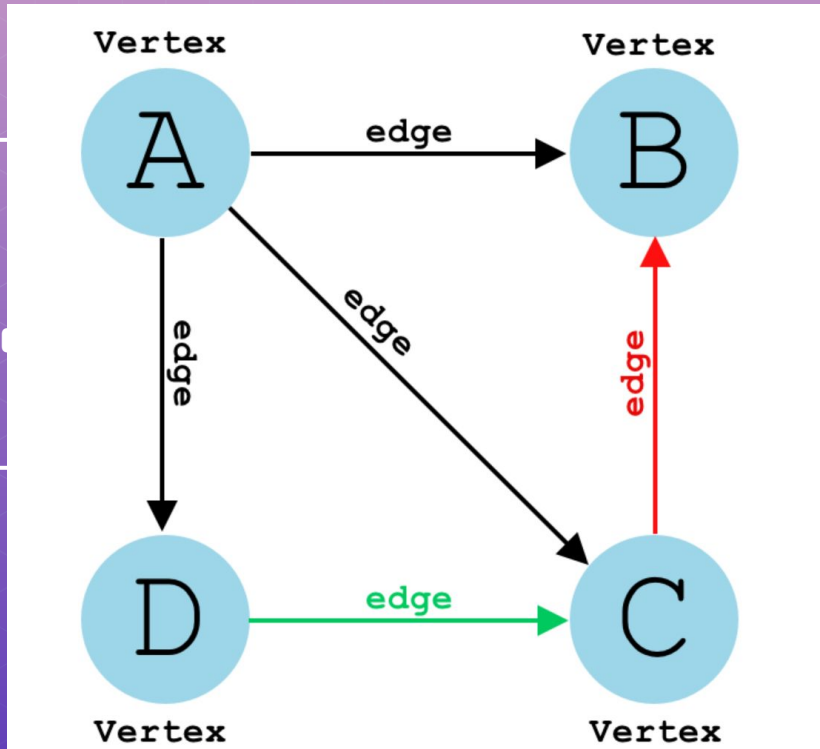
Graph

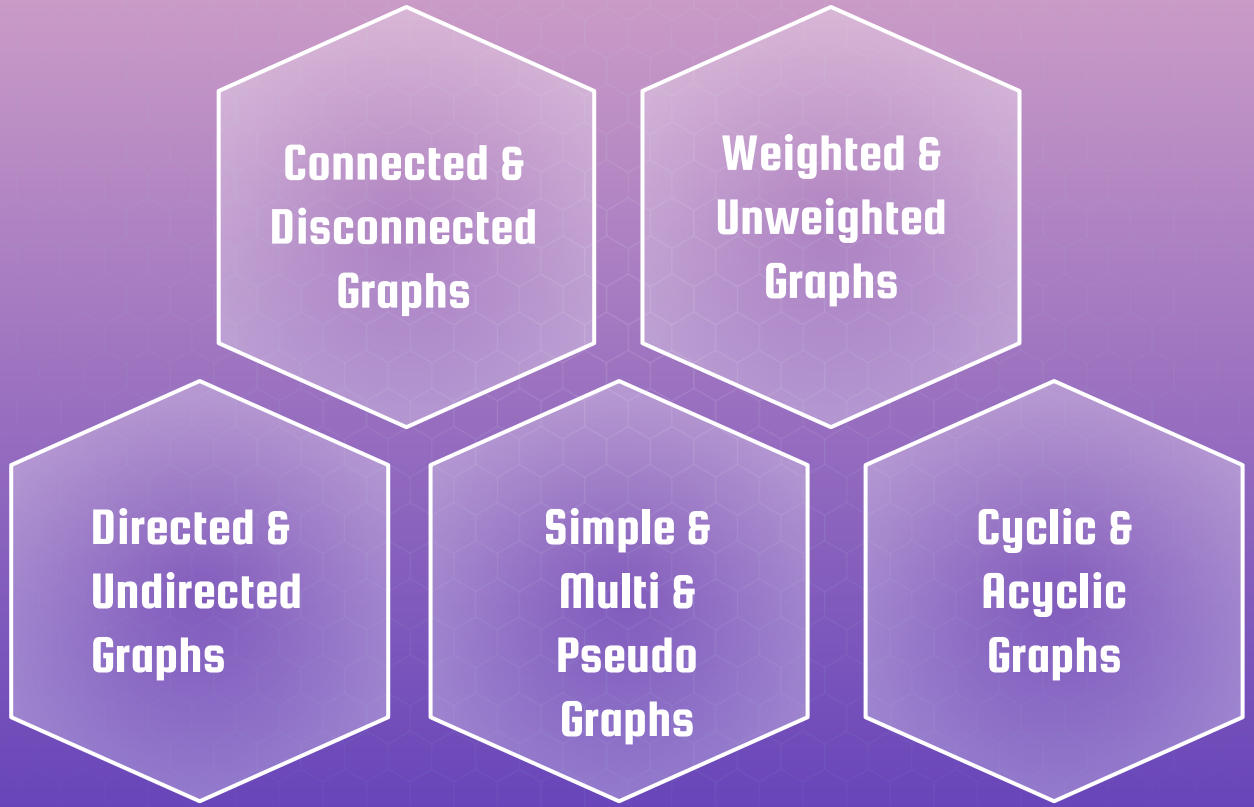
Hengjun Pei

What is a Graph?

Definition: A mathematical structure that represents a function by connecting a set of points, called vertices, and lines between those points, called edges.



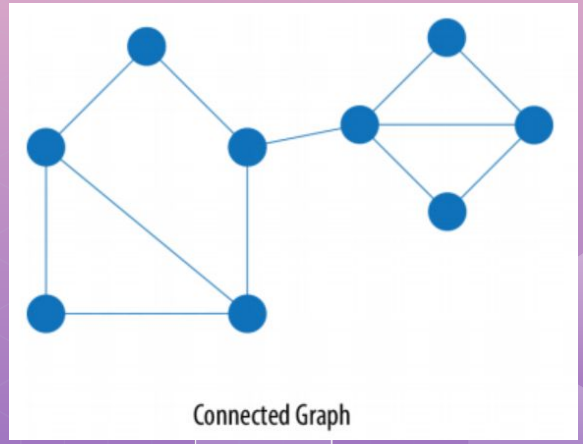
Graph Characteristics

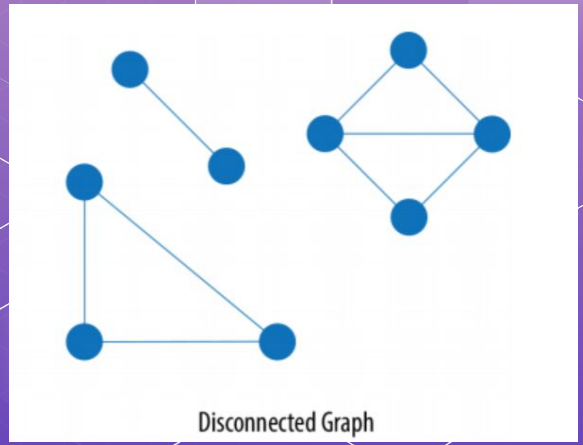


Connected & Disconnected Graphs

Connected Graph – A graph is a connected graph if, for each pair of vertices, there exists at least one single path which joins them.

Disconnected Graph – A disconnected graph is a graph that is not connected. There is at least one pair of vertices that have no path connecting them.

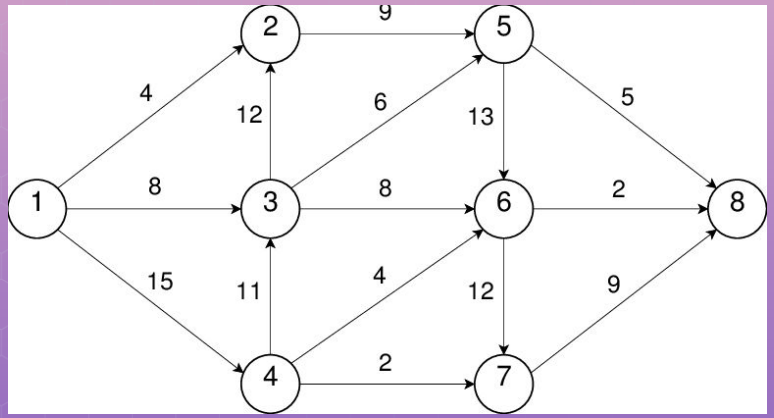


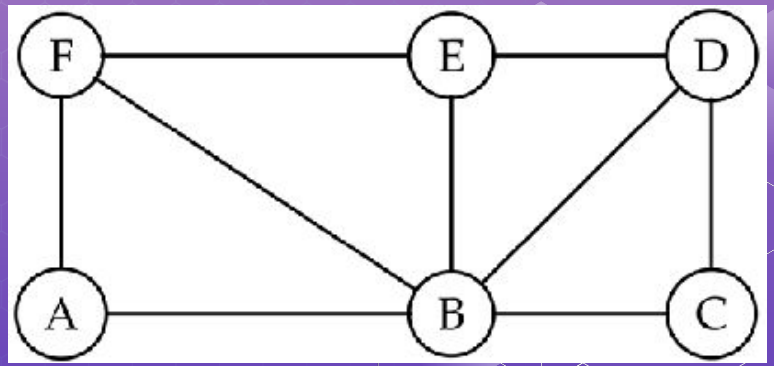


Weighted & Unweighted Graphs

Weighted Graph – A weighted graph is a graph in which a number (the weight) is assigned to each edge

Unweighted Graph – An unweighted graph is a graph where the edges do not have any weight or cost associated with them.

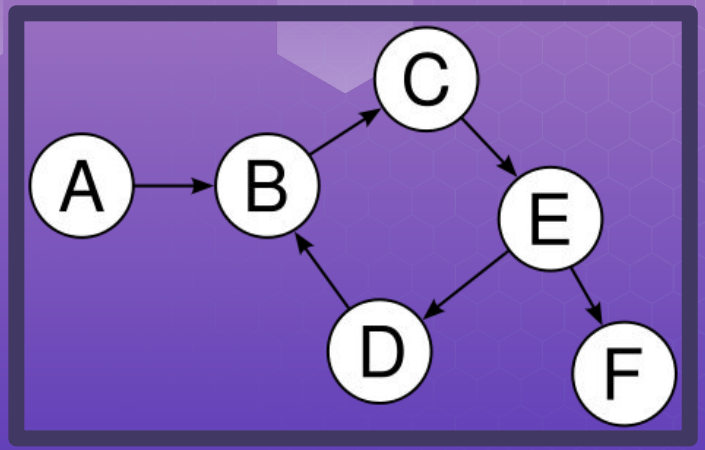


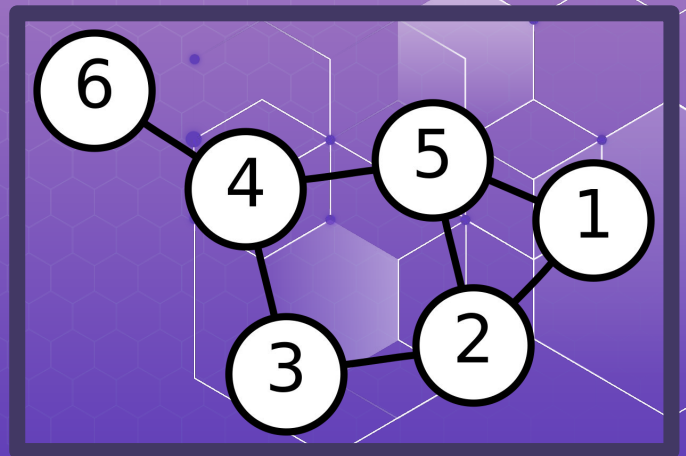


Directed & Undirected Graphs

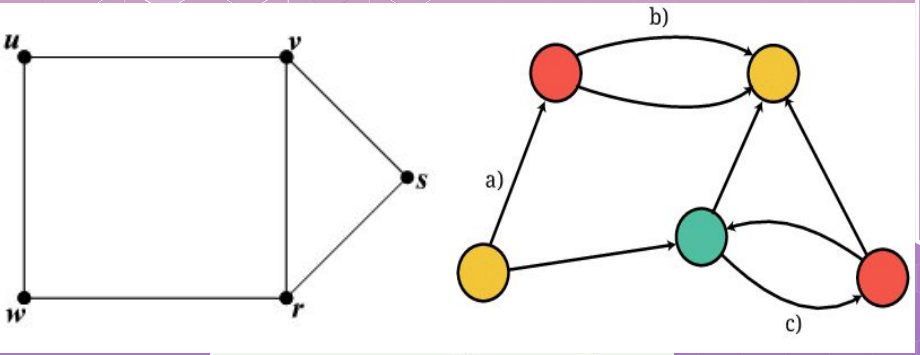
Directed Graph – A directed graph, also known as a digraph, is a graph made up of a set of vertices, or nodes, connected by directed edges, or arcs.

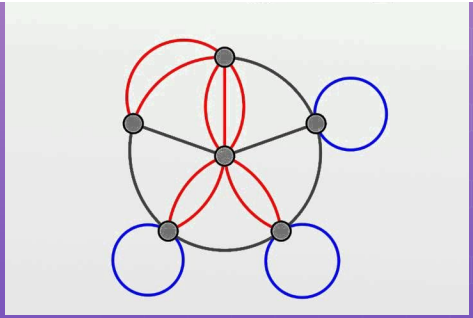
Undirected Graph – An undirected graph is a graph where edges have no direction, and nodes can be connected in either direction.





Simple & Multigraph & Pseudo Graphs





Simple Graph – A simple graph is an undirected, unweighted graph with no loops or multiple edges.

Multigraph – A multigraph is a graph in graph theory that allows multiple edges, or parallel edges, to connect the same pair of vertices.

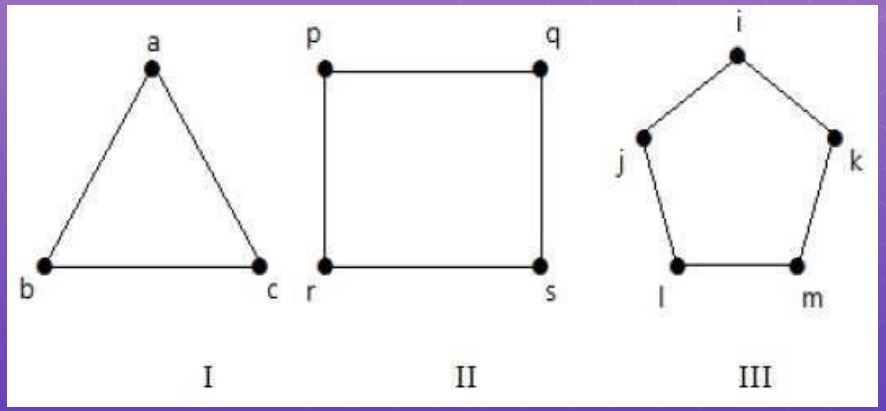
Puedo Graph – a multigraph with multiple edges and loops, or edges that connect a vertex to itself.

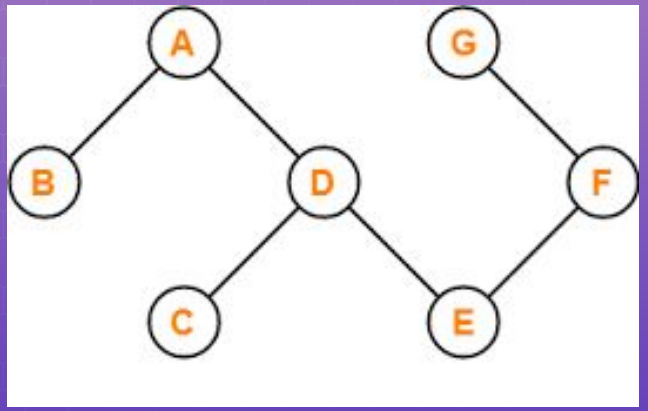
*Note – The number of edges between two vertices is called the edge multiplicity.

Cyclic & Acyclic Graphs

Cyclic Graph – A cyclic graph, also known as a circular graph, is a graph that contains a cycle, or a closed chain of vertices.

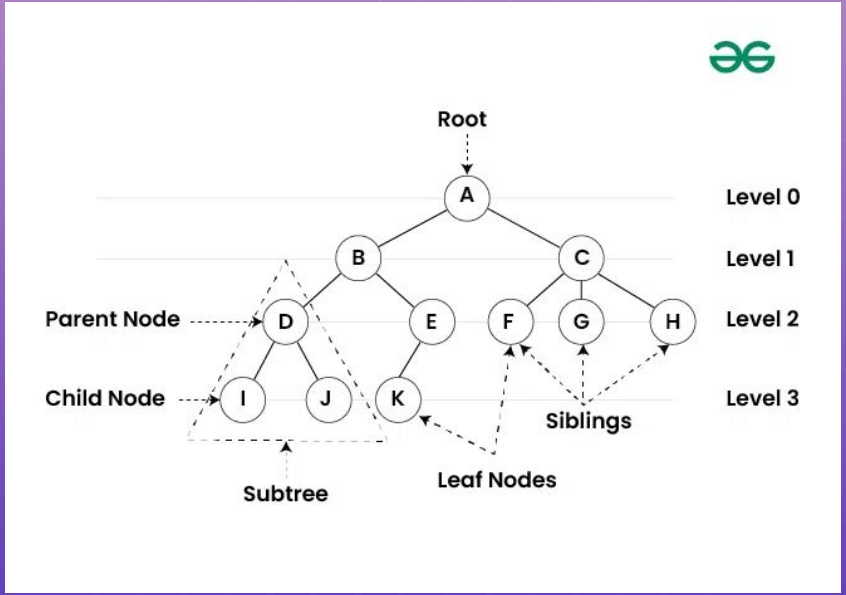
Acyclic Graph – An acyclic graph is a graph that has no cycles, or closed paths.





Tree Structure

A tree data structure is a collection of nodes connected by edges and represents a hierarchical tree structure with a set of connected node. The first node of the tree is called the root and the most bottom nodes are called leaves. Every Tree is a graph, but not every graph is a tree.



Graph representation

Adjacency Set

Adjacency List

TABLE 2 An Adjacency List for a Directed Graph.

Initial Vertex	*Terminal Vertices*
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

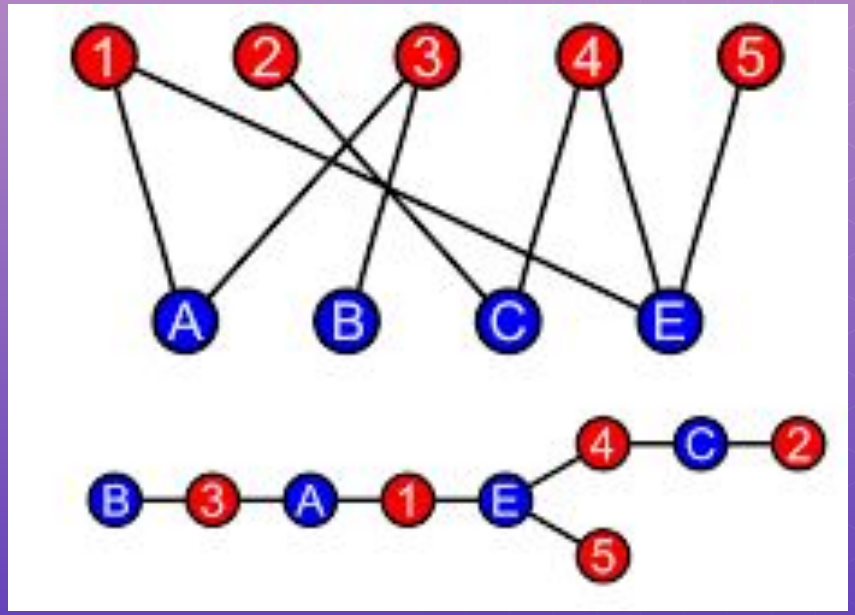
Adjacency Matrix

*Example:

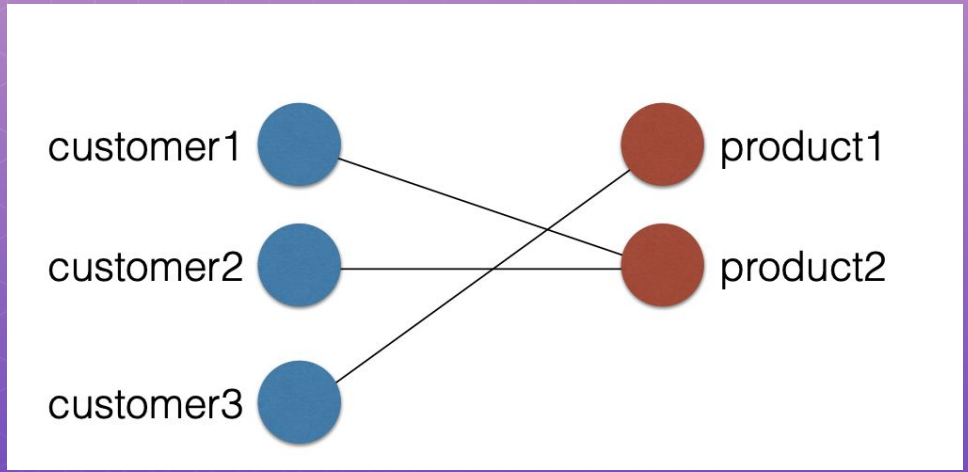
	A	B	C	D	E
A	0	0	0	0	1
B	0	0	1	0	0
C	0	1	0	0	1
D	1	0	0	1	0
E	0	1	1	0	0

Bipartite Graph

Definition - A bipartite graph, also known as a bigraph, consists of a set of graph vertices divided into two separate sets in such a way that no two graph vertices within the same set are adjacent.

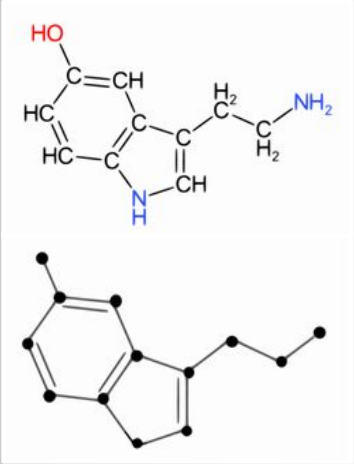


Real World Application



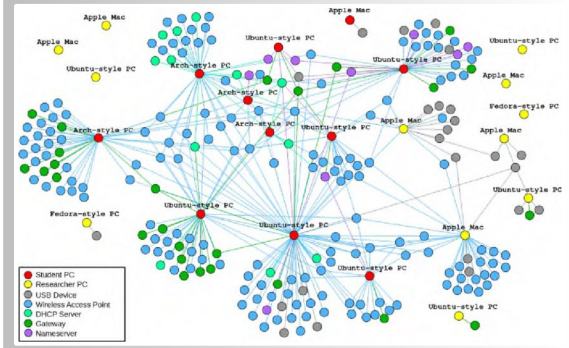
Graph Applications in Real Life

Chemical Graph



Transportation Map Graph

Network Graph



Breadth First Search

- **Traversal Mechanism** : It starts from a given vertex and explores all its neighbors level by level before moving on to vertices at the next level.
- **Implementation** : Can be implemented using a queue.
- **Applications** : Used in finding the shortest path in unweighted graphs, level-order traversal in trees, and in algorithms like Ford-Fulkerson for computing maximum flow in a flow network.
- **Complexity** : The time complexity is O(V+E) where V is the number of vertices and E is the number of edges.

```
BFS(graph, start_vertex):
    let queue be a queue
    queue.enqueue(start_vertex)
    visited[start_vertex] = true
    while queue is not empty:
        vertex = queue.dequeue()
        for each neighbor in graph[vertex]:
            if not visited[neighbor]:
                queue.enqueue(neighbor)
                visited[neighbor] = true
```

Depth First Search

- **Traversal Mechanism:** It goes deep into the graph by moving to an adjacent unvisited vertex and repeats the process until it can no longer go further, at which point it backtracks to the most recent vertex with unexplored neighbors.
- **Implementation:** Can be implemented using a stack or recursively.
- **Applications:** Used in topological sorting, finding connected components, solving puzzles (like mazes), and pathfinding in games.
- **Complexity:** The time complexity is O(V+E) where V is the number of vertices and E is the number of edges.

```
DFS(graph, start_vertex):
    let stack be a stack
    stack.push(start_vertex)
    while stack is not empty:
        vertex = stack.pop()
        if vertex is not visited:
            visited[vertex] = true
            for each neighbor in graph[vertex]:
                if not visited[neighbor]:
                    stack.push(neighbor)
```

Dijkstra's Algorithm

Mechanism : Dijkstra's Algorithm operates by maintaining a set of known shortest distances from the source to each vertex, progressively expanding this set by exploring the closest least valued unexplored vertices.

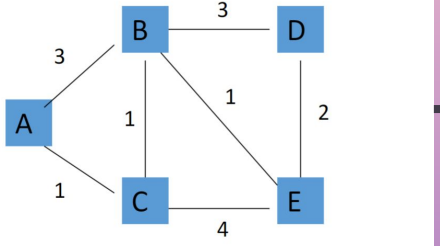
Implementation: Use a priority queue (min-heap) to keep track of vertices to explore, ordered by their current shortest distance. Insert the source vertex with a distance of 0.

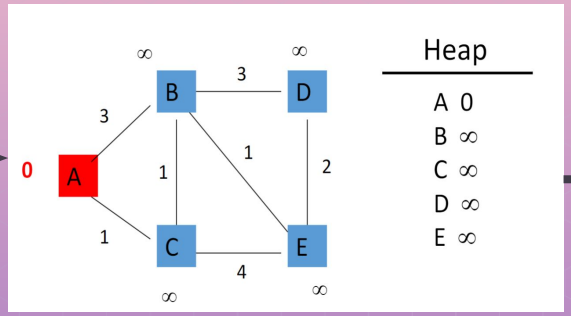
Application: Dijkstra's Algorithm efficiently finds the shortest path from a starting vertex to all other vertices in a graph with edge weights, making it essential for various applications in routing, navigation, and optimization problems.

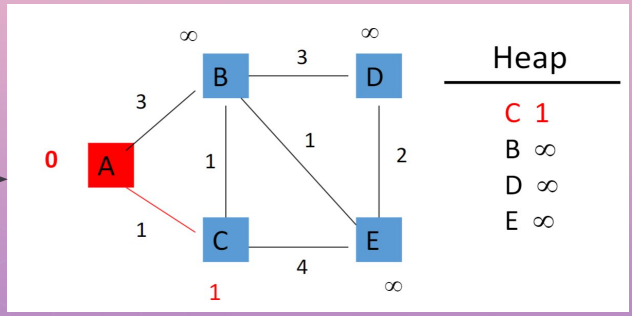
Complexity: O(V^2) when using an adjacency matrix representation of a graph, and O((V + E)logV) when using an adjacency list representation.

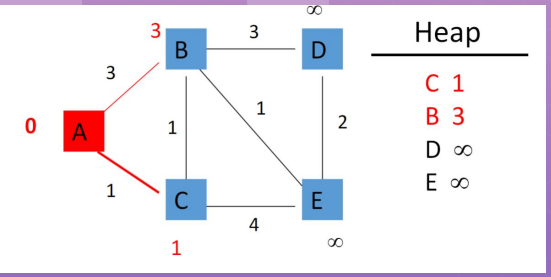
```
for each vertex v:
    dist[v] = ∞
    prev[v] = none
dist[source] = 0
set all vertices to unexplored
while destination not explored:
    v = least-valued unexplored vertex
    set v to explored
    for each edge (v,w)
        if dis[v] + len(v,w) < dist[w]:
            dis[w] = dist[v] + len(v,w)
            prev[w] = v
```

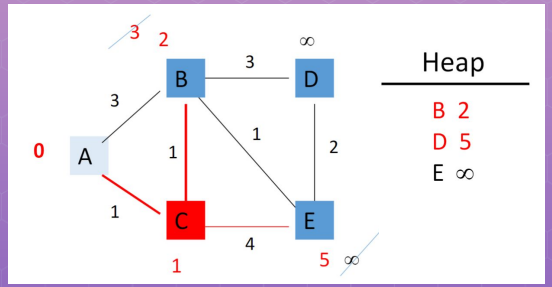
**Only works when there is no negatively weighted edges

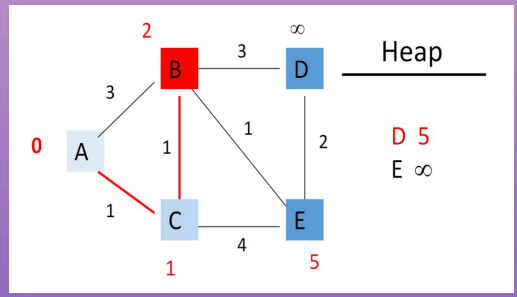

A
B
C
D
E
3
3
1
1
1
2
4

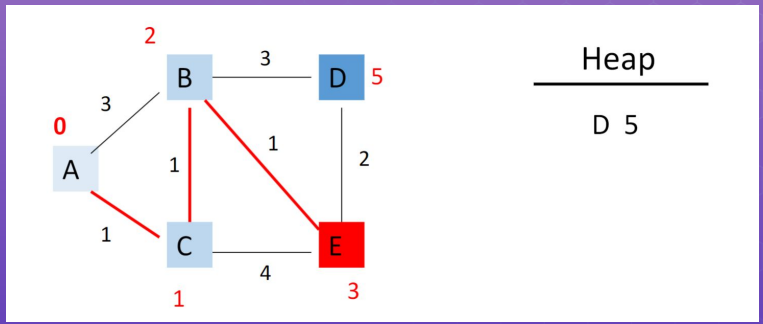

∞
∞
0
∞
∞
Heap
A 0
B ∞
C ∞
D ∞
E ∞

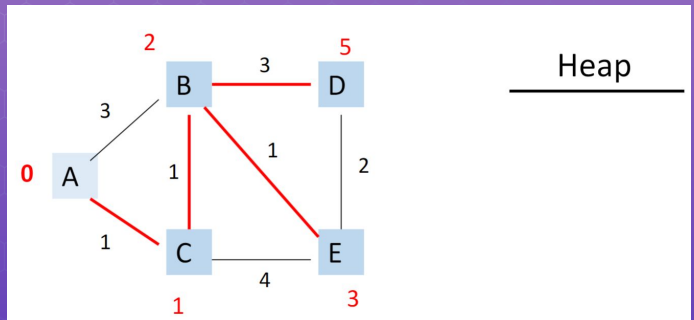

∞
∞
0
1
Heap
C 1
B ∞
D ∞
E ∞


3
∞
0
1
∞
Heap
C 1
B 3
D ∞
E ∞


3
2
0
1
5
∞
Heap
B 2
D 5
E ∞


2
∞
0
1
5
Heap
D 5
E ∞


2
5
0
1
3
Heap
D 5


2
5
0
1
3
Heap

Bellman–Ford's Algorithm

Mechanism: The Bellman–Ford algorithm works by grossly underestimating the length of the path from the starting vertex to all other vertices. The algorithm then iteratively relaxes those estimates by discovering new ways that are shorter than the previously overestimated paths.

Implementation: The algorithm can use the adjacency list to keep track of the path from source to each vertex.

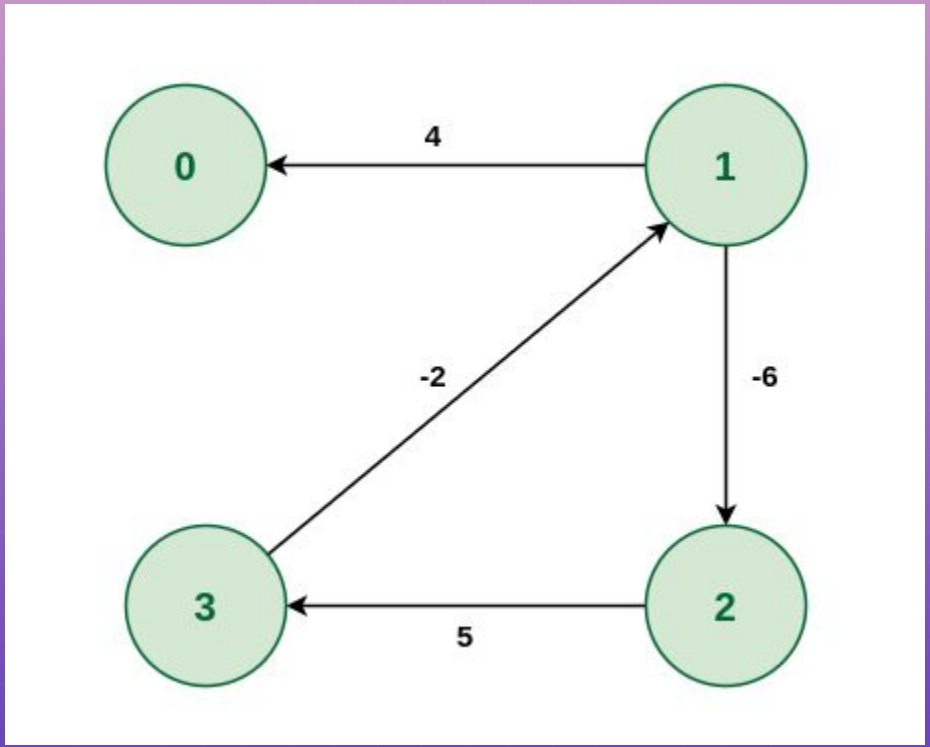
Application: Bellman–Ford's Algorithm can find the shortest path in graphs with negative weights such as networks where link costs may vary, potentially including negative values, and finance where detecting arbitrage opportunities in currency exchange.

Complexity: The Bellman–Ford algorithm has a time complexity of O(VE).

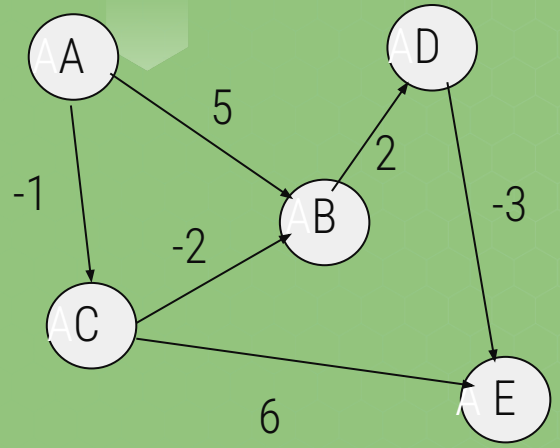
```
Bell-Ford Algorithm:
    For each vertex v
        dist[v] = ∞
        dist[source] = 0
    for i = 1 to |V|-1
        for each edge (u,v)
            if d[v] > d[u] + w(u,v)
                then d[v] = d[u] + w(u,v)
    For each edge (u,v)
        if d[v] > d[u] + w(u,v)
            then a negative-weight cycle exists
```

* *Works for graphs with negative edges but not with the negative cycle

Negative Cycle



How is works?



	A	B	C	D	E
d(0)	0	∞	∞	∞	∞
d(1)	0	5	-1	7	4
d(2)	0	-3	-1	-1	-4
d(3)	0	-3	-1	-1	-4
d(4)	0	-3	-1	-1	-4

THANKS