Chapter 2: Graph

By: Sandeep Godbole

Imagine a person who wants to travel from one point in a city to another but doesn't know the route to the destination. He needs an application that provides clear directions on how to get there. To build such an application, we need a way to represent the city map as a data structure within the program. The developer now faces the question: which data structure can efficiently represent a city map and help determine optimal routes?

A graph data structure is well-suited for this purpose. By representing landmarks as nodes and roads connecting them as edges, a graph can model the city layout in memory. This structure allows the program to explore possible routes, find the shortest path, and provide helpful navigation guidance.

In this chapter, we will explore the essential aspects of graphs, including key terms, operations, and algorithms that make graphs such a powerful tool.

Introduction to Graph

A graph is a fundamental data structure in computer science and mathematics, made up of nodes (also called vertices) and edges. The edges connect pairs of nodes. Graphs are widely used to model various real-world structures, such as networks, social connections, paths on maps, and task dependencies. Typically, a graph is drawn as shown below.

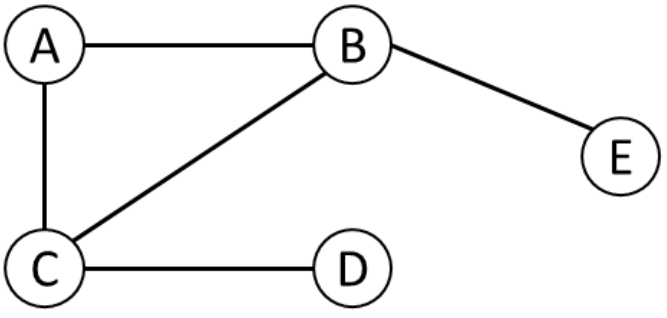


Fig. 1: Graph representation

Introduction to Graph Terms and Concepts

Basic Terms

1. **Graph**: A collection of vertices (nodes) and edges (connections between nodes).
   Refer to Fig. 2(a).
2. **Vertex (Node)**: A fundamental unit in a graph, representing an entity or a point in the graph.
   Refer to Fig. 2(a).
3. **Edge (Link)**: A connection between two vertices in a graph. Edges can be directed or undirected. Refer to Fig. 2(a) and Fig. 2(b).
4. **Directed Graph (Digraph)**: A graph in which edges have a direction, meaning each edge goes from one vertex to another specific vertex. Refer to Fig. 2(b).
5. **Undirected Graph**: A graph where edges have no direction, meaning they connect two vertices without any specific direction. Refer to Fig. 2(d).

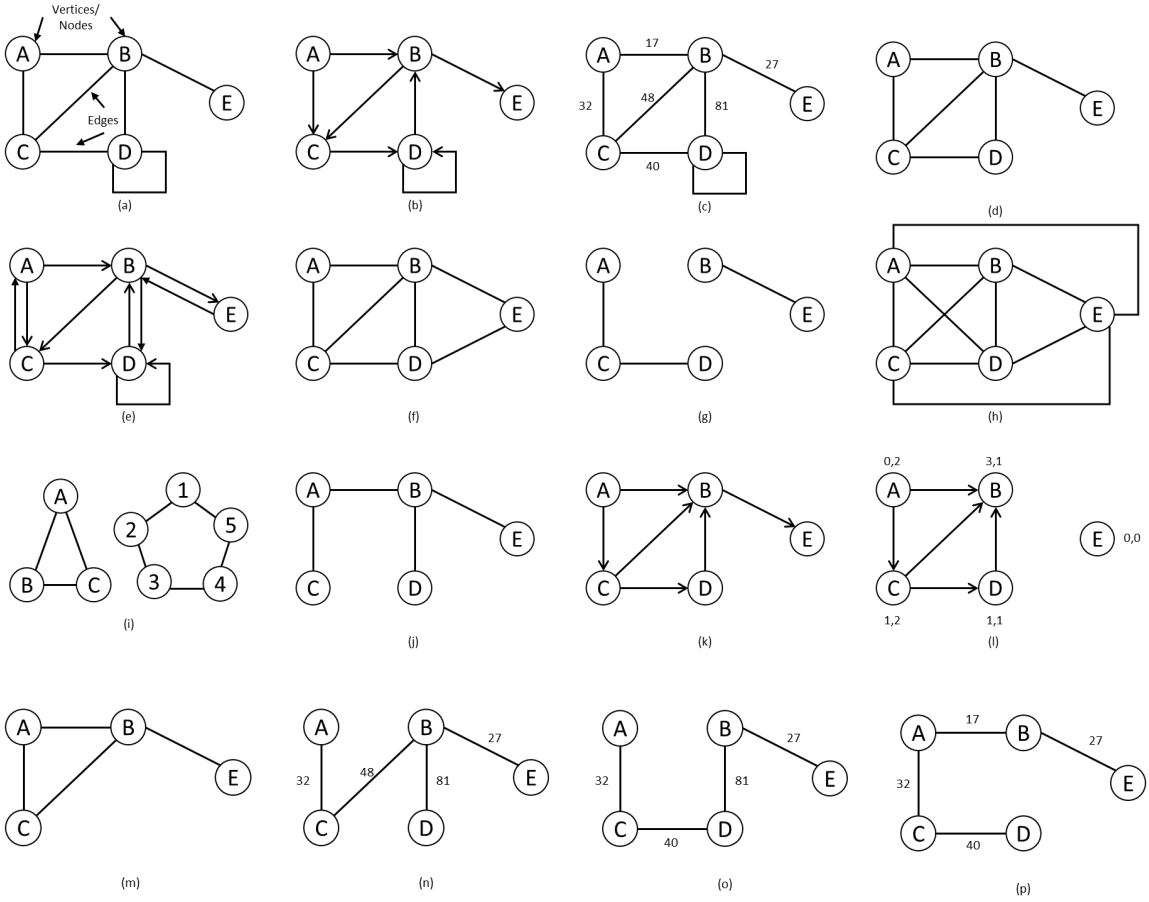


Fig. 2: Reference graphs for learning various terms in graph theory.

6. **Weighted Graph**: A graph in which each edge has an associated numerical weight (or cost).

Refer to Fig. 2(c).

7. **Unweighted Graph**: A graph where edges do not have any weights associated with them.
   Refer to Fig. 2(d).
8. **Simple Graph**: A graph with no loops (edges that connect a vertex to itself) and no multiple edges between vertices. Refer to Fig. 2(d).
9. **Multigraph**: A graph where multiple edges between the same set of vertices are allowed.
   Refer to Fig. 2(e).
10. **Dense Graph**: A graph in which there are many edges relative to the number of vertices.
    Refer to Fig. 2(f) and Fig. 2(h).
11. **Sparse Graph**: A graph in which there are few edges relative to the number of vertices.
    Refer to Fig. 2(g).

Types of Graphs

1. **Complete Graph**: A graph in which there is an edge between every pair of vertices.
   Refer to Fig. 2(h)
2. **Cyclic Graph**: A graph that contains at least one cycle. Refer to Fig. 2(b) and Fig. 2(d).
3. **Acyclic Graph**: A graph that has no cycles. Refer to Fig. 2(j).
4. **Directed Acyclic Graph (DAG)**: A directed graph with no cycles, often used in scheduling and task dependencies. Refer to Fig. 2(k).

Vertex-Related Terms

1. **Degree of a Vertex**: The number of edges connected to a vertex. Refer to Fig. 2(l).
   - **In-Degree**: The number of incoming edges for a vertex in a directed graph.
   - **Out-Degree**: The number of outgoing edges for a vertex in a directed graph.

2. **Adjacent Vertices**: Vertices that are connected by an edge. Refer to Fig. 2(b) and Fig. 2(d).
   A-B, A-C, B-C, C-D, D-B, B-E are examples of adjacent vertices.
3. **Isolated Vertex**: A vertex with a degree of zero (no edges connected to it). Refer to Fig. 2(l).

Path and Cycle Terms

1. **Path**: A sequence of vertices where each adjacent pair is connected by an edge.
   Refer to Fig. 2(b). A-B-E is an example of a path.
2. **Simple Path**: A path with no repeated vertices. Refer to Fig. 2(b) A-C-D-B-E is an example of simple path.
3. **Cycle**: A path that starts and ends at the same vertex without repeating any edges or vertices, except the starting/ending vertex. Refer to Fig. 2(b). B-C-D-B is an example of a cycle.
4. **Eulerian Path**: A path that visits every edge of a graph exactly once.
   Refer to Fig. 2(d). E-B-A-C-D-B-C is an example of a Eulerian path.
5. **Eulerian Cycle**: A cycle that visits every edge of a graph exactly once and returns to the starting vertex. Refer to Fig. 2(i). A-B-C-A is an example of a Eulerian cycle.
6. **Hamiltonian Path**: A path that visits every vertex of a graph exactly once.
   Refer to Fig. 2(j).
   1-2-3-4-5 is an example of a Hamiltonian path.
7. **Hamiltonian Cycle**: A cycle that visits every vertex of a graph exactly once and returns to the starting vertex. Refer to Fig. 2(j). 1-2-3-4-5-1 is an example of a Hamiltonian cycle.

Connectivity Terms

1. **Connected Graph**: A graph in which there is a path between every pair of vertices.
   Refer to Fig. 2(h).
2. **Disconnected Graph**: A graph in which some pairs of vertices have no path between them.
   Refer to Fig. 2(b).

3. **Strongly Connected (for Directed Graphs)**: A directed graph is strongly connected if there is a path from each vertex to every other vertex. Refer to Fig. 2(e).
4. **Weakly Connected (for Directed Graphs)**: A directed graph is weakly connected if replacing all directed edges with undirected edges would make it connected. Refer to Fig. 2(b).
5. **Bridge (Cut-Edge)**: An edge that, if removed, would increase the number of disconnected components in the graph. Refer to Fig. 2(j). Removing edge A-B results in two disconnected graphs. So, edge A-B forms a bridge.
6. **Articulation Point (Cut Vertex)**: A vertex is said to be an articulation point in a graph if removal of the vertex and associated edges disconnects the graph. Refer to Fig. 2(d). Removing vertex B results in two disconnected graphs. So, vertex B forms an articulation point.

Special Types of Graph

1. **Subgraph**: A graph formed from a subset of the vertices and edges of another graph.
   Refer to Fig. 2(g) and Fig. 2(j). They are the subgraphs of the graph shown in Fig. 2(f).
2. **Induced Subgraph**: A subgraph formed by a subset of the vertices of a graph and all the edges between them in the original graph. Refer to Fig. 2(m). It is an induced subgraph of the graph shown in Fig. 2(f).
3. **Spanning Tree**: A tree that includes all the vertices of a graph and a subset of its edges that connect all vertices without forming any cycles. Refer to Fig. 2(n) and Fig. 2(o).
4. **Minimum Spanning Tree (MST)**: A spanning tree with the minimum possible total edge weight in a weighted graph. Refer to Fig. 2(p).

Graph Storage Structure

Adjacency Matrix

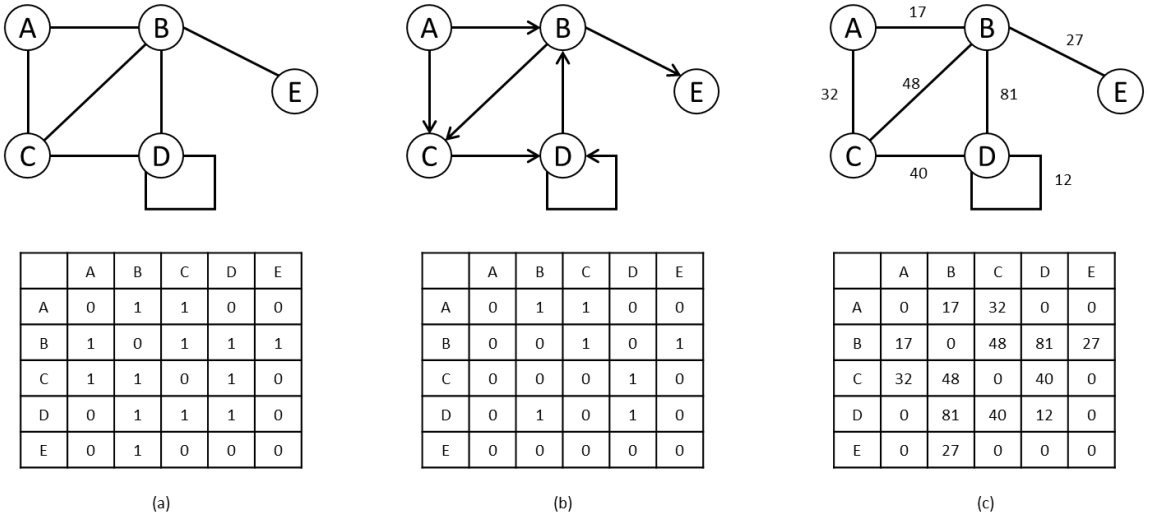
An adjacency matrix is a way to represent a graph using a 2D array. In this matrix, both rows and columns represent the vertices of the graph. The matrix shows whether or not an edge connects a pair of vertices in each cell.

- For an **unweighted graph**, a cell contains `1` if there is an edge between the vertices, or `0` if there is no edge.
- For a **weighted graph**, the cell contains the weight of the edge instead of `1`.

If the graph is **undirected**, the adjacency matrix is symmetric along the diagonal. In a **directed graph**, the matrix is not necessarily symmetric.

Advantages and Disadvantages

- **Advantage**: Fast access to check if an edge exists between two vertices; ideal for dense graphs.
- **Disadvantage**: Requires more space for sparse graphs, as it uses O(V^2) memory where V is the number of vertices.



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

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

	A	B	C	D	E
A	0	17	32	0	0
B	17	0	48	81	27
C	32	48	0	40	0
D	0	81	40	12	0
E	0	27	0	0	0

Fig. 3: Adjacency matrix representation for (a) undirected unweighted graph, (b) directed unweighted graph, and (c) undirected weighted graph.

Adjacency List

An adjacency list is a way to represent a graph using a list of lists. Each vertex in the graph has a list associated with it, which contains all the vertices it is directly connected to. This representation is especially efficient for sparse graphs, where the number of edges is much lower than the maximum possible.

- For an **unweighted graph**, each list entry simply contains the adjacent vertices.

- For a **weighted graph**, each entry can store a pair, where the first element is the adjacent vertex, and the second is the weight of the edge.

Advantages and Disadvantages

- **Advantage**: Efficient for space, especially for sparse graphs; uses O($V + E$) memory, where V is the number of vertices and E is the number of edges.
- **Disadvantage**: Slower to check if an edge exists between two specific vertices, as this requires searching through the list of adjacent vertices.

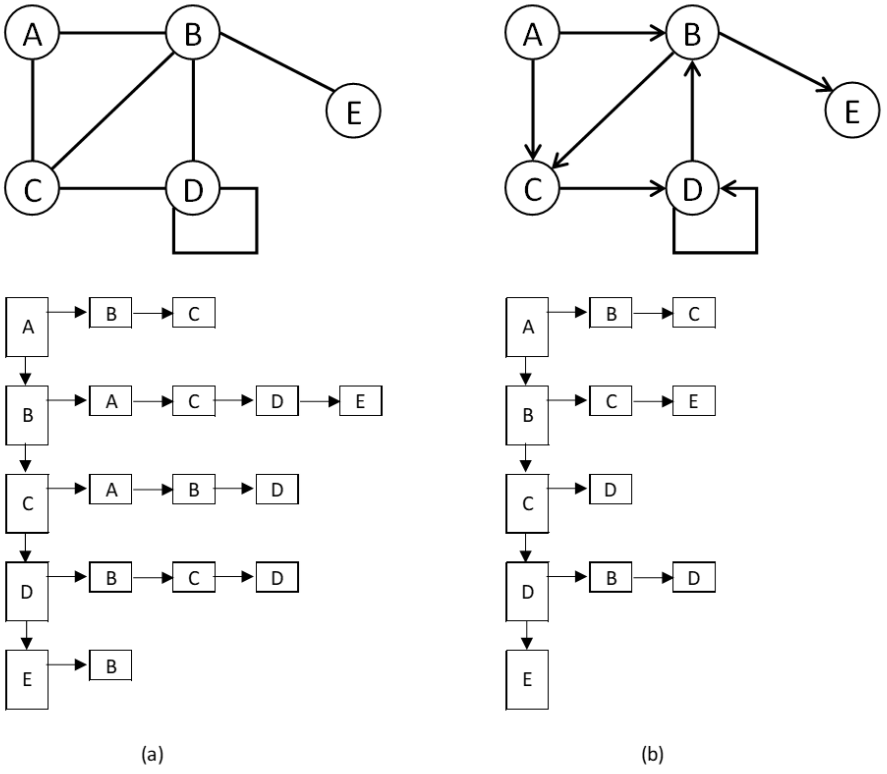


Fig. 4: Adjacency list representation for (a) undirected and (b) directed unweighted graphs

A hash map can be used to store vertices (the vertical linked list), and a vector can be used to store the edges (the horizontal linked lists).

Graph Operations

Add Vertex

Introduces a new disjoint vertex to the graph. Fig. 5(a) shows the graph before and after the Add Vertex operation.

Delete Vertex

Removes a vertex and all associated edges. Fig. 5(b) shows the graph before and after the Delete Vertex operation.

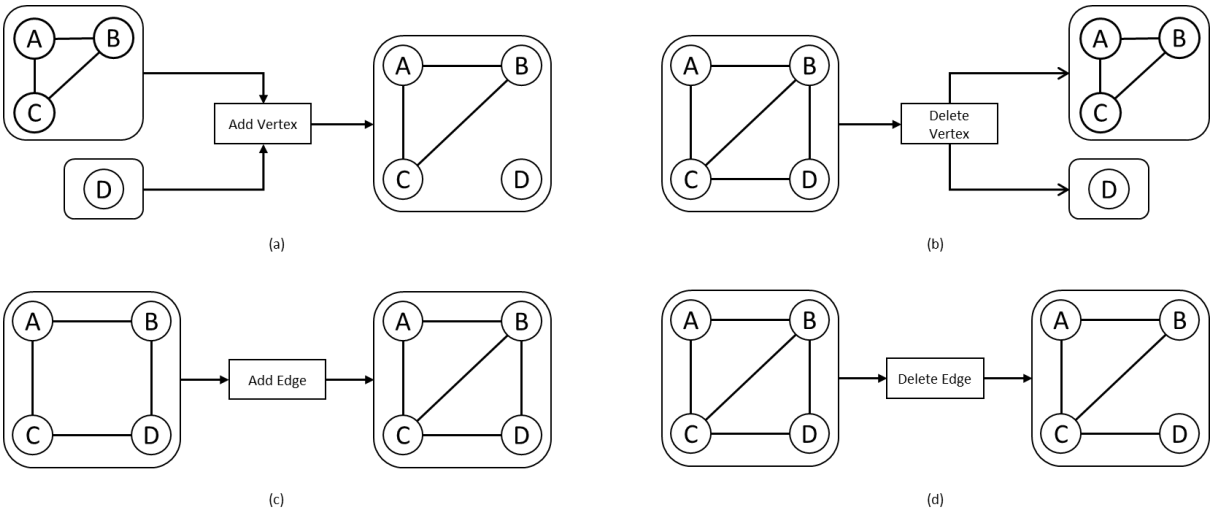


Fig. 5: Graph operations.

Add Edge

Adds a new edge between two existing vertices. Fig. 5(c) shows the graph before and after the Add Edge operation.

Delete Edge

Removes an existing edge between two vertices. Fig. 5(d) shows the graph before and after the Delete Edge operation.

Graph Algorithms

Graph traversal is the process of visiting all vertices in a graph in a specific order. Depth First Search (DFS) and Breadth First Search (BFS) are algorithms used to traverse a graph.

Depth First Search (DFS)

In DFS, descendant vertices of a vertex are processed before moving to an adjacent vertex. A stack is used to implement the DFS algorithm, which

proceeds as follows:

1. Initialize a stack and mark the starting vertex as visited.
2. Push the starting vertex onto the stack.
3. While the stack is not empty:
    - Pop the top vertex from the stack and consider it the current vertex.
    - For each adjacent, unvisited vertex of the current vertex:
        - Mark it as visited.
        - Push it onto the stack.
4. Repeat until the stack is empty, ensuring all vertices in the connected component are visited.

Refer to Fig. 6 for a diagrammatic illustration of DFS.

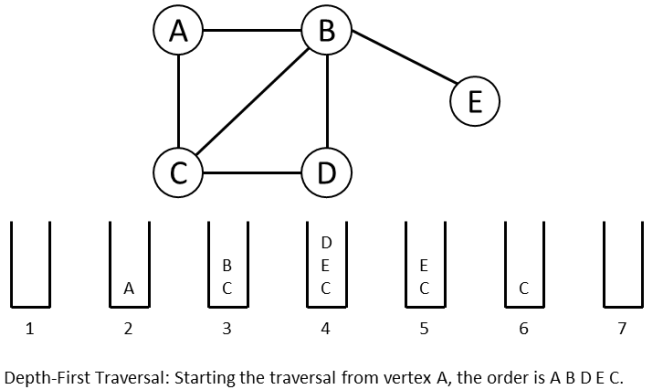


Fig. 6: Depth First Search/Traversal.

Breadth First Search (BFS)

In BFS, adjacent vertices of a vertex are processed before moving to the descendent vertex. A queue is used to implement the BFS algorithm, which proceeds as follows:

1. Initialize a queue and mark the starting vertex as visited.
2. Enqueue the starting vertex.
3. While the queue is not empty:
    - Dequeue the front vertex from the queue, making it the current vertex.
    - For each adjacent, unvisited vertex of the current vertex:

- Mark it as visited.
- Enqueue it.

4. Repeat until the queue is empty, ensuring all reachable vertices are visited level by level.

Refer to Fig. 7 for a diagrammatic illustration of BFS.

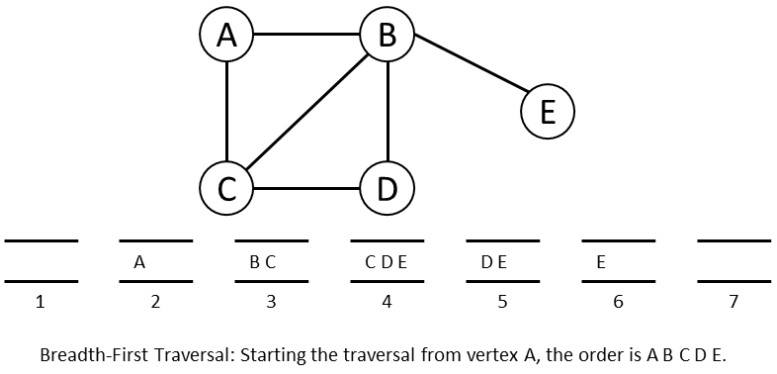


Fig. 7: Breadth First Search/Traversal.

What is a network in graph theory?

In graph theory, a **network** is a graph where vertices represent entities and edges represent connections or relationships between them, often with weights indicating distances, costs, or capacities.

Dijkstra's Algorithm

Dijkstra's Algorithm is a fundamental algorithm in graph theory used to find the shortest path from a single source vertex to all other vertices in a weighted, non-negative edge graph. Named after Dutch computer scientist Edsger W. Dijkstra, this algorithm is widely applied in fields such as network routing, geographical mapping, and urban planning.

Steps of Dijkstra's Algorithm

1. **Initialize Distances**: Assign an initial distance value to each vertex. Set the distance of the source vertex to zero and all other vertices to infinity (indicating they are initially unreachable from the source).
2. **Mark the Source Vertex as Visited**: Begin with the source vertex, marking it as visited. This ensures each vertex is processed only once, as Dijkstra's

algorithm does not revisit vertices.

3. **Update Neighbour Distances**: For the current vertex, examine all adjacent vertices. For each adjacent vertex, calculate a potential shorter path by summing the distance from the source vertex to the current vertex with the weight of the edge connecting them. If this calculated distance is shorter than the previously known distance, update the adjacent vertex's distance.
4. **Select the Next Vertex**: Choose the unvisited vertex with the smallest distance value and mark it as the new current vertex. This vertex becomes the new focus for distance updates.
5. **Repeat Until All Vertices are Visited**: Continue the process of updating distances and selecting the next vertex with the smallest distance until all vertices are visited.
6. **Output the Shortest Paths**: Upon completion, the algorithm provides the shortest path distances from the source vertex to each reachable vertex in the graph.

Dijkstra's algorithm guarantees the shortest path in graphs with **non-negative weights**, making it an efficient and reliable approach for a wide range of applications.

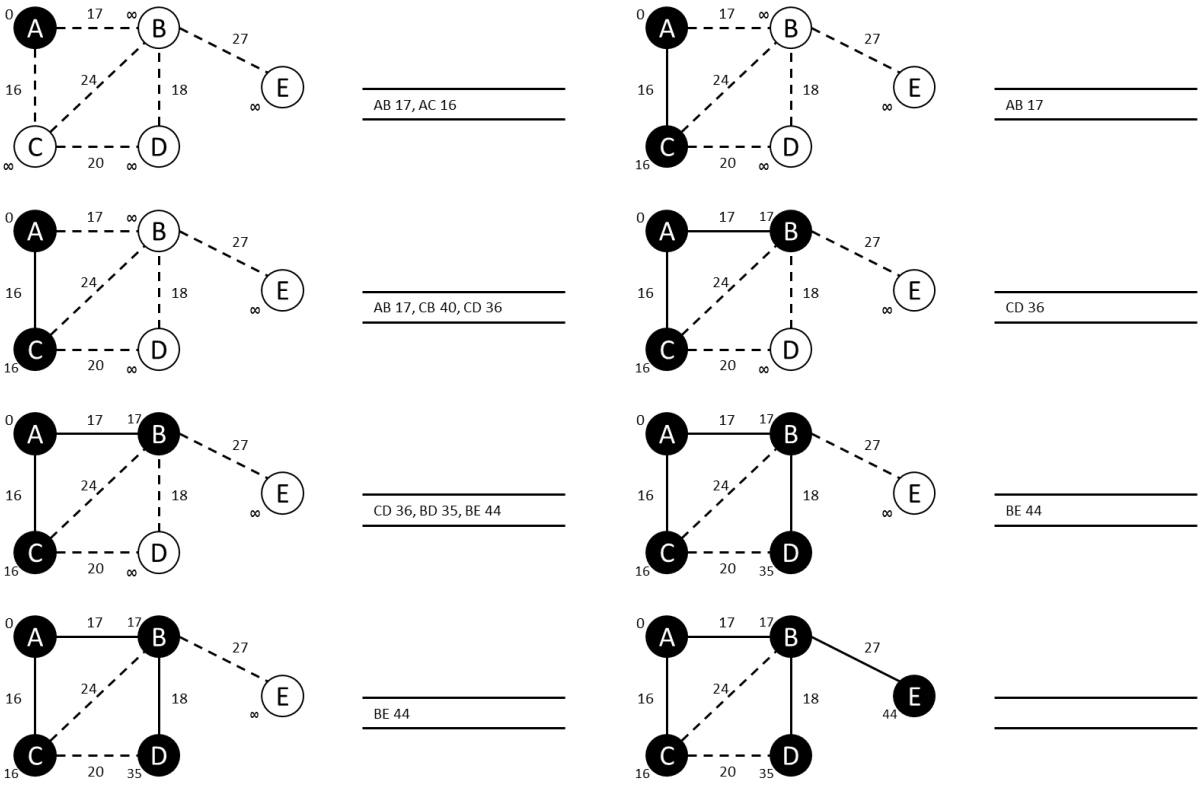


Fig. 8: Dijkstra algorithm illustration.

Why does Dijkstra’s algorithm fail for -ve weights?

Refer to the following graph. After applying Dijkstra's algorithm, the minimum distance between vertex A and C is found to be 16 units. However, this result is incorrect. By traversing from A to B to C, the distance is 0, which is less than 16. Therefore, Dijkstra's algorithm does not appear to work correctly for this graph.

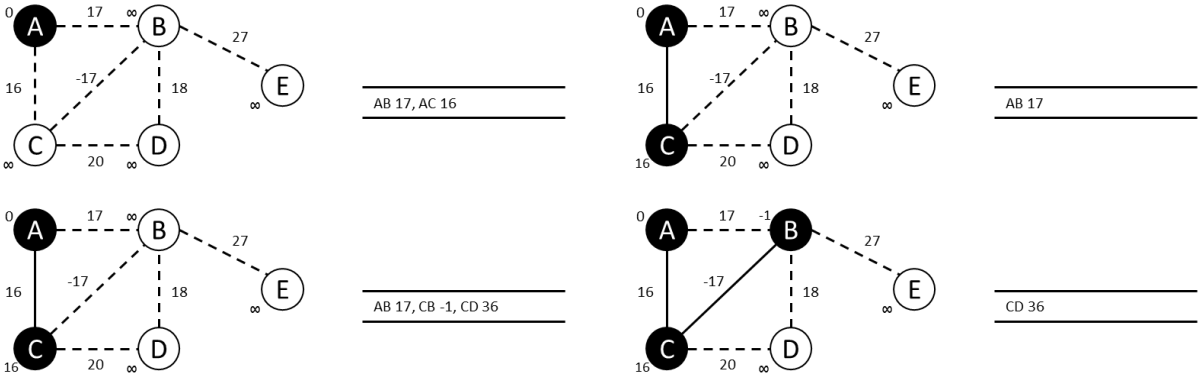


Fig. 9: Dijkstra algorithm failure illustration.

Bellman-Ford Algorithm

The Bellman-Ford Algorithm is a shortest-path algorithm designed to find the minimum distance from a single source vertex to all other vertices in a graph. Unlike Dijkstra’s algorithm, Bellman-Ford can handle graphs with negative edge weights, making it particularly useful for networks where costs may fluctuate or decrease along certain paths. However, the algorithm is not suitable for graphs with negative cycles, as paths with infinite reductions can occur.

Steps of the Bellman-Ford Algorithm

1. **Initialize Distances**: Assign an initial distance of infinity to each vertex, with the source vertex set to a distance of zero, indicating it as the starting point.
2. **Relax Edges Repeatedly**: For each vertex, perform the following process up to v - 1 times (where v is the number of vertices). For each edge (u, v) with weight w, check if the current distance to v can be minimized by taking the path through u. If so, update the distance to v as the sum of the distance to u and the weight w.
3. **Check for Negative Cycles**: After completing v - 1 iterations, go through all edges one more time. If a shorter path to any vertex is still found, a negative weight cycle exists, indicating that no finite shortest path exists for certain vertices.

4. **Output Shortest Paths**: If no negative cycles are detected, the algorithm provides the shortest path distances from the source vertex to each reachable vertex.

The Bellman-Ford Algorithm is versatile, handling graphs with negative weights while also identifying negative cycles. Its use cases include financial modelling and transportation networks where costs may vary along routes.

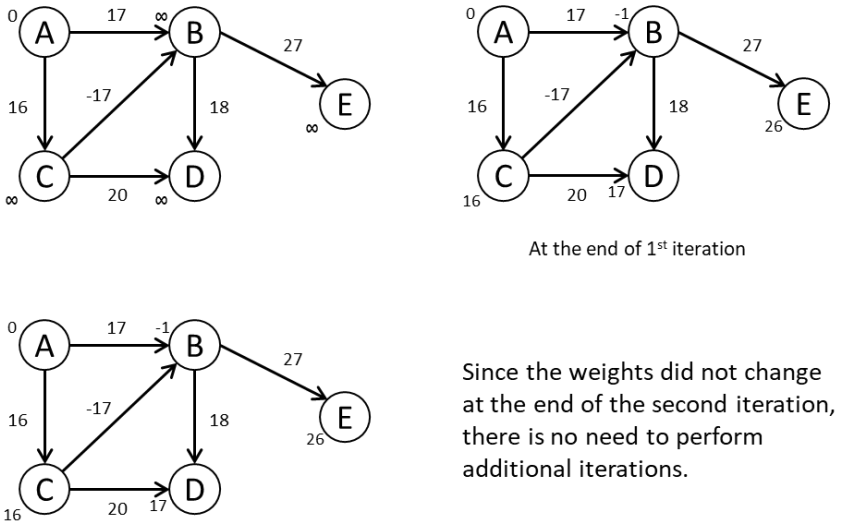


Fig. 10: Bellman-Ford Algorithm illustration

A negative cycle in a graph is a cycle where the sum of the edge weights is negative. It means that by traversing the cycle repeatedly, you can continually decrease the total path weight. Refer to Fig. 11. The cycle CBDC forms a negative cycle.

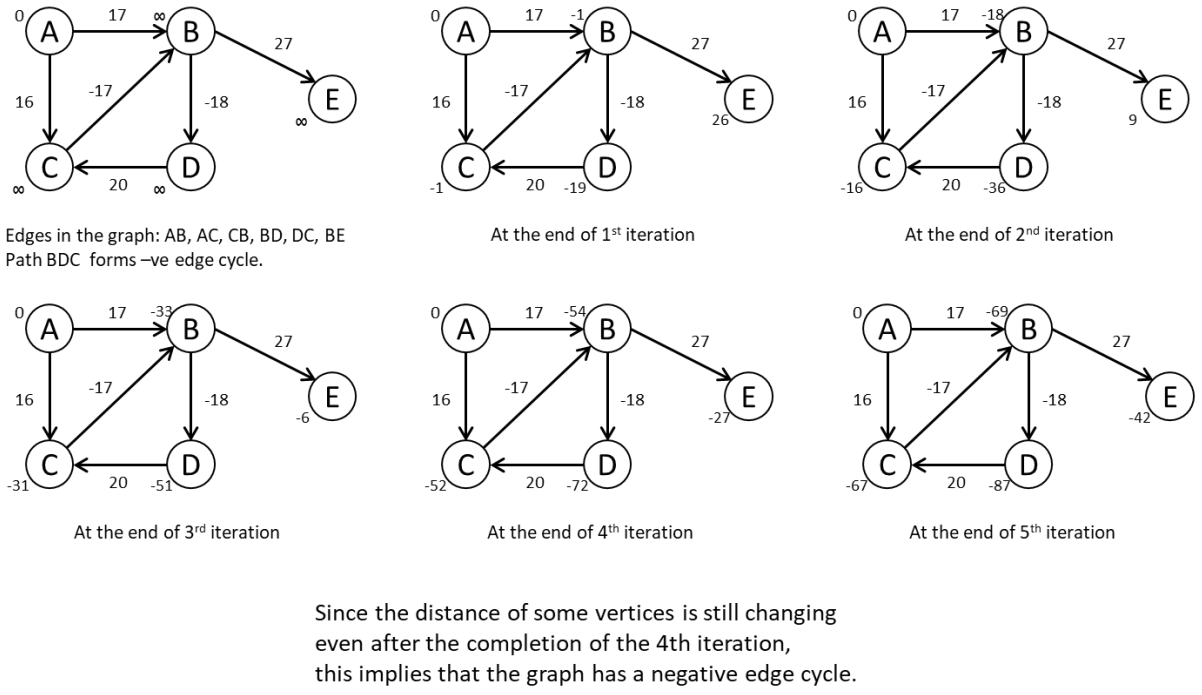


Fig. 11: Bellman-Ford Algorithm failure illustration

The Bellman-Ford Algorithm does not work correctly for undirected graphs if there are any negative edges.

Spanning Tree

A **spanning tree** of a connected, undirected graph is a subgraph that includes all the vertices of the original graph and forms a tree. This means the spanning tree has no cycles and connects all vertices with the minimum possible number of edges. Given an undirected graph with V vertices, any spanning tree will have exactly V - 1 edges.

Properties of a Spanning Tree

- **Connectivity**: A spanning tree connects all vertices of the graph, ensuring there is a path between any two vertices.
- **Acyclic**: A spanning tree has no cycles; adding any additional edge would create a cycle.
- **Uniqueness for Each Graph**: A graph can have multiple spanning trees, especially if it has multiple paths between vertices. However, each spanning tree maintains the same number of vertices and edges.

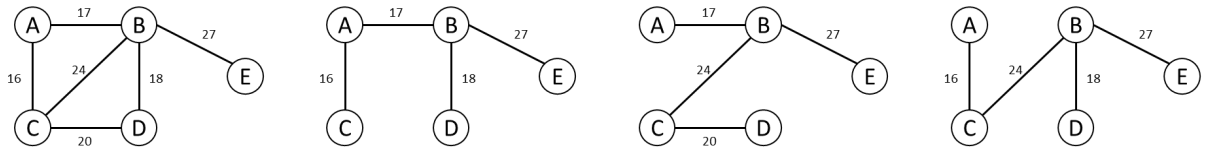


Fig. 12: Original graph and various spanning trees.

Minimum Spanning Tree (MST)

In weighted graphs, a **minimum spanning tree (MST)** is a spanning tree that has the minimum possible total edge weight among all spanning trees of the graph. MST algorithms, such as Kruskal's and Prim's, are used to find the MST, which is particularly useful in applications like network design, where minimizing costs is essential.

Spanning trees are fundamental in fields such as computer networking, circuit design, and clustering, where connecting nodes efficiently without redundancy is required.

Kruskal's Algorithm

Kruskal's Algorithm is a classic algorithm in graph theory used to find the **minimum spanning tree (MST)** of a connected, weighted, undirected graph. A minimum spanning tree is a subset of the graph's edges that connects all vertices with the minimum possible total edge weight and without any cycles. Kruskal's Algorithm is efficient and often used in network design, where reducing total connection cost is essential.

Steps of Kruskal's Algorithm

1. **Sort All Edges by Weight**: Arrange all edges in ascending order based on their weights. This ensures that the algorithm starts by examining the smallest weights first.
2. **Initialize Isolated Vertices:** Include all vertices of the graph without connecting them initially.
3. **Add Edges to the MST**: Begin iterating through the sorted edge list. For each edge, check if it connects two vertices from different subsets:
    - If it does, include the edge in the MST and merge the subsets of the two vertices.
    - If it does not (i.e., it forms a cycle with already-added edges), discard the edge.

4. **Repeat Until MST is Complete**: Continue this process until the MST contains exactly V - 1 edges, where V is the number of vertices. This guarantees that the MST spans all vertices without cycles.
5. **Output the MST**: The resulting set of edges forms the minimum spanning tree, providing the least total weight to connect all vertices.

> 💡 **What is a sparse graph?**
>
> A sparse graph is a type of graph in which the number of edges is relatively small compared to the number of vertices.

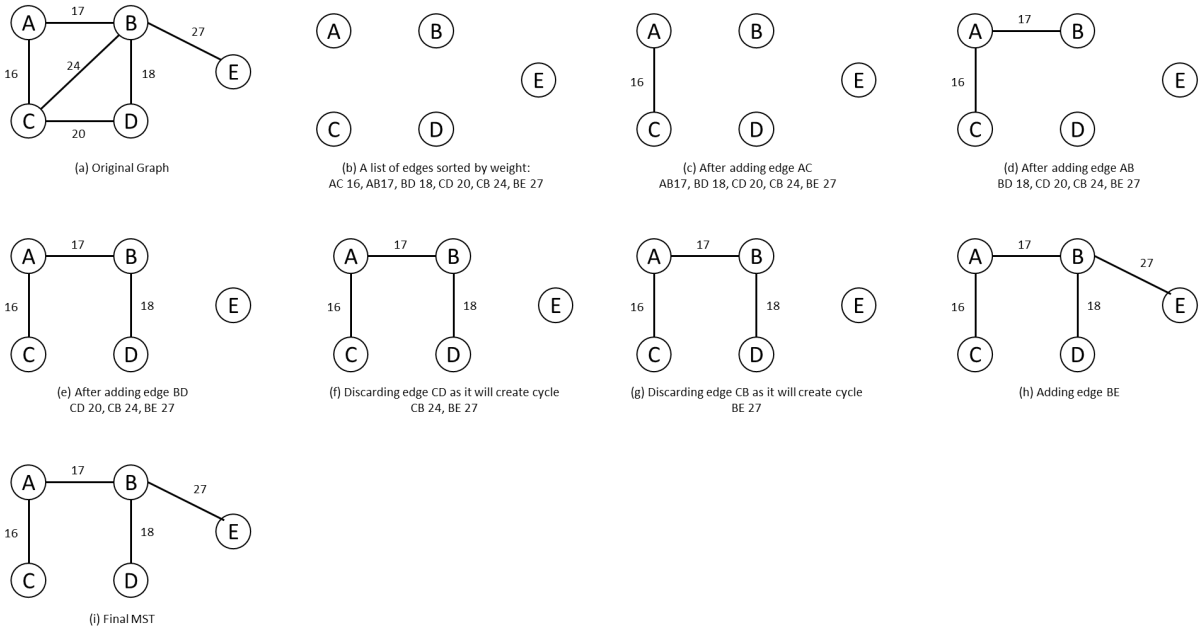


Fig. 13: Kruskal's Algorithm illustration

Prim's Algorithm

Prim's Algorithm is a popular greedy algorithm used to find the **minimum spanning tree (MST)** of a connected, weighted, undirected graph. The MST generated by Prim's Algorithm connects all vertices in the graph with the minimum possible total edge weight, ensuring no cycles. This algorithm is efficient for dense graphs and is widely used in network design, where minimizing the cost of connections is essential.

Steps of Prim's Algorithm

Step 1: Choose any random vertex to serve as the MST's initial vertex.

Step 2: Until there are vertices that are not part of the MST, proceed with steps 3 through 5.

Step 3: Identify the edges that connect any tree vertex to the unconnected vertices.

Step 4: Determine which of these edges is the minimum.

Step 5: If the selected edge does not form a cycle, add it to the MST.

Step 6: Return the MST and exit

Prim's Algorithm is particularly efficient with priority queues and is commonly used in real-world applications such as designing efficient road networks, electrical grids, and other minimum-cost spanning systems.

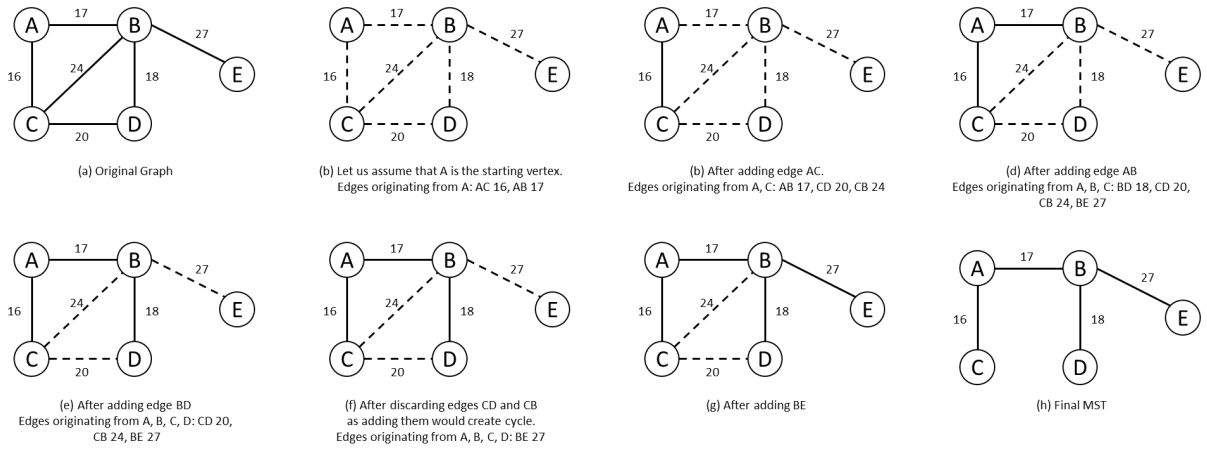


Fig. 14: Prim's Algorithm illustration.

Applications

Representing a City as a Graph

A developer has been tasked with modelling a city to better understand and optimize its infrastructure. The developer decides to use a graph data structure, as it provides a clear and effective way to represent the city's layout and connections.

In this model, each **vertex** (or node) represents a distinct location within the city, such as intersections, landmarks, or neighbourhoods. These vertices capture all the key points that people may travel to or from within the city.

The **edges** in the graph represent the **roads** or **pathways** that connect these locations. By defining edges between vertices, the developer can represent all

major and minor routes between areas in the city. This structure enables a straightforward approach to analyzing travel paths, distances, and connectivity.

Using this graph representation, the developer can implement and apply various algorithms to solve real-world urban problems, such as finding the shortest path between two locations, optimizing traffic flow, designing public transport routes, and ensuring efficient routing for emergency services. The graph data structure thus becomes a powerful tool for modelling, analyzing, and improving city infrastructure.

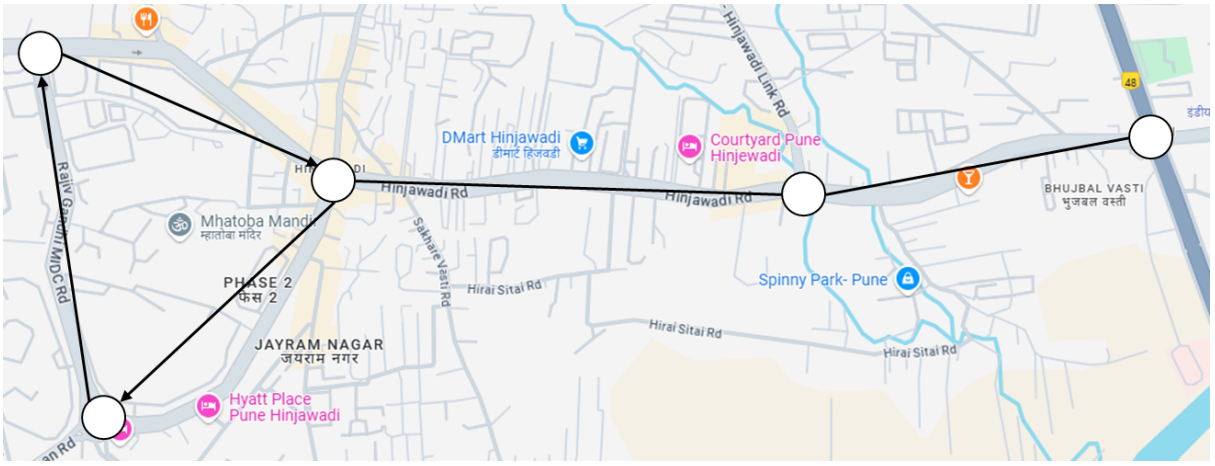


Fig. 15: Representing a city as a graph.

Modelling a PERT Chart as a Graph

A graph can be effectively used to represent a PERT (Program Evaluation and Review Technique) chart, which is a tool for project management that helps visualize task sequences, dependencies, and the critical path for completing a project.

Each node in the graph represents a **specific event** or **milestone** in the project, marking the start or end of a particular task or a set of tasks. The nodes help track when each phase of the project begins or ends. The directed edges (arrows) between nodes indicate **tasks** or **activities** that connect these events, showing the flow from one task to the next and highlighting dependencies.

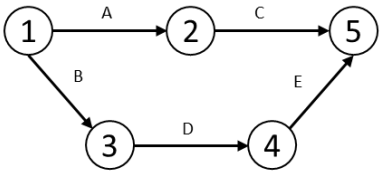


Fig. 16: PERT chart as a graph.

Alternatively, nodes can represent individual tasks and edges as task dependencies, particularly in PERT versions that emphasize task-level tracking.

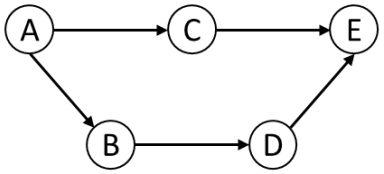


Fig. 17: Task-level tracking.