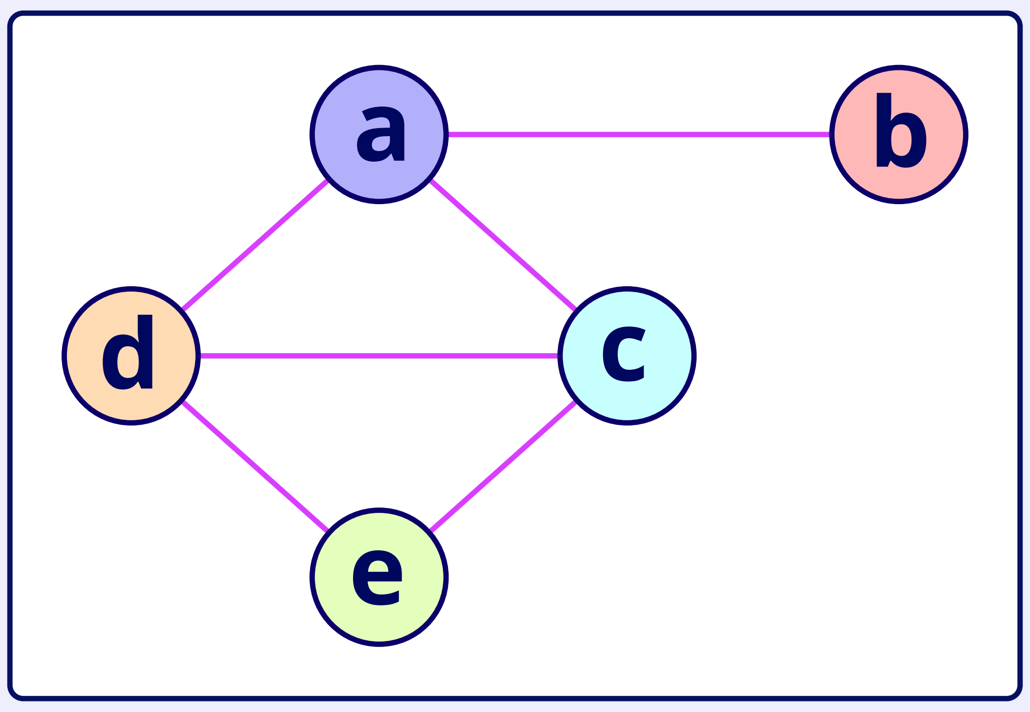
Data structures - 301

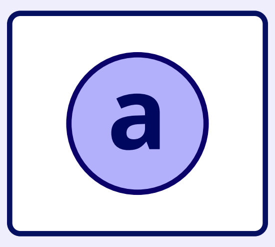
General Concepts

Graph: A graph is a collection of two sets of entities called the vertices and the edges.



A Graph

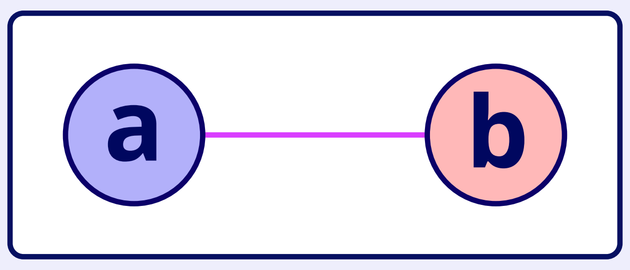
Vertex: A vertex is a point in a graph where two lines intersect; it is typically illustrated as a circle.



A vertex assigned a label “a”

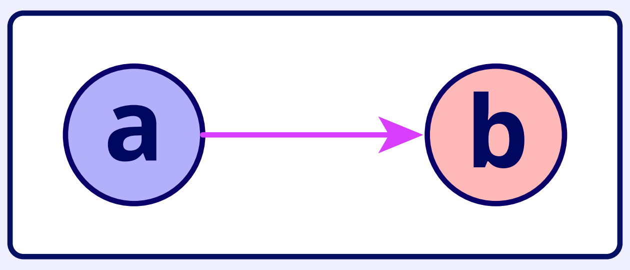
Edge: An edge is a relationship between two vertices in a graph, or a relationship of a vertex with itself. It’s typically drawn as a line or a curve.

- **Undirected edge:** This refers to an edge without a direction that indicates a two-way relationship between vertices.



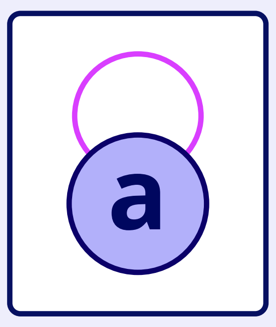
An undirected edge (a, b)

- **Directed edge:** This is an edge with a direction representing a one-way relationship.

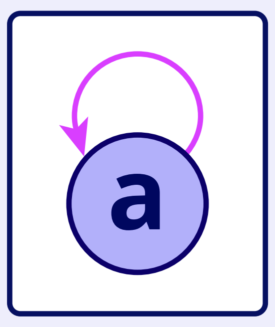


A directed edge (a, b)

- **Loop:** This is an edge (directed or undirected) that represents a relation of a vertex with itself.

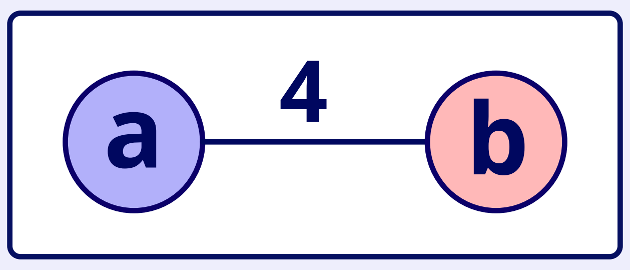


An undirected loop (a, a)

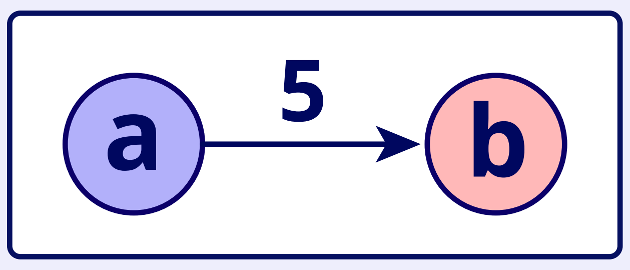


A directed loop (a, a)

Weighted edge: This is an edge (directed or undirected) with an associated numeric label called the weight of the edge.

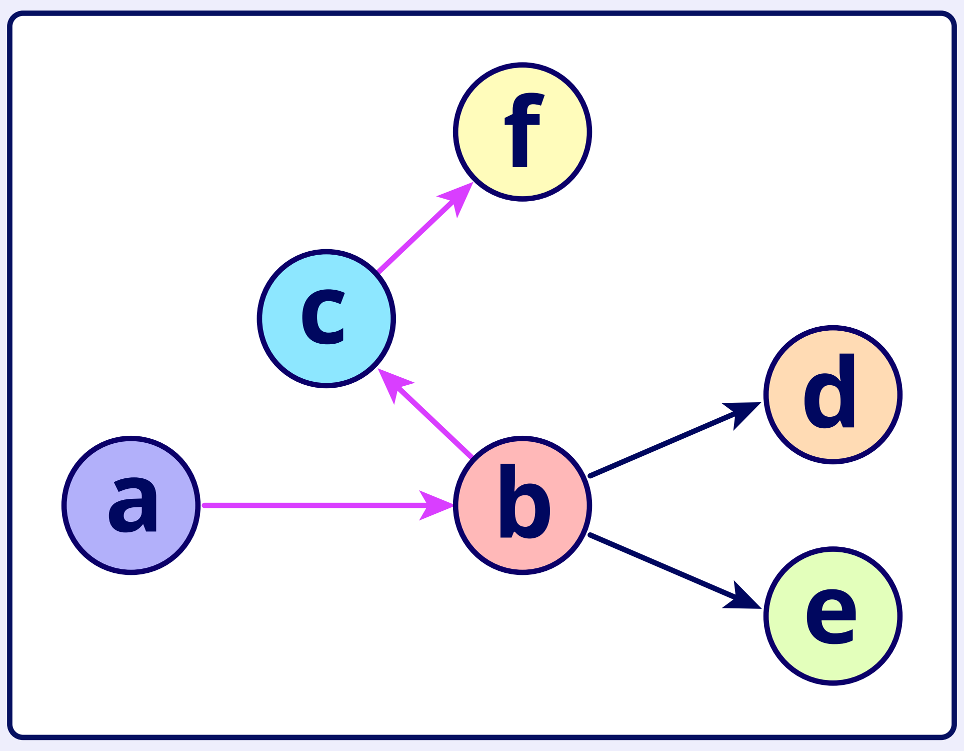


Undirected weighted edges

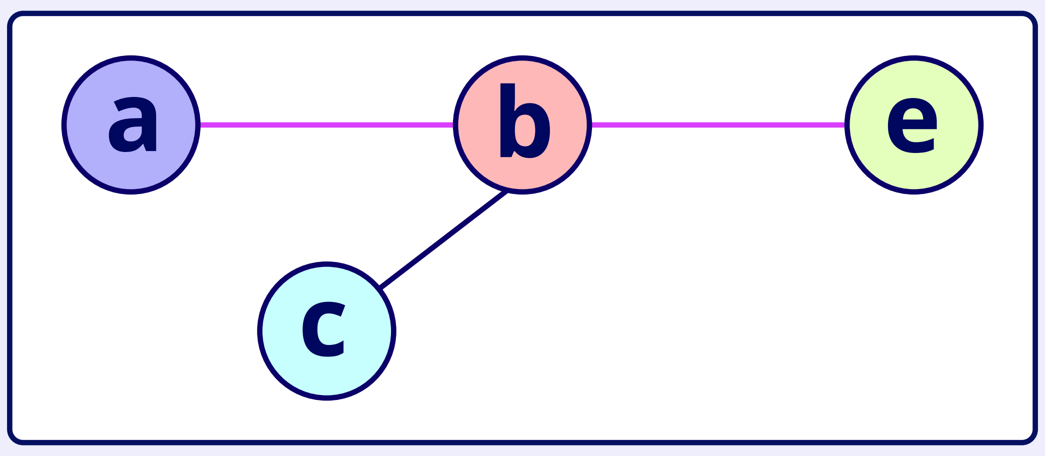


Directed weighted edges

Paths: A (directed or undirected) path is a sequence of distinct vertices where each successive pair is joined by an edge. A directed path has directed edges that point in the same direction along the path.

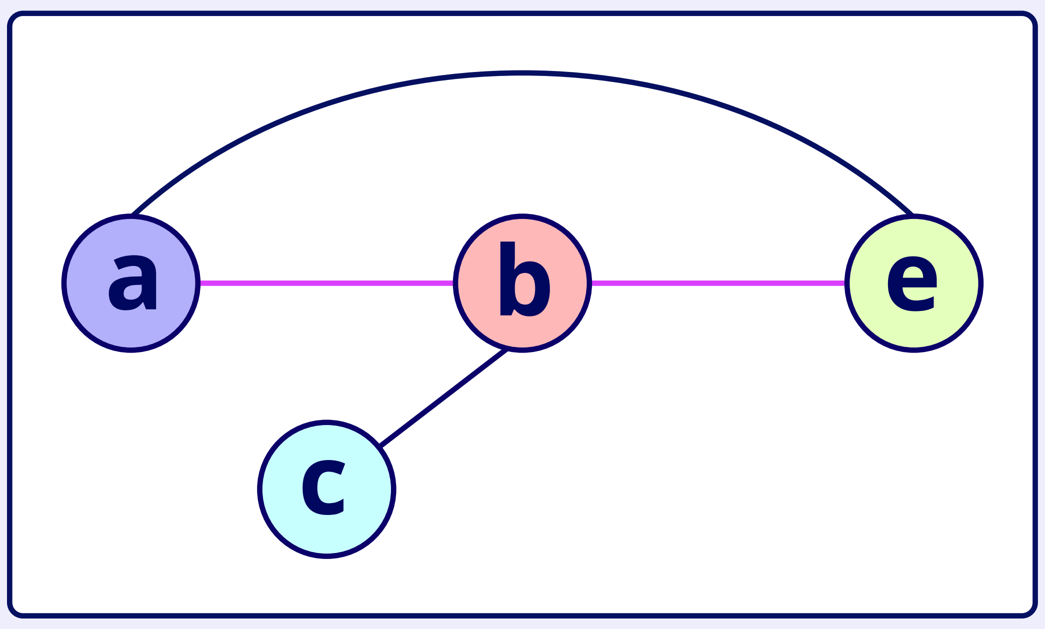


A directed path from vertex a to f

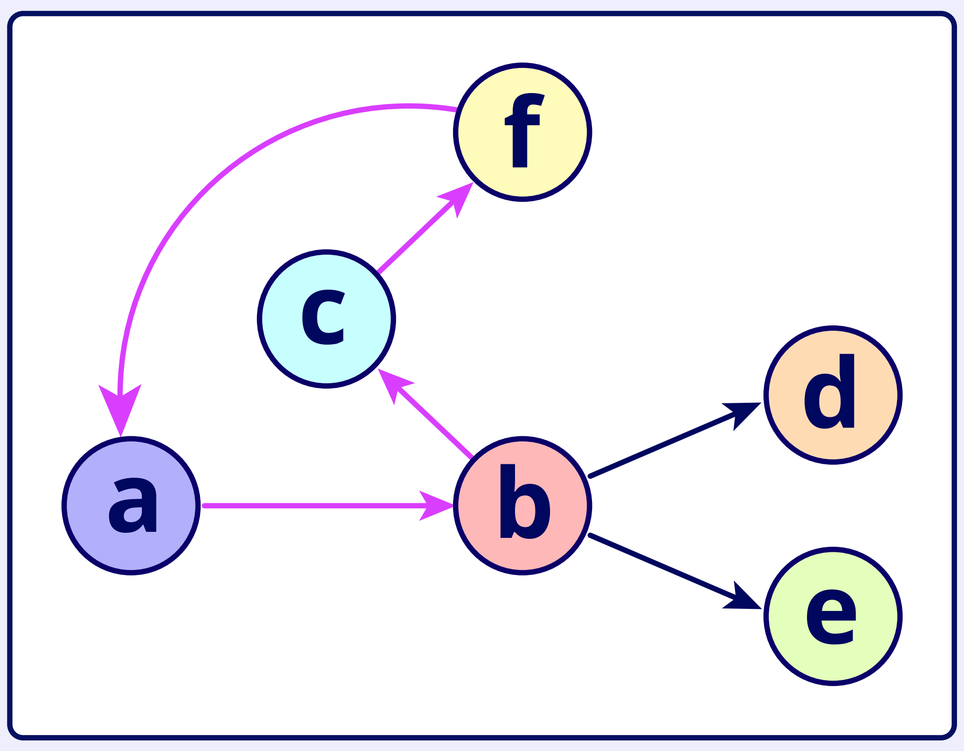


An undirected path from vertex a to e

Cycle: This is a path with an additional edge added between the vertices at both ends of the path. If a cycle is directed, the edge directions must point in the same direction around the cycle.



A cycle with undirected edges

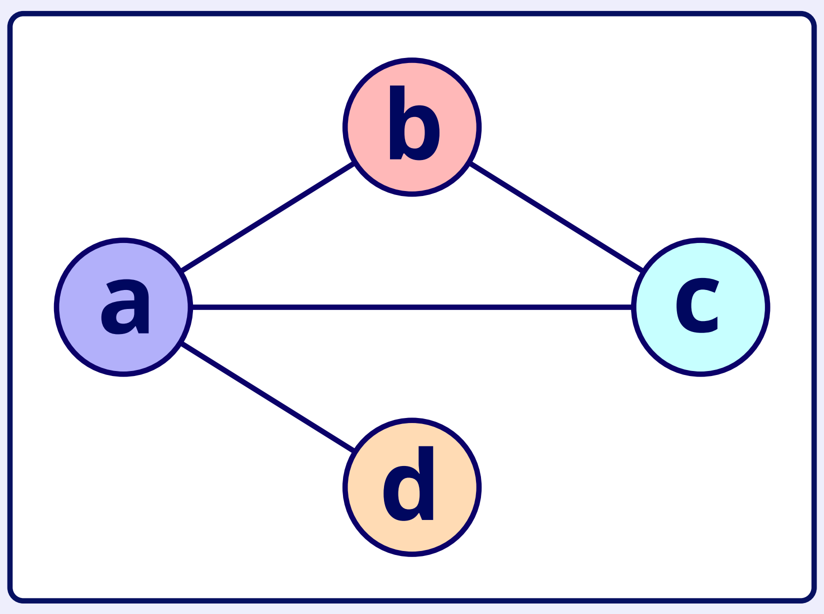


A cycle with directed edges

Graphs

Undirected graphs

A graph in which all edges are undirected.



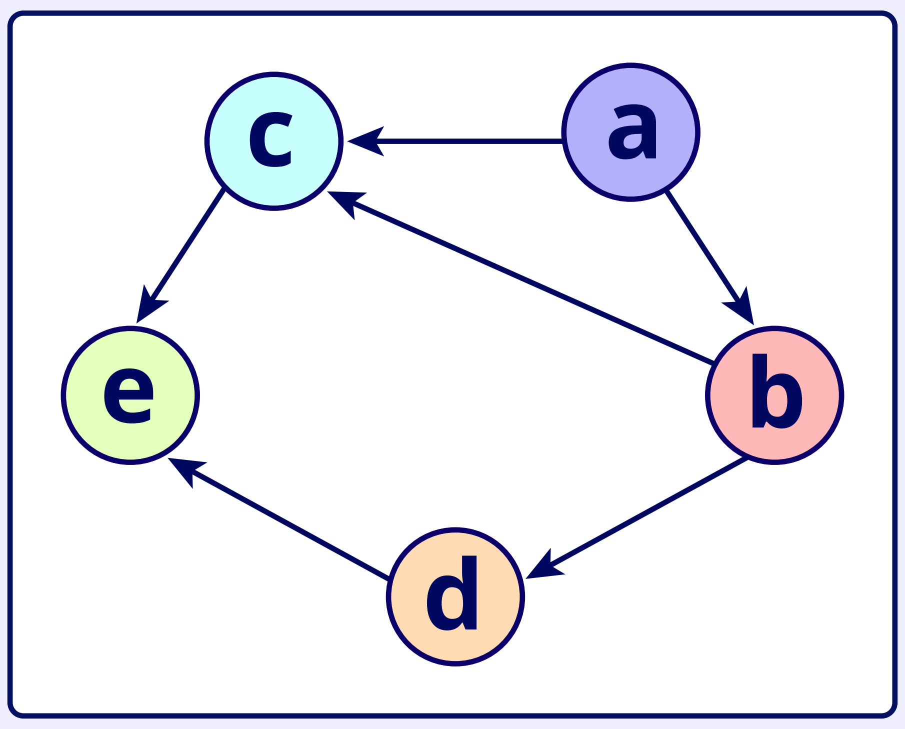
An undirected graph

Applications:

- Modeling two-way friendship relationships in social networks.
- Representing bidirectional physical networks like roads.
- Image segmentation and analysis.
- Solving mazes.

Directed graphs

A graph in which all edges are directed.



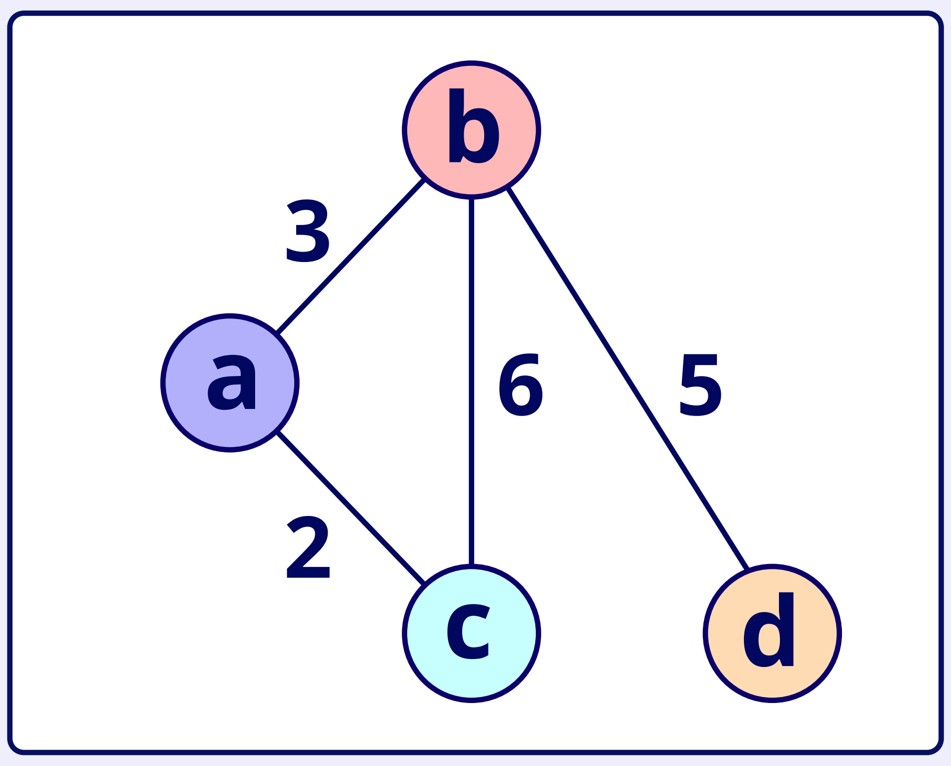
A directed graph

Applications:

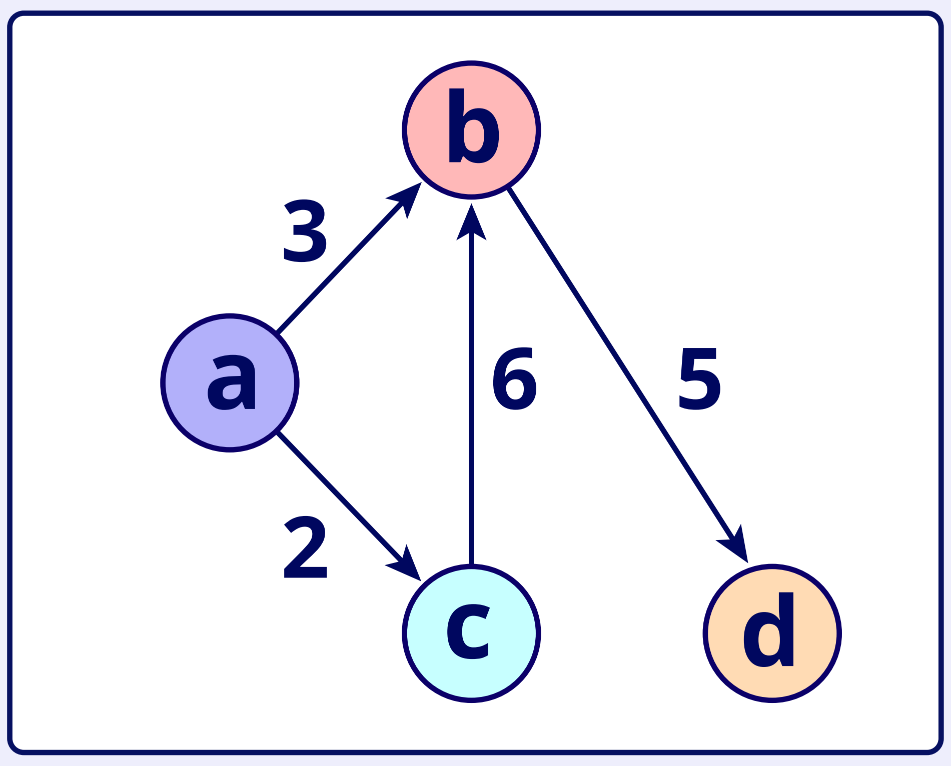
- Useful for representing asymmetric relationships, such as in social networks.
- Modeling web pages and links (web graph).
- Developing project schedules.

Weighted graph

An undirected or directed graph in which all edges have numeric weights.



An undirected weighted graph



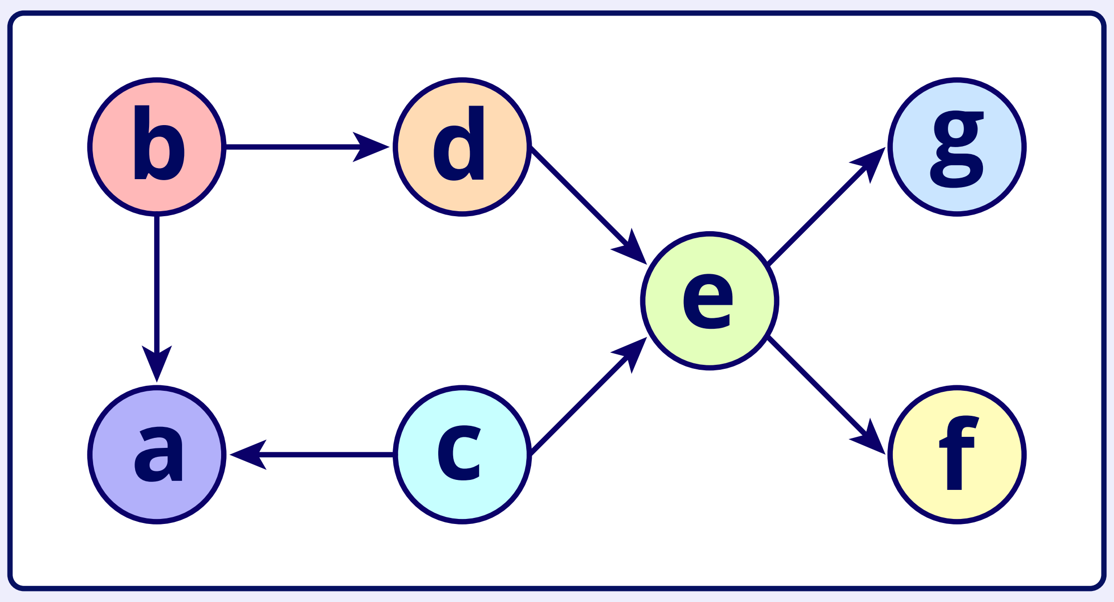
A directed weighted graph

Applications:

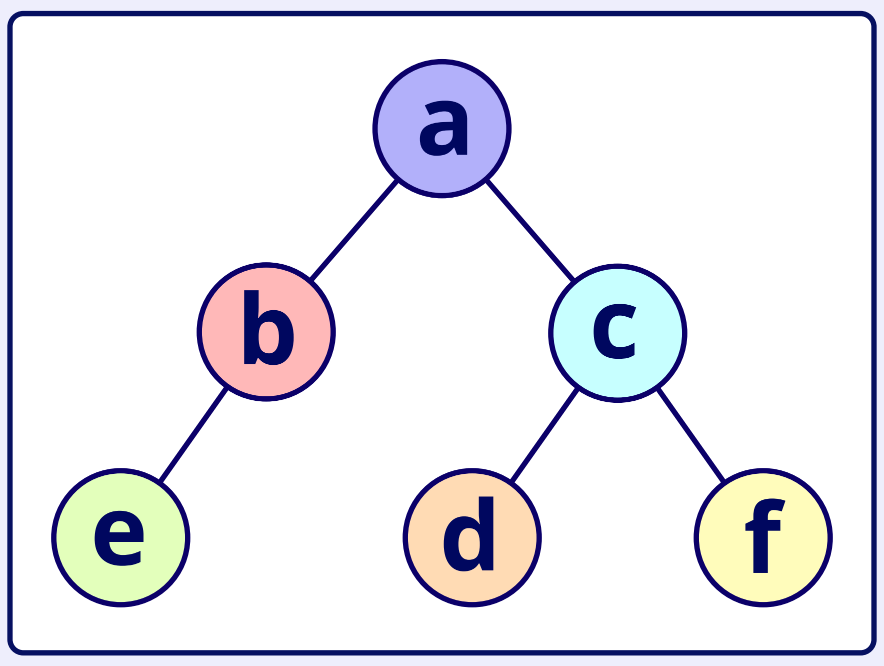
- Can model real-world scenarios where relationships have quantitative values.
- Useful for optimization problems.
- Network routing (finding the shortest path in a network).
- Mapping and geographical information systems (finding shortest routes).
- Scheduling (finding the optimal way to allocate resources).
- Logistics and transportation (finding the least costly paths).

Acyclic graph

An acyclic graph is a graph that does not have any cycle.



An acyclic, directed graph



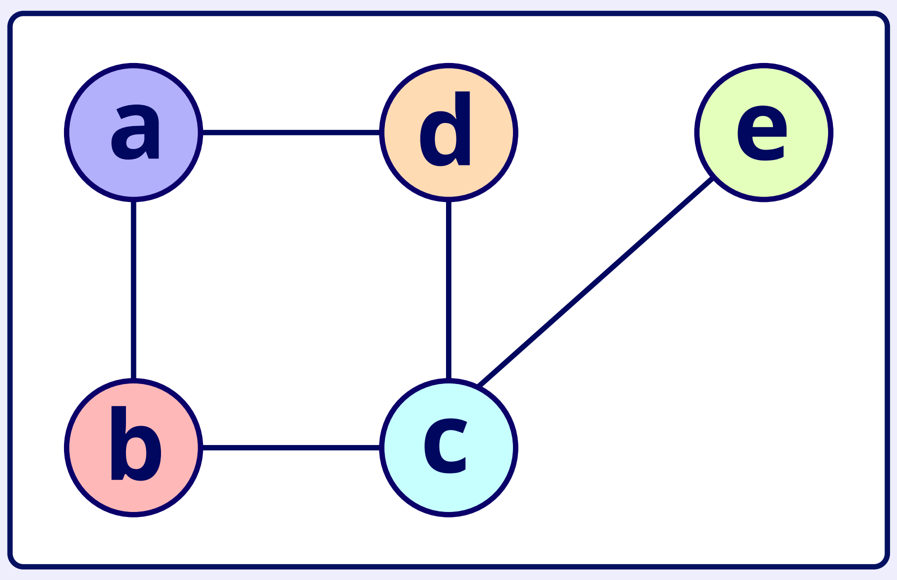
An acyclic, undirected graph

Applications:

- A good fit for problems where cycles are not allowed.
- Task scheduling and project management (dependency graphs).

Connected graph

A connected graph is an undirected graph where there is a path between any two vertices.



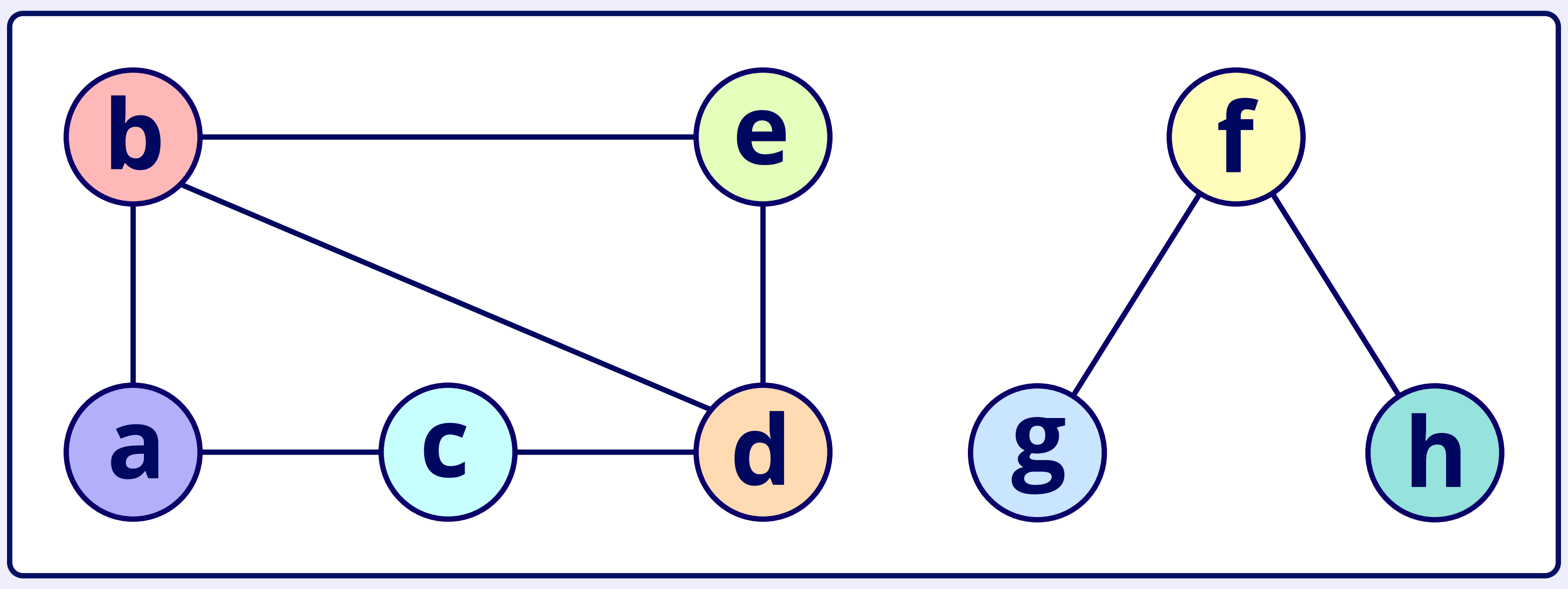
A connected graph

Applications:

- Used to model networks and geographical systems.

Disconnected graph

A disconnected graph is one where, starting from a vertex, it is not possible to reach every other vertex.



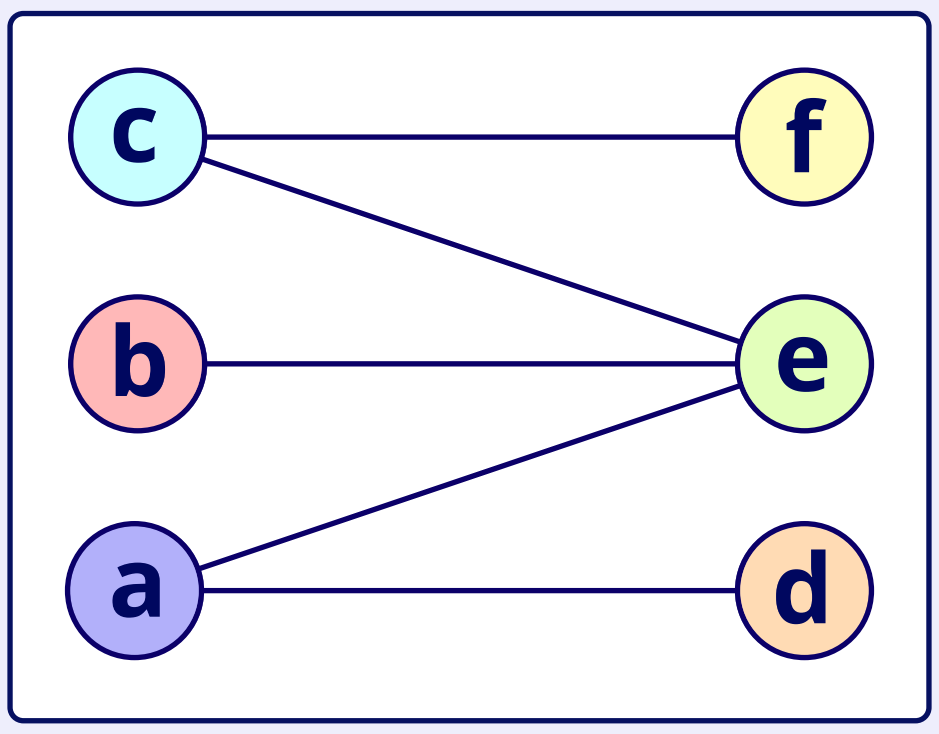
A disconnected graph

Applications:

- Network analysis where some vertices or clusters might not be directly connected to others.
- Modeling systems with distinct, isolated components.
- Graph partitioning in parallel computing and distributed systems.

Bipartite graph

An undirected graph is bipartite if we can partition all of its vertices into two groups, say L and R, such that every edge has one end in L and one end in R.



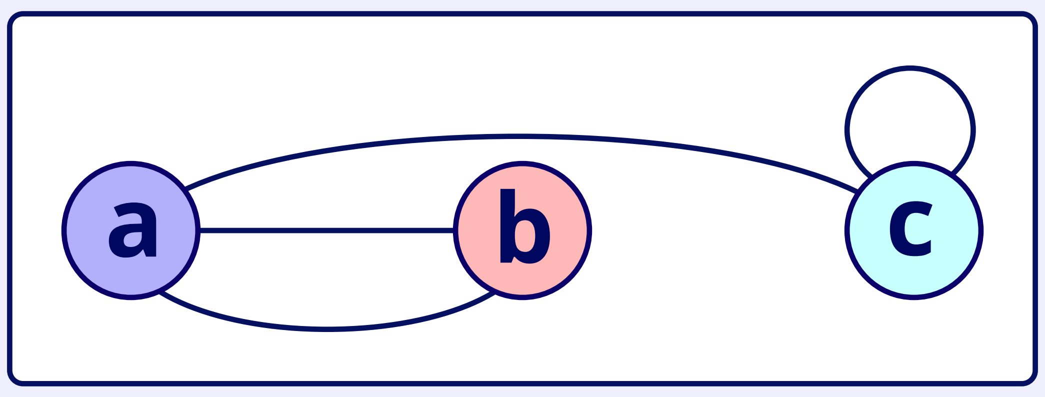
An undirected partite graph

Applications:

- Modeling problems with two distinct groups where no two members of the same group share a direct relationship.
- Task scheduling, where tasks need to be divided between two resources without conflicts.
- Circuit design and analysis where components can be categorized into input and output sets.

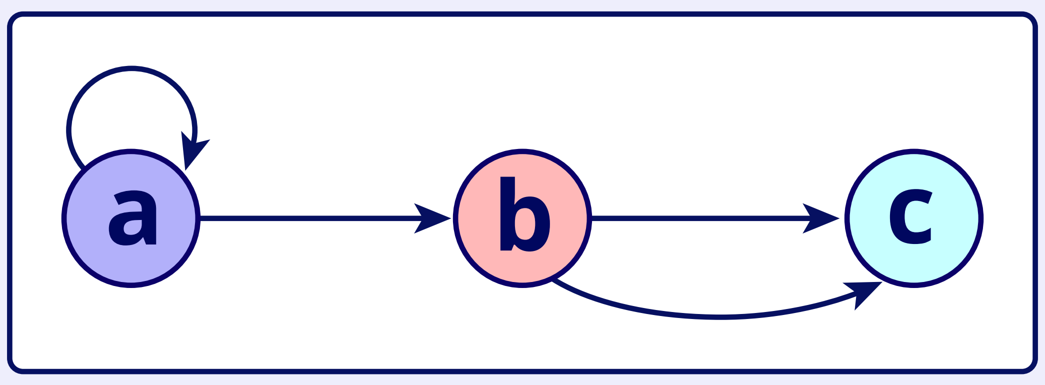
Non-simple graph

An undirected graph is non-simple if it contains a loop or multiple edges between the same pair of vertices.



A non-simple undirected graph

A directed graph is non-simple if it contains a loop or multiple edges between the same pair of vertices that point in the same direction.

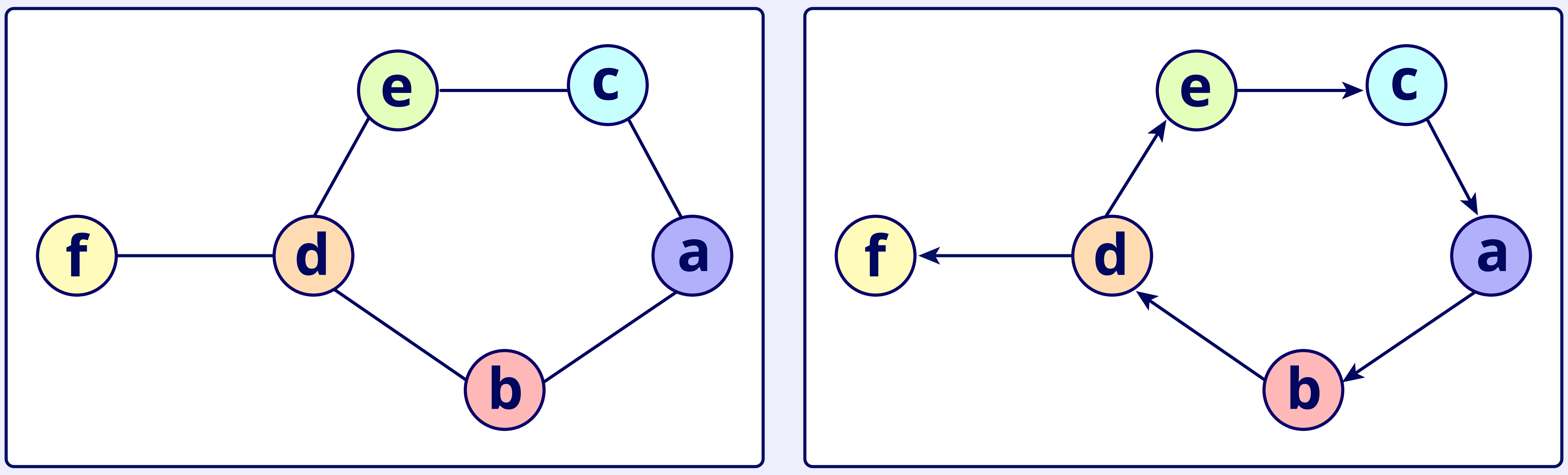


A non-simple directed graph

Applications:

- Fights network where multiple flights can take place between a pair of cities.
- Academic networks where multiple collaborations can exist between a pair of authors.
- Biological networks where a loop can be considered a feedback mechanism.

Data Structure for Implementation



Adjacency matrix

An adjacency matrix for a graph is a 2D array representing the graph.

For an undirected (simple) graph, a cell of the adjacency matrix contains a 1 if there's an edge between the corresponding vertices or a 0 if there isn't.

	a	b	c	d	e	d
a	0	1	1	0	0	0
b	1	0	0	1	0	0
c	1	0	0	0	1	0
d	0	1	0	0	1	1
e	0	0	1	1	0	0
d	0	0	0	1	0	0

An adjacency matrix for the given undirected graph

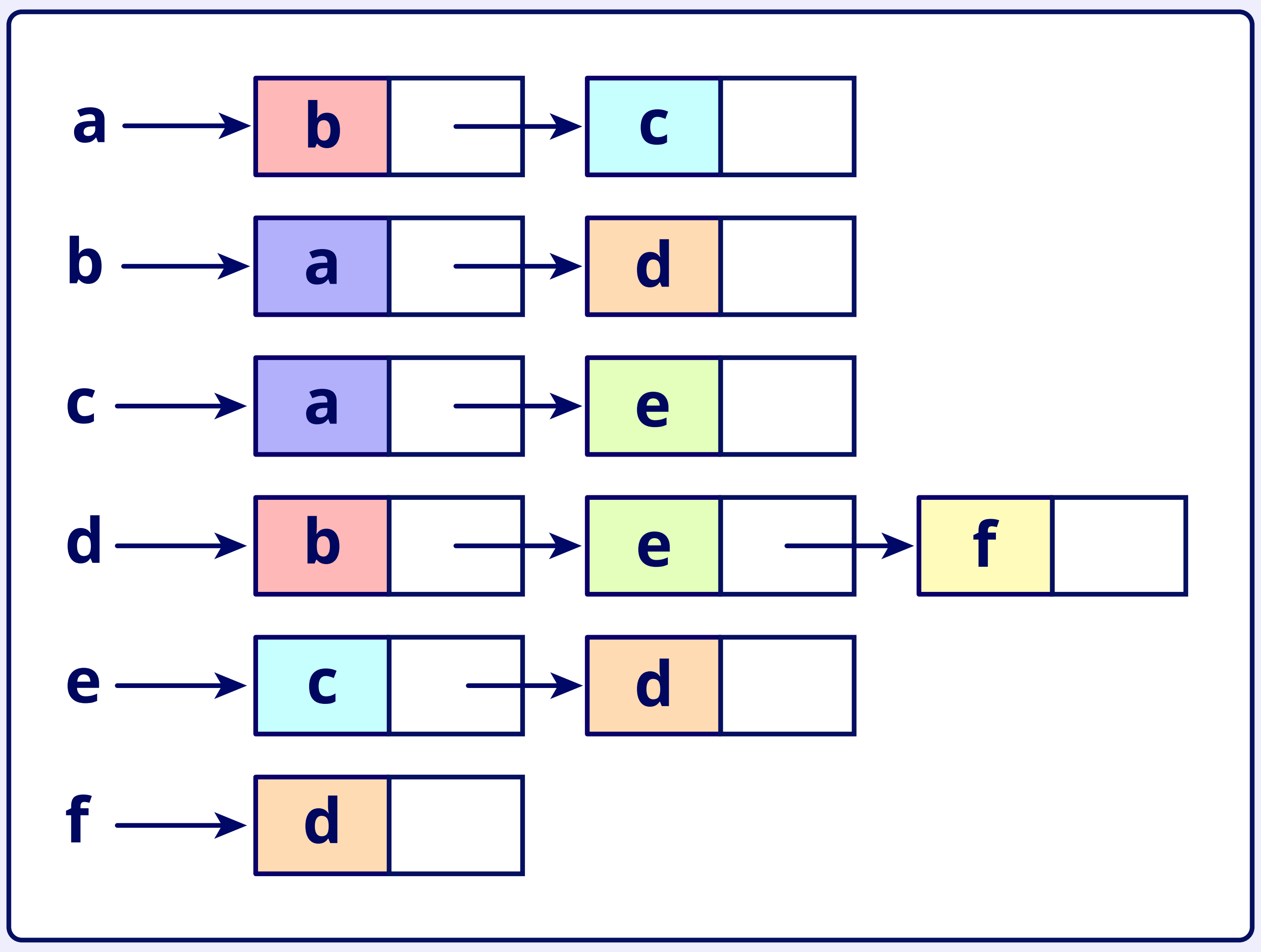
	a	b	c	d	e	d
a	0	1	0	0	0	0
b	0	0	0	1	0	0
c	1	0	0	0	0	0
d	0	0	0	0	1	1
e	0	0	1	0	0	0
d	0	0	0	0	0	0

An adjacency matrix for the given directed graph

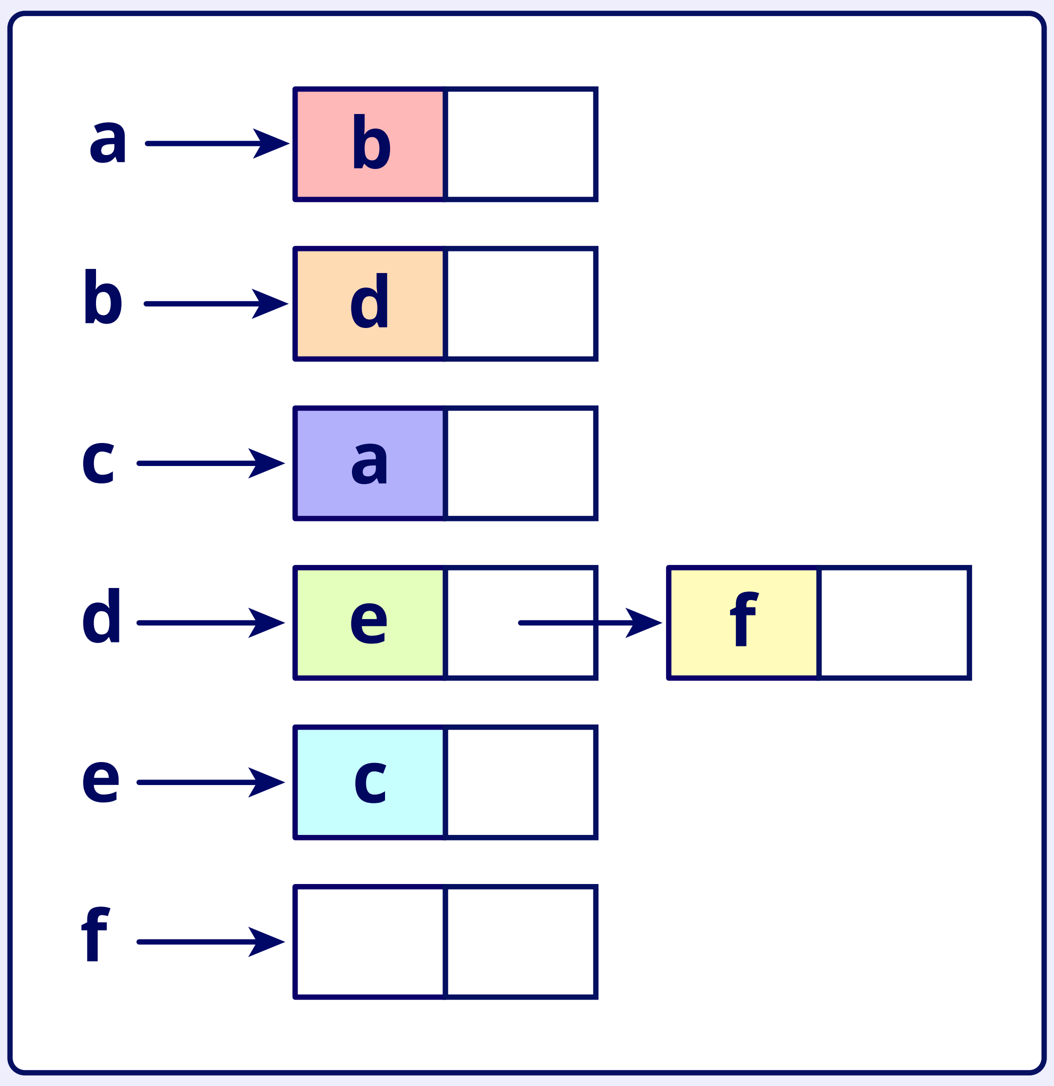
For the adjacency matrix of a directed (simple) graph, a cell in the row u and column v contains a 1 if there's an edge from u to v, or a 0 if there isn't.

Adjacency list

An adjacency list represents a graph as a list of lists. Each vertex has a list that contains all the vertices to which it is connected through an edge.



An adjacency list for the given undirected graph



An adjacency list for the given directed graph

Comparison

	Adjacency Matrix	Adjacency List
Adjacency Matrix	A 2D array	An array of lists
Space Complexity	O(V^2)	O(V + E)
Edge Lookup	O(1), directly accessible	O(E), requires traversing a list
Edge Additional/Remove	O(1)	O(1) for adding an edge, O(E) for removing an edge

Algorithms

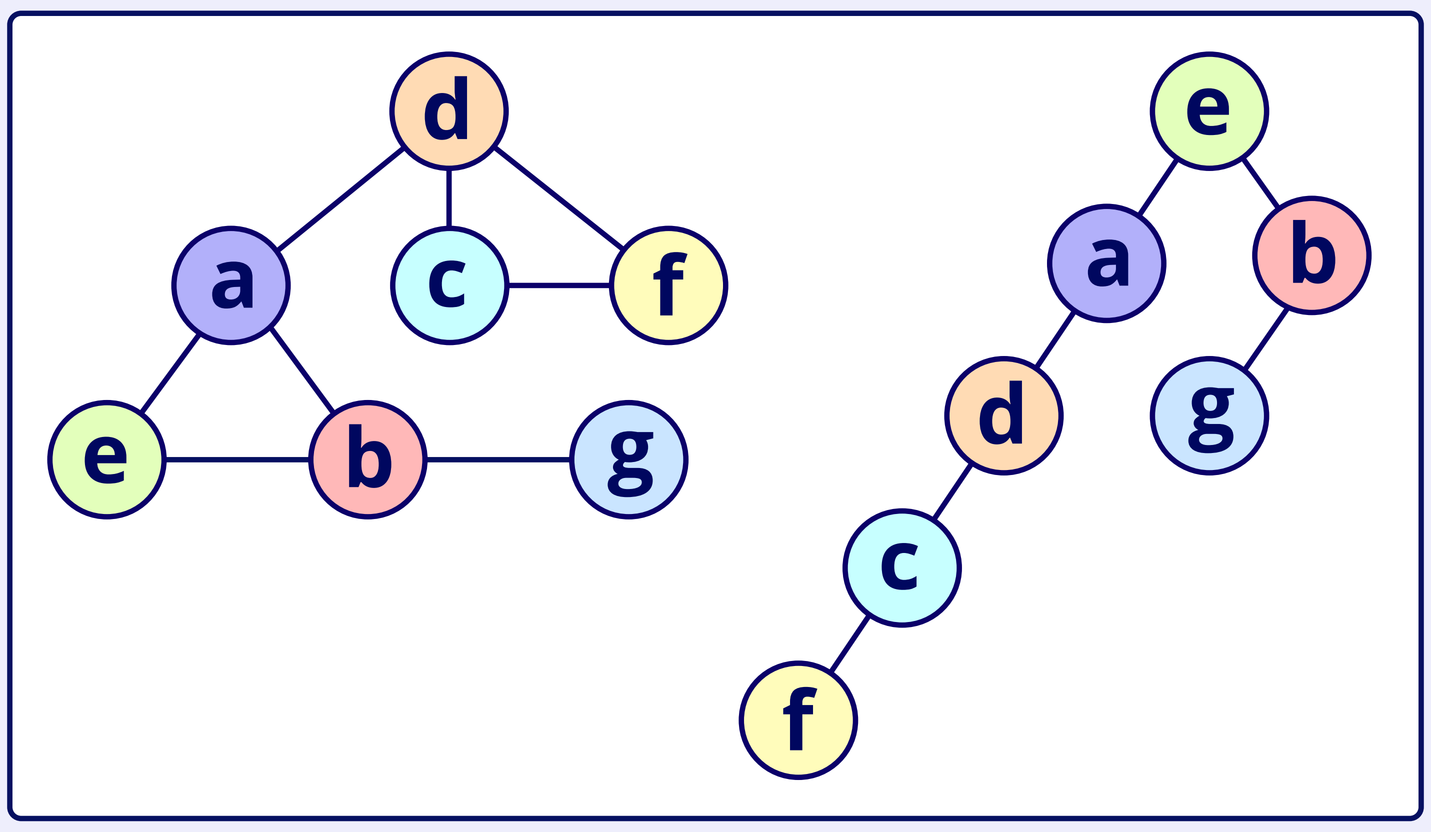
Graphs traversal algorithms

Traversal algorithms like BFS and DFS are used for exploring graphs by beginning at a vertex and traversing edges to get to other vertices.

- An undirected edge can be traversed in any direction.
- A directed edge can only be explored in the direction it specifies.

Breadth-first search

Breadth-first search (BFS) is a graph exploration algorithm that starts from a vertex and visits all its adjacent vertices before moving on to visit their adjacent vertices, and so on till all vertices are explored. The vertices and edges traversed form a tree called a **BFS tree.**



A graph and its corresponding BFS tree

Steps of BFS

1. **Initialization:** Choose a vertex, enqueue it into a queue.
2. **Explore:** While the queue is not empty, dequeue a vertex and mark it visited. Enqueue the vertices adjacent to it.
3. **Repeat:** Continue till the queue is empty.
4. If the search ends and there are still unvisited vertices, restart search from any unvisited vertex till all vertices are explored.

Time Complexity: O(V+E)

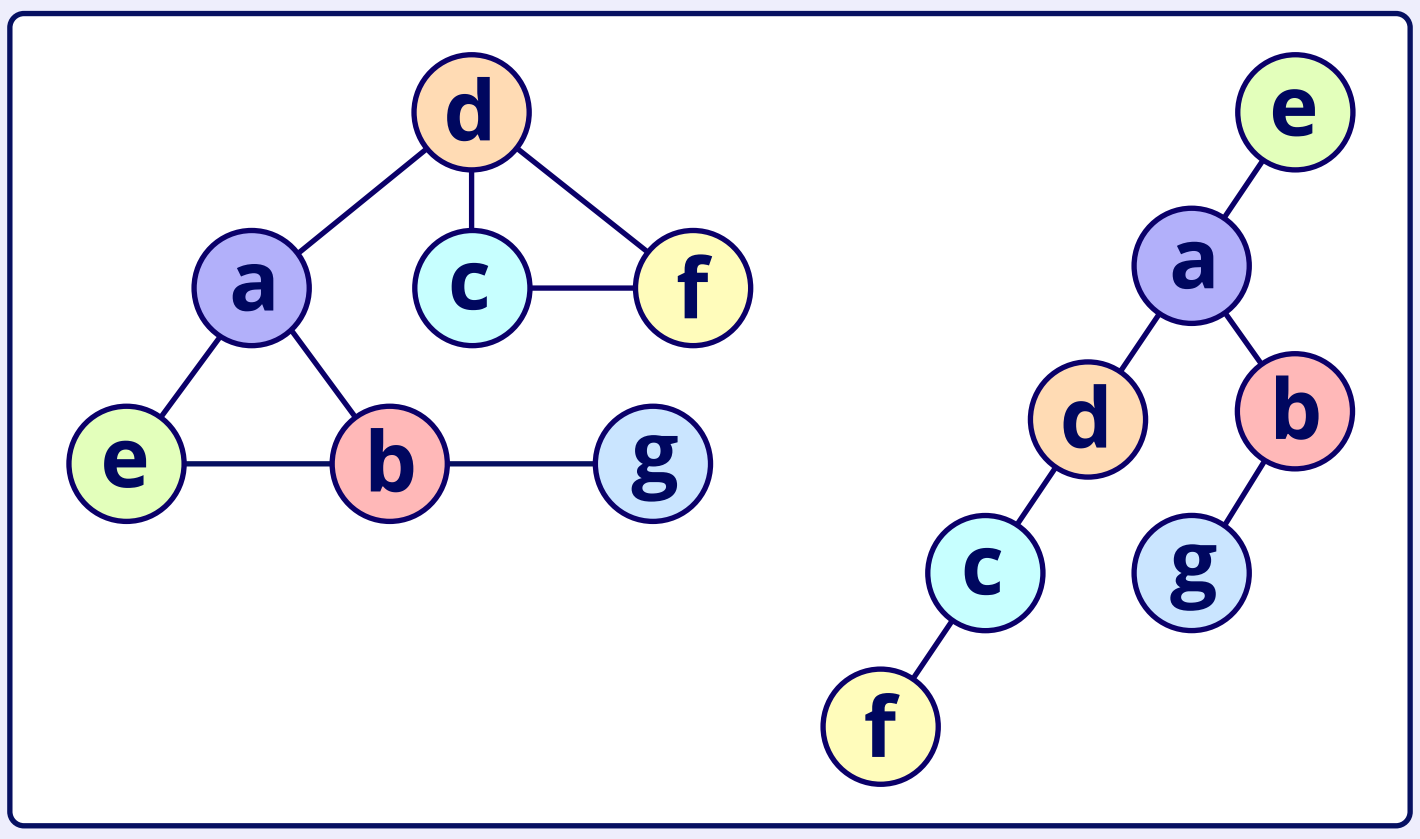
Applications:

- Finding the shortest path in unweighted graphs.
- Checking if a graph is bipartite.
- Finding connected components in a graph.
- Web crawling or social network analysis to explore connections.

Depth-first search

Breadth-first search (BFS) is a graph traversal algorithm that explores as deeply as possible along each branch before backtracking. It starts from a designated vertex and continues along a path until it can go no further, then it backtracks to explore other paths.

The vertices and edges traversed form a tree called a **DFS tree.**



A graph and its corresponding DFS tree

Steps of DFS

1. **Initialization:** Choose a vertex and mark it as visited. Start the DFS traversal from that vertex.
2. **Explore:** Visit a vertex that's adjacent to the current vertex and continue exploring from there.
3. **Backtrack:** If a vertex has no unvisited neighbors, backtrack to the predecessor of the current vertex and continue exploring.
4. If the search ends and there are still unvisited vertices, restart the search from any unvisited vertex till all vertices are explored.

Time Complexity: O(V+E)

Applications:

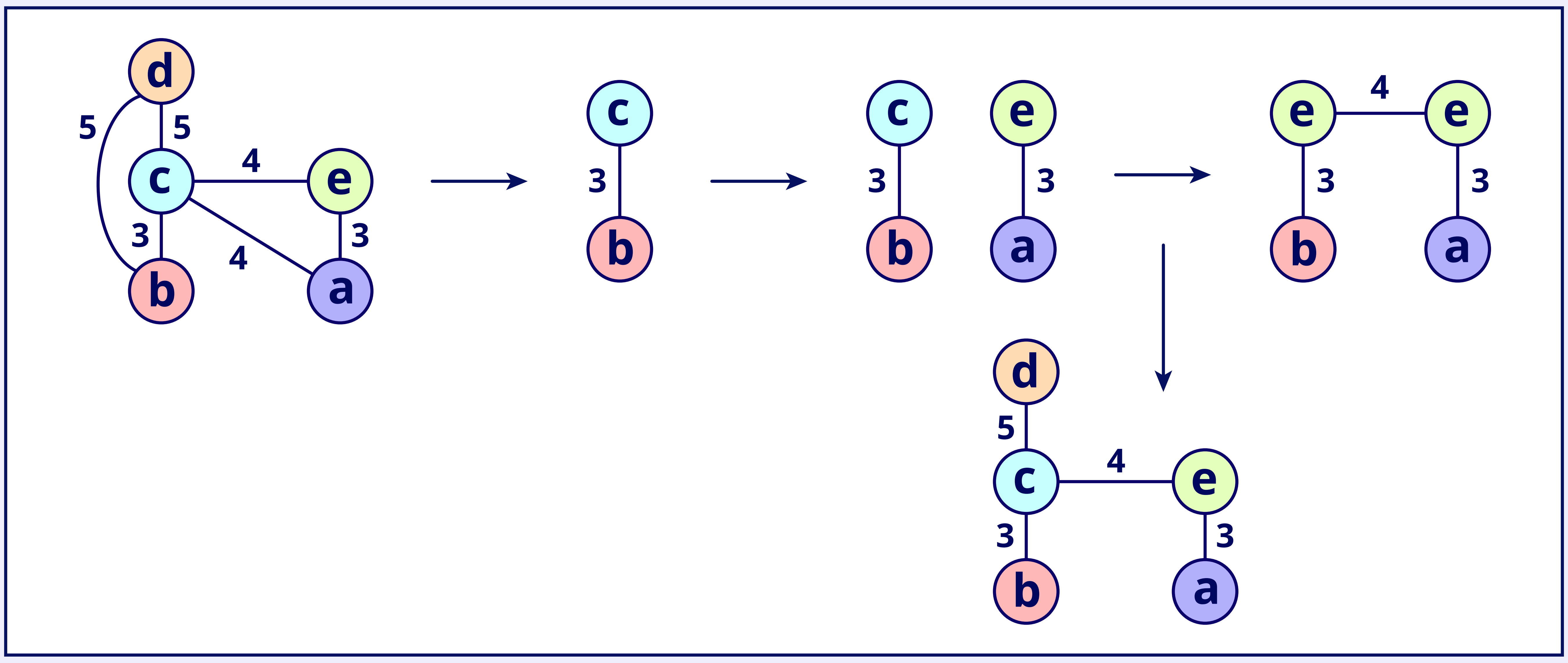
- Topological sorting in directed acyclic graphs (DAGs).
- Finding connected components in a graph.
- Detecting cycles in a graph.
- Solving puzzles and games with state-space search.

Minimum spanning tree

The minimum spanning tree (MST) of a connected, weighted, undirected graph is the subset of its edges with the minimum possible total edge weight such that the edges connect all vertices of the graph without forming cycles.

Kruskal's algorithm

Kruskal's algorithm is used to find the MST of a connected, weighted, undirected graph that starts with a collection of vertices and adds edges in a greedy manner to build an MST.



Kruskal's Algorithm

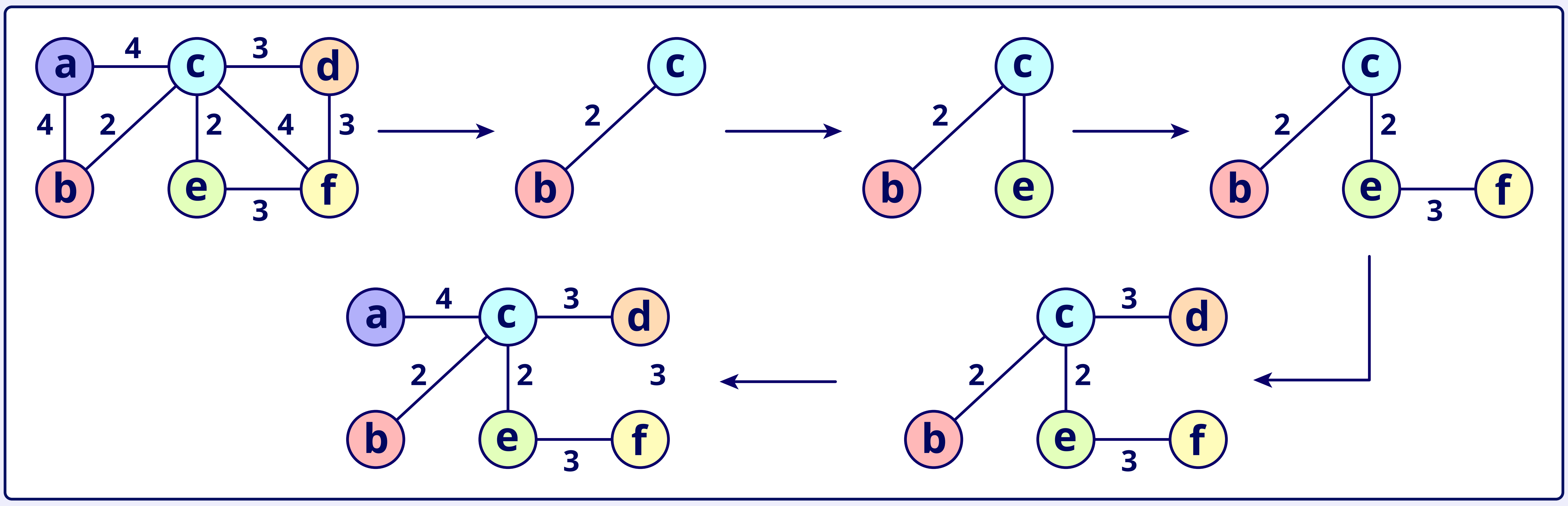
Steps of Kruskal's algorithm

1. **Initialization:** Include all vertices in the MST being built.
2. Sort all edges of the graph by their weights in a non-decreasing order.
3. **Contruct MST:** Iterate through the sorted edges. For each edge, add it to the MST if adding it does not form a cycle.
4. **Repeat:** Continue adding edges to the MST until there are V-1 edges (where V is the number of vertices).

Implementation details: Union-Find data structure is used to keep track of connected components formed at each step for efficient cycle detection.

Prim’s algorithm

Prim’s algorithm is a greedy approach used to find the MST of a weighted, undirected graph. It works by iteratively adding edges of minimum weight to a set so that the edges in the set form a tree.



Prim’s algorithm

Steps of Prim’s Algorithm

1. **Initialization:** Start with an arbitrary vertex and add it to a set that represents the MST being built.
2. **Contruct MST:**
   a. Iteratively, in each round, select the edge with the smallest weight that connects a vertex in the MST-set to a vertex outside it.
   b. Add the edge and the (latter) vertex to the MST-set.
3. **Repeat:** Continue adding edges (and associated vertices) until all vertices are a part of the MST-set.
4. **Output:** The edges included in the MST-set represent the minimum spanning tree of the graph.

Implementation details: A priority queue is used to contain vertices that have not yet become part of the MST set. Each vertex stores the minimum weight of an edge (over all edges) that joins it to a vertex in the MST set.

Prim’s vs Kruskal’s

	Prim’s Algorithm	Kruska’s Algorithm
Approach	Expands from one vertex, adding the smallest weight edges joined to the selected set	Selects edges of the smallest weight globally
Implementation	Priority queue	Sorting + disjoint set
Time complexity	O(E + V log V)	O(E log V)
Space complexity	O(V)	O(E+V)
Use	Better for dense graphs with number of edges close to V2	Comparable for sparse graphs with number of edges close to V

Applications of both algorithms:

- Designing network topologies, such as computer networks and telecommunications.
- Constructing efficient road and rail networks.
- Cluster analysis in data mining.
- Image segmentation—dividing images into regions based on similarities.
- In data clustering, MSTs are used to identify clusters by removing the most expensive edges
- in the tree.

Source-destinations algorithms

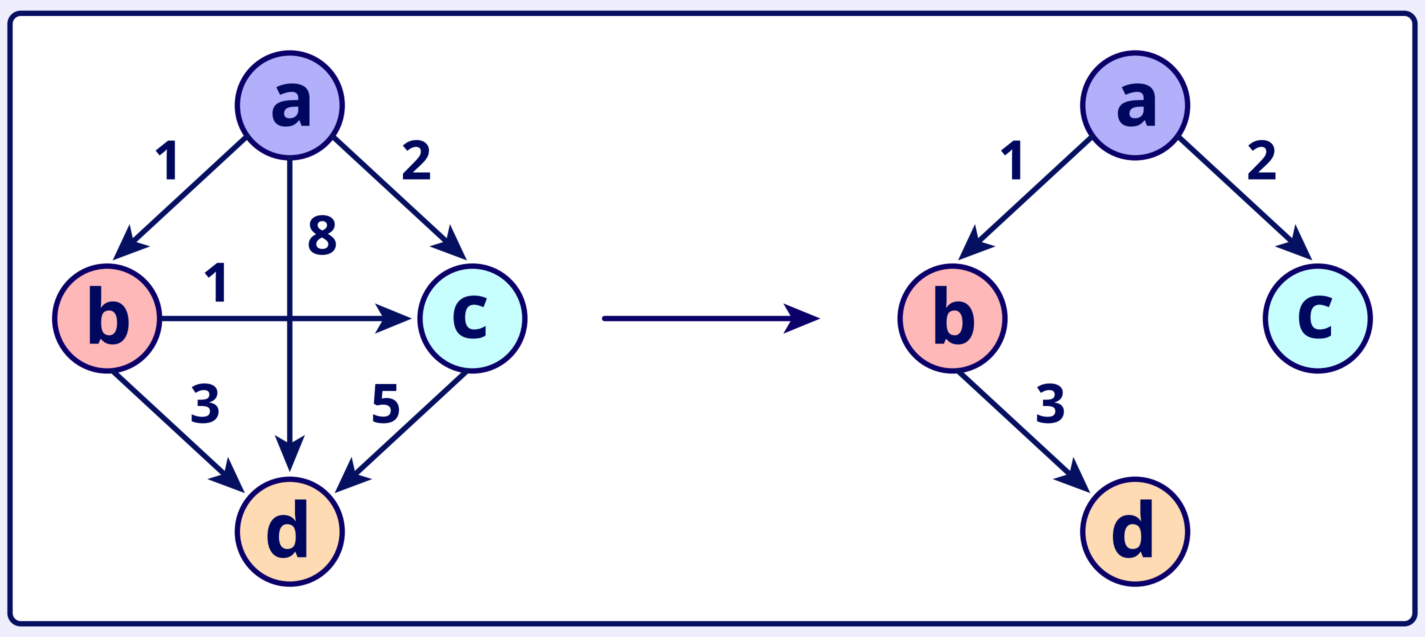
In weighted directed graphs, the shortest path from a vertex u to a vertex v is the path from u to v with the smallest sum of edge weights.

Common algorithms to find the shortest path from a vertex (called the source vertex) to all other vertices are:

- Dijkstra’s algorithm
- Bellman-Ford algorithm

Dijkstra’s algorithm

Dijkstra’s algorithm is a greedy algorithm for finding the shortest paths from a source vertex to all other vertices in a weighted directed graph with non-negative edge weights. It works by ascertaining at each step the shortest path to one more vertex.



A graph and the shortest paths from the source vertex to all others

Steps of Dijkstra’s Algorithm

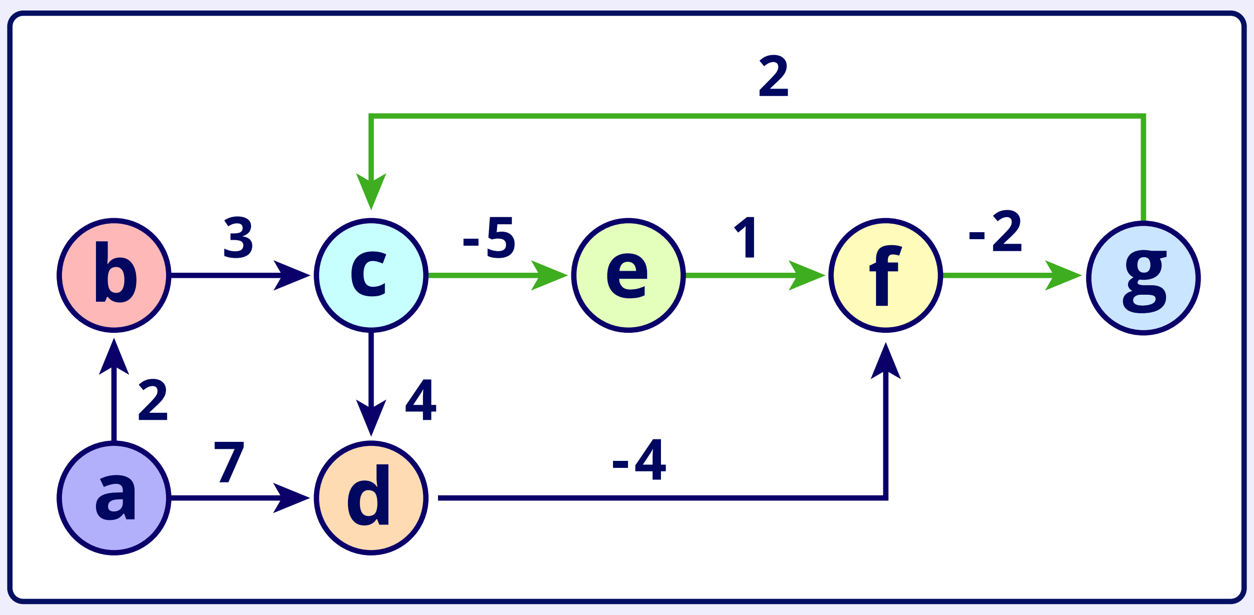
1. **Initialization:** Set the distance from the source vertex to itself as 0 and to all other distances as infinity. Insert all vertices into a priority queue.
2. **Extraction:** Remove the vertex with the smallest distance from the priority queue.
3. **Relaxation:** For each vertex v adjacent to the extracted vertex u, if the distance to v through u is shorter than the currently known distance to v, update v's distance.
4. **Repeat:** Continue extracting vertices from the priority queue and performing relaxation until the priority queue is empty.
5. **Output:** The shortest path distances from the source vertex to all other vertices are obtained.

Applications of both algorithms:

- Finding the shortest path in road networks and GPS navigation systems.
- Network routing protocols to determine optimal paths for data packets.
- Optimization problems where minimizing the sum of weights (distances) is essential.

Bellman-Ford Algorithm

Definition: The Bellman-Ford Algorithm is a shortest-path algorithm that finds the shortest paths from a single source vertex to all other vertices in a weighted graph, even in the presence of negative weight edges.



Negative weight cycle of weight -4

Steps of Bellman-Ford Algorithm

1. **Initialization:** Set the distance to the source vertex to 0 and the distance to all other vertices to infinity.
2. **Relaxation:**
   - Iterate over all edges (u, v), and if the distance to v through u is shorter than the currently known distance to v, update v's distance.
   - Repeat this process V-1 times (where V is the number of vertices).
3. **Negative cycle detection:** After V-1 iterations, iterate over all edges again and relax them again to check if there's a shorter path to any vertex. If so, it indicates the presence of a
4. negative weight cycle.
   Output: The shortest path distances are obtained if there's no negative-weight cycle in the graph.

Applications:

- Finding shortest paths in graphs with negative weight edges.
- Detecting negative weight cycles in graphs.
- Routing algorithms in networks where edge weights might be negative.
- Currency arbitrage problems in financial markets.

Dijkstra's vs Bellman-Ford

	Dijkstra's	Bellman-Ford
Works with negative weights	No	Yes
Processes an edge	Only once	Multiple times
Time complexity	O(E log V)	O(VE)