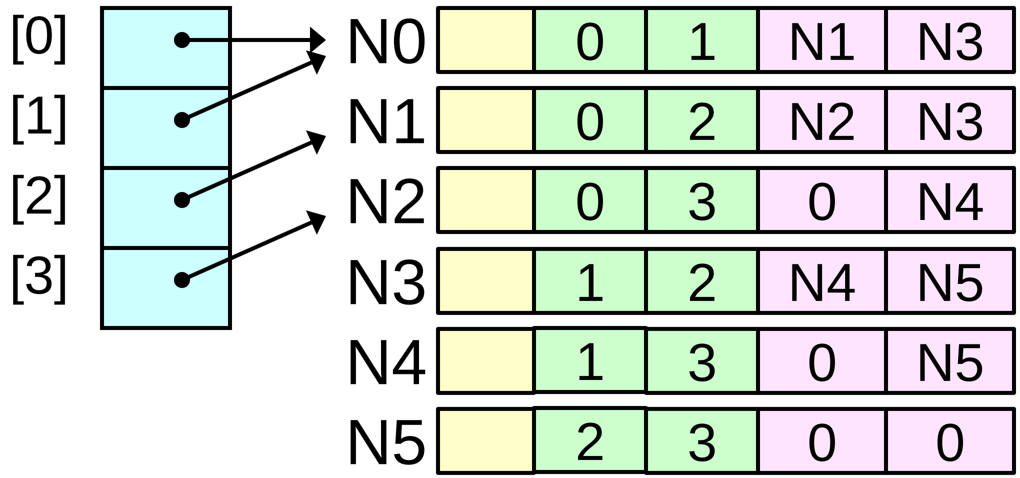
- For an undirected graph, each edge appears twice in the adjacency list.
  - Any update to the edge has to be done twice.
- An option is to use a **multilist** – sharing a node among several lists – to simplify this problem.
  - Each edge is represented by a node.
  - Each node appears in two links.
- The list representation in chapter 4 for sparse matrices is an example of multilists.

Node structure:

flag, etc.	vertex 1	vertex 2	pointer to next edge containing vertex 1	pointer to next edge containing vertex 2

Adjacency Multilists





The lists are: vertex 0: N0→N1→N2

vertex 1: N0→N3→N4

vertex 2: N1→N3→N5

vertex 3: N2→N4→N5

Weighted Edges

- Very often the edges of a graph have weights associated with them. Examples:
  - Distances
  - Costs
- We need an additional field, **weight**, for each edge entry.
- A graph with weighted edges is called a **network**.

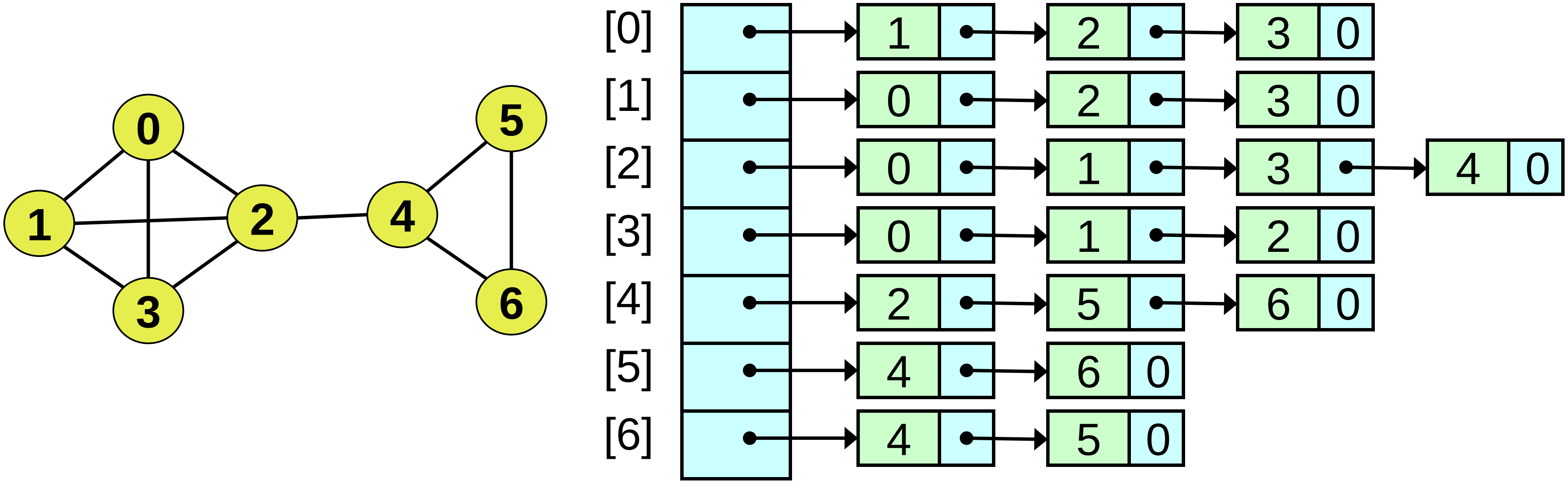
Searching in a Graph

- Goal: To reach every reachable vertex in a graph from a particular (starting) vertex.
- Two methods considered here:
  - **Depth-first search (DFS)**
  - **Breadth-first search (BFS)**
- They work on both directed and undirected graphs.
- Complexity of DFS and BFS
  - Graphs represented by adjacency matrices: **$O(|V|^2)$**
  - Graphs represented by adjacency lists: **$O(|E|)$**

Depth-First Search (DFS)

- A generalization of preorder traversal.
- Starting from vertex **v**, process **v** and then recursively (or using a stack to) traverse all vertices adjacent to **v**.
- To avoid cycles, mark visited vertices.
- The same idea was used in the maze problem in chapter 3.

Depth-First Search (DFS)

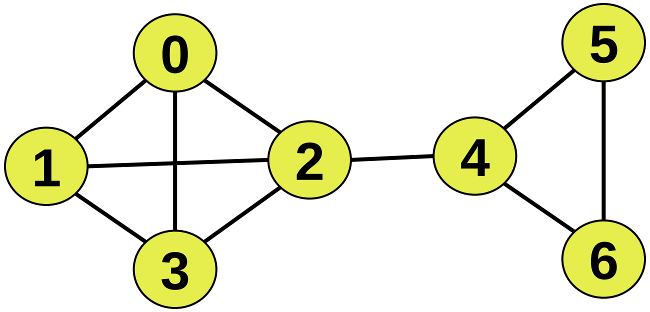


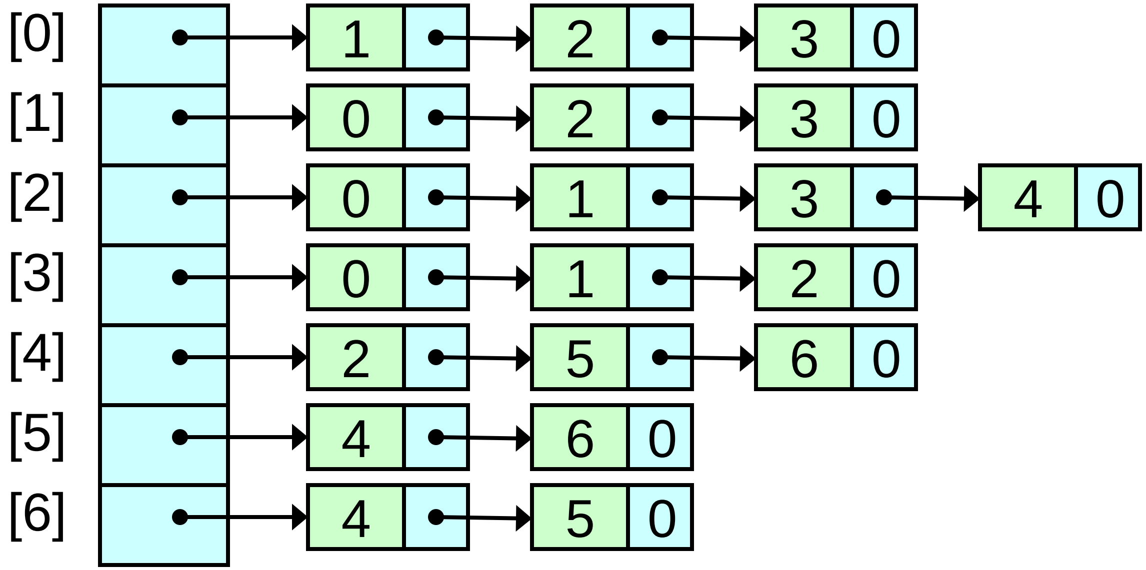
Order of visiting (starting from **0**):

Breadth-First Search (BFS)

- A generalization of level-order traversal.
- A queue is used.
- To avoid cycles, mark visited vertices.
- BFS can be used off-line in the maze problem to find the shortest path.

Breadth-First Search (BFS)





Order of visiting (starting from **0**):

Finding Connected Components

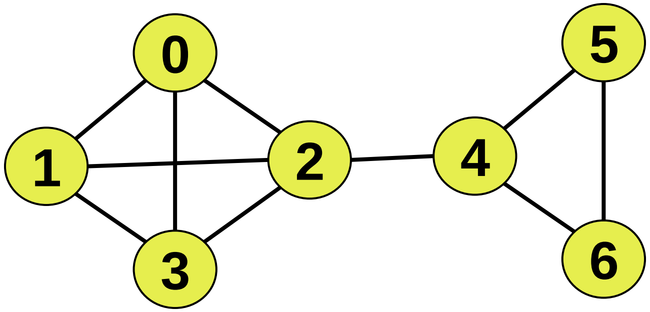
- In an undirected graph, both DFS and BFS can be used to find the connected component that contains a given vertex.
- To look for all the connected components, just start the search from any unvisited vertex, and repeat the process until all the vertices are already visited.
- The is exactly the method used in the equivalence class problem in chapter 4.

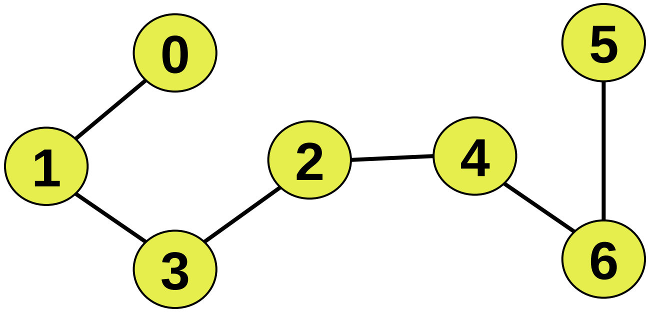
Spanning Trees

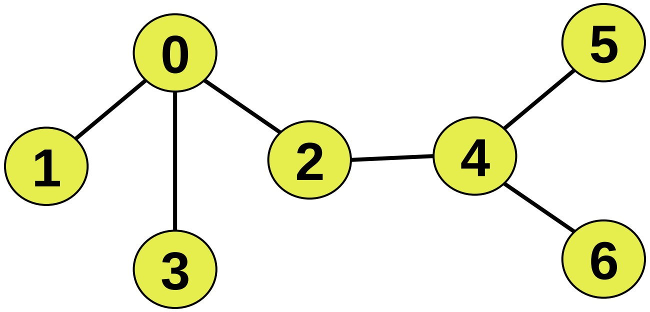
A spanning tree **T** of a graph **G**:

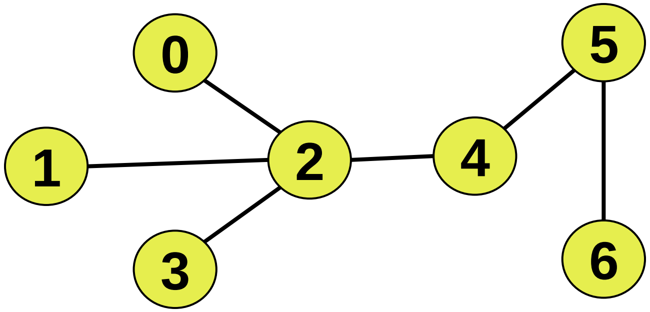
- A minimal connected subgraph of **G** with **V(G)=V(T)**
- Has **|V|–1** edges
- Is no longer connected if any edge is deleted
- Has no cycles

Spanning Trees

Example spanning trees of this graph: 

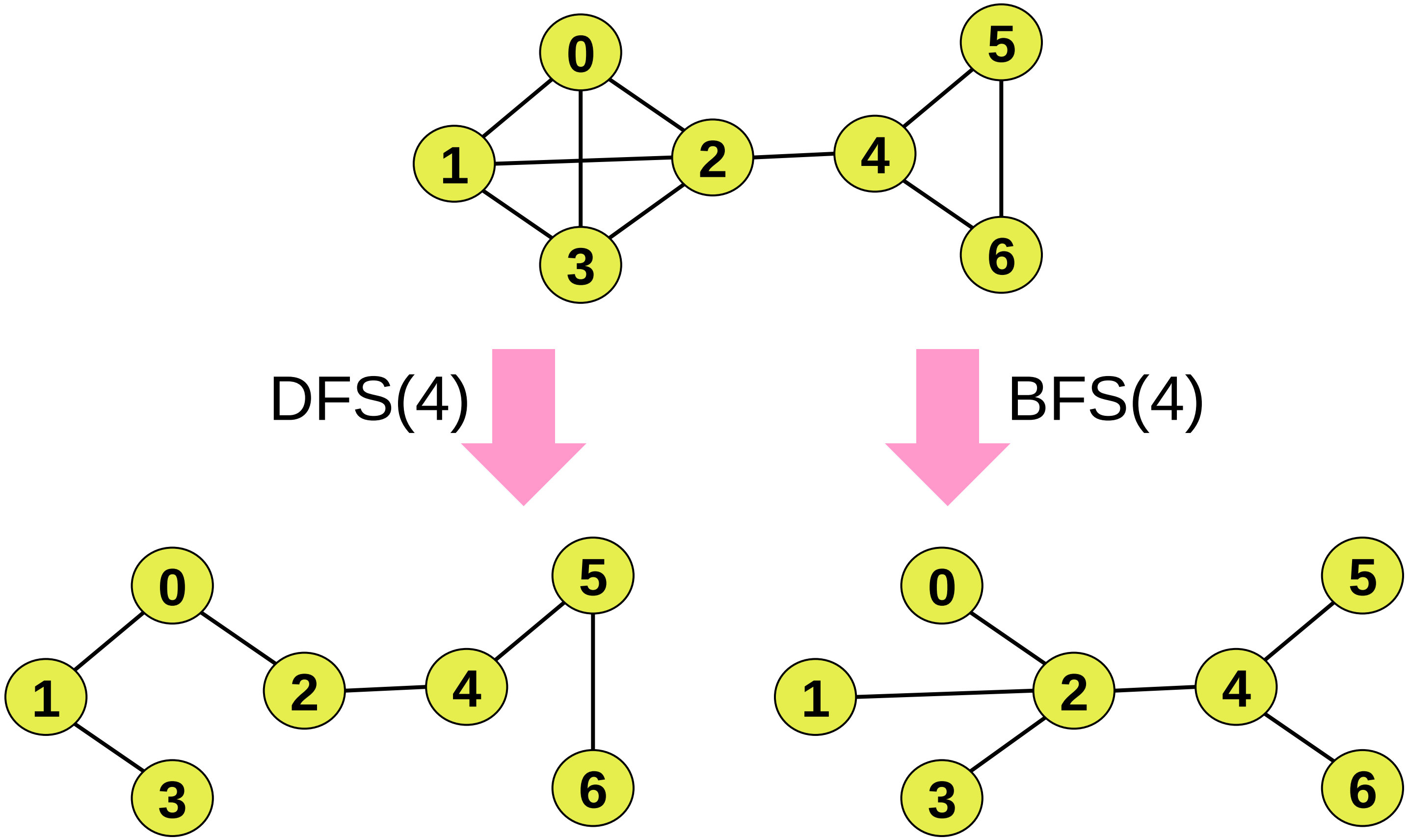






Spanning Trees

DFS and BFS can be used to find spanning trees:



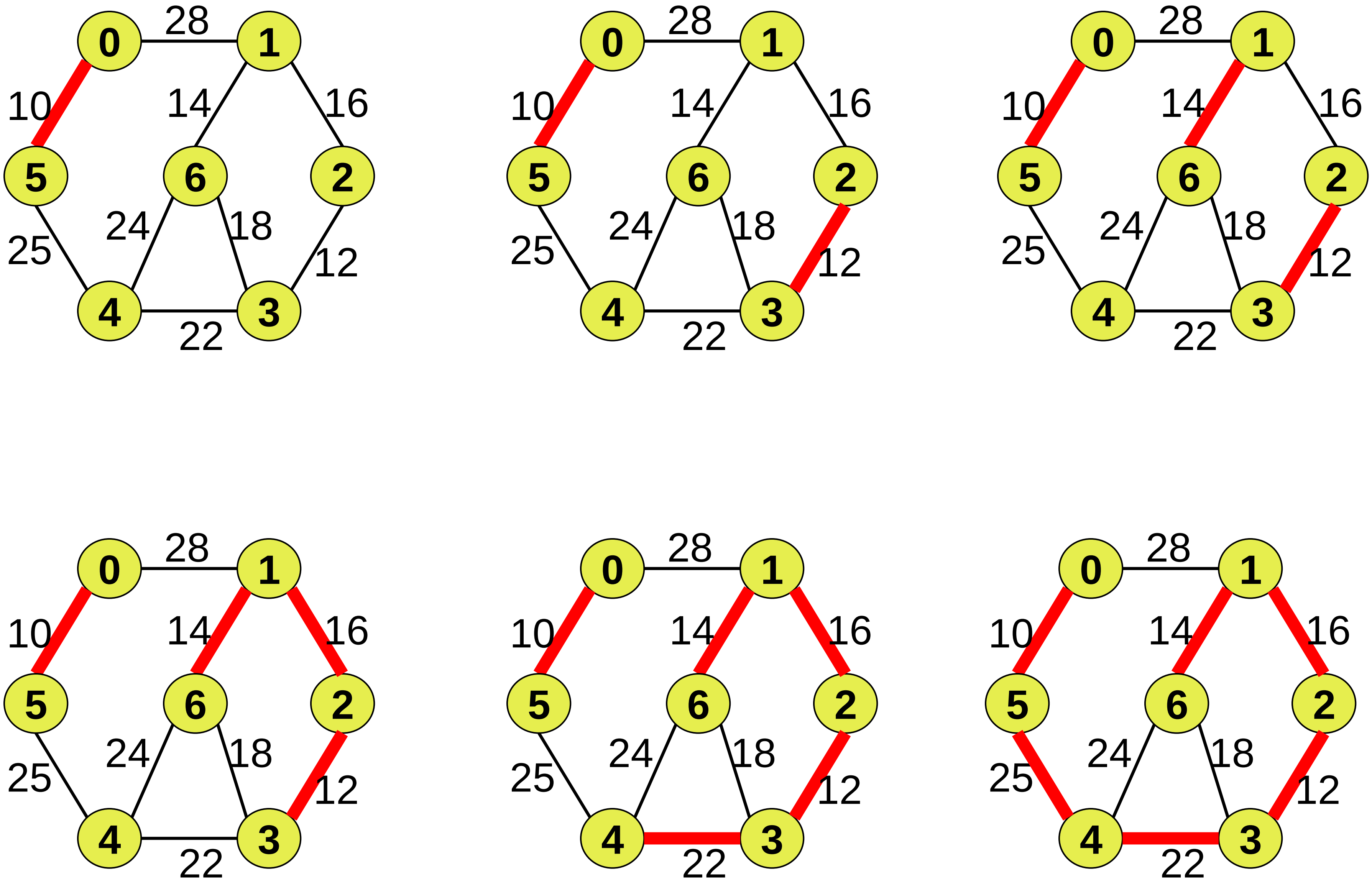
Minimum (Cost) Spanning Tree

- For a weighted graph, A **minimum-cost spanning tree** (or **minimum spanning tree**, or **MST** for short) is the spanning tree that has the lowest total weight (cost) in all its edges.
- Example application:
  - Build the roads that can connect a set of cities with minimal total distance.
- Three algorithms for finding the MST of a graph:
  - **Kruskal's algorithm**
  - **Prim's algorithm**
  - **Sollin's algorithm**

MST: Kruskal's Algorithm

- Kruskal's algorithm builds a minimum-cost spanning tree **T** by adding edges to **T** one at a time.
- The algorithm selects the edges for inclusion in **T** in non-decreasing order of their costs.
- An edge is added to **T** if it does not form a cycle with the edges that are already in **T**.
- In intermediate stages of the algorithm, the edges of **T** may not be in the same connected component.
- The algorithm repeats until **T** has **$|V|-1$** edges.

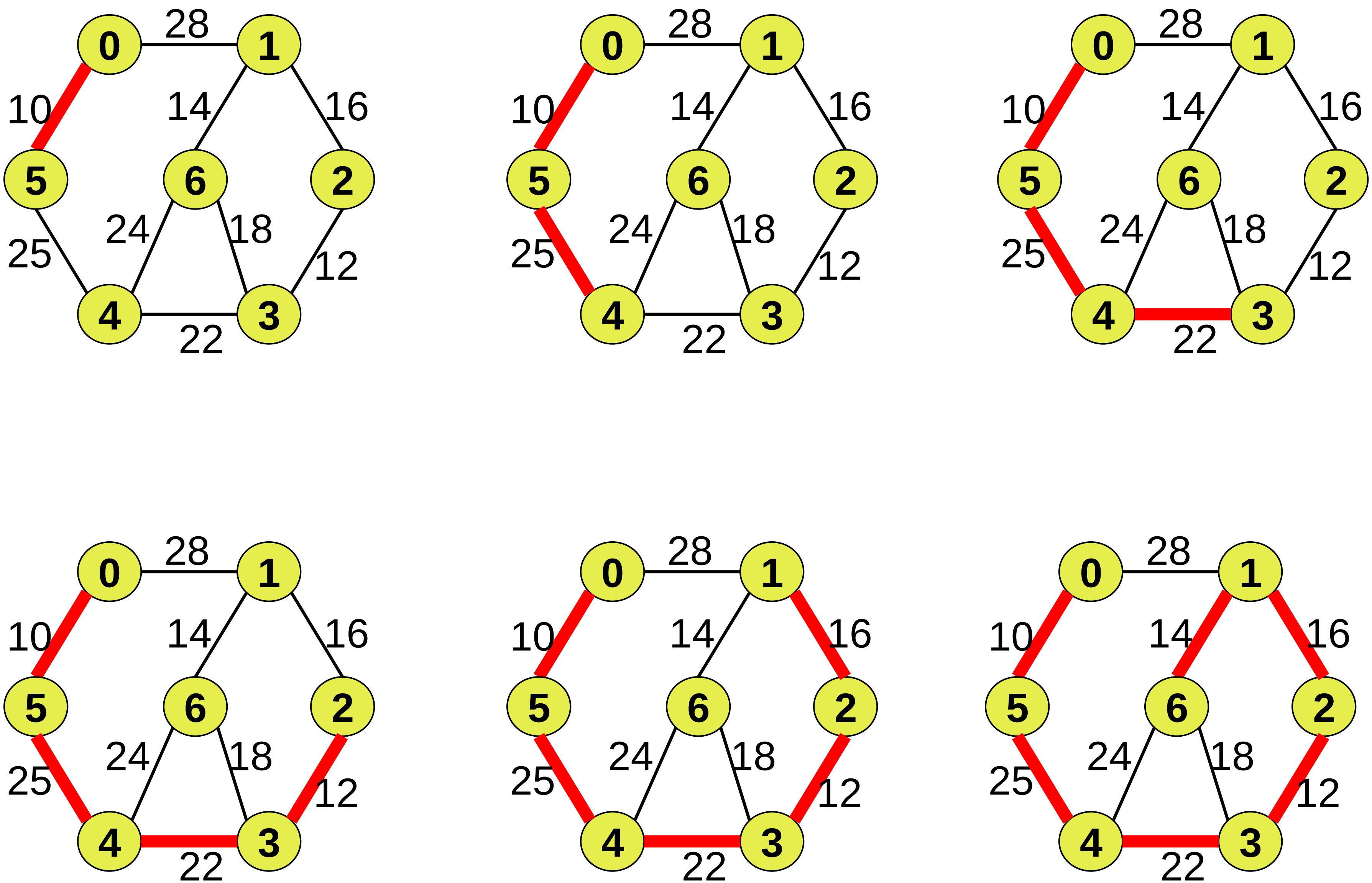
MST: Kruskal's Algorithm



MST: Prim's Algorithm

- Prim's algorithm builds a minimum-cost spanning tree **T** by adding edges to **T** one at a time.
- At any time of the algorithm, the edges in **T** form a tree (i.e., they are in the same connected component and forms no cycle).
- The algorithm always selects the lowest-cost edge for inclusion in **T** such that the resulting set of edges satisfies the condition above.
- The algorithm repeats until **T** has **|V|–1** edges.

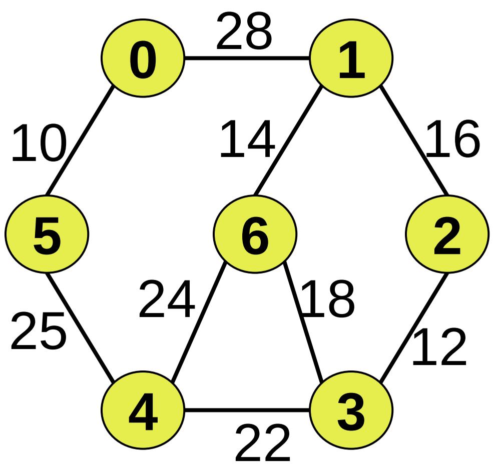
MST: Prim's Algorithm



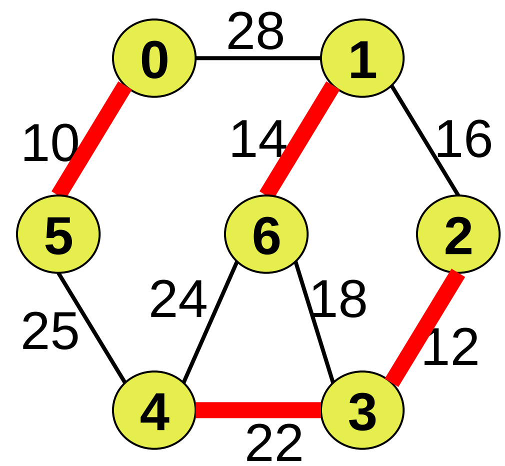
MST: Sollin's Algorithm

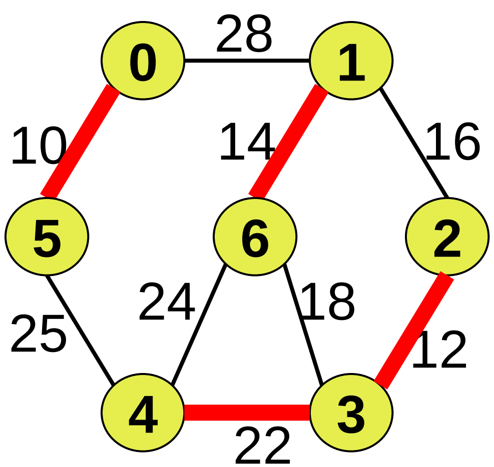
- Sollin's algorithm builds a minimum-cost spanning tree **T** by adding multiple edges to **T** at a time.
- At any time of the algorithm, the vertices and edges in **T** form a spanning forest (i.e., **$V(T)=V(G)$**, **$E(T)\subseteq E(G)$**, and **$E(T)$** forms no cycle).
- In each step, each tree in the spanning forest selects one lowest-cost edge that connects itself to another tree.
  - If two trees select the same edge, keep only one.
  - If two trees select two different edges of the same cost to connect them, keep only one.
- The algorithm repeats until **T** has **$|V|-1$** edges.

MST: Sollin's Algorithm

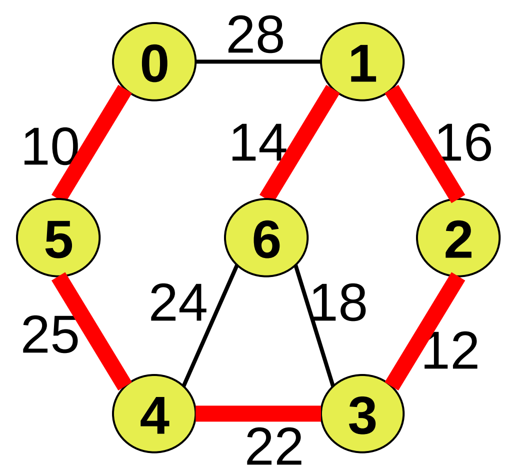


vertex 0: (0,5)
vertex 1: (1,6)
vertex 2: (2,3)
vertex 3: (2,3)
vertex 4: (3,4)
vertex 5: (0,5)
vertex 6: (1,6)





vertex {0,5}:
(4,5)
vertex {1,6}:
(1,2)
vertex {2,3,4}:
(1,2)

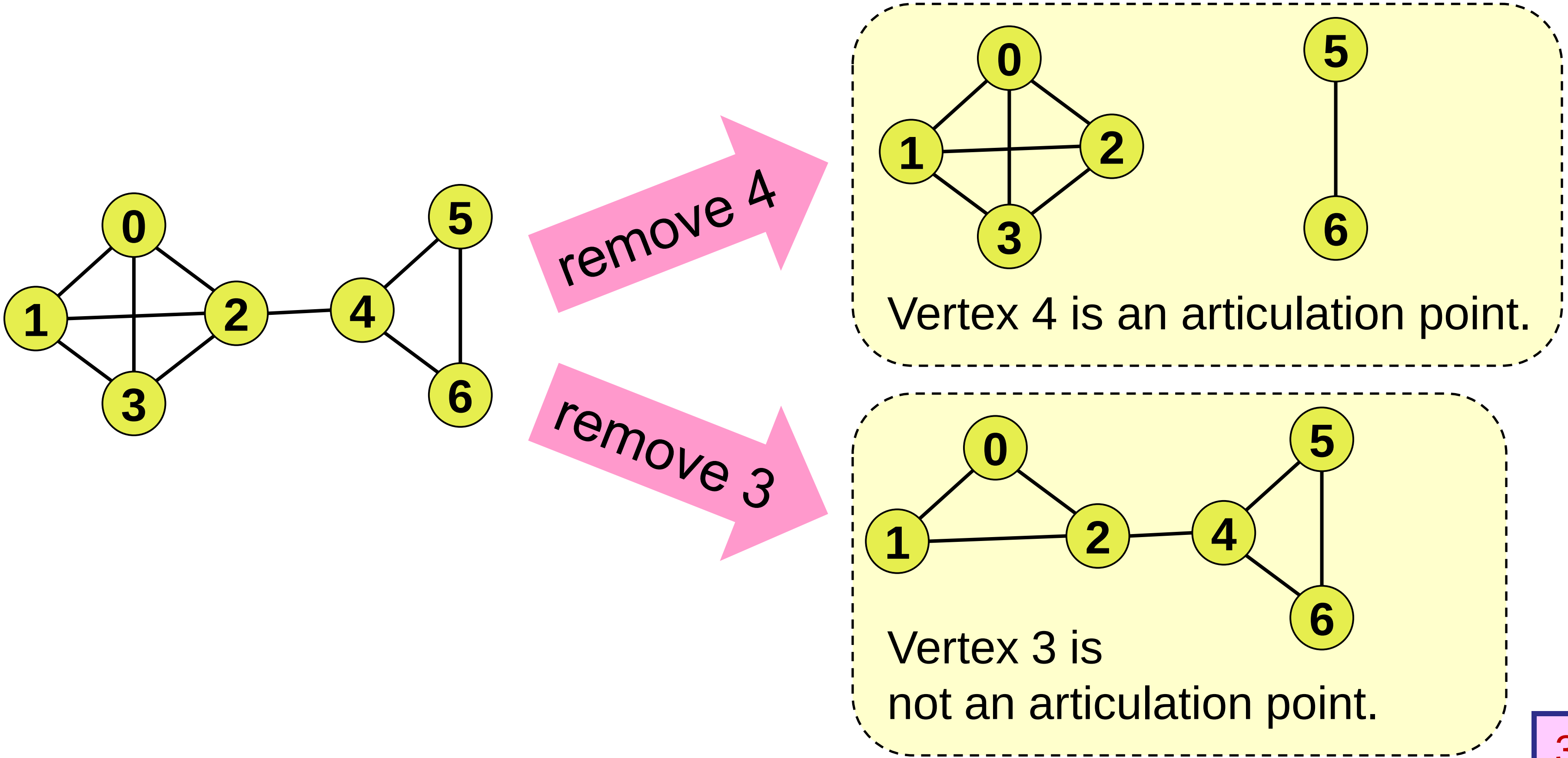


Articulation Points

Vertex **v** is an articulation point of graph **G**

$\Leftrightarrow$ [If **v** (and all the edges incident on **v**) is removed

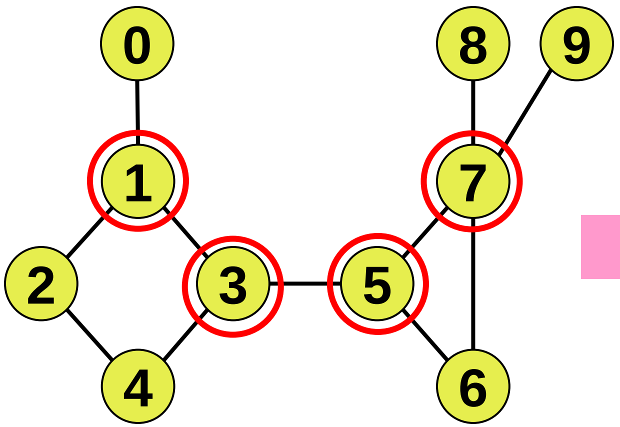
$\Rightarrow$ the connected component containing **v** becomes two or more connected components]



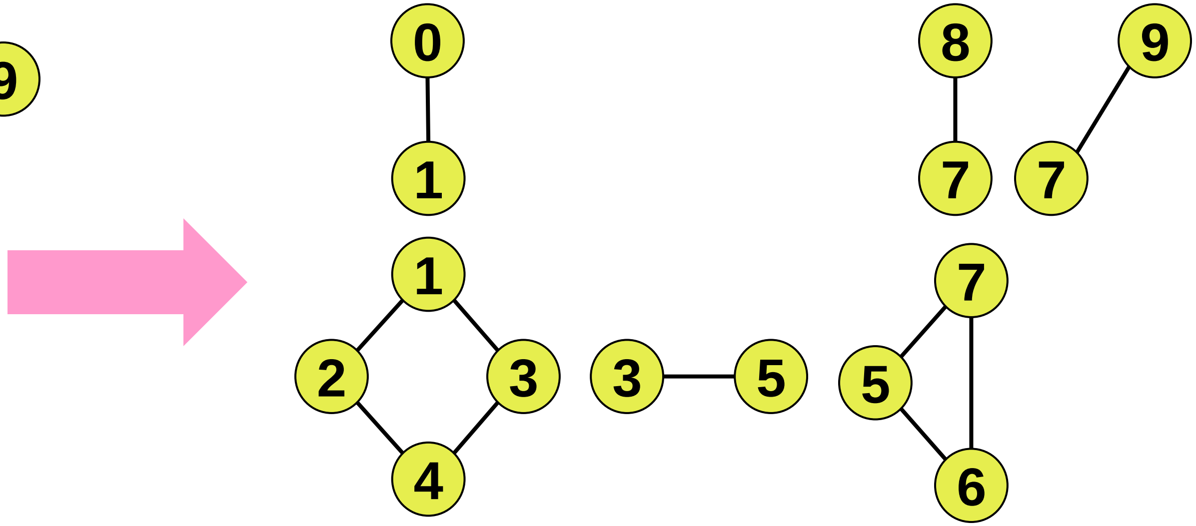
Biconnected Components

- A **biconnected graph** is a connected graph with no articulation points.
  - For a biconnected graph with $\geq$3 vertices, between any pair of vertices there are at least two paths with no common intermediate vertices.
- A **biconnected component** of a graph is a <u>maximal biconnected subgraph</u>.
  - Two biconnected components of a graph can <u>share at most one vertex</u>.
  - Two biconnected components of a graph have <u>no common edge</u> (i.e., edges are partitioned into the connected components).

Biconnected Components



articulation points

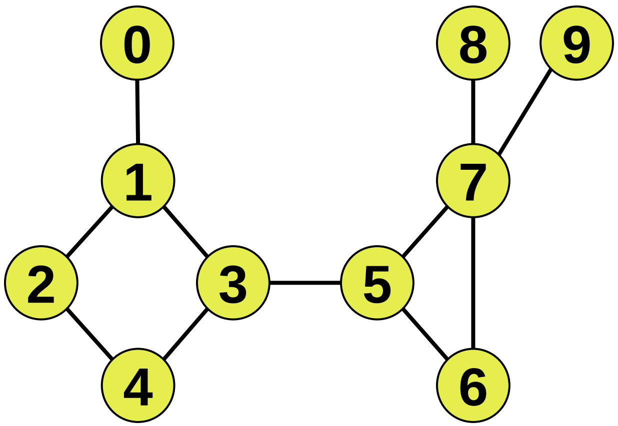


biconnected components

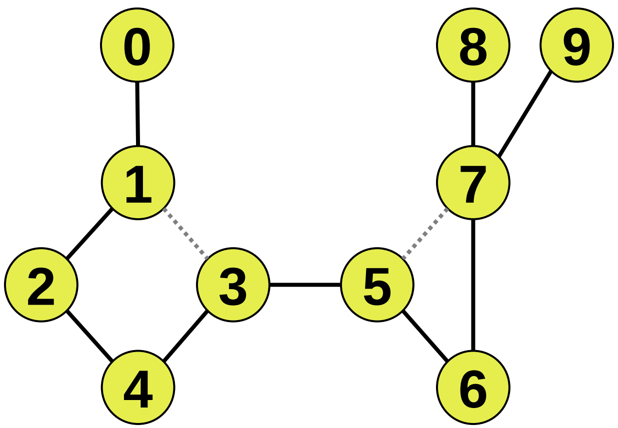
Depth-First Spanning Trees

- A **depth-first spanning tree** is a spanning tree found by DFS.
  - We can use a depth-first spanning tree to find the articulation points and biconnected components.
- A spanning tree partitions the edges of a graph into <u>tree edges</u> and <u>non-tree edges</u>:
  - **Back edges**: Non-tree edges whose two vertices are <u>ancestor and descendant of each other</u> in the spanning tree.
  - **Cross edges**: Non-tree edges that are not back edges. (*Depth-first spanning trees have no cross edges. Why?*)

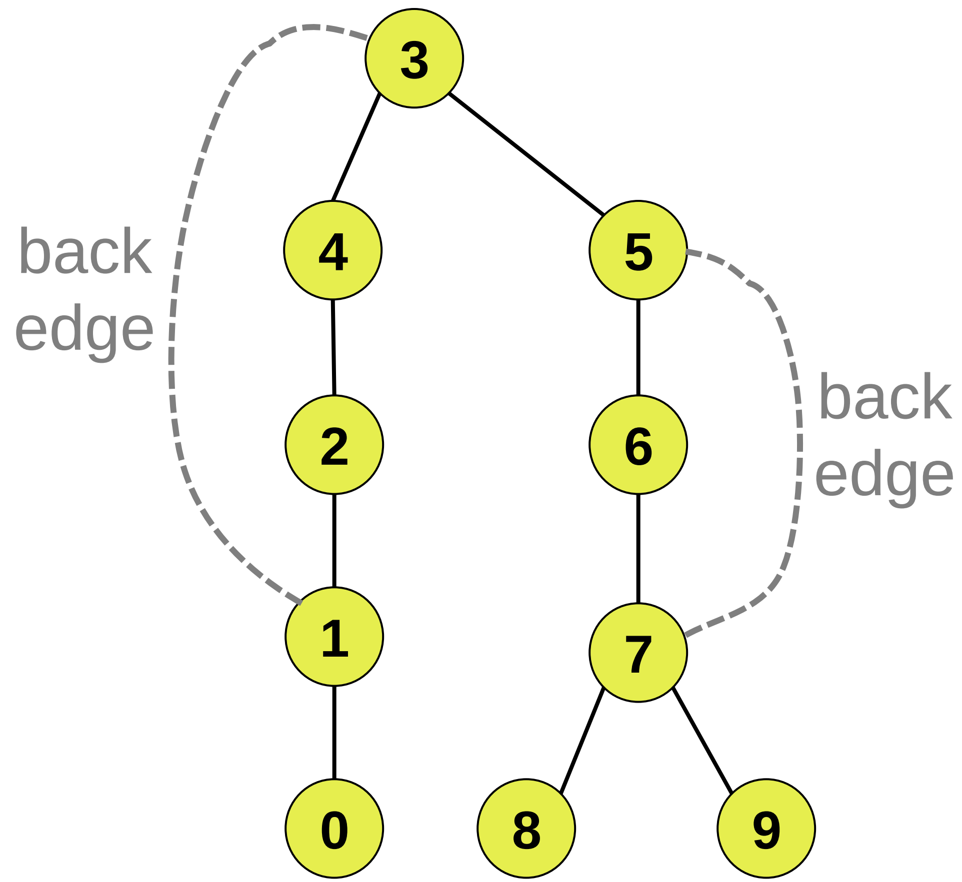
Depth-First Spanning Trees



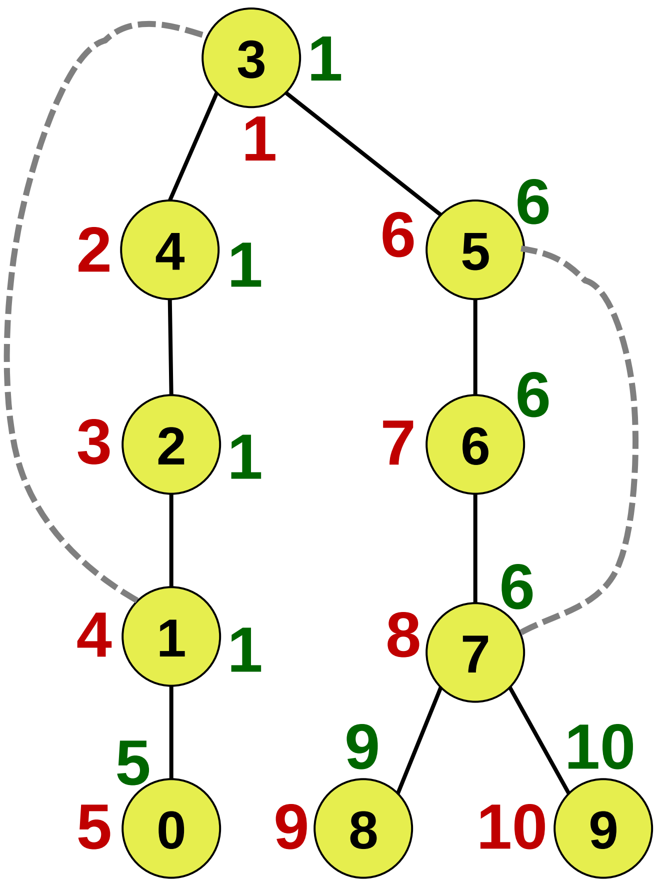
A possible DFS(3)



Drawn in tree form:



Depth-First Spanning Trees



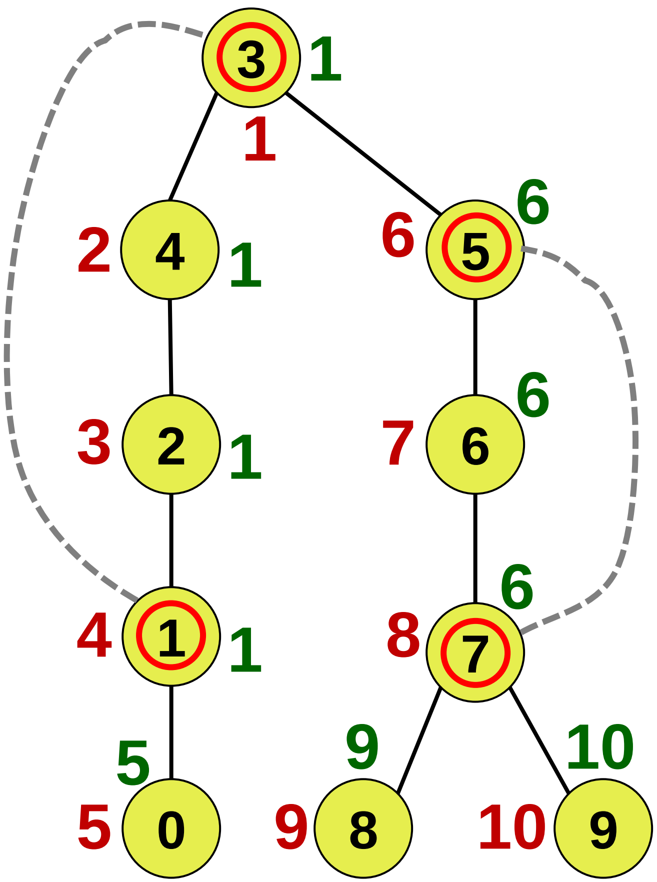
dfn(**v**): The order of visiting **v** when generating this tree.

low(**v**): The smallest *dfn* of the vertices in any cycle containing **v**.

low(**v**) is computed as the <u>smallest</u> of the following:

- *dfn*(**v**)
- min[*low*(**w**)], with **w** a child of **v**
- *dfn*(**u**), with **(u,v)** a <u>back edge</u>

Depth-First Spanning Trees



Determining articulation points:

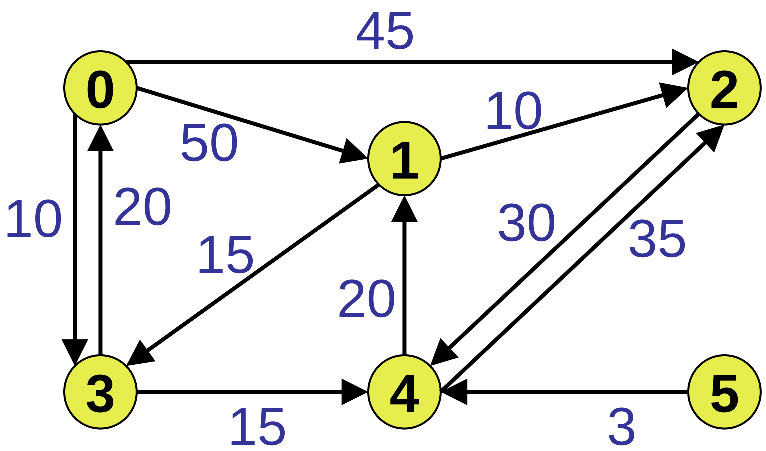
- If vertex **u** is the <u>root</u> and its has ≥2 children, then **u** is an articulation point.
- If vertex **u** is not the root and it has a child **w** with *low*(**w**) ≥ *dfn*(**u**), then **u** is an articulation point.

Finding the Shortest Path

- This is a very practical problem ...
  - Physical routing (Google maps, etc.)
  - Network routing
  - Action planning
  - etc.
- Types:
  - Single source, single destination
  - Single source, all destinations
  - All pairs

Finding the Shortest Path

Example problem:



Short paths from 0:

to 1: 0→3→4→1 (dist=45)

to 2: 0→2 (dist=45)

to 3: 0→3 (dist=10)

to 4: 0→3→4 (dist=25)

to 5: none (dist=∞)

Dijkstra's Algorithm

- Initialize the distances (source: 0; others: ∞), dist[]
- Initialize a set of "unvisited vertices" with all the vertices
- In each iteration
  - Select the unvisited vertex u with the smallest dist[u]
  - Remove u from the set of unvisited vertices
  - For each edge <u,v> such that dist[u] + w(<u,v>) < dist[v]
    - Consider only those v in the set of unvisited vertices
    - Set dist[v] = dist[u] + w(<u,v>)
    - Set pred[v] = u   not in textbook; required to remember the path
- Termination:
  - The set of unvisited vertices is empty
  - The smallest dist[u] among unvisited vertices is infinity.

Dijkstra's Algorithm

Example:

Selected vertex: 0 (distance = 0)

- edge 0→1 (dist=50<∞): ➔ dist[1] = 50; pred[1] = 0
- edge 0→2 (dist=45<∞): ➔ dist[2] = 45; pred[2] = 0
- edge 0→3 (dist=10<∞): ➔ dist[3] = 10; pred[3] = 0

Selected vertex: 3 (distance = 10)

- edge 3→4 (dist=25<∞): ➔ dist[4] = 25; pred[4] = 3

Selected vertex: 4 (distance = 25)

- edge 4→1 (dist=45<50): ➔ dist[1] = 45; pred[1] = 4
- edge 4→2 (dist=60>45): ➔ no update

Selected vertex: 2 (distance = 45): no edge to check

Selected vertex: 1 (distance = 45): no edge to check

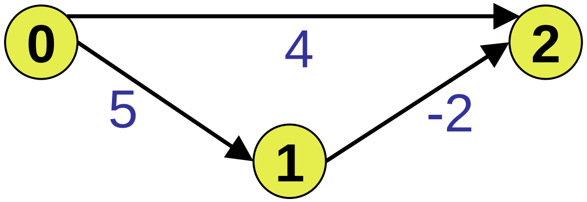
Selected vertex: 5 (distance = ∞): termination

Dijkstra's Algorithm

- Assumes non-negative weights for all the edges
- Time complexity: $O(|V|^2)$
  - $|V|$ iterations, each iteration is $O(|V|)$ to select the vertex with the smallest distance
  - Each edge only processed once
- Improved time complexity: $O(|E| + |V| \log|V|)$
  - Use a min priority queue (Fibonacci heap; chp. 9) for vertex selection.

Dijkstra's Algorithm

- Dijkstra's algorithm does not handle edges with negative weights for all the edges.
  - It assumes that adding an edge to a path can not reduce its total distance.
  - Therefore, once a vertex is considered visited (i.e., shortest distance to that vertex is known), its distance is never updated again.
  - Example:



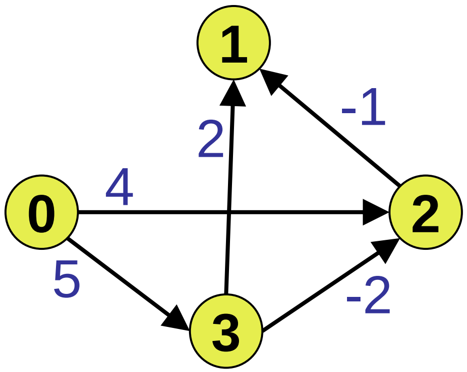
Note: No shortest path exists when a graph contains cycles of negative total weights.

Bellman-Ford Algorithm

- Assumes no negative cycles
- Initialize the distances (source: 0; others: ∞), dist[]
- Repeat for |V| times
  - For each edge <u,v> such that dist[u] + w(<u,v>) < dist[v]
    - Set dist[v] = dist[u] + w(<u,v>)
    - Set pred[v] = u not in textbook; required to remember the path
- Early termination: When there is no update in an iteration of the main loop.
- Time complexity: *O*(|V|*|E|)

Bellman-Ford Algorithm

Example:



It#	Edge	dist[0]	dist[1]	dist[2]	dist[3]
		0	∞	∞	∞
1	0→2			4	
	0→3				5
	2→1		3		
	3→1				
	3→2			3	
2	0→2				
	0→3				
	2→1		2		
	3→1				
	3→2				

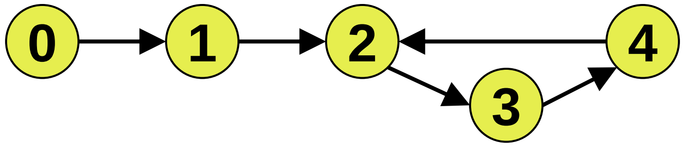
No update in the next iteration and the algorithm will terminate.

Transitive Closure

- A **transitive closure** matrix of a graph is a binary matrix such that
  - $A^+[i][j] = 1 \Leftrightarrow$ There exists a path of **length >0** from i to j.
- A **reflective transitive closure** matrix of a graph is a binary matrix such that
  - $A^*[i][j] = 1 \Leftrightarrow$ There exists a path of **length ≥0** from i to j.
- Floyd's algorithm can be used to identify the transitive closure.

Transitive Closure

Example:



Adjacency

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0

A^+

	0	1	2	3	4
0	0	1	1	1	1
1	0	0	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

A^*

	0	1	2	3	4
0	1	1	1	1	1
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

Activity Networks

- **Activity networks** are directed graphs that represent the relations between activities.
  - Activities have precedence, meaning some activities can not start before some other activities have been completed.
- Types of activity networks:
  - **Activity-on-vertex (AOV)** networks
  - **Activity-on-edge (AOE)** networks

Activity-on-Vertex Networks

- Each vertex represents an activity.
- Edges represent the precedence between activities.
  - An edge **<i,j>** means that activity **i** has to occur before activity **j**.
- Definitions:
  - There exists a directed path from **i** to **j**

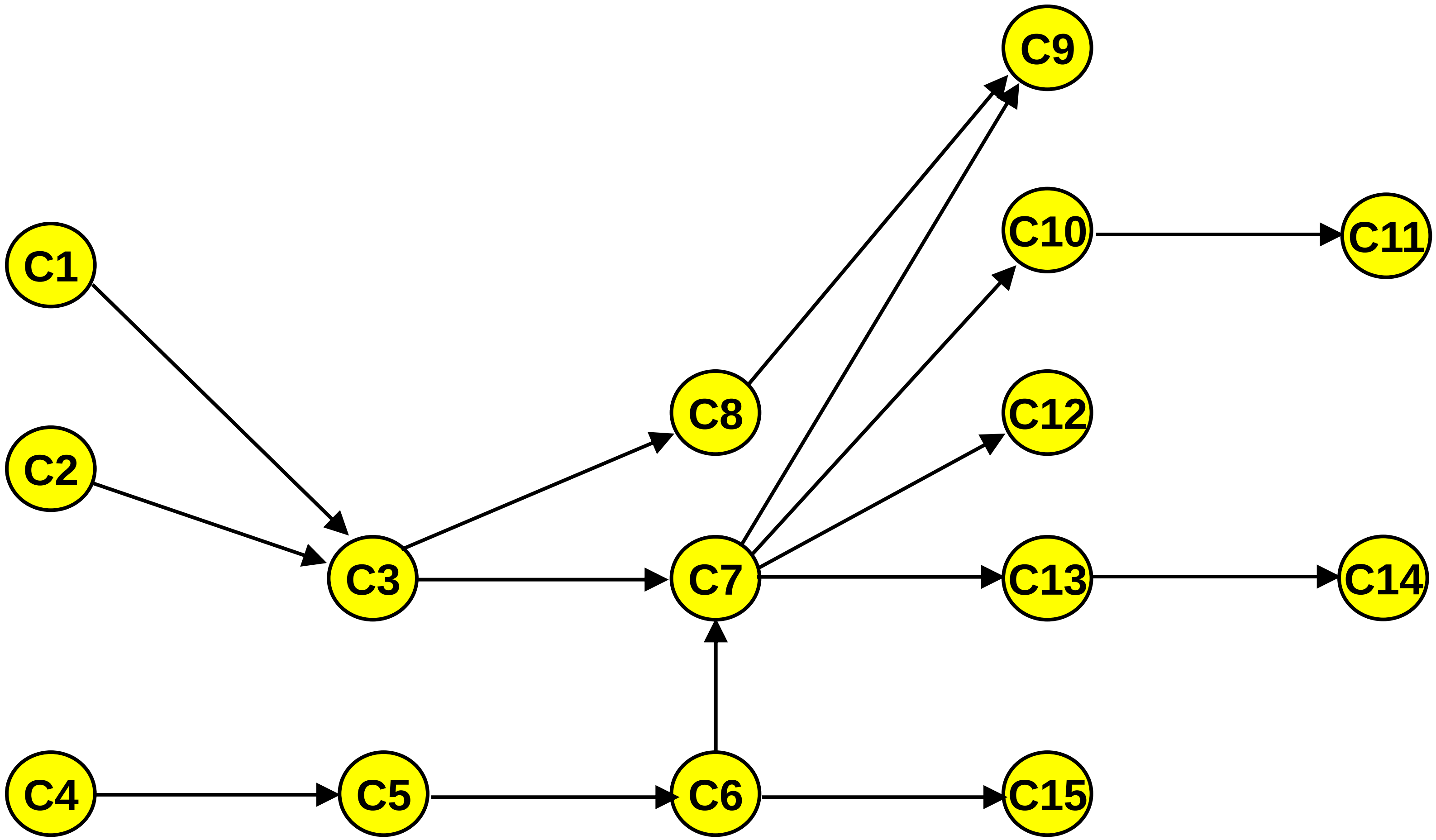
    $\Leftrightarrow$ **i** is a **predecessor** of **j** , and **j** is a **successor** of **i**.
  - **<i,j>** is an edge

    $\Leftrightarrow$ **i** is an **immediate predecessor** of **j** , and **j** is an **immediate successor** of **i**.

AOV Network: Example

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

AOV Network: Example



Topological Order

A **topological order** is a linear ordering of the vertices of a graph with the following property:

i is a predecessor of **j** $\Leftrightarrow$ **i** precedes **j** in the linear ordering.

Example topological orders for the course-taking example:

C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15

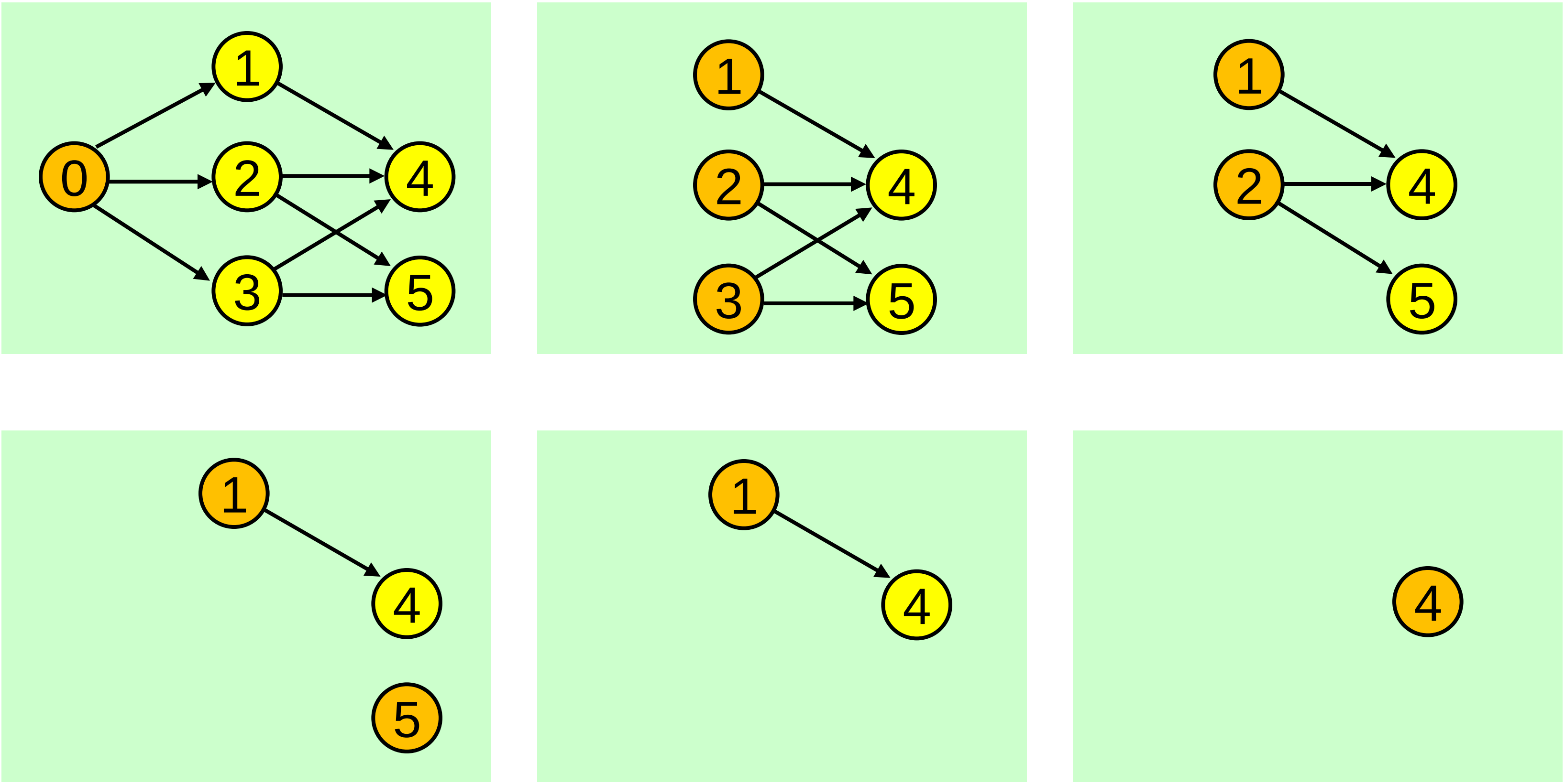
C1, C2, C4, C3, C5, C8, C6, C7, C9, C10, C12, C13, C15, C11, C14

Finding a Topological Order

- Goal: Given an AOV network, find a topological order of the activities.
- Method: Iteratively find a vertex with no predecessor, removes it from the network, and put it in the linear ordering.
  - Use a `count` field for each vertex in the graph representation. This field keeps track of the number of its immediate predecessors that are not processed yet.
  - We can put vertices whose `count` field becomes zero into a queue or stack.
  - If we end up with a non-empty network with all the vertices having predecessors, than there is a cycle in the network and no topological order exists.

Finding a Topological Order

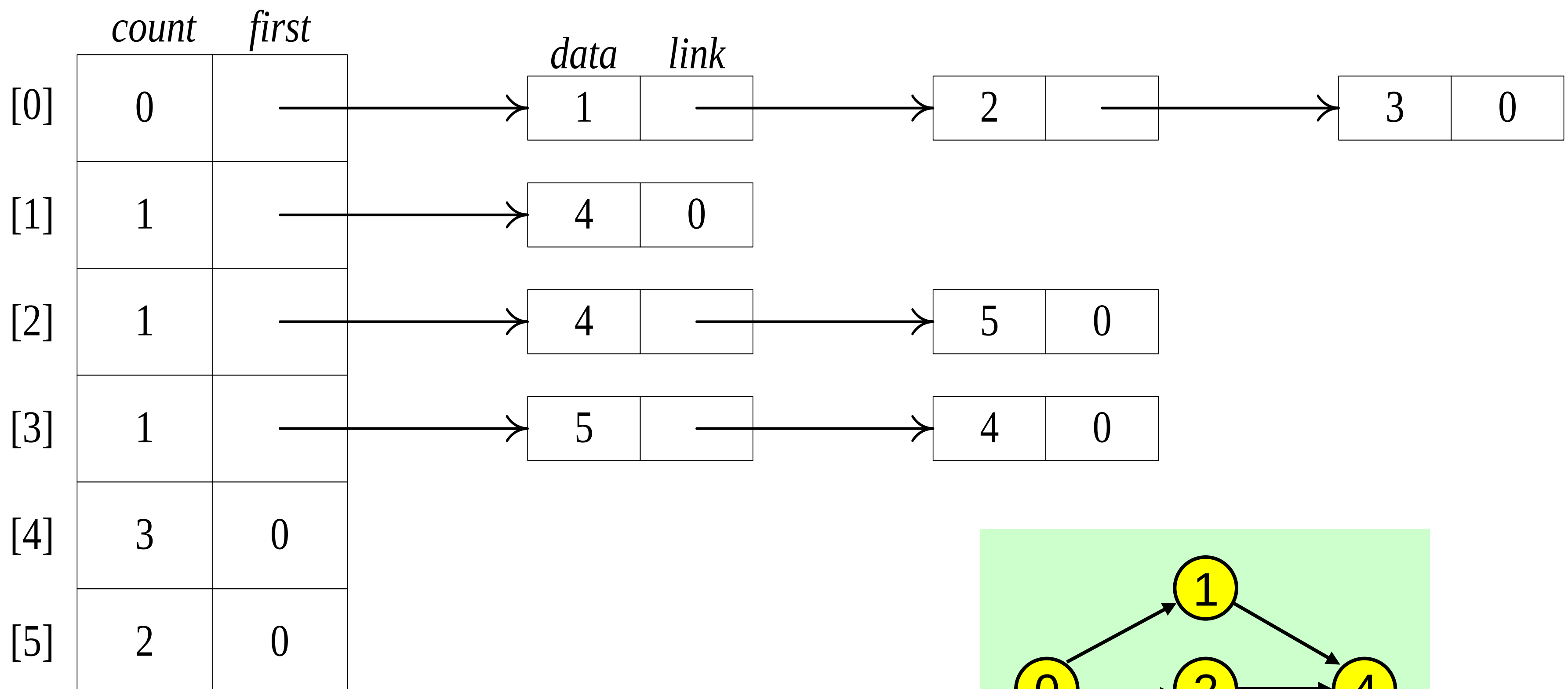
Example:

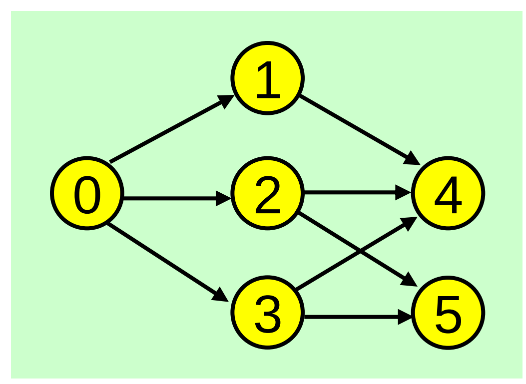


Resulting linear ordering: **0 → 3 → 2 → 5 → 1 → 4**

Finding a Topological Order

Internal representation for computing the topological order of an AOV network (note the `count` field):

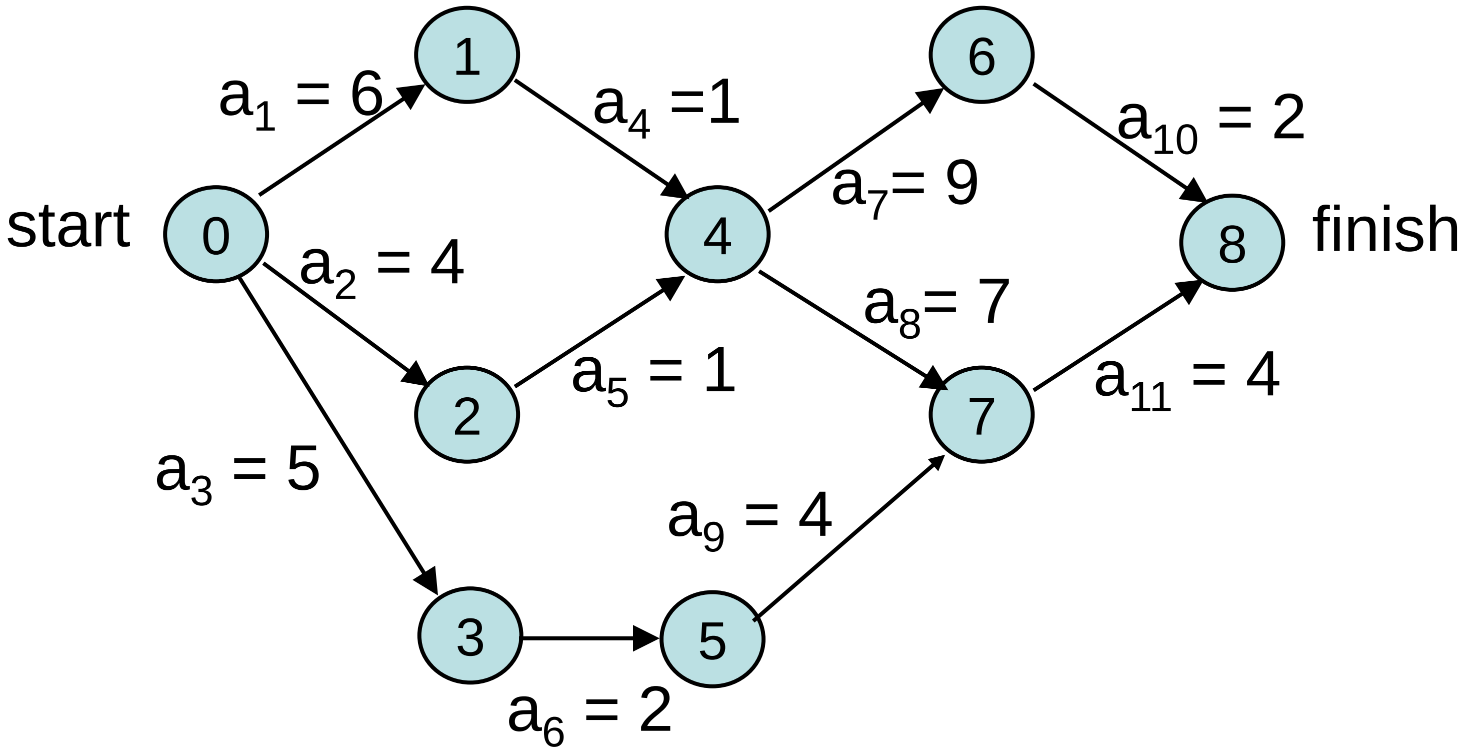




Activity-on-Edge Networks

- Each edge represents an activity, and its weight represents the duration required for completing that activity.
- Each vertex represents an event (state). Outgoing activities from a vertex can not start until all the incoming activities to that vertex are completed.
- Such a network can be used to plan a complex project. Some common questions:
  - How much time is required to complete all the activities?
  - What is the earliest time an activity can be started?
  - What is the earliest time an activity can be finished?
  - Is there any activity that, when delayed, will delay the whole project?

AOE Network: Example



Definitions for AOE Networks

- **Critical path**: A path with the longest length. This is the shortest time possible to complete the whole project. (There may be several critical paths, and they are all critical.)
- The **earliest event time** for a vertex **k**, **EE(k)**, is the length of the longest path from the start vertex to **k**. This is also the **earliest start time** for activity $\mathbf{a_i}$, **E(i)**, if $\mathbf{a_i}$ originates from **k**.
- The **latest event time** for a vertex **k**, **LE(k)**, is the latest time to reach in vertex (event) **k** without increasing the duration of the whole project.
- The **latest time** for activity $\mathbf{a_i}$, **L(i)**, is the latest time to start $\mathbf{a_i}$ without increasing the duration of the whole project. If $\mathbf{a_i}$ ends in vertex (event) **k**, then $\mathbf{L(i)=LE(k)-duration(a_i)}$.
- Activity $\mathbf{a_i}$ is a **critical activity** if $\mathbf{E(i)=L(i)}$, and $\mathbf{L(i)-E(i)}$ is a measure of the criticality of $\mathbf{a_i}$.

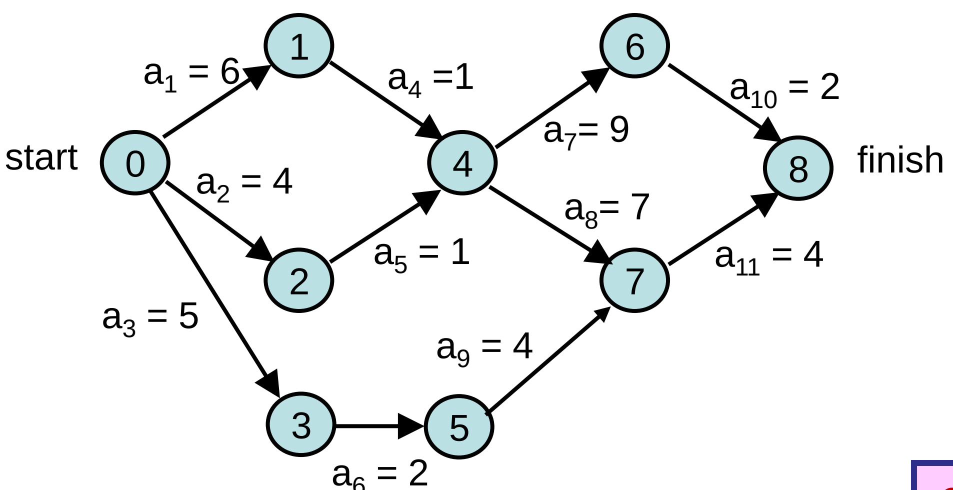
Computing EE

- Process the vertices in topological order. (We just need to modify the code that generates the topological order.)
- In each item in a adjacency list, add a field to represent the activity duration.
- **EE** of the "finish vertex" is the length of critical paths.
- Computing **EE**:
  - **EE(0) = 0** (start event)
  - **EE(k) = max[EE(j) + duration(<j,k>)]** for all the immediate predecessors **j** of **k** (this is the set **P(k)**).

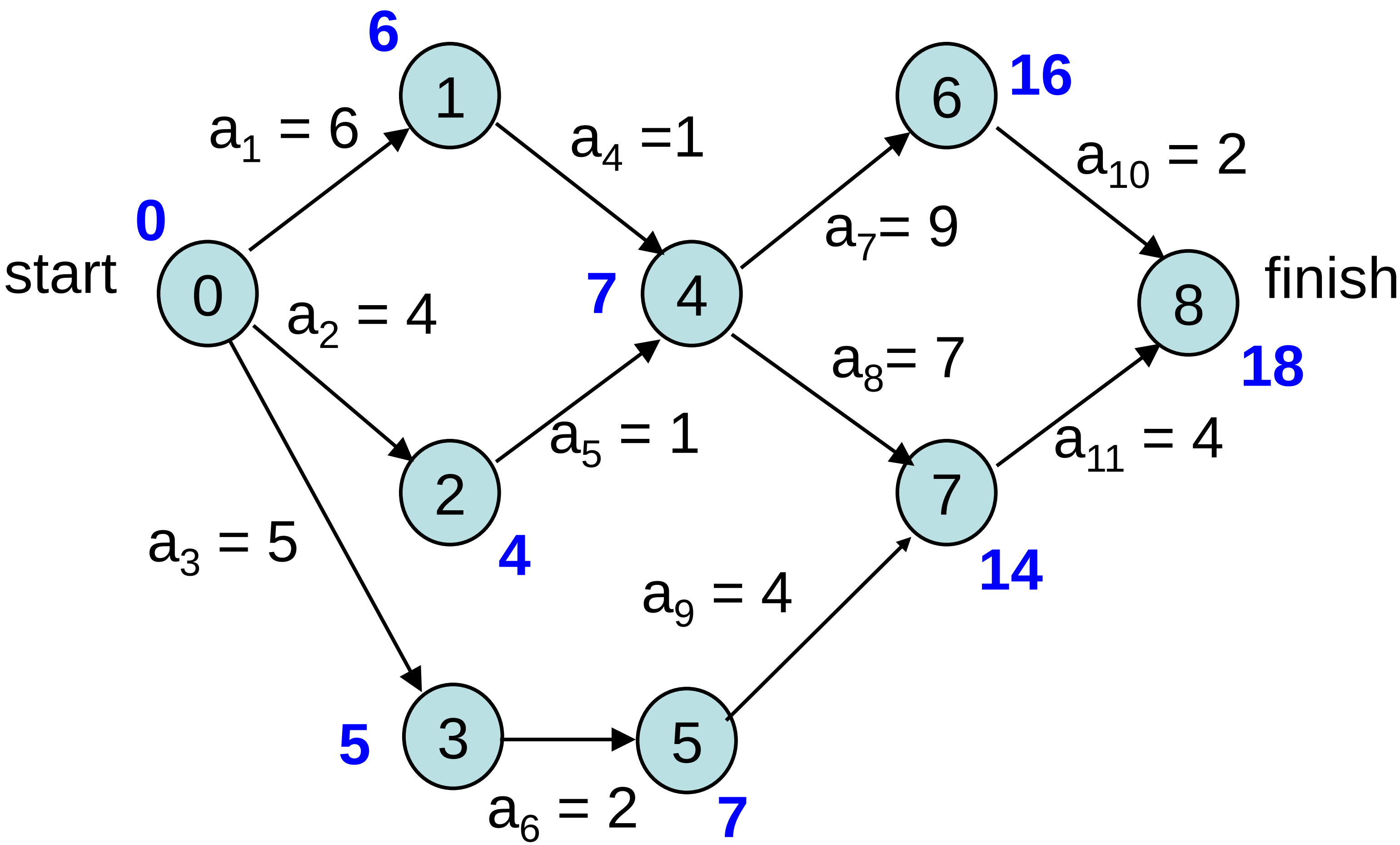
Computing EE

Internal representation for computing EE for an AOE network (note the `count` field):

	count	first	vertex	dur	link	vertex	dur	link	vertex	dur	link
[0]	0	→	1	6	→	2	4	→	3	5	0
[1]	1	→	4	1	0						
[2]	1	→	4	1	0						
[3]	1	→	5	2	0						
[4]	3	→	6	9	→	7	7	0			
[5]	2	→	7	4	0						
[6]	2	→	8	2	0						
[7]	2	→	8	4	0						
[8]	2	0									



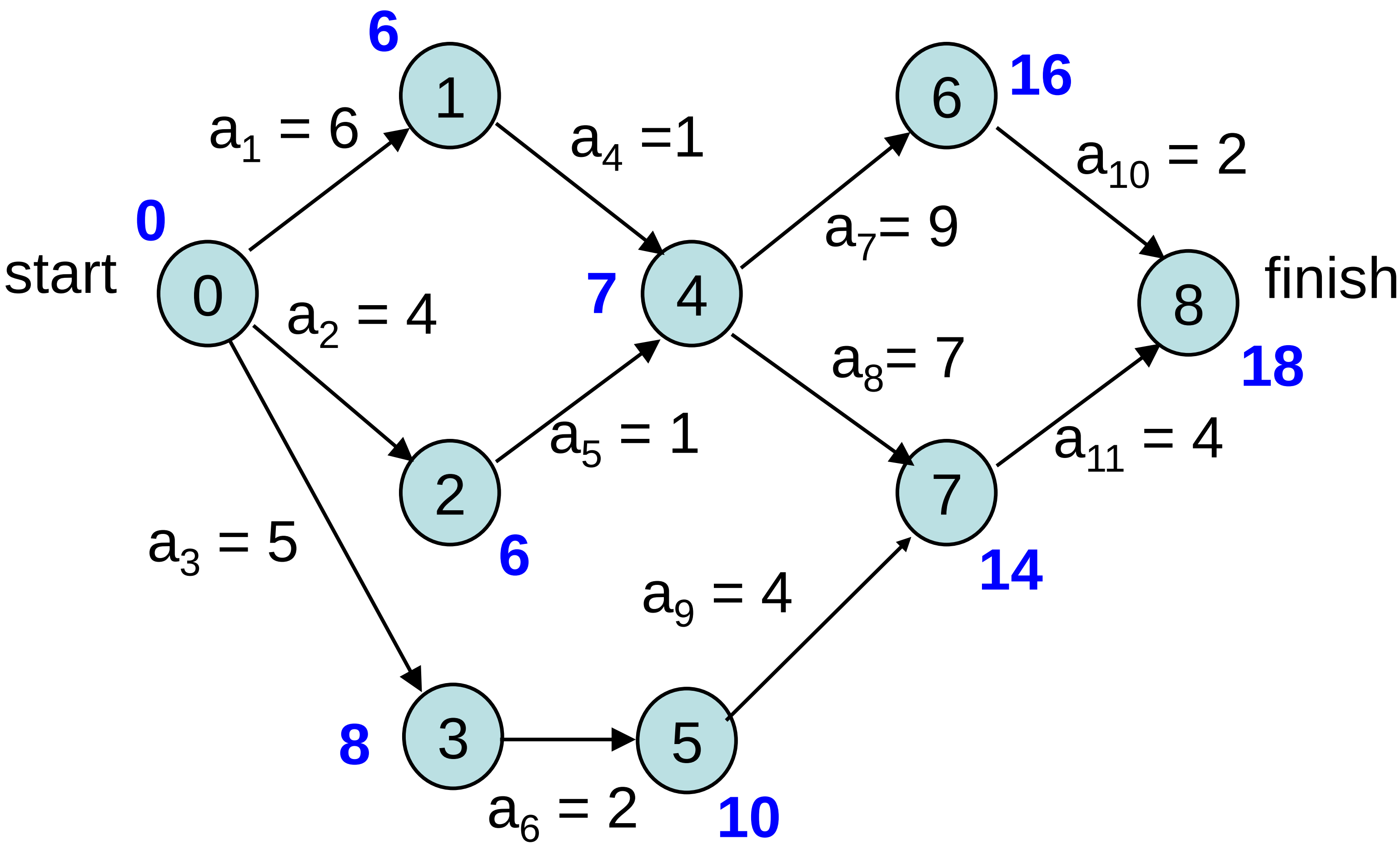
Computing EE



Computing LE

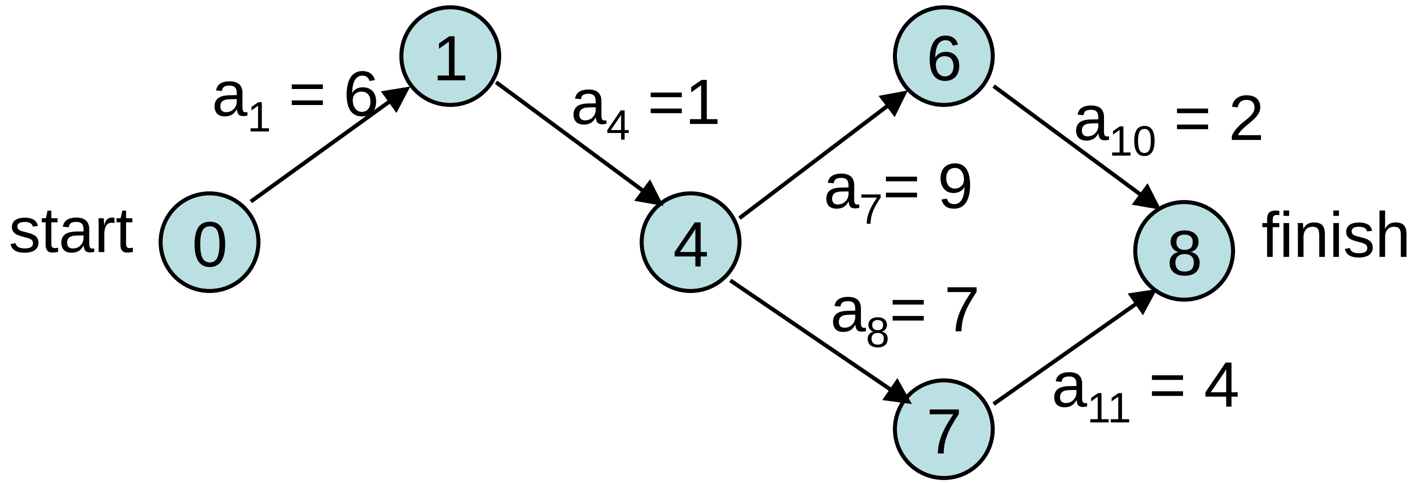
- Process the vertices in reverse topological order.
- Computing **LE**:
  - **LE(k) = EE(k)** (**k** is the "finish" event)
  - **LE(k) = min[LE(j) – duration(<k,j>)]** for all the immediate successors **j** of **k** (this is the set **S(k)**).

Computing LE



Determining L, E, and Critical Activities

- **E(i)=EE(k)** (**a_i** originates from **k**)
- **L(i)=LE(k)-duration(a_i)** (**a_i** ends at **k**)
- **L(i)–E(i)**: **Slack**: The time allowed to wait at a vertex without increasing the duration of the whole project.
- Critical activities are activities with **L(i)=E(i)**.
- A critical path consists of critical activities.



Extra Reading Assignments

- From the textbook: Section 6.4.3. This is Floyd's Algorithm.