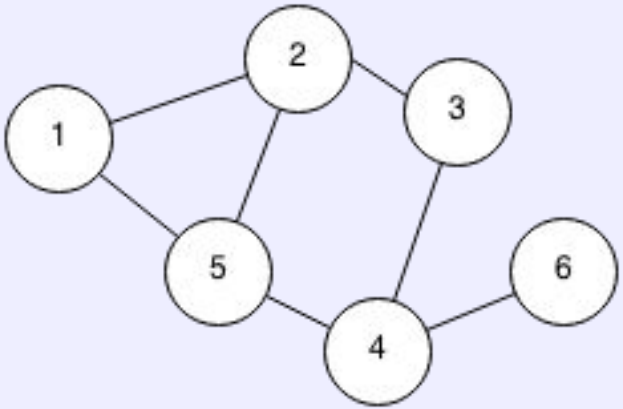
Graph Basics

by Emma and Maha

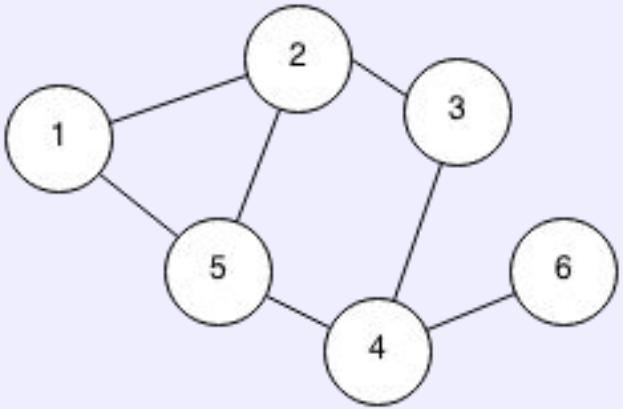
What is a Graph?



Graph: structure/concept that expresses a set of objects and the *relationships* between them (for mostly nonlinear data)

- **Leonhard Euler** → author of first graph theory paper (1736)

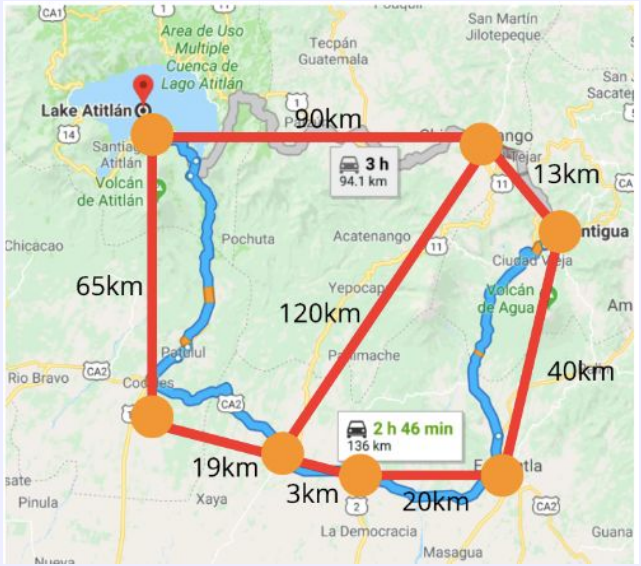
What is a Graph?



Key Terms:

- **vertices:** objects (dots)
- **edges:** connections/links (lines)
- **path:** sequence of edges that can be followed from one vertex to another
  - (path from 1 to 4?)
- 2 vertices are **adjacent** if directly connected by an edge
- edge e is **incident** to vertex v if v is an endpoint of e

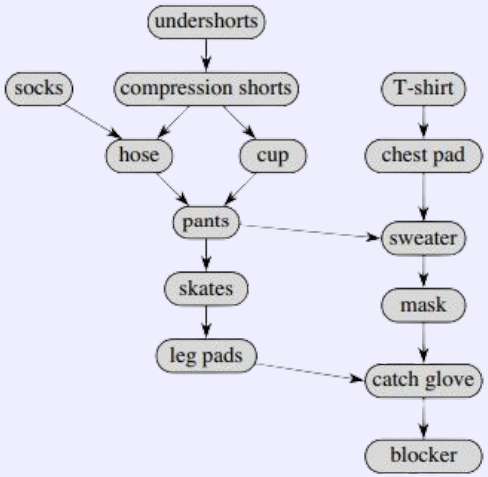
What is a Graph?

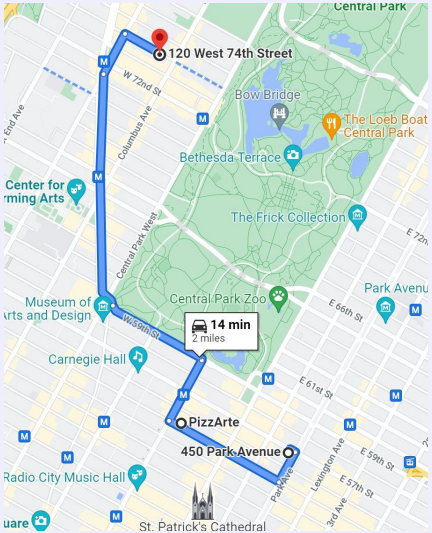


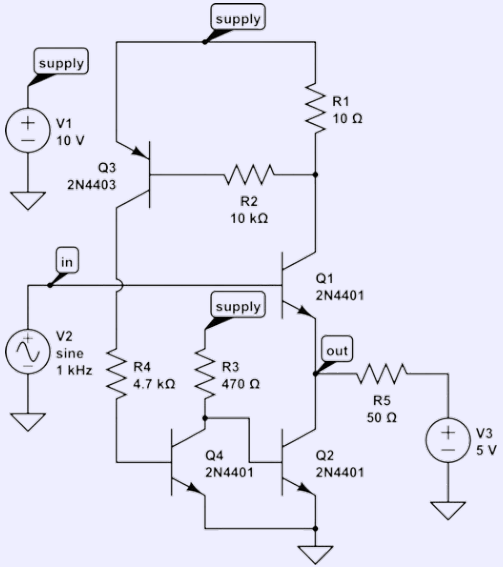
Supplementary Terms:

- degree: # edges connected to a vertex
  - in-degree; out-degree
- weight: value assigned to an edge
- loop: edge connecting vertex to itself
- parallel edge: multiple edges connect same pair of vertices

Why ~~What~~ is a Graph?

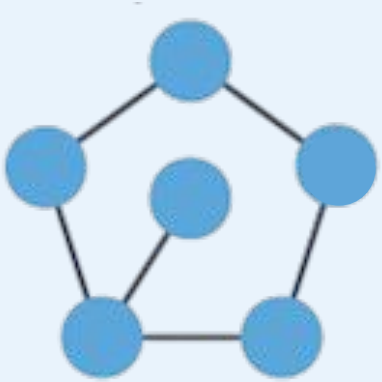






Types of Graphs

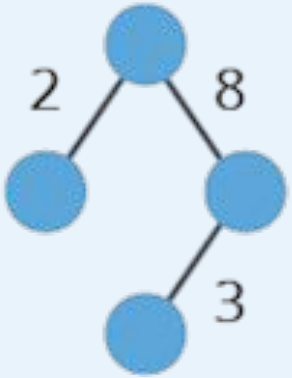
Sparse



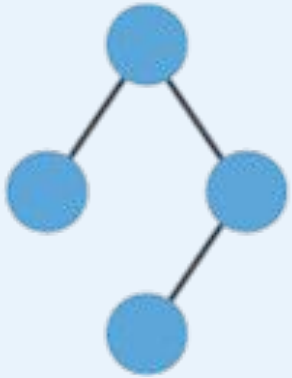
Dense

Types of Graphs

Weighted

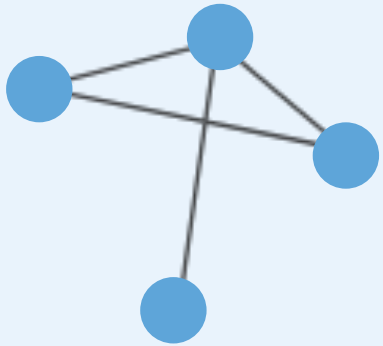


Unweighted

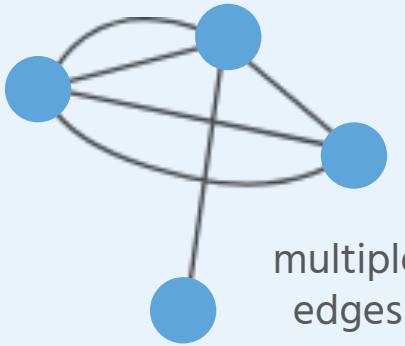


Types of Graphs

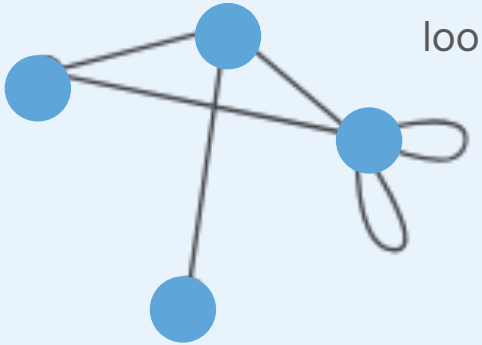
Simple



Non-Simple



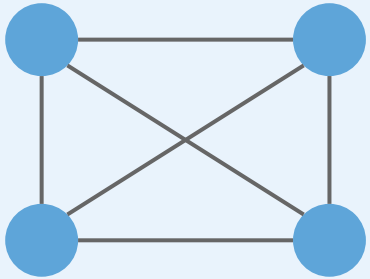
multiple edges



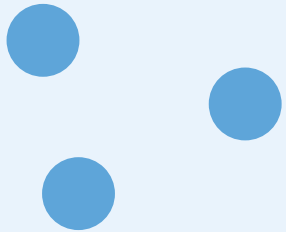
loops

Types of Graphs

Complete



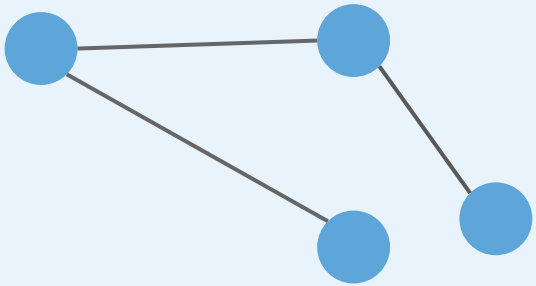
Null



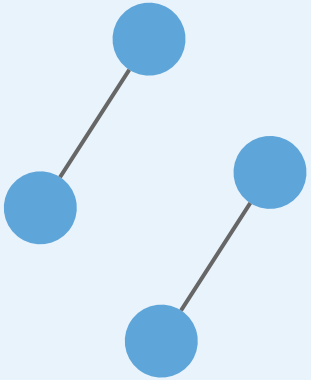
Trivial

Types of Graphs

Connected

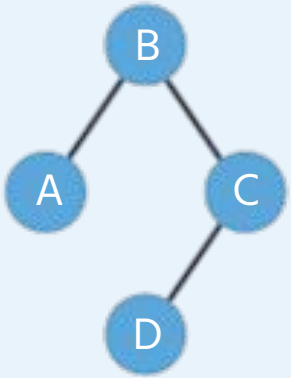


Disconnected

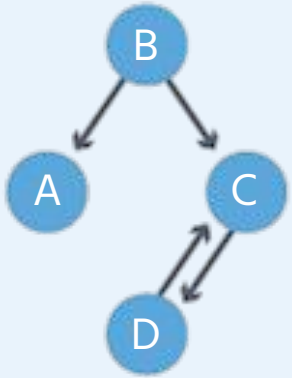


Types of Graphs

Undirected



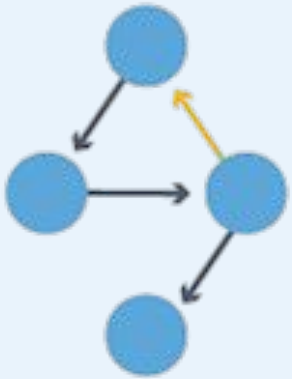
Directed



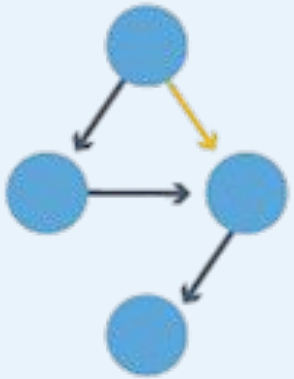
- in-degree
- out-degree

Types of Graphs

Cyclic



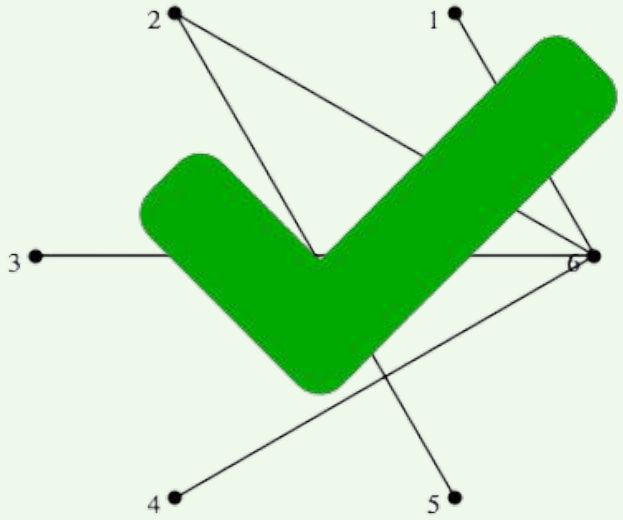
Acyclic



Time for a Game!

are the following graphs?

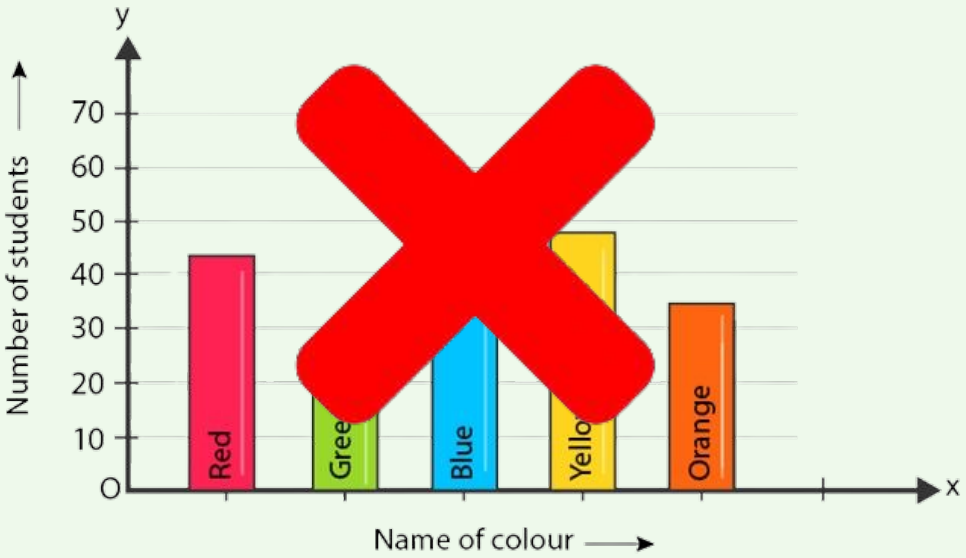
Is this a Graph?



Is this a Graph?

*bonus points if you can tell me **what kind** of graph!

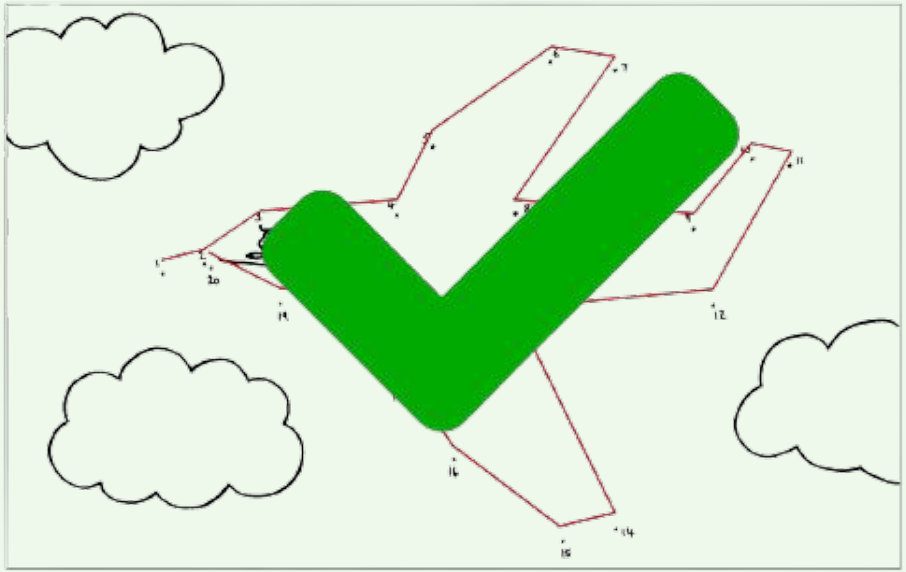
Is this a Graph?



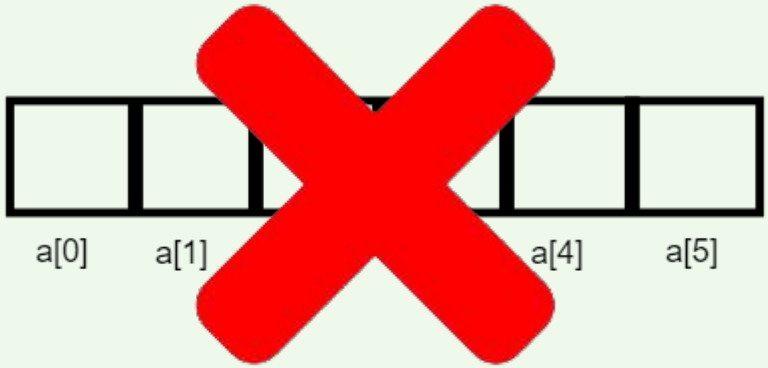
Is this a Graph?

*bonus points if you can tell me **what kind** of graph!

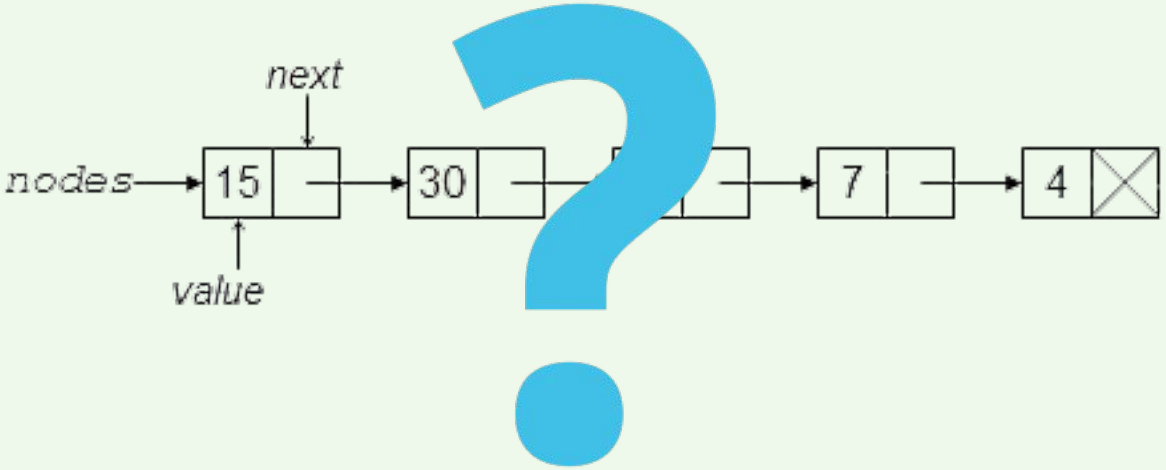
Is this a Graph?



Is this a Graph?



Is this a Graph?



Is this a Graph?

Representing Graphs

Typical graph operations (non-exhaustive):

- insert/delete vertex/edge
- list all vertices/edges
- are the vertices x and y adjacent?
- what/how many edges are connected to vertex x?

Main approaches to represent/store a graph:

- adjacency list
- adjacency matrix
- edge list

Representing Graphs

Typical graph operations (non-exhaustive):

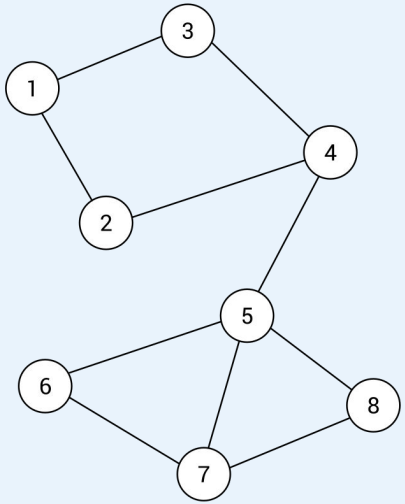
- insert/delete vertex/edge
- list all vertices/edges
- are the vertices x and y adjacent?
- what/how many edges are connected to vertex x?

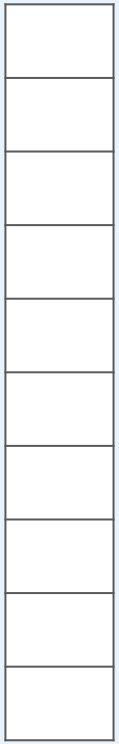
Main approaches to represent/store a graph:

- adjacency list
- adjacency matrix
- edge list

Edge List

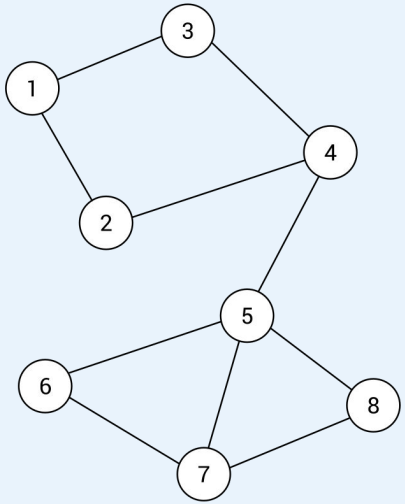
- List of pairs to represent an edge (A, B)
  - if directed → A: source; B: receiver





Edge List

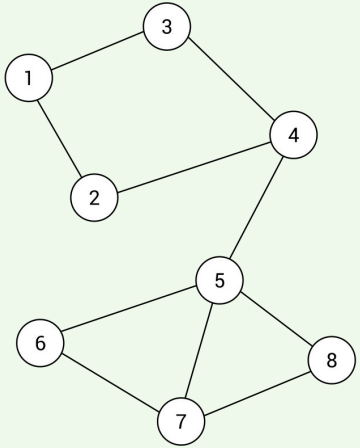
- List of pairs to represent an edge (A, B)
  - if directed → A: source; B: receiver
- pros: easy to implement
- cons: inefficient checking for the existence of an edge & finding all adjacent vertices of a given vertex

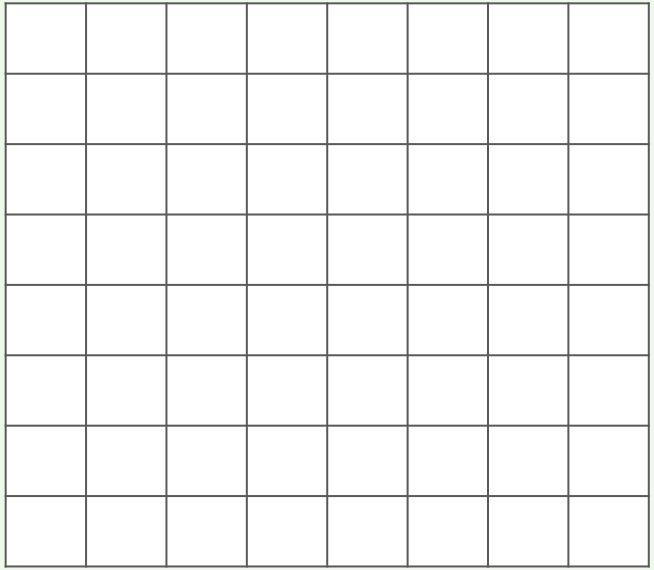


(1, 2)
(1, 3)
(2, 4)
(3, 4)
(4, 5)
(5, 6)
(5, 7)
(5, 8)
(6. 7)
(7, 8)

Adjacency Matrix

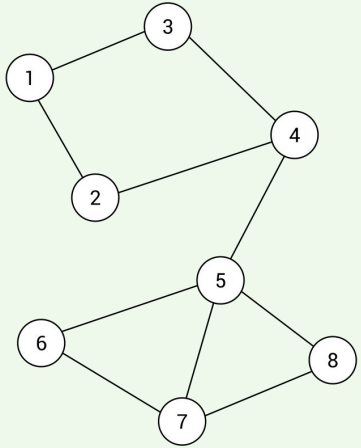
- N * N matrix
  - where N = # of vertices
- A[i][j] = 0 or 1
  - if edge 1
  - else 0





Adjacency Matrix

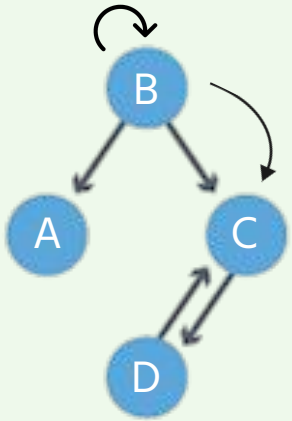
- undirected graphs have symmetrical adjacency matrices
- directed graphs have asymmetrical adjacency matrices

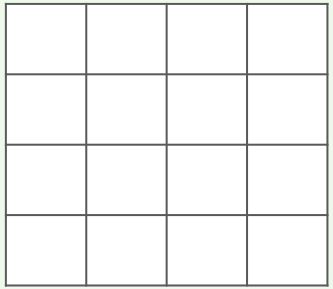


	1	1					
1			1				
1			1				
	1	1		1			
			1		1	1	1
				1		1	
				1	1		1
				1		1	

Adjacency Matrix

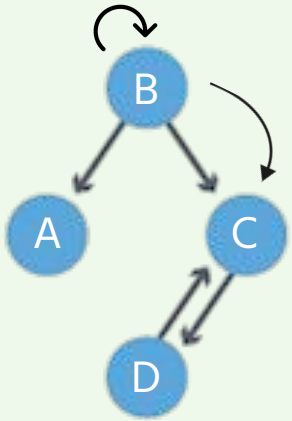
- try with directed graph:
- if graph has parallel edges, can't store boolean 1 or 0
  - store the number of edges in A[i][j]
- what happens with loops?





Adjacency Matrix

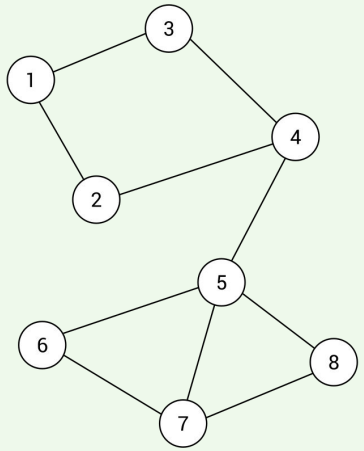
- try with directed graph:
- if graph has parallel edges, can't store boolean 1 or 0
  - store the number of edges in A[i][j]
- what happens with loops?



1	1	2	
			1
		1	

Adjacency Matrix

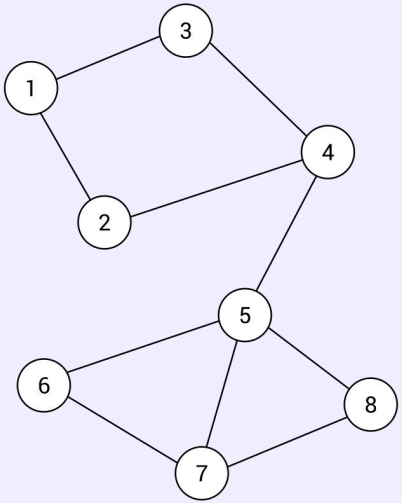
- pros:
  - O(1) constant time to check edge existence
  - easy to find adj. vertices
    - what vertices are adjacent to 4?
- cons:
  - N^2 space
  - hard to find in/out-degree

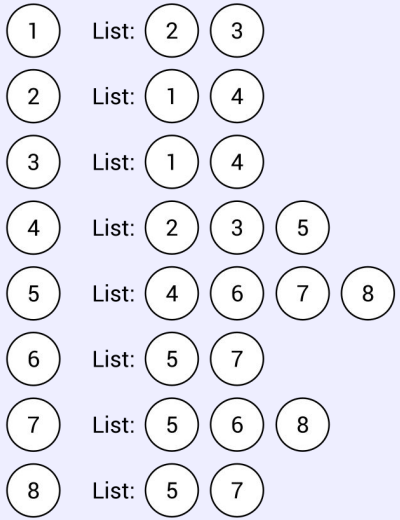


	1	1					
1			1				
1			1				
	1	1		1			
			1		1	1	1
				1		1	
				1	1		1
				1		1	

Adjacency List

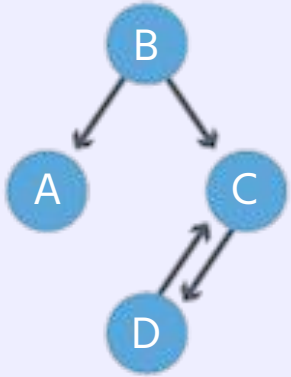
- edge list + adj. matrix hybrid
- array of N lists
  - where N = # of vertices
- A[1] = (linked) list of all vertices connected to vertex 1

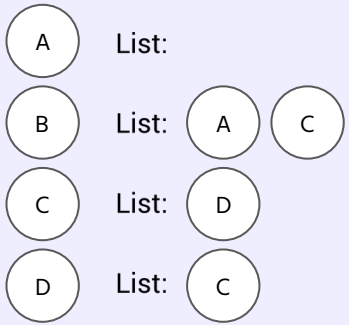




Adjacency List

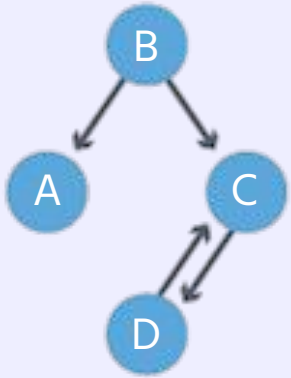
- directed graph
  - list receiving vertices
- slightly easier to count out-degree (in degree still not that good)

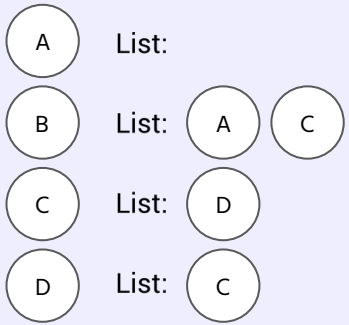




Adjacency List

- pros:
  - more dynamically sized
  - less space (N + M)
    - N = # vertices
    - M = # edges
- cons:
  - slower edge existence checking
    - (worst case O(M)?)



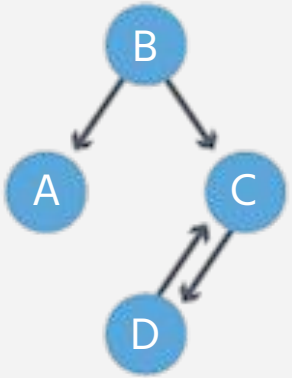


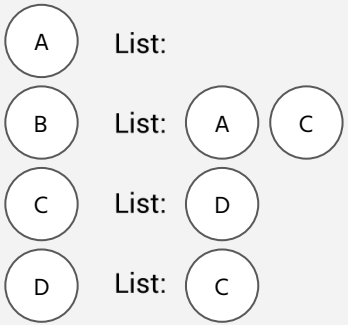
Which one to Pick?

Space Efficiency:

- sparse vs dense?

1		1	
			1
		1	



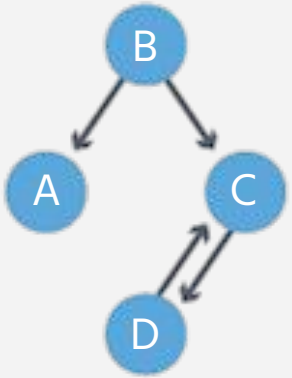


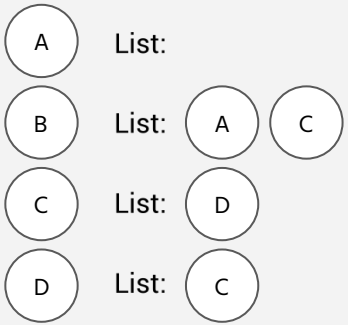
Which one to Pick?

Time Efficiency:

- add an edge?
- find list of adjacent vertices?

1		1	
			1
		1	



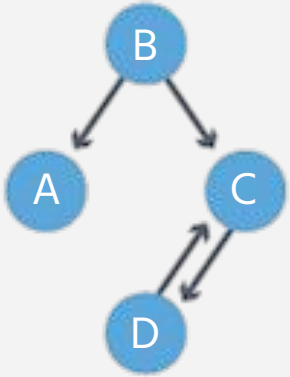


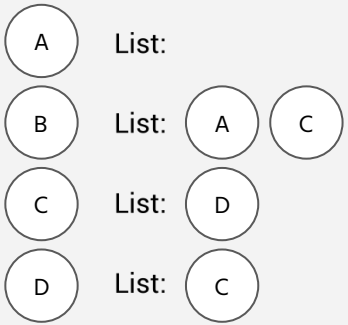
Which one to Pick?

Performance Estimate:

- what is the worst case of determining if vertex v is adjacent to vertex w?

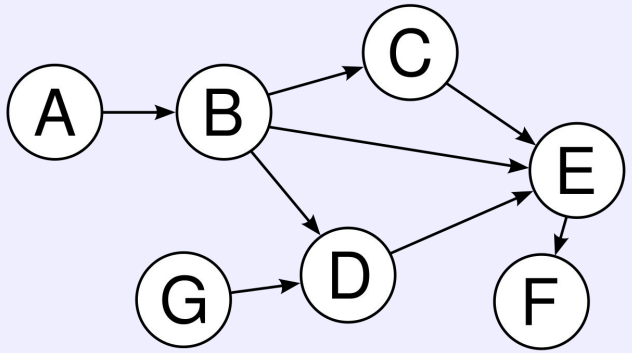
1		1	
			1
		1	



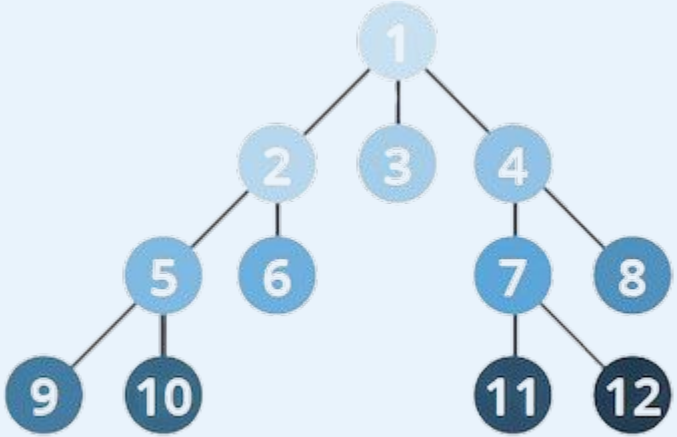


Directed Acyclic Graphs

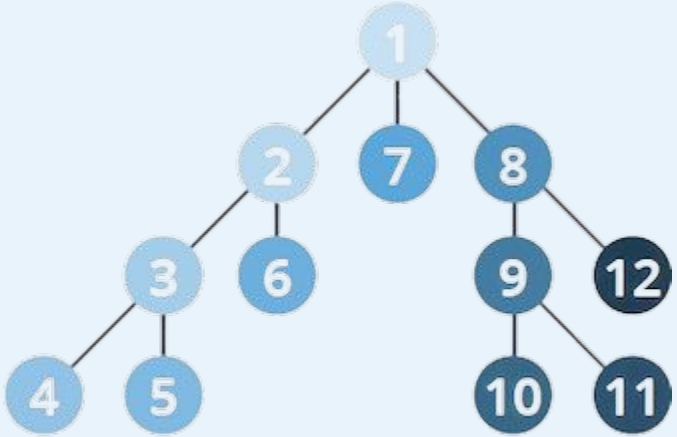
- Combination of a directed + acyclic graph:
  - Directed : arrows direction represents some relationship between the nodes
  - Acyclic : no cycles
- Used to represent causal relationships and model dependencies.



Graph Traversal



Breadth-First



Depth-First

Breadth First Traversal

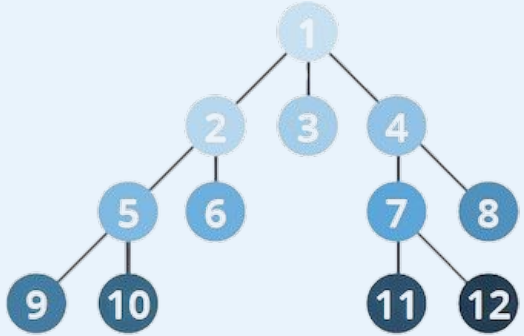
```
BFS(graph, start vertex):
    visited = set()
    queue = Queue()

    // Mark the start vertex as visited and enqueue it
    visited.add(start vertex)
    queue.enqueue(start_vertex)

    // Begin BFS traversal
    while queue is not empty:
        // Dequeue a vertex from the queue
        current_vertex = queue.dequeue()

        // Process the current vertex (optional)

        // Visit all unvisited neighbors of the current vertex
        for each neighbor of current vertex:
            if neighbor is not in visited:
                // Mark neighbor as visited and enqueue it
                visited.add(neighbor)
                queue.enqueue(neighbor)
```



# of edges = m, # of vertices = n
Time Complexity: O(m + n)
Space Complexity: O(m + n)
(typically on unweighted graphs)

Depth First Traversal

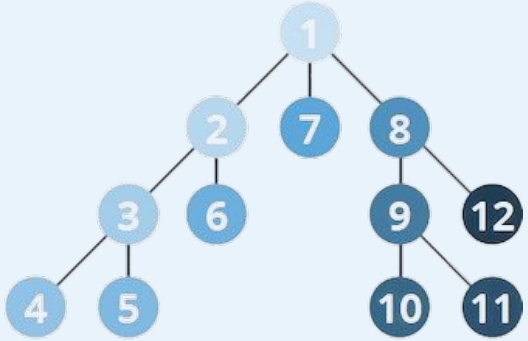
```
DFS(graph, start vertex):
    // Initialize data structures
    visited = set()

    // Call recursive helper function to perform DFS
    DFSHelper(graph, start_vertex, visited)

DFSHelper(graph, current_vertex, visited):
    // Mark the current vertex as visited
    visited.add(current_vertex)

    // Process the current vertex if needed (optional)
    // Example: print(current_vertex)

    // Explore all unvisited neighbors of the current
vertex recursively
    for each neighbor of current vertex:
        if neighbor is not in visited:
            DFSHelper(graph, neighbor, visited)
```



# of edges = m, # of vertices = n
Time Complexity: O(m + n)
Space Complexity: O(n)
(typically on unweighted graphs)

Honorable Mentions (weighted)

1. Dijkstra's Algorithm

2. A* Search Algorithm

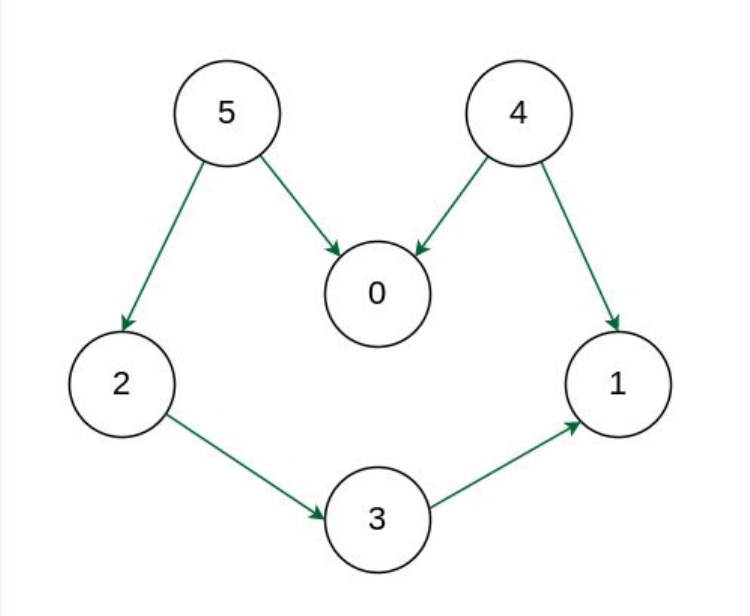
3. Bellman-Ford Algorithm

Uses:

- Network routing
- Mapping
- Navigation
- Game development
- Robotic
- Network optimization

Topological Sorting

- Only for DAGs (sometimes called linearization)
- Linear ordering of vertices such that for every directed edge u-v, vertex u comes before v in ordering.
- Useful for task ordering/ scheduling
- The graph needs to have at least 1 node with an in-degree = 0 (Cormen 77)



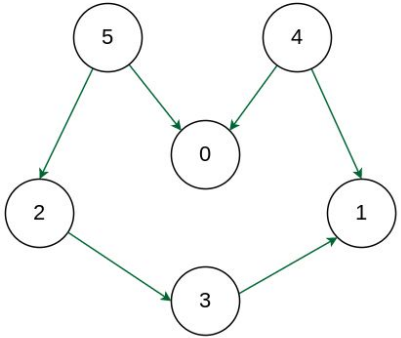
Topological Sorting

Procedure TOPOLOGICAL-SORT(G)

Input: G: a directed acyclic graph with vertices numbered 1 to n.

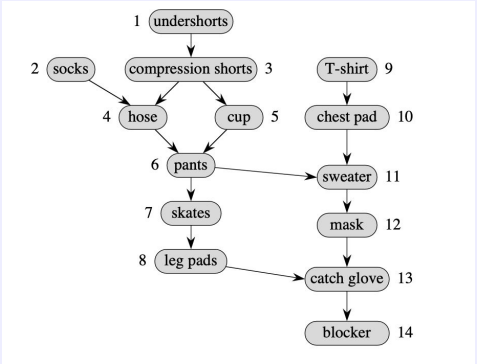
Output: A linear order of the vertices such that u appears before v in the linear order if (u, v) is an edge in the graph.

1. Let *in-degree*$[1..n]$ be a new array, and create an empty linear order of vertices.
2. Set all values in *in-degree* to 0.
3. For each vertex u:
   A. For each vertex v adjacent to u:
      i. Increment *in-degree*$[v]$.
4. Make a list *next* consisting of all vertices u such that *in-degree*$[u] = 0$.
5. While *next* is not empty, do the following:
   A. Delete a vertex from *next*, and call it vertex u.
   B. Add u to the end of the linear order.
   C. For each vertex v adjacent to u:
      i. Decrement *in-degree*$[v]$.
      ii. If *in-degree*$[v] = 0$, then insert v into the *next* list.
6. Return the linear order.



This is Kahn's implementation, authored by Arthur Kohlberg Kahn. He introduced it in 1962.

There is a DFS implementation of topological search, where the program recurses over the remaining vertices instead of removing explored vertices.



Runtime + space

Run time for topological sorting is **Θ(n + m)** where **n** is the number of vertices and **m** is the number of edges.
Worst-case:

- **Complete directed graph = O(n*n)**
- **DAG (Kahn's topological sort) = O(n + m)**

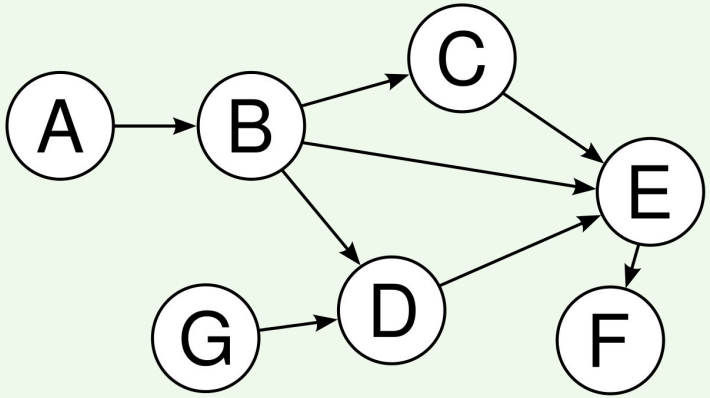
- **Space complexity for Kahn's topological sort is O(n)**
  - In-degree array = O(n)
  - 'Next' list = O(n)
  - Linear Order List = O(n)
  - Other variables have constant space complexity

*** n = number of vertices**

Paths / Critical Paths

- **Path** = “A ***path*** in a graph is a sequence of vertices and edges that allow you to get from one vertex to another (or back to itself)” (Cormen 82)

- **Critical Path** = longest path in a graph + Sum of task times add up to maximum + Represents the sequence of tasks that need to be completed without delay to minimize the project duration

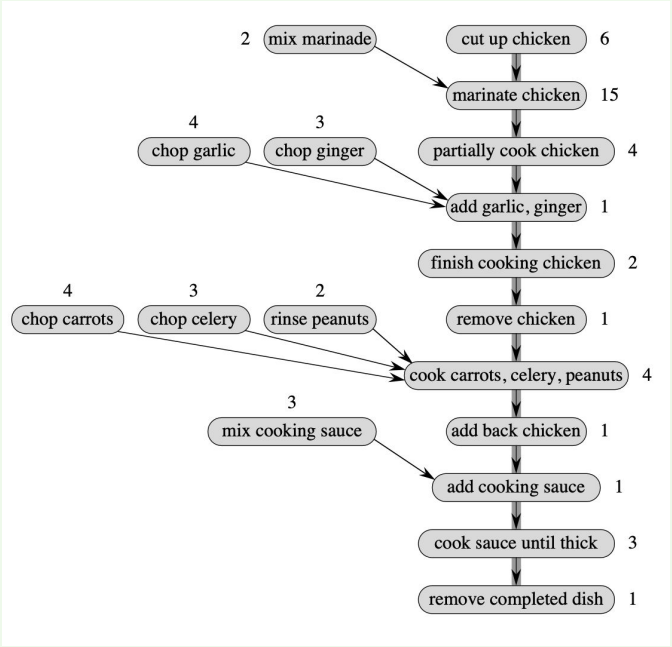


What is the critical path from source (A) to sink (F)? Is there more than one?

PERT Charts

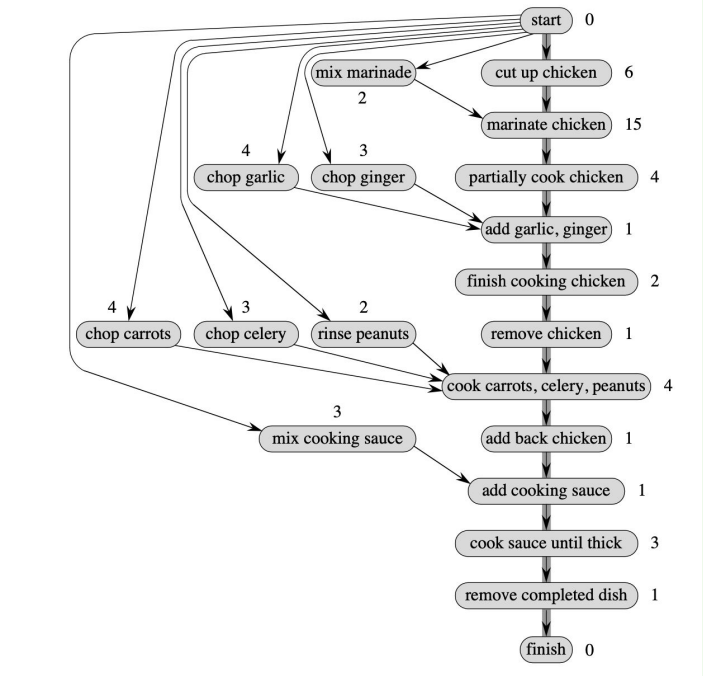
Program **E**valuation and **R**eview **T**echnique
- Project management tool
- Visual representation of tasks needed to be completed in order to finish a project
- Determines project duration

Cormen uses the example of making Kung Pao Chicken (81).



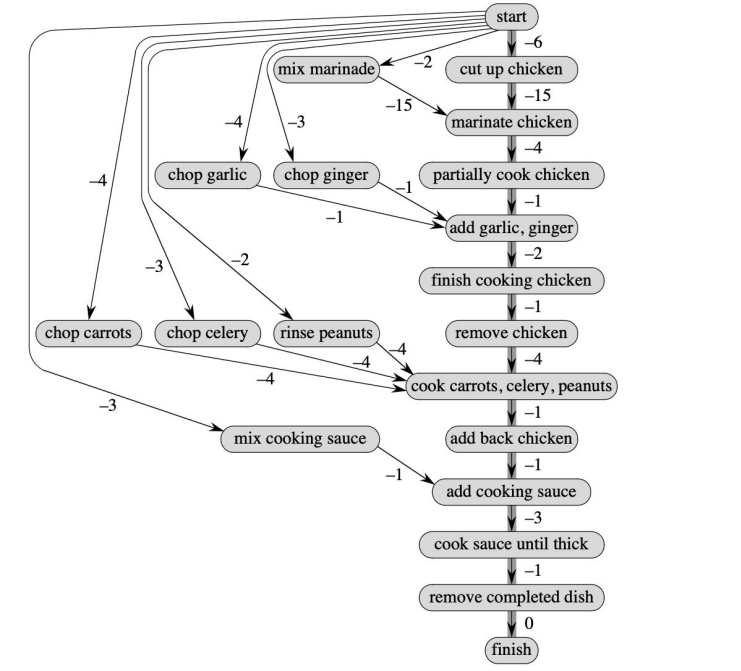
Critical paths: PERT Charts

- Dummy start and end vertices with time = 0
- Task times (weights) associated with vertices rather than edges: weight of path = sum of weights of vertices
- **Shortest path :** Negate the task times!!! The shortest path would be the minimum sum of weights from the start to the end.



Weighted Directed Graphs

- Pushing task times onto the edges going out of a node and negating them converts a PERT chart into a **weighted directed graph**
- The **shortest path** is the one with most negative sum of negated task times
- Shortest path in a WDG corresponds to a critical path in a PERT chart



UP NEXT...

Shortest path in a Directed Acyclic Graph:

- Gives us an idea of how to find the shortest path in all directed graphs (see Cormen pages 85 to 88).

The “Death of a Salesman” Puzzle – Traveling sales from Washington to Washington, DC

Sources

"Applications of Graph Data Structure." *GeeksforGeeks*, 16 Aug. 2018, www.geeksforgeeks.org/applications-of-graph-data-structure/.

Cormen, Thomas H. *Algorithms Unlocked*. MIT Press, 2013.

"Graph Definitions." *EECS University of California, Berkeley*, inst.eecs.berkeley.edu/~cs61bl/r//cur/graphs/definitions.html?topic=lab23.topic&step=3&course=. Accessed 6 Mar. 2024.

"Graph Theory." *Wikipedia*, 31 Aug. 2020, en.wikipedia.org/wiki/Graph_theory#History.

"Graph Theory - the Computer Science Handbook." *Www.thecshandbook.com*, www.thecshandbook.com/Graph_Theory.

Hoppa, Jocelyn. "Graph Algorithms in Neo4j: Graph Algorithm Concepts." *Neo4j Graph Data Platform*, 26 Nov. 2018, neo4j.com/blog/graph-algorithms-neo4j-graph-algorithm-concepts/.

https://www.geeksforgeeks.org/topological-sorting/

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/